Geoinformatica (2019) 23: 37–78
https://doi.org/10.1007/s10707-018-0330-9

Spatial data management in apache spark: the GeoSpark perspective and beyond

Jia Yu[1] · **Zongsi Zhang**[1] · **Mohamed Sarwat**[1]

Received: 19 July 2017 / Revised: 16 July 2018 / Accepted: 27 September 2018 /
Published online: 22 October 2018

Abstract
The paper presents the details of designing and developing GeoSpark, which extends the core engine of Apache Spark and SparkSQL to support spatial data types, indexes, and geometrical operations at scale. The paper also gives a detailed analysis of the technical challenges and opportunities of extending Apache Spark to support state-of-the-art spatial data partitioning techniques: uniform grid, R-tree, Quad-Tree, and KDB-Tree. The paper also shows how building local spatial indexes, e.g., R-Tree or Quad-Tree, on each Spark data partition can speed up the local computation and hence decrease the overall runtime of the spatial analytics program. Furthermore, the paper introduces a comprehensive experiment analysis that surveys and experimentally evaluates the performance of running de-facto spatial operations like spatial range, spatial K-Nearest Neighbors (KNN), and spatial join queries in the Apache Spark ecosystem. Extensive experiments on real spatial datasets show that GeoSpark achieves up to two orders of magnitude faster run time performance than existing Hadoop-based systems and up to an order of magnitude faster performance than Spark-based systems.

Keywords Spatial databases · Distributed computing · Big geospatial data

1 Introduction

The volume of available spatial data has increased tremendously. Such data includes but is not limited to: weather maps, socioeconomic data, vegetation indices, geo-tagged social media, and more. Furthermore, several cities are beginning to leverage the power of sensors

✉ Jia Yu
jiayu2@asu.edu

Zongsi Zhang
zzhan236@asu.edu

Mohamed Sarwat
msarwat@asu.edu

[1] Arizona State University, 699 S. Mill Avenue, Tempe, AZ, USA

to monitor the urban environment. For instance, the city of Chicago started installing sensors across its road intersections to monitor the environment, air quality, and traffic. 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 climate change analysis [1], study of deforestation [2], population migration [3], and variation in sea levels [4], (2) Urban Planning: assisting government in city/regional planning, road network design, and transportation / traffic engineering, (3) Commerce and Advertisement [5]: e.g., point-of-interest (POI) recommendation services. These applications need a powerful data management platform to handle spatial data.

Existing spatial database systems (DBMSs) [6] extend relational DBMSs with new data types, operators, and index structures to handle spatial operations based on the Open Geospatial Consortium standards [7]. Even though such systems sort of provide full support for spatial data, they suffer from a scalability issue. That happens because the massive scale of available spatial data hinders making sense of it when using traditional spatial query processing techniques. Recent works (e.g., [8, 9]) extend the Hadoop ecosystem to perform spatial analytics at scale. Although the Hadoop-based approach achieves high scalability, it still exhibits slow run time performance and the user will not tolerate such delays.

Apache Spark, on the other hand, provides a novel data abstraction called Resilient Distributed Datasets (RDDs) [10] 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, the native Spark ecosystem does not provide support for spatial data and operations. Hence, Spark users need to perform the tedious task of programming their own spatial data processing jobs on top of Spark.

In this paper, we present the details of designing and developing GeoSpark [1], which extends the core engine of Apache Spark and SparkSQL to support spatial data types, indexes, and geometrical operations at scale. In other words, the system extends the Resilient Distributed Datasets (RDDs) concept to support spatial data. It also introduces Spatial SQL interface that follows SQL/MM-Part 3 standard [11]. Moreover, the GeoSpark optimizer can produce optimized spatial query plans and run spatial queries (e.g., range, knn and join query) on large-scale spatial datasets. In addition, the map visualization function of GeoSpark creates high resolution maps in parallel. In summary, the key contributions of this paper are as follows:

- The design and development of GeoSpark, an open-source full-fledged cluster computing framework to load, process, and analyze large-scale spatial data in Apache Spark. GeoSpark is equipped with an out-of-the-box Spatial Resilient Distributed Dataset (SRDD), which provides in-house support for geometrical and distance operations necessary for processing geospatial data. The spatial RDD provides an Application Programming Interface (API) for Apache Spark programmers to easily develop their spatial analysis programs using operational (e.g., Java and Scala) and declarative (i.e., SQL) languages.

  A detailed analysis of the technical challenges and opportunities of extending the core Apache Spark engine and SparkSQL to support state-of-the-art spatial data partitioning techniques: uniform grid, R-tree, Quad-Tree, and KDB-Tree. Each partitioning technique repartitions data based upon the spatial proximity among spatial objects to achieve load balancing in the Spark cluster. The paper also shows how building local

[1] source code: https://github.com/DataSystemsLab/GeoSpark

spatial indexes, e.g., R-Tree or Quad-Tree, on each Spark data partition can speed up the local computation and hence decrease the overall runtime of the spatial analytics program. The GeoSpark optimizer adaptively selects a proper join algorithm to strike a balance between the run time performance and memory/cpu utilization in the cluster.

A comprehensive experimental analysis that surveys and experimentally evaluates the performance of running de-facto spatial operations like spatial range, spatial K-Nearest Neighbors (KNN), and spatial join queries in the Apache Spark ecosystem. The experiments also compare the performance of GeoSpark to the state-of-the-art Spark-based and Hadoop-based spatial systems. The experiments show that GeoSpark achieves up to two orders of magnitudes better run time performance than the existing Hadoop-based system and up to an order of magnitude faster performance than Spark-based systems in executing spatial analysis jobs.

