GeoSpark: A Cluster Computing Framework for Processing 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 RDD 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). The 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 and Join) on SRDDs. GeoSpark adaptively decides whether a spatial index needs to be created locally on an SRDD partition to strike a balance between the run time performance and memory/cpu utilization in the cluster. Extensive experiments show that GeoSpark achieves better run time performance (with reasonable memory/CPU utilization) than its Hadoop-based counterparts (e.g., SpatialHadoop) in various spatial data processing applications.

1. INTRODUCTION

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. In consequence, everything we do on the mobile internet leaves breadcrumbs of spatial digital traces, e.g., geo-tagged tweets, venue checkins. 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. , (3) Commerce and Advertisement: e.g., point-of-interest (POI) recommendation services. The aforementioned applications needs a powerful data management platform to handle the large volume of spatial data. 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 query processing techniques. Moreover, big 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, effectively stores it, and allows applications to efficiently retrieve it when necessary.

- **Challenge II: Interactive Performance.** User will not tolerate delays introduced by the underlying spatial database system to execute queries efficiently. Instead, the user needs to see useful information quickly. Hence, the underlying spatial data processing system must figure out effective ways to process user's request in a sub-second response time.

Existing spatial database systems (DBMSs) [14] extend relational DBMSs with new data types, 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, 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., [2, 6]) extend the Hadoop [5] ecosystem to perform spatial analytics at scale. Although the Hadoop-based approach achieves high scalability and exhibits excellent performance in batch-processing jobs, it shows poor performance handling applications that require interactive performance. Apache Spark [17], on the other hand, is an in-memory cluster computing system. Spark provides a novel data abstraction

called resilient distributed datasets (RDDs) [18] 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 presents 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. This problem is quite challenging due to the fact that (1) spatial data may be quite complex, e.g., rivers' and cities' geometrical boundaries, (2) spatial (and geometric) operations (e.g., Overlap, Intersect, Convex Hull, Cartographic Distances) 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 Geospatial Consortium (OGC) [13] 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 and Join query) on large-scale spatial datasets. After geometrical objects are retrieved in the Spatial RDD layer, users can invoke spatial query processing operations provided in the Spatial Query Processing Layer, decide how spatial objects could be stored, indexed, and accessed using SRDDs, and return the spatial query results required by user. In summary, the key contributions of this paper are as follows:

- GeoSpark as a full-fledged cluster computing framework to load, process, and analyze large-scale spatial data in Apache Spark.
- 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.
- Spatial data indexing strategies that partition the input Spatial RDD using a grid structure and assigns grids to machines for parallel execution. GeoSpark also adaptively decide 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.
- Extensive experimental evaluation that benchmarks the performance of GeoSpark in spatial analysis applications like spatial join, spatial aggregation, and spatial co-location pattern recognition. The experiments also compare and contrast the GeoSpark with existing Hadoop-based systems (i.e., SpatialHadoop).

[1]The source code is available at `https://github.com/Sarwat/GeoSpark`

The rest of this paper is organized as follows. Section 2 highlights the related work. GeoSpark architecture is given in Section 3. Section 4 presents the Spatial Resilient Distributed Datasets (SRDDs) and Section 5 explains how GeoSpark efficiently processes spatial queries on SRDDs. Section 7 experimentally evaluates GeoSpark . Finally, Section 8 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. There are some real scenarios in life: Return all parks in Phoenix or return all restaurants within one mile of my current location. In terms of the format, spatial range query needs one set of points or polygons and one query window as inputs and returns all the points / polygons which lie in the query area. Spatial join queries are queries that combine two datasets or more with a spatial predicate, such as distance relations. There are also some real scenarios in life: tell me all of the parks which have rivers in Phoenix and tell me all of the gas stations which have grocery stores within 500 feet. Spatial join query needs one set of points, rectangles or polygons (Set A) and one set of query windows (Set B) as inputs and returns all points and polygons that lie in each one of the query window set. Spatial query processing algorithms usually make use of spatial indexes to reduce the query latency. For instance, R-Tree [8] 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. R-Tree is a balanced search tree and obtains better search speed and less storage utilization. However, its performance could be reduced by heavy update activities. Quad-Tree [7, 16] is another spatial index which is used to recursively divide a two-dimensional space into four quadrants. Quad-Tree fits uniform data well and heavy update activities do not affect its performance.

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 [2] utilizes global partition indexing and customizable on demand local spatial indexing to achieve efficient query processing. And also Hadoop-GIS can support declarative spatial queries with an integrated architecture wit HIVE. SpatialHadoop [6], a comprehensive extension to Hadoop, has native support for spatial data by modifying the underlying code of Hadoop. The spatial functions it can provide include Grid, R-tree, R+-tree, spatial range query, kNN and spatial join query. MD-HBase [12] 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 [9] combines Hadoop with SECONDO, a database which can handle non-standard data types, like spatial data, usually not supported by standard systems. It uses Hadoop as the distributed task manager and does distributed operations on spatial DBMS of multiple nodes. 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.