The rest of this paper is organized as follows. GeoSpark architecture is described in Section 3. Section 4 presents the Spatial Resilient Distributed Datasets (SRDDs) and Section 5 explains how to efficiently process spatial queries on SRDDs. Query optimization is discussed in Section 5.4. Section 6 provides three spatial applications to depict how the users leverage GeoSpark in practice. Section 7 experimentally evaluates GeoSpark system architecture. Section 2 highlights the related work. Finally, Section 8 concludes the paper.

2 Background and related work

In this section, we summarize the background, related work and existing systems that support spatial data processing operations. To be precise, we study the systems' source code and research papers (if exist).

2.1 Spatial database operations

Spatial database operations are deemed vital for spatial analysis and spatial data mining. Users can combine query operations to assemble a complex spatial data mining application.

Spatial Range query A spatial range query [12] returns all spatial objects that lie within a geographical region. For example, a range query may find all parks in the Phoenix metropolitan area or return all restaurants within one mile of the user's current location. In terms of the format, a spatial range query takes a set of points or polygons and a query window as input and returns all the points / polygons which lie in the query area.

Spatial join Spatial join queries [13] 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 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 K nearest neighbors (KNN) query Spatial KNN query takes a query center point, a spatial object set as inputs and finds the K nearest neighbors around the center points. For instance, a KNN query finds the 10 nearest restaurants around the user.

Spatial indexing Spatial query processing algorithms usually make use of spatial indexes to reduce the query latency. For instance, R-Tree [14] provides an efficient data partitioning strategy to efficiently index spatial data. The key idea is to 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. Quad-Tree [15] recursively divides a two-dimensional space into four quadrants.

2.2 Spatial data processing in the Hadoop ecosystem

There exist systems that extend state-of-the-art Hadoop to support massive-scale geospatial data processing. A detailed comparison of the existing Hadoop-based systems is given in Table 1. Although these systems have well-developed functions, all of them are implemented on top of the Hadoop MapReduce framework, which suffers from a large number of reads and writes on disk.

SpatialHadoop [9] provides native support for spatial data in Hadoop. It supports various geometry types, including polygon, point, line string, multi-point and so on, and multiple spatial partitioning techniques [16] including uniform grids, R-Tree, Quad-Tree, KD-Tree, Hilbert curves and so on. Furthermore, SpatialHadoop provides spatial indexes and spatial data visualization [17]. The SQL extension of SpatialHadoop, namely Pigeon [18], allows users to run Spatial SQL queries following the standard SQL/MM-Part 3 [11] but does not provide a comprehensive spatial query optimization strategy.

Parallel-Secondo [19] integrates Hadoop with SECONDO, a database that can handle non-standard data types, like spatial data, usually not supported by standard systems. It employs Hadoop as the distributed task manager and performs operations on a multi-node spatial DBMS. It supports the common spatial indexes and spatial queries except KNN . However, it only supports uniform spatial data partitioning techniques, which cannot handle the spatial data skewness problem. In addition, the visualization function in Parallel-Secondo needs to collect the data to the master local machine for plotting, which does not scale up to large datasets.

HadoopGIS [8] utilizes SATO spatial partitioning [20] (similar to KD-Tree) and local spatial indexing to achieve efficient query processing. Hadoop-GIS can support declarative spatial queries with an integrated architecture with HIVE [21]. However, HadoopGIS doesn't offer standard Spatial SQL [11] as well as spatial query optimization. In addition, it lacks the support of complex geometry types including convex/concave polygons, line string, multi-point, multi-polygon and so on. HadoopGIS visualizer can plot images on the master local machine.

2.3 Apache spark and spark-based spatial data processing systems

Apache Spark [10] is an in-memory cluster computing system. Spark provides a novel data abstraction called resilient distributed datasets (RDDs) 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. For fault tolerance, Spark rebuilds lost data on failure using lineage: each RDD remembers how it was built from other datasets (through transformations) to recover itself. The Directed Acyclic Graph in Spark consists of a set of RDDs (points) and directed Transformations (edges). There are two transformations can be applied to RDDs, narrow transformation and wide transformation. The relation between the two

Table 1 Compare related systems with GeoSpark

Feature name	GeoSpark	Simba	Magellan	Spatial Spark	GeoMesa	Spatial Hadoop	Parallel Secondo	Hadoop GIS
RDD API	✓	✗	✗	✓	✓	✗	✗	✗
DataFrame API	✓	✓	✓	✗	✓	✗	✗	✗
Spatial SQL [11, 28]	✓	✗	✗	✗	✓	✓	✗	✗
Query optimization	✓	✓	✓	✗	✓	✗	✓	✗
Complex geometrical operations	✓	✗	✗	✗	✓	✓	✗	✗
Spatial indexing	R-Tree Quad-Tree	R-Tree Quad-Tree	✗	R-Tree	Grid file	R-Tree Quad-Tree	R-Tree	R-tree
Spatial partitioning	Multiple	Multiple	Z-Curve	R-Tree	R-Tree	Multiple	Uniform	SATO
Range / Distance query	✓	✓	✓	✓	✓	✓	✓	✓
KNN query	✓	✓	✗	✗	✗	✓	✗	✓
Range / Distance Join	✓	✓	✓	✓	✓	✓	✓	✓

RDDs linked by the transformations are called narrow dependency and wide dependency respectively:

- Narrow dependency does not require the data in the former RDD (or RDDs) to be shuffled across partitions for generating the later RDD.
- Wide dependency requires the data in the former RDDs to be shuffled across partitions to generate the new RDD. For example, Reduce, GroupByKey, and OuterJoin. Basically, wide dependency results in stage boundaries and starts a new stage.

SparkSQL [22] is an independent Spark module for structured data processing. It provides a higher-level abstraction called DataFrame over Spark RDD. A DataFrame is structured to the format of a table with column information. SparkSQL optimizer leverages the structure information to perform query optimization. SparkSQL supports two kinds of APIs: (1) DataFrame API manipulates DataFrame using Scala and Java APIs; (2) SQL API manipulates DataFrame using SQL statements directly. Unfortunately, Spark and SparkSQL do not provide native support for spatial data and spatial operations. Hence, users need to perform the tedious task of programming their own spatial data exploration jobs on top of Spark.

Limitations of Spark-based systems We have studied four popular Spark-based systems including their research papers and source code: Simba [23], Magellan [24], SpatialSpark [25] and GeoMesa [26]. Before diving into the details of each system, we want to summarize their common limitations (see Table 1):

- Simple shapes only: Simba only supports simple point objects for KNN and join. Magellan can read polygons but can only process spatial queries on the Minimum Bounding Rectangle (MBR) of the input polygons. SpatialSpark only works with point and polygon. Spatial objects may contain many different shapes (see Section 4.1) and even a single dataset may contain heterogeneous geometrical shapes. Calculating complex shapes is time-consuming but indeed necessary for real life applications.
- Approximate query processing algorithms: Simba does not use the Filter and Refine model [14, 27] (see Section 4.4), which cannot guarantee the query accuracy for complex shapes. The Filter-Refine model is a must in order to guarantee R-Tree/Quad-Tree query accuracy although it takes extra time. Magellan only uses MBR in spatial queries instead of using the real shapes. Simba and GeoMesa do not remove duplicated objects introduced by spatial partitioning (i.e., polygon and line string, see Section 4.3) and directly returns inaccurate query results.
- RDD only or DataFrame only: Simba and Magellan only provide DataFrame API. However, the Java / Scala RDD API allows users to achieve granular control of their own application. For complex spatial data analytics algorithm such as spatial co-location pattern mining (see Section 6), Simba and Magellan users have to write lots of additional code to convert the data from the DataFrame to the RDD form. That leads to additional execution time as well as coding effort. On the other hand, SpatialSpark only provides RDD API and does not provide support for spatial SQL. The lack of a SQL / declarative interface makes it difficult for the system to automatically optimize spatial queries.
- No standard Spatial SQL [11]: Simba's SQL interface doesn't follow either OpenGIS Simple Feature Access [28] or SQL/MM-Part 3 [11], the two de-facto Spatial SQL standards. Magellan only allows the user to issue queries via its basic SparkSQL DataFrame API and does not allow users to directly run spatial queries in a SQL statement.

Simba [23] extends SparkSQL to support spatial data processing over the DataFrame API. It supports several spatial queries including range query, distance join query, KNN and KNN join query. Simba builds local R-Tree indexes on each DataFrame partition and uses R-Tree grids to perform the spatial partitioning. It also optimizes spatial queries by: (1) only using indexes for highly selective queries. (2) selecting different join algorithms based on data size.

Magellan [24] is a popular industry project that received over 300 stars on GitHub. It extends SparkSQL to support spatial range query and join query on DataFrames. It allows the user to build a "z-curve index" on spatial objects. Magallan's z-curve index is actually a z-curve spatial partitioning method, which exhibits slow spatial join performance [16].

SpatialSpark [25] builds on top of Spark RDD to provide range query and spatial join query. It can leverage R-Tree index and R-Tree partitioning to speed up queries.

GeoMesa [26] is an open-source spatial index extension built on top of distributed data storage systems. It provides a module called GeoMesaSpark to allow Spark to read the pre-processed and pre-indexed data from Accumulo [29] data store. GeoMesa also provides RDD API, DataFrame API and Spatial SQL API so that the user can run spatial queries on Apache Spark. GeoMesa must be running on top of ZooKeeper [30] with 3 master instances. GeoMesa supports range query and join query. In particular, it can use R-Tree spatial partitioning technique to decrease the computation overhead. However, it uses a grid file as the local index per DataFrame partition. Grid file is a simple 2D index but cannot well handle spatial data skewness in contrast to R-Tree or Quad-Tree index. Most importantly, GeoMesa does not remove duplicates introduced by partitioning the data and hence cannot guarantee join query accuracy. In addition, GeoMesa does not support parallel map rendering. Its user has to collect the big dataset to a single machine then visualize it as a low resolution map image.

3 System overview

Figure 1 gives an overview of GeoSpark. Users can interact with the system using either a Spatial SQL API or a Scala/Java RDD API. The Scala/Java RDD API allows the user to use an operational programming language to write her custom made spatial analytics application. The user can create a Spatial RDD, call the geometrical library and run spatial operations on the created RDDs. The Spatial SQL API follows the SQL/MM Part 3 Standard [11]. Specifically, three types of Spatial SQL interfaces are supported: (1) Constructors: initialize a Spatial RDD. (2) Geometrical functions: that represent geometrical operations on a given Spatial RDD (3) Predicates: issue a spatial query and return data that satisfies the given predicate such as Contains, Intersects and Within.

The Spatial Resilient Distributed Dataset (SRDD) layer extends Spark with Spatial RDDs (SRDDs) that efficiently partition spatial data elements across the Apache Spark cluster. This layer also introduces parallelized spatial transformations and actions (for SRDD) that provide a more intuitive interface for programmers to write spatial data analytics programs. A Spatial RDD can accommodate heterogeneous spatial objects which are very common in a GIS area. Currently, GeoSpark allows up to seven types of spatial objects to co-exist in the same Spatial RDD. The system also provides a comprehensive geometrical operations library on-top of the Spatial RDD.