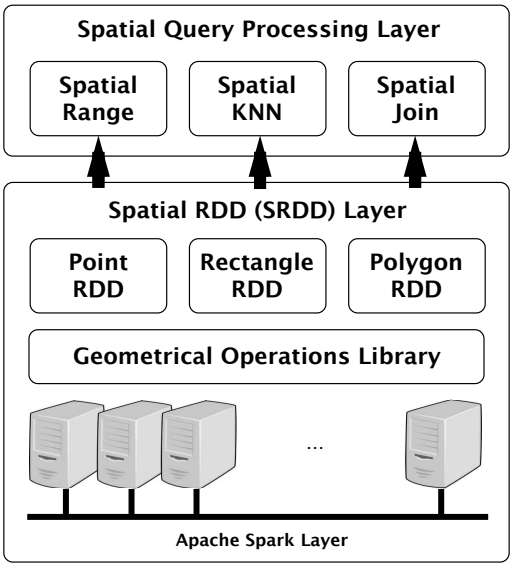


Figure 1: GeoSpark Overview

3. GEOSPARK OVERVIEW

As depicted in Figure 1, GeoSpark consists of three main layers: (1) *Apache Spark Layer*: this layer provides the basic Apache Spark functionality which includes the RDD concept along with its actions and transformations. (2) *Spatial Resilient Distributed Dataset (SRDD) Layer*: this layer extends the regular RDD to support geometrical objects (i.e., points, rectangles, and polygons) as well as geometrical operations on these objects. (3) *Spatial Query Processing Layer*: this layer harnesses and extends the SRDD layer to execute spatial queries (e.g., Range query and Join query) on large-scale spatial datasets.

Apache Spark Layer. The Apache Spark Layer consists of regular operations that are natively supported by Apache Spark. That consists of loading / saving data from / to persistent storage (e.g., stored on local disk or Hadoop file system HDFS). For instance, the `PersistToFile()` persists the dataset in one instance to a file on disk or on Hadoop Distributed File System. This operation requires a file system path from users and then persists the dataset in this instance to that path.

Spatial RDDs 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 Query Processing Layer. Based on Spatial RDD layer, Spatial Query Processing Layer supports spatial queries (e.g., Range query and Join query) for large-scale spatial datasets. After geometrical objects are stored and processed in Spatial RDD layer, user can call spatial queries provided in Spatial Query Processing Layer and GeoSpark processes such query in the in-memory cluster and returns the final results to the user.

`SpatialRangeQuery()` – This query requires a query area from users as the input, which can be one circle or random polygon, accesses the Spatial RDD, and finds all of the points, rectangles or polygons which fall into the query area. The result of this query is stored in one instance of Spatial RDD. User can decide whether to do further calculations or persist it on disk.

`SpatialJoinQuery()` – In this query, one query area set is joined with one Spatial RDD. The query area set which is composed of rectangles or polygons can be also stored in Spatial RDD. GeoSpark then joins the two Spatial RDDs and returns a new Spatial RDD instance which is extended from the original SRDDs. For one query area, the object contained by it will be attached behind it in this instance.

4. SPATIAL RDD (SRDD) LAYER

This section describes the details inside spatial RDDs. Spatial RDDs are intuitively extension of traditional RDDs. Spatial data can be easily stored in SpatialRDD, processed by geometrical library and queried by spatial query layer swiftly.

4.1 Spatial Objects Support (SRDDs)

GeoSpark supports various spatial data input format (e.g., Comma Separated Value, Tab Separated Value and Well-Known Text). Different from the era users spend time on parsing input format by themselves, GeoSpark users only need to specify the format name and the start column of spatial data and GeoSpark will take over the data transformation and store processed data in SpatialRDDs automatically.

At the storage level, GeoSpark takes advantages of JTS Topology Suite [4] to support spatial objects. Each spatial object is stored as a point, rectangle or polygon type. In terms of the type of spatial objects, spatial RDDs (SRDDs) are defined as follows:

PointRDD. PointRDD supports all of the 2D Point objects (that represent points on the surface of the earth) with the following format:⟨ Longitude, Latitude ⟩. All points in a PointRDD are automatically partitioned by Apache Spark Layer and assigned to machines accordingly.

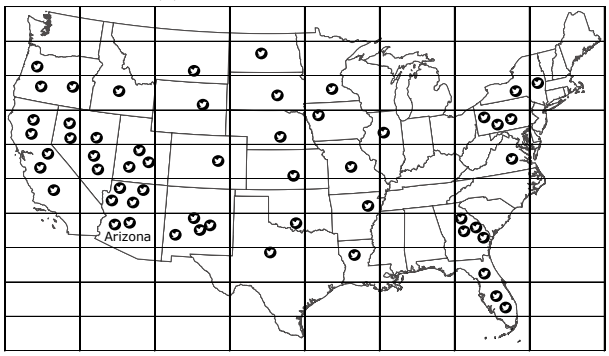
RectangleRDD. RectangleRDD supports rectangle objects in the following format: ⟨ Point A (Longitude, Latitude) and Point B (Longitude, Latitude)⟩. Point A and Point B are a pair of vertexes on the diagonal of one rectangle. Rectangles in a RectangleRDD are also distributed to different machines by Apache Spark Layer.