The spatial query processing layer allows programmers to execute spatial query operators over loaded Spatial RDDs. Such a layer provides an efficient implementation of the most-widely used spatial query operators, e.g., range filter, distance filter, spatial k-nearest

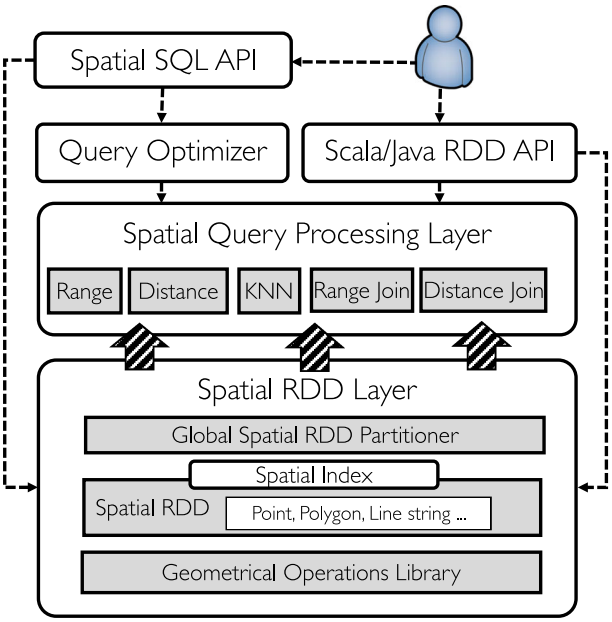


Fig. 1 GeoSpark Overview

neighbors, range join and distance join. Given a Spatial SQL statement, the optimizer takes into account the execution time cost and interleaves different queries to produce a good query execution plan. It mainly offers two types of optimizations: (1) cost-based join query optimization: pick the faster spatial join algorithm based on the size of input Spatial RDDs (2) predicate pushdown: detect the predicates which filter the data and push them down to the beginning of the entire plan in order to reduce data size.

4 Spatial RDD (SRDD) layer

GeoSpark Spatial RDDs are in-memory distributed datasets that intuitively extend traditional RDDs to represent spatial objects in Apache Spark. A Spatial RDD consists of many partitions and each partition contains thousands of spatial objects. Large-scale spatial data cannot be easily stored in Spark's native RDD like plain objects because of the following challenges:

Heterogeneous data sources Different from generic datasets, spatial data is stored in a variety of special file formats that can be easily exchanged among GIS libraries. These formats include CSV, GeoJSON [31], WKT [32], NetCDF/HDF [33] and ESRI Shapefile [34]. Spark does not natively understand the content of these files and straightforward loading of such data formats into Spark may lead to inefficient processing of such data.

Complex geometrical shapes There are many different types of spatial objects each of which may possess very complex shapes such as concave/convex polygons and multiple sub-shapes. In addition, even a single dataset may contain multiple different objects such as Polygon, Multi-Polygon, and GeometryCollection. These objects cannot be efficiently

partitioned across machines, serialized in memory, and processed by spatial query operators. It requires too much effort to handle such spatial objects, let alone optimize the performance in terms of run time cost and memory utilization.

Spatial partitioning The default data partitioner in Spark does not preserve the spatial proximity of spatial objects, which is crucial to the efficient processing of spatial data. Nearby spatial objects are better stored in the same RDD partition so that the issued queries only access a reduced set of RDD partitions instead of all partitions.

Spatial index support Spark does not support any spatial indexes such as Quad-Tree and R-Tree. In addition, maintaining a regular tree-like spatial index yields additional 15% storage overhead [35, 36]. Therefore, it is not possible to simply build a global spatial index for all spatial objects of an RDD in the master machine memory.

In order to tackle the challenges mentioned above, GeoSpark offers an integrated solution, Spatial RDD, that allows for efficient loading, partitioning, and indexing of complex spatial data. The rest of this section highlights the details of the Spatial RDD layer and explains how GeoSpark exploits Apache Spark core concepts for accommodating spatial objects. For the sake of understanding the concepts, we use the New York City Taxi Trip dataset [37] (TaxiTripTable) as a running example in all sections. The dataset contains detailed records of over 1.1 billion individual taxi trips in the city from January 2009 through December 2016. Each record includes pick-up and drop-off dates/times, pick-up and drop-off precise location coordinates, trip distances, itemized fares, and payment methods.

4.1 SRDD spatial objects support

As mentioned before, Spatial RDD supports various input formats (e.g., CSV, WKT, GeoJSON, NetCDF/HDF, and Shapefile), which cover most application scenarios. Line delimited file formats (CSV, WKT and GeoJSON) that are compatible with Spark can be created through the GeoSpark Spatial SQL interface. Binary file formats (NetCDF/HDF and Shapefile) need to be handled by GeoSpark customized Spark input format parser which detects the position of each spatial object.

Since spatial objects have many different types [31–34], GeoSpark uses a flexible implementation to accommodate heterogeneous spatial objects. Currently, GeoSpark supports seven types of spatial objects, Point, Multi-Point, Polygon, Multi-Polygon, LineString, Multi-LineString, GeometryCollection, and Circle. This means the spatial objects in a Spatial RDD can either belong to the same geometry type or be in a mixture of many different geometry types.

GeoSpark users only need to declare the correct input format followed by their spatial data without any concern for the underlying processing procedure. Complex data transformation, partitioning, indexing, in-memory storing are taken care of by GeoSpark and do not bother users. A SQL and Scala example of constructing a Spatial RDD from WKT strings is given below.

```
/* Spatial SQL API*/
SELECT ST_GeomFromWKT(TaxiTripRawTable.pickuppointString)
FROM TaxiTripRawTable

/* Scala/Java RDD API */
```

```
var TaxiTripRDD = new SpatialRDD(sparkContext, dataPath)
```

4.2 SRDD built-in geometrical library

GeoSpark provides a built-in library for executing geometrical computation on Spatial RDDs in parallel. This library provides native support for many common geometrical operations (e.g., Dataset boundary, polygon union and reference system transform) that follow the Open Geospatial Consortium (OGC) [7] standard. Operations in the geometrical computation library can be invoked through either GeoSpark Spatial SQL interface or GeoSpark RDD APIs. Each operation in the geometrical library employs a distributed computation algorithm to split the entire geometrical task into small sub-tasks and execute them in parallel. We explain the algorithms used in Dataset Boundary and Reference System Transformation as examples (SQL is also given); other operations have similar algorithms.

DatasetBoundary (SQL: ST_Envelope_Aggr) This function returns the rectangle boundary of the entire Spatial RDD. In GeoSpark Spatial SQL, it takes as input the geometry type column of the dataset. It uses a Reduce-like algorithm to aggregate the boundary: it calculates the merged rectangular boundary of spatial objects two by two until the boundaries of all the objects are aggregated. This process first happens on each RDD partition in parallel. After finding the aggregated boundary of each partition, it aggregates the boundary of partitions two by two until the end. For instance, the following function returns the entire rectangular boundary of all taxi trips' pickup points.

```
/* Spatial SQL */
SELECT ST_Envelope_Aggr(TaxiTripTable.pickuppoint)
FROM TaxiTripTable

/* Scala/Java RDD API */
var envelopeBoundary = TaxiTripRDD.boundary()
```

ReferenceSystemTransform (SQL: ST_Transform) Given a source and a target Spatial Reference System code, this function changes the Spatial Reference System (SRS) [7] of all spatial objects in the Spatial RDD. In GeoSpark Spatial SQL, this function also takes as input the geometry type column of the dataset. It uses a Map-like algorithm to convert the SRS: for each partition in a Spatial RDD, this function traverses the objects in this partition and converts their SRS.

4.3 SRDD partitioning

Apache Spark loads the input data file into memory, physically splits its in-memory copy to many equally sized partitions (using hash partitioning or following HDFS partitioned file structure) and passes each partition to each worker node. This partitioning method doesn't preserve the spatial proximity which is crucial for improving query speed.

GeoSpark automatically repartitions a loaded Spatial RDD according to its internal spatial data distribution. The intuition of Spatial RDD partitioning is to group spatial objects into the same partition based upon their spatial proximity. Spatial partitioning accelerates the query speed of a join query. It achieves that by reducing the data shuffles across the

cluster (see Section 5.3) and avoiding unnecessary computation on partitions that are impossible to have qualified data. A good spatial partitioning technique keeps all Spatial RDD partitions balanced in terms of memory space and spatial computation, aka. load balancing.

Spatial RDD represents a very large and distributed dataset so that it is extremely time consuming to traverse the entire Spatial RDD for obtaining the spatial distribution and partition it according to its distribution. GeoSpark employs a low overhead spatial partitioning approach to take advantage of global spatial distribution awareness. Hence, GeoSpark swiftly partitions the objects across the cluster. The spatial partitioning technique incorporates three main steps as follows (see Algorithm 1):

Algorithm 1 SRDD spatial partitioning

```
Data: An original SRDD
Result: A repartitioned SRDD
/* Step 1: Build a global grid file at master node */
Take samples from the original SRDD A partitions in parallel;
Construct the selected spatial structure on the collected sample at master node;
Retrieve the grids from built spatial structures;
/* Step 2: Assign grid ID to each object in parallel */
foreach spatial object in SRDD A do
    foreach grid do
        if the grid intersects the object then
            Add (grid ID, object) pair into SRDD B;
    // Only needed for R-Tree partitioning
    if no grid intersects the object then
        Add (overflow grid ID, object) pair into SRDD B;
/* Step 3: Repartition SRDD across the cluster */
Partition SRDD B by ID and get SRDD C;
Cache the new SRDD C in memory and return it;
```

Step 1: **Building a global spatial grid file:** In this step, the system takes samples from each Spatial RDD partition and collects the samples to Spark master node to generate a small subset of the Spatial RDD. This subset follows the Spatial RDD's data distribution. Hence, if we split the subset into several load balanced partitions that contain a similar number of spatial objects and apply the boundaries of partitions to the entire Spatial RDD, the new Spatial RDD partitions should still be load-balanced. Furthermore, the spatial locations of these records should also be of a close spatial proximity to each other. Therefore, after sampling the SRDD, GeoSpark constructs one of the following spatial data structures that splits the sampled data into partitions at the Spark master node (see Fig. 2). As suggested by [16], GeoSpark takes 1% percent data of the entire Spatial RDD as the sample:

- **Uniform Grid:** GeoSpark partitions the entire two-dimensional space into equal sized grid cells which have the same length, width and area. The boundaries of these grid cells are applied to the entire Spatial RDD. The partitioning approach generates non-balanced grids which are suitable for uniform data.
- **R-Tree | Quad-Tree | KDB-Tree:** This approach exploits the definition of capacity (fanout) in the classical spatial tree index structures, R-Tree [14],