PolygonRDD. All random polygon objects are supported by PolygonRDD. The required format of PolygonRDD is as follows: ⟨ Point A (longitude, Latitude), Point B (Longitude, Latitude), Point C,...⟩. The number of columns has no upper limit. Underlying Apache Spark Layer partitions PolygonRDDs to distributed clusters.

4.2 SRDDs Built-in Geometrical Operations

GeoSpark provides built-in operations for spatial RDDs. Once a spatial RDD is initialized, the built-in operations of this spatial RDD come to be available to users. Via well-defined invocation API, users can execute complex operations on spatial data stored in spatial RDDs efficiently without involving the implementation details of these functionalities. From an implementation perspective, these operations interact with Apache Spark Layer through Map, Sort, Filter, Reduce and so on. In this transparent procedure, users

(a) Tweets spatial distribution



(b) Tweets spatial distribution with a grid

Figure 2: Geo-Tagged Tweets in the United States

only focus on spatial analysis programming details without any knowledge about the underlying processes.

GeoSpark provides a set of geometrical operations which is called Geometrical Operations Library. This library provides native support for geometrical operations that follow the Open Geospatial Consortium (OGC) [13] standard. Example of the geometrical operations provided by GeoSpark are as follows:

`Initialize()` – The functionality of this operation is to initialize a Spatial PointRDD, RectangleRDD or PolygonRDD which supports three common geometrical objects, point, rectangle and polygon, and related operations. The operation parses input data and stores it with spatial object type. The dataset should follow the required format in the corresponding GeoSpark Spatial RDDs.

`Overlap()` – In one Spatial RDD, the goal of this operation is to find all of the internal objects which are intersected with others in geometry.

`Inside()` – In one Spatial RDD, this operation can find all of the internal objects which are contained by others in geometry.

`Disjoint()` –In one Spatial RDD, this operation returns all of the objects which are not intersected or contained by one particular object or object set in geometry. The parameter of this operation is the instance of the object or object set.

`MinimumBoundingRectangle()` – This operation 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()` – In one Spatial PolygonRDD, this operation could return the union polygon of all polygons in this RDD.

4.3 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 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. Global grids are low cost in either file byte-size or data partitioning. On the other hand, the construction of global grid file is an iterative job which requires multiple coordinate sorting on the same datasets.

Data: Input Spatial RDDs
`/* Step1:Create a global grid file has` N `grids */`
Find the minimum geo-boundary for two inputs;
Create a grid file; `/* Each grid has equal geo-size */`
`/* Step2:Assign gridID to each element */`
foreach *spatial object in the SRRDs* **do**
  for $grid = 0$ **to** N **do**
    if *this grid contains / intersects with this Spatial Object* **then**
      Assign this grid ID to this Spatial Object;
    end
    `/* Duplicates happen when one spatial object intersects with multiple grids */`
  end
end

Algorithm 1: Data Partitioning

To partition the input datasets, GeoSpark performs the following steps for all spatial RDDs: (1) Load the original datasets from the data source, transform the original datasets to extract spatial information and store them in regular RDD. Meanwhile, cache the RDD into memory for the next iterative job. (2) Traverse the coordinates in the input RDDs multiple times to find their Minimum Bounding Rectangles. (3) Calculate the grid file boundary (GLB) which is the intersection of the two MBRs. (4) Pre-filter the two datasets by GLB to remove the elements never overlap with others. This step is especially suitable for the case that two spatial datasets are not in the same spatial space and only have a small intersection area.

An example of global grids is shown in Figure 2b. Elements, points (Tweets) or polygons (States), lie in a lot of same size grids. The number of grids may impact query performance. Section 6 provides more explanation about it. Global grids construction needs to sort the original dataset by coordinates iteratively to find the geographical boundary (Minimum Bounding Rectangle) . For instance, sort by X-coordinate and Y-coordinate in point set. It might be the most time-consuming step in this algorithm if it relies on other computing framework (i.e., Hadoop). However, GeoSpark counteracts this perfectly with Apache Spark cache in memory feature. GeoSpark caches (or partially caches) the target dataset in memory meanwhile executes the first time sorting. The next sorting for the cache will only cost millisecond time.

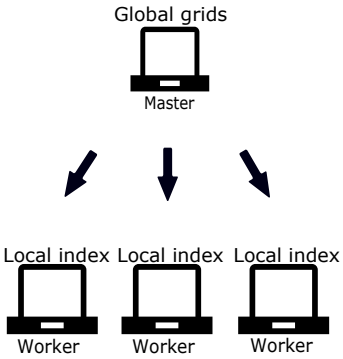


Figure 3: GeoSpark execution model

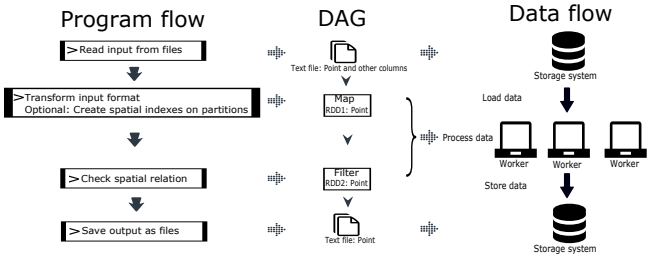


Figure 4: Range query program flow, DAG and data flow

5. SPATIAL QUERY PROCESSING LAYER

The section describes the implementation details of spatial query processing layer in GeoSpark .