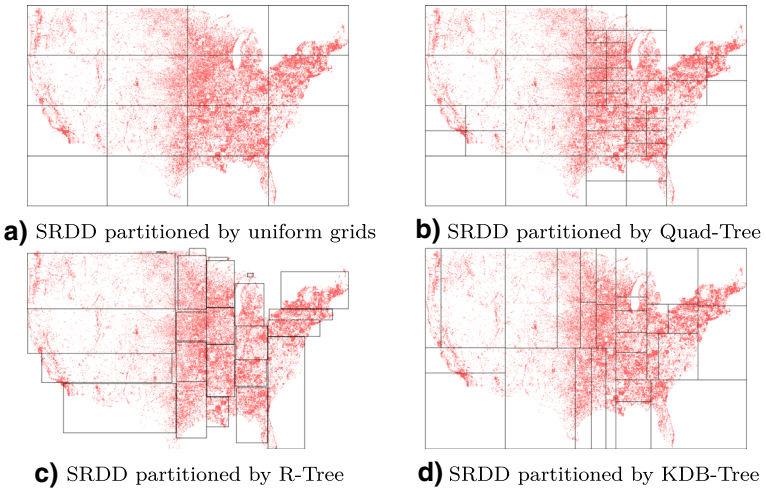


Fig. 2 Grids generated by SRDD spatial partitioning techniques

Quad-Tree [27] and KDB-Tree [38]: each tree node contains the same number of child nodes. GeoSpark builds an R-Tree, Quad-Tree or KDB-Tree on the sample subset and collects the leaf node boundaries to a grid file. It is worth noting that: the grids of R-Tree that builds on the sample data do not cover the entire space of the dataset. Thus, we need to have an overflow partition to accommodate the objects that do not fall in any grid of R-Tree partitioning. However, since Quad-Tree and KDB-Tree always start the splitting from the entire dataset space (the dataset boundary is either from the user or calculated by the geometrical library in Section 4.2), we don't the overflow partition.

Step 2: **Assigning a grid cell ID to each object:** After building a global grid file, GeoSpark needs to know the grid inside which each object falls and then repartition the Spatial RDD in accordance with the grid IDs. Therefore, GeoSpark duplicates the grid files and broadcasts the copies to each Spatial RDD partition. After receiving the broadcasted grid file, each original Spatial RDD partition simultaneously starts to check each internal object against the grid file. The results are stored in a new Spatial RDD whose schema is <Key, Value>. If an object intersects with a grid, the grid ID will be assigned to this object and a <grid ID, object>pair will be added to the result Spatial RDD. Because some objects span across multiple grids and some grids overlap with each other, an object may belong to multiple grids and hence the resulting SRDD may contain duplicates. To guarantee the query accuracy, GeoSpark spatial query processing layer handles this issue using two different techniques to remove duplicates (see Section 5.3).

Step 3: **Re-partitioning SRDD across the cluster:** The Spatial RDD generated by the last step already has <Key, Value>pair schema. The Key represents a grid cell ID. In this step, GeoSpark repartitions the Spatial RDD by Key and then spatial objects which have the same grid cell ID (Key) are grouped into the same

partition. These partitions constitute a new Spatial RDD. This step results in massive data being shuffled across the cluster due to passing spatial objects to their assigned worker nodes.

4.4 SRDD indexing

Spatial indexes such as R-Tree and Quad-Tree can speed up a spatial query significantly. That is due to the fact that such indexes group nearby spatial objects together and represent them with a tight bounding rectangle in the next higher level of the tree. A query that does not intersect with the rectangle cannot intersect with any of the objects in the lower levels.

Since many spatial analysis algorithms (e.g., spatial data mining, geospatial statistical learning) have to query the same Spatial RDD many times until convergence, GeoSpark allows the user to build spatial indexes and the built indexes can be cached, persisted and re-used many times. However, building a spatial index for the entire dataset is not possible because a tree-like spatial index yields additional 15% storage overhead [35, 36]. No single machine can afford such storage overhead when the data scale becomes large.

Build local indexes To solve the problem, if the user wants to use a spatial index, GeoSpark will build a set of local spatial indexes rather than a single global index. In particular, GeoSpark creates a spatial index (R-Tree or Quad-Tree) per RDD partition. These local R-Trees / Quad-Trees only index spatial objects in their associated partition. Therefore, this method avoids indexing all objects on a single machine.

To further speed up the query, the indexes in GeoSpark are clustered indexes. In that case, spatial objects in each partition are stored directly in the spatial index of this partition. Given a query, the clustered indexes can directly return qualified spatial objects and skip the I/O of retrieving spatial index according to the qualified object pointers.

Query local indexes When a spatial query is issued by the spatial query processing layer, the query is divided into many smaller tasks that are processed in parallel. In case a local spatial index exists in a certain partition, GeoSpark will force the spatial computation to leverage the index. In many spatial programs, the built indexes will be re-used again and again. Hence, the created spatial indexes may lead to a tremendous saving in the overall execution time and the index construction time can be amortized.

In addition, since spatial indexes organize spatial objects using their Minimum Bounding Rectangle (MBR) instead of their real shapes, any of the queries that leverage spatial indexes have to follow the Filter and Refine model [14, 27] (explained in Section 5): In the filter phase, we find candidate objects (MBR) that intersect with the query object (MBR); in the refine phase, we check the spatial relation between the candidate objects and the query object and only return the objects that truly satisfy the required relation (contain or intersect).

Persist local indexes To re-use the built local indexes, GeoSpark users first need to store the indexed spatial RDD using one of the following ways (DataFrame shares similar APIs) - (1) cache to memory: call IndexedSpatialRDD.cache() (2) persist on disk: call IndexedSpatialRDD.saveAsObjectFile(HDFS/S3_PATH). Both methods make use of the same algorithm: (1) go to each partition of the RDD in parallel (2) call SRDD customized serializer to serialize the local index on each partition to a byte array, one array per partition (see Section 4.5) (3) write the generated byte array of each partition to memory or disk. The user can directly use the name of the cached indexed SpatialRDD in his program because

Spark manages the corresponding memory space and de-serializes byte arrays in parallel to recover the original RDD partition information. For an indexed SpatialRDD on disk, the user needs to call IndexedSpatialRDD.readFromObjectFile(HDFS/S3_PATH) to explicitly read it back to Spark. Spark will read the distributed file partitions in parallel and the byte array of each partition is de-serialized to a local index.