5.1 Execution Model

Figure 3 gives the general execution model followed by GeoSpark . To accelerate a spatial query, GeoSpark leverages the grid partitioned Spatial RDDs, the fast in-memory computation and DAG scheduler of Apache Spark to parallelize the query execution. Spatial indexes like Quad-Tree and R-Tree are also provided in spatial query processing layer. Spatial query algorithms avoid reduce-like tasks as possible as they can. As we mentioned before, reduce like tasks may result in data shuffle across the cluster and data shuffle is expensive in terms of either time or resources. Users specify whether GeoSpark should consider local spatial indexes. GeoSpark adaptively decides whether a local spatial index should be created for a certain SRDD 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. Since index building is an additional overhead, GeoSpark executes a full spatial object scan or nested loops (in case of join) for some SRDD partitions that only have very few spatial objects. This execution model has the following advantages:

- **Efficient data partitioning:** After the SRDD data is partitioned according the girds, only elements lie inside the same grid need to be calculated the spatial relations. Clusters do not need to spend time on those spatial objects in different grid cells which have are guaranteed not to intersect.
- **Available local spatial indexes:** For elements lie in the same grid, GeoSpark can create local spatial indexes like Quad-Tree or R-Tree on-the-fly. Index-based spatial query may exhibit much higher efficiency than scan-based or nested loop algorithms.

In the rest of this section, we present how GeoSpark uses the aforementioned execution model to process range and join queries. The algorithm for running K-Nearest Neighbor queries (however the main idea is similar) is omitted for the sake of the space.

5.2 Spatial Range Query

Generally speaking, spatial range query is fast and less resource-consuming. Therefore, the first priority of spatial range query is to walk around the unnecessary overhead and keep the algorithm neat and efficient. Spatial indexes may also improve the query performance. GeoSpark implements the spatial range query algorithm in the following steps:

1. Broadcast the query window to each machine in the cluster and Create a spatial index on each Spatial RDD partition if necessary.
2. For each SRDD partition, if a spatial index is created, use the query window to query the spatial index. Otherwise, check the spatial predicate between the query window and each spatial object in the SRDD partition. If the spatial predicate holds true, the algorithm adds the spatial object to the result set.
3. Remove spatial objects duplicates that existed due to the global grid partitioning phase.
4. Return the result to the next stage of the spark program (if needed) or Persist the result set to disk.

For a better understanding of GeoSpark spatial range query, the DAG, data flow and programming flow are described in Figure 4. As the data flow shows, GeoSpark processes data in parallel without data shuffle to achieve better performance.

5.3 Spatial Join Query

As mentioned earlier, to accelerate the speed of a spatial join query, almost all of the algorithms creates a spatial index or grid file. However, a spatial join operation usually iterates the original dataset multiple times to have some global parameters like boundary or spatial layout. This procedure is routine and time-consuming. Thanks to the in-memory computing feature of Apache Spark, the efficiency of iterative jobs could be significantly improved. Therefore, spatial join queries in GeoSpark can achieve significantly better performance.

GeoSpark implements the parallel join algorithm proposed by [19] and [10]. The algorithm first traverses the spatial objects in the two input SRDDS. If spatial object lies inside one grid cell, the algorithm assigns the grid ID to this element. 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. The algorithm then 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, the algorithm calculates their spatial relations. If two elements from two SRDDS are overlapped, the algorithm keeps them in the final results. The algorithm then

Data: PointRDD and RectangleRDD
Result: One *Key-Values* set in (Rect, Point, Point,...)
Return two *Key-Value* sets in (gridID, Rectangle) and (gridID, Point);
`/* Step1: Local spatial join execution */`
for $gridID = 0$ **to** N **do**
  for *each PointRDD partition in gridID* **do**
    Create a spatial index if necessary;
  end
  foreach *RectangleRDD partition in gridID* **do**
    foreach *Rectangle in RectangleRDD partition* **do**
      `/* Index-based query */`
      if *index exists* **then**
        Search the spatial index ;
        Record the result for this rectangle;
      end
      `/* Nested loop query */`
      else
        foreach *point has this gridID* **do**
          if *this rectangle contains this point* **then**
            Record this point for this rectangle;
          end
        end
      end
    end
  end
end
Return a *Key-Values* set in (Rectangle, Point, Point, ...);
`/* Step 2:Remove duplicates */`
foreach *rectangle occurs > one times as Key in the result* **do**
  Combine their *Values* and delete duplicates;
end
Return a *Key-Values* set in (Rectangle, Point, Point, ...);

Algorithm 2: Spatial join with global grids and local index

groups the results for each rectangle. The grouped results are in the following format: Rectangle, Point, Point, Point. Finally, the algorithm removes the duplicated points and returns the result to other operations in the Spark DAG or saves the final result to disk.

Minimized data shuffle. GeoSpark pays lots of effort on decreasing the data shuffle scale to achieve a better performance. Data shuffle appears two times in GeoSpark 's spatial join algorithm. They are caused by Join and GroupByKey respectively as shown in the DAG execution diagram given in Figure 5. These two data shuffle operations are inevitable. However, the scale of data shuffling operations has been significantly decreased by the filters performed by GeoSpark before them. Actually, there is another small data shuffle when the cached Spatial RDDs are sorted for the MBRs. It is not included in the figure because it doesn't change the data and has no impacts on the main data flow.

6. GEOSPARK USE CASES

This section describes three example applications as use cases for GeoSpark .

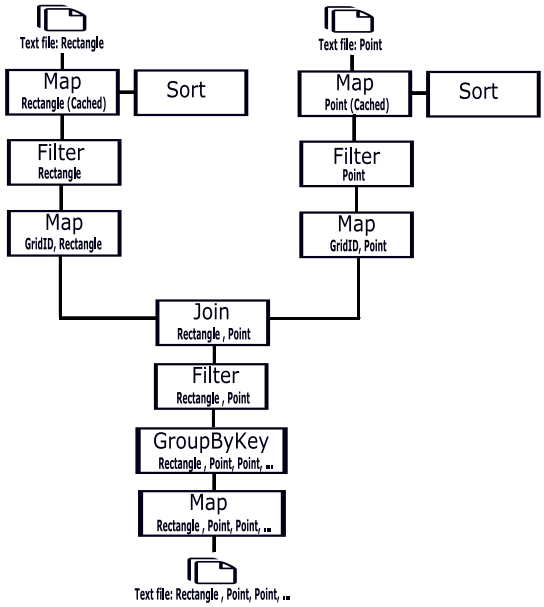


Figure 5: Spatial join query DAG

6.1 Application 1: 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 regions or polygons which compose the map of San Francisco. The number of regions depends on the display resolution, or granularity, in the heat map. The GeoSpark program is as follows (code given in Figure 6): (1) Call GeoSpark PointRDD initialization method to store the dataset of trees in memory. (2) Call GeoSpark `SpatialJoinQuery()` in PointRDD. The first parameter is a set of polygons which can be stored in Spatial PolygonRDD or regular list. The second one is "count" which means count the number of trees fall in each of the regions.(3) Use a new instance of Spatial PolygonRDD to store the result of Step (2). Step (2) returns the count for each polygon. The format of each tuple is 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.

6.2 Application 2: 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 exponents in spatial autocorrelation. Based on the exponents, analysts can tell whether these objects influence each other. Global Moran's I and Geary's C reflect the spatial autocor-

```
/* San Francisco Trees Heat Map */
public void SFTreeHeatMap(){
  PointRDD USTrees =
   PointRDD.Initialization (SparkContext,DatasetLocation);
  PolygonRDD SFTreesRegions =
  USTrees.SpatialJoinQueryWithIndex(SFRegions,"RTree");
  SpatialPairRDD SFTreesCount =  SFTreesRegions.CountByKey();
   SFTreesCount.Persistence(SFTreesCountResultPath);
 }
```

Figure 6: *Trees Heat Map* (Java code) in GeoSpark

```
/* Global Adjacency Matrix */
public void GlobalAdjMat() {
  PointRDD targetSet =
   PointRDD.Initialization (SparkContext,DatasetLocation);
  RDD globalAdjacentMatrix =
   targetSet.SpatialJoinQuery(targetSet, WITHIN, 10);
  globalAdjacentMatrix.Persistence(MatrixLocation);
}
```

Figure 7: *Adjacency Matrix* (Java code) in GeoSpark

relation for the whole dataset. Compared with global exponents, local Moran's I and Geary's C reflect the correlation between one specific object and its neighbors. Moran's I and Geary's C indexes are defined by two specific formulas. An important part of these formulas is to find the spatial adjacent matrix. For global and local exponents, there are corresponding global adjacent matrix and local adjacent matrix. In this matrix, each tuples stands for whether two objects, such as points, rectangles or polygons, are neighbors or not in the spatial space.

An application programmer may leverage the Spatial RDDs and the spatial query processing layer provided by GeoSpark to implement the spatial autocorrelation analysis procedure (Code given in Figure 7). 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. (2) Call GeoSpark spatial join query in PointRDD. The first parameter is the query point set itself and the second one is the query distance. In this case, we assume the query distance is 10 miles. (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 in 10 miles. The format is like this: Point coordinates (longitude, latitude), neighbor 1 coordinates (longitude, latitude), neighbor 2 coordinates (longitude, latitude), ... (4) Call persistence method in Spark to persist the resulting PointRDD.

6.3 Application 3: Spatial Co-location

Spatial co-location is defined as two or more species are ofter located in a neighborhood relationship. Ripley's K function [15] is often used in judging co-location. It usually executes multiple times and form a 2-dimension curve for observation. The calculation of K function also needs the adjacent matrix between two type of objects. As we mentioned in spatial autocorrelation analysis, adjacent matrix is the result of a join query.

The procedure in GeoSpark to find this matrix has the following steps: (1) Call GeoSpark PointRDD initialization method to store the two datasets in memory. (2) Call GeoSpark spatial join query in one of the PointRDDs. The first parameter is another PointRDD and the second one is the query distance. In this case, we assume the query distance is 10 miles. (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 in 10 miles. The format is like this: Point coordinates (longitude, latitude), neighbor 1 coordinates (longitude, latitude), neighbor 2 coordinates (longitude, latitude), ... (4) Call persistence method in Apache Spark Layer to persist the resulting PointRDD.