4.5 SRDD customized serializer

When Spark transfers objects across machines (e.g., data shuffle), all objects have to be first serialized in byte arrays. The receiver machines will put the received data chunk in memory and then de-serialize the data. Spark default serializer can provide a compact representation of simple objects (e.g., integers). However, for objects such as spatial objects that possess very complex geometrical shapes, the default Spark serializer cannot efficiently provide a compact representation of such objects [39]. That may lead to large-scale data shuffled across the network and tremendous memory overhead across the cluster.

To overcome this issue, GeoSpark provides a customized serializer for spatial objects and spatial indexes. The proposed serializer uses a binary format to serialize a spatial object and indexes. The serialized object and index are put in byte arrays.

The way to serialize a spatial object is as follows:

- **Byte 1** specifies the type of the spatial object. Each supported spatial object type has a unique ID in GeoSpark.
- **Byte 2** specifies the number of sub-objects in this spatial object.
- **Byte 3** specifies the type of the first sub-object (only needed for GeometryCollection, other types don't need this byte).
- **Byte 4** specifies the number of coordinates (n) of the first sub-object. Each coordinate is represented by two double type (8 bytes * 2) data X and Y.
- **Byte 5 - Byte 4+16*n** stores the coordinate information.
- **Byte 16*n+1** specifies the number of coordinates (n) of the second sub-object...
- **Until the end** Here all sub-objects have been serialized.

The way to serialize a single local spatial index (Quad-Tree or R-Tree, explained in Section 4.4) is detailed below. It uses the classic N-ary tree serialization/deserialization algorithm. It is also worth noting that, spatial objects are stored inside tree nodes.

Serialization phase It uses the Depth-First Search (DFS) to traverse each tree node from the root following the pre-order strategy (first write current node information then write its children nodes). This is a recursive procedure. In the iteration of each tree node (each recursion), it first serializes the boundary of this node, and then serializes all spatial objects in this node one by one (use the object serializer explained above). Eventually, it goes to the children nodes of the working node. Since each N-ary tree node may have various internal spatial objects and children nodes, it also writes a memo to note the number of spatial objects and children nodes in this node.

De-serialization phase It still utilizes the same traverse strategy as the serialization phase (DFS, pre-order). It starts from the root and runs a recursive algorithm. In each recursion, it first re-constructs the boundary and internal spatial objects of the working node. Then it starts reading the bytes of the children nodes of the working node and hands over the work to the next recursion.

Based on our experiments in Section 7, GeoSpark serializer is faster than Spark kryo serializer and has smaller memory footprint when running complex spatial operations, e.g., spatial join query.

5 Spatial query processing layer

After the Spatial RDD layer loads, partitions, and indexes Spatial RDDs, GeoSpark can run spatial query processing operations on the SRDDs. The spatial query proccessing layer provides support for a wide set of popular spatial operators that include range query, distance query, K Nearest Neighbors (KNN) query, range join query and distance join query.

5.1 Processing spatial range and distance queries in spark

A spatial range query is fast and less resource-consuming since it only returns all the spatial objects that lie within the input query window object (point, polygon, line string and so on). Such a query can be completed by a parallelized Filter transformation in Apache Spark, which introduces a narrow dependency. Therefore, it is not necessary to repartition the Spatial RDD since repartitioning might lead to a wide dependency in the Apache Spark DAG. A more efficient way is to broadcast the query window to all worker nodes and parallelize the computation across the cluster. For non-closed query window objects such as a point or a line string, the range query processing algorithm only checks the "intersect" relation rather than "contain".

The spatial distance query conducts the same operation on the given Spatial RDD but adds an additional distance buffer between the query window and the candidate objects.

SQL API Spatial predicates such as "ST_Contains" can be used to issue a range query in GeoSpark Spatial SQL. For instance, "ST_Contains (A, B)" returns true if A contains B. "ST_Distance (A, B)$\leq$ d" returns true if the distance between A and B is equal to or less than d. The following two Spatial SQL examples depict (1) return taxi trips that are picked up in Manhattan (2) return taxi trips that are picked up within 1 mile from Manhattan. The other two Scala/Java examples take as input a Spatial RDD and a query window object and then run the same operations.

```
/* Spatial SQL Range query */
SELECT *
FROM TaxiTripTable
WHERE ST_Contains(Manhattan, TaxiTripTable.pickuppoint)

/* Spatial SQL Distance query */
SELECT *
FROM TaxiTripTable
WHERE ST_Distance(Manhattan, TaxiTripTable.pickuppoint) <= 1

/* Scala/Java RDD Range query */
RangeQuery.SpatialRangeQuery(PickupPointRDD, Manhattan)

/* Scala/Java RDD Distance query */
RangeQuery.SpatialDistanceQuery(PickupPointRDD, Manhattan, 1)
```

Algorithm For a given spatial range query, GEOSPARK broadcasts the query window to each machine in the cluster. For each Spatial RDD partition (see Algorithm 2), if a spatial index exists, it follows the Filter and Refine model: (1) uses the query window's MBR to query the spatial index and return the candidate results. (2) checks the spatial relation between the query window and candidate objects using their real shapes. The truly qualified spatial objects are returned as the partition of the new resulting Spatial RDD. If no spatial index exists, GEOSPARK filters spatial objects using the query window and collects qualified objects to be a partition of the new result Spatial RDD. The result Spatial RDD is sent to the next stage of the Spark program (if needed) or persisted on disk. For a distance query, we added a distance buffer to the query window such that it extends the boundary of the query window to cover more area. The remaining part of distance query algorithm remains the same with range query.