7. EXPERIMENTS

This section provides a comprehensive experimental evaluation that studies the performance of GeoSpark .

Compared approaches. we compare the following spatial data processing approaches:

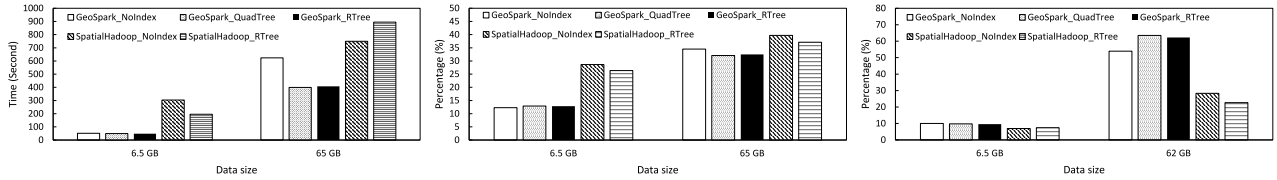
- `GeoSpark_NoIndex`: GeoSpark approach without spatial index. In this approach, data is only partitioned according grids.
- `GeoSpark_QuadTree`: GeoSpark approach with spatial Quad-Tree index. In this approach, spatial Quad-Tree is created on each partitions after data partitioned according grids.
- `GeoSpark_RTree`: GeoSpark approach with spatial R-Tree index. In this approach, spatial R-Tree is created on each partitions after data partitioned according grids.
- `SpatialHadoop_NoIndex`: SpatialHadoop approach without spatial index.
- `SpatialHadoop_RTree`: SpatialHadoop approach with spatial R-Tree index.

Cluster. Our cluster setting on Amazon EC2 is as follows: (1) Cluster size: 17 nodes which have 16 r3.2xlarge workers and 1 c4.2xlarge master. (2) Operating System per node: Ubuntu Server 14.04 LTS 64-bit. (3) CPU per worker node: Eight Intel Xeon Processor operating at 2.5 GHz with Turbo up to 3.3 GHz. (4) Memory per worker node: 61 GB in total and 50 GB registered memory in Spark and Hadoop. (5) Storage per worker node: Amazon Elastic Block Store general purpose SSD 100 GB. (6) Max throughput per worker node: 800 MBps.

Datasets. We use three real spatial datasets from TIGER project [3] 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 [1].

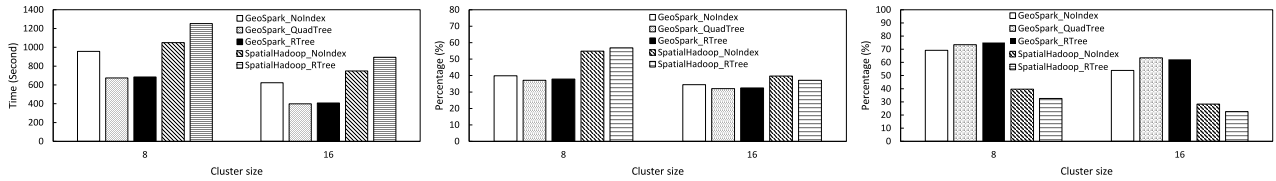
Metrics. We use three metrics to measure GeoSpark performance. They are: (1) Run time: It stands for the total program run time of one spatial analysis application. (2) Memory utilization: It stands for average memory utilization during one spatial analysis application. (3) CPU utilization: It stands for average CPU utilization / computation power during one spatial analysis application.

Monitoring tool. We deploy Ganglia [11], a scalable distributed monitoring system for high performance computing



(a) Run time (b) Necessary memory utilization (c) CPU utilization

Figure 8: Approaches on different size data with GeoSpark and SpatialHadoop



(a) Run time (b) Necessary memory utilization (c) CPU utilization

Figure 9: Approaches on different size clusters with GeoSpark and SpatialHadoop

Table 1: Join query time with different grid numbers

Grids	2500	10000	40000
NoIndex	1094s	624s	916s
QuadTree	397s	399s	709s
RTree	385s	408s	735s

systems such as clusters, on our Amazon EC2 experimental cluster. Our Ganglia deployment is similar with that of Apache Spark as well as Apache Hadoop. The Ganglia client program records CPU utilization and memory utilization of its host machine per second and the Ganglia server program collects the metadata from the clients across the cluster automatically.

7.1 Global grid number in spatial join query

As we mentioned in Section 5, the number global grids may impact the spatial join query performance. Generally speaking, small grid number means more heavy local query on each node while the large one means more time on assigning grid IDs to elements. To have a better understanding of the grid number setting, we choose different global grid number and join TIGER Zcta510 1.5 GB dataset with Edges 62 GB dataset on our 16 workers cluster. The experiment result is shown in Table 1.

As Table 1 shows, GeoSpark join query without local spatial index has the least run time when the grid number is 10000 while the run time of join query with Quad-Tree and R-Tree is in sub-linear growth according to the growing of grid number. This makes sense because that join queries spend more time on local query if the grid number is too small. When the grid number is too large, they spend more time on the loop for assigning grid IDs to elements. And also, small size grid cells caused by large number of grids might be covered by elements in datasets and result in much more duplications when assign grid IDs as well as more time when remove duplications. However, join query with help of spatial index still spends relative less time on local query even if the grid number is very small.

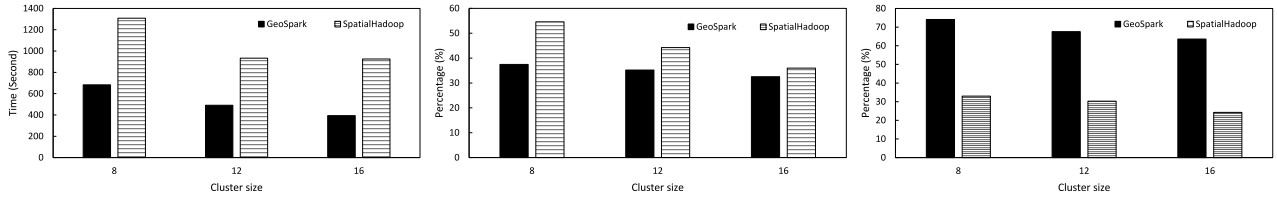
Based on the curve in Table 1, GeoSpark decides the grid number automatically with a default grid element factor which means the upper limit of the elements can lie inside one grid. Users can also set their own grid element factor for their own cases.

7.2 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 8.

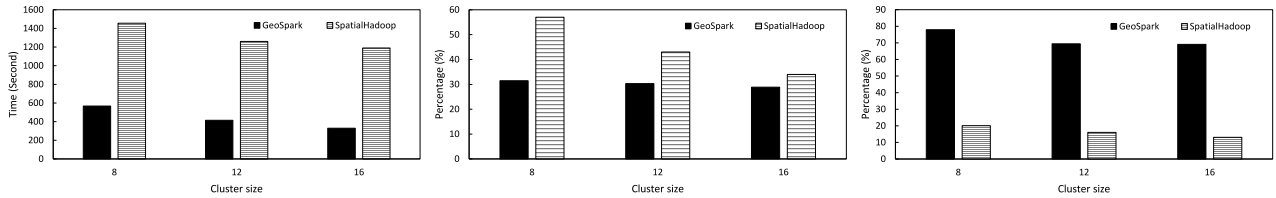
As depicted in Figure 8, GeoSpark and SpatialHadoop cost more run time on the large dataset than that on the small one. However, GeoSpark achieves much better run time performance than SpatialHadoop in both of the two datasets. This superiority is more obvious on the small dataset. The reason is that GeoSpark can cache more percentage of the intermediate data in memory on the small scale input than that on the large one. Caching more intermediate data can accelerate the processing speed.

From a memory utilization perspective, GeoSpark and SpatialHadoop may crash if the registered memory is lower than the necessary memory. Normally, Spark will use the left whole memory which is the difference between registered memory (50 GB per worker node in our setting) and necessary memory to cache intermediate data because the simply encoded intermediate data in GeoSpark is very large for fast reading speed in the later steps. The intermediate data doesn't fit the cache memory will be spilled to disk. As we see in Figure 8, GeoSpark necessary memory utiliza-



(a) Run time (b) Necessary memory utilization (c) CPU utilization

Figure 10: Spatial aggregation with GeoSpark and SpatialHadoop R-Tree index join queries



(a) Run time (b) Necessary memory utilization (c) CPU utilization

Figure 11: Spatial co-location with GeoSpark and SpatialHadoop R-Tree index join queries

tion is lower than SpatialHadoop on both of the two different size data because GeoSpark can spill intermediate data into cache memory swiftly instead of spill it into disk slowly. Slowly spilling may make lots of intermediate stay in memory for a while. However, this advantage on the large dataset is not as obvious as that on the small dataset. The reason behind is that, due to the large scale input, more intermediate data which doesn't fit the cache memory has to be spilled into disk slowly which has to stay in memory.

In terms of CPU utilization, GeoSpark costs more CPU computation power than SpatialHadoop. This sacrifice is for caching intermediate data to increase the processing speed. Caching intermediate data to memory in GeoSpark needs encoding and decoding which is computation consuming. This overhead is much more obvious on the large dataset because of its large scale intermediate data.

`GeoSpark_QuadTree` and `GeoSpark_RTree` have less run time even though these local indexes are created on-the-fly while `SpatialHadoop_RTree` query time which includes in advance index creating is longer than that `SpatialHadoop_NoIndex`. The reason is that GeoSpark only creates small spatial indexes in the grid cells which have the elements might be overlapped by others while SpatialHadoop creates spatial indexes for all of the elements. And also the size of indexes in GeoSpark is smaller but the number is larger. Therefore, GeoSpark index creating has better parallelism. It also makes sense that in both of GeoSpark and SpatialHadoop, `GeoSpark_QuadTree`, `GeoSpark_RTree` and `SpatialHadoop_RTree` save memory for smaller query scale but consume CPU computation power for creating indexes.

7.3 Effect of cluster size

This section compares GeoSpark performance on differ size clusters with SpatialHadoop. They are tested on TIGER Edges 62 GB dataset. Figure 9 shows the results.

As described in Figure 9, GeoSpark and SpatialHadoop run time performance improves with the increasing number of the machines in the cluster. GeoSpark consumes less run time than SpatialHadoop especially on `GeoSpark_QuadTree` and `GeoSpark_RTree`. The reason we mentioned before is the cache of intermediate data and the better parallelism of GeoSpark index creating.

The memory utilization of GeoSpark and SpatialHadoop on more powerful clusters is also better and GeoSpark also has less necessary memory than SpatialHadoop on different cluster due to the cache of intermediate data. But the difference of necessary memory utilization between is more significant on the small size cluster. The fact is: although the parallelism of GeoSpark and SpatialHadoop is worse on the small size cluster, GeoSpark still can cache most parts of the intermediate data in to cache memory swiftly while SpatialHadoop slow intermediate data spilling on disk is affected by it seriously.

GeoSpark and SpatialHadoop exhibits less CPU utilization on the large size cluster than that on the small size cluster. It makes sense because GeoSpark has higher CPU consumption than SpatialHadoop due to the encoding and decoding for caching intermediate data.

7.4 Performance of different Applications

We implement the spatial aggregation and spatial co-location analysis mentioned in Section 6 with `GeoSpark_RTree` and `SpatialHadoop_RTree`. To show the excellence of GeoSpark on iterative analysis. In spatial co-location, 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. Figure 10 and 11 describe their performances.

For spatial aggregation, we join TIGER Zcta 1.5 GB dataset with TIGER Edges 62 GB dataset. For spatial co-

location, we use the first point column in both of TIGER Zcta 1.5 GB dataset and TIGER Edges 62 GB dataset and join them together.

As shown in Figure 10 and 11, GeoSpark outperforms SpatialHadoop in both applications. And their performances are also being improved with the increasing of cluster size. Since the spatial aggregation doesn't need complex transformation on the result of spatial join, the performance of GeoSpark and SpatialHadoop is similar with that on spatial join query. However, the gap between GeoSpark and SpatialHadoop performance in spatial co-location is more obvious than in spatial aggregation. For the run time, GeoSpark only costs the quarter time of SpatialHadoop. There are two reasons: (1) The sets we use in spatial co-location are two point sets. GeoSpark spatial PointRDD provides optimized run time performance and lower memory overhead for points while SpatialHadoop treats them as regular spatial objects which have MBRs without any optimizations. (2) 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. While SpatialHadoop has to read and transform the original datasets again and again.

For the memory utilization on spatial co-location, the difference between GeoSpark and SpatialHadoop is also more obvious than that on spatial aggregation because the two reason we just mentioned. In terms of CPU utilization on spatial co-location, GeoSpark consumes much more CPU computation power than SpatialHadoop. Due to less data transformation and loading, GeoSpark spends more run time on encoding and decoding intermediate data in cache memory. And this step is highly computation consuming.

8. 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 indexing and query processing algorithms in Apache Spark to efficiently analyze spatial data at scale. Extensive experiments on data sizes and cluster sizes show that GeoSpark achieves better run time performance (with reasonable memory/cpu utilization) than its MapReduce-based counterparts (e.g., SpatialHadoop) in various spatial data analysis scenarios. The proposed ideas are packaged into an open source software artifact. In the future, we plan to extend Spark SQL engine with a set of SQL User-Defined-Functions (UDFs) that maps to spatial data types and proximity constraints. The input/output of these UDFs would be quite similar to UDFs defined in PostGIS, an extension to PostgreSQL that provides a SQL interface for users to express spatial operations on geographical data. We also 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.

9. ACKNOWLEDGMENT

This project is supported in part by the National Geospatial-Intelligence Agency (NGA) Foresight Project.

10. REFERENCES

[1] http://spatialhadoop.cs.umn.edu/datasets.html.

[2] 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*, 6(11):1009–1020, 2013.

[3] U. Bureau. Topologically integrated geographic encoding and referencing (tiger).

[4] M. Davis. Secrets of the jts topology suite. *Free and Open Source Software for Geospatial*, 2007.

[5] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of ACM*, 51:107–113, 2008.

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

[7] R. A. Finkel and J. L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta informatica*, 4(1):1–9, 1974.

[8] A. Guttman. *R-trees: a dynamic index structure for spatial searching*, volume 14. ACM, 1984.

[9] J. Lu and R. H. Guting. Parallel Secondo: Boosting Database Engines with Hadoop. In *International Conference on Parallel and Distributed Systems*, pages 738 –743, 2012.

[10] 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.

[11] M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.

[12] S. Nishimura, S. Das, D. Agrawal, and A. E. Abbadi. MD-Hbase: A Scalable Multi-dimensional Data Infrastructure for Location Aware Services. In *Proceedings of the International Conference on Mobile Data Management, MDM*, pages 7–16, 2011.

[13] Open Geospatial Consortium. `http://www.opengeospatial.org/`.

[14] PostGIS. `http://postgis.net`.

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

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

[17] Spark. `https://spark.apache.org`.

[18] 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*, pages 15–28, 2012.

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