Algorithm 2 Range query and distance query

```
   Data: A query window A, a Spatial RDD B and spatial relation predicate
   Result: A Spatial RDD that contains objects that satisfy the predicate
1  foreach partition in the SRDD B do
2      if an index exists then
           // Filter phase
3          Query the spatial index of this partition using the window A's MBR;
           // Refine phase
4          Check the spatial relation predicate using real shapes of A and candidate
           objects;
5      else
6          foreach object in this partition do
7              Check spatial relation predicate between this object and A;
8              Record this object if it is qualified;
9  Generate the result Spatial RDD;
```

DAG and iterative spatial data mining The DAG and data flow of the range query and the distance query are described in Fig. 3. The query processing algorithm only introduces a single narrow dependency, which does not require data shuffle. Thus, all it needs is just one stage. For a compute-intensive spatial data mining program, which executes range queries many times (with different query windows), all queries access data from the same cached indexed Spatial RDD fluently without any interruptions from wide dependencies so that the procedure is very fast.

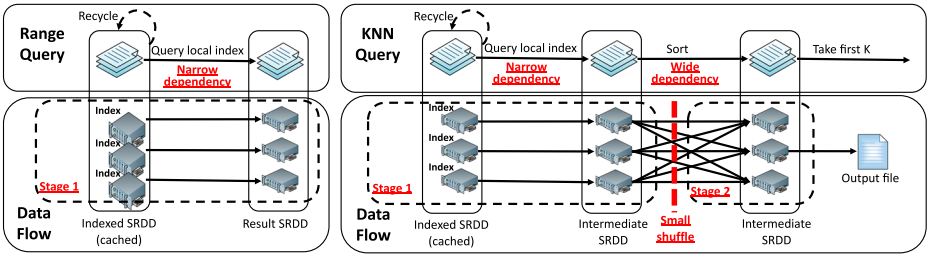


Fig. 3 Spatial range query and KNN query DAG and data flow

5.2 Spatial K nearest neighbors (KNN) query

The straightforward way to execute a KNN query is to rank the distances between spatial objects and the query location, then pick the top K nearest neighbors. However, ranking these distances in a large SRDD should be avoided if possible to avoid a large amount of data shuffle, which is time-consuming and bandwidth-consuming. In addition, it is also not necessary to spatially partition a Spatial RDD that incurs a single wide dependency.

SQL API: In Spatial SQL, "ST_Neighbors(A, B, K)" issues a spatial KNN query which finds the K nearest neighbors of A from Column B. The SQL example below returns the 100 nearest taxi trip pickup points of New York Time Square. The Scala/Java example performs the same operation.

```
/* Spatial SQL API */
SELECT ST_Neighbors(TimeSquare, TaxiTripTable.pickuppoint, 100)
FROM TaxiTripTable

/* Scala/Java RDD API */
KnnQuery.SpatialKnnQuery(PickupPointRDD, TimeSquare, 100)
```

Algorithm To parallelize a spatial KNN query more efficiently, GeoSpark modifies a popular top-k algorithm [40] to fit the distributed environment (1) to be able to leverage local spatial indexes if they exist (2) to reduce the data shuffle scale of ranking distances. This algorithm takes an indexed/non-indexed SRDD, a query center object (point, polygon, line string and so on) and a number K as inputs. It contains two phases (see Algorithm 3): selection and sorting.

Algorithm 3 K nearest neighbor (KNN) query

```
   Data: A query center object A, a Spatial RDD B, the number K
   Result: A list of K spatial objects
   /* Step 1: Selection phase                                          */
1  foreach partition in the SRDD B do
2      if an index exists then
3          Return K nearest neigbors of A by querying the index of this partition;
4      else
5          foreach object in this partition do
6              Check the distance between this object and A;
7              Maintain a priority queue that stores the top K nearest neighbors;
   /* Step 2: Sorting phase                                            */
8  Sort the spatial objects in the intermediate Spatial RDD C based on their distances to A;
9  Return the top K objects in C
```

- **Selection phase** For each SRDD partition, GeoSpark calculates the distances from the given object to each spatial object, then maintains a local priority queue by adding or removing objects based on the distances. Such a queue contains the nearest K objects around the given object and becomes a partition of the new intermediate SRDD. For the indexed Spatial RDDs, GeoSpark can query the local indexes (only R-Tree supports this, see [41]) in partitions to accelerate the distance calculation. Similarly, GeoSpark needs to follow Filter and Refine model to recheck the results returned by the index search using their real shapes.
- **Sorting phase** Each partition in the Spatial RDD generated by the selection phase only contains K objects. GeoSpark sorts this intermediate Spatial RDD in ascending order

according to the distances. Sorting the small scale intermediate Spatial RDD is much faster than sorting the original Spatial RDD directly. The sorting phase also outputs an intermediate Spatial RDD. GeoSpark collects the first K objects in the intermediate Spatial RDD across the cluster and returns those objects as the final result.

DAG and iterative spatial data mining Figure 3 depicts the DAG and data flow of spatial KNN query. The query processing algorithm includes two transformations: selection, sorting. The former incurs a narrow dependency and the latter introduces a wide dependency that results in a small data shuffle. These two transformations will be scheduled to two stages without pipeline execution. However, because the intermediate Spatial RDD generated by the first transformation only has K objects per partition, the shuffle caused by transforming this Spatial RDD is very small and will not impact the execution much. For an iterative spatial data mining using KNN query, the two intermediate Spatial RDDs are dropped after each query execution but the indexed Spatial RDD still resides in memory cache. Recycling the indexed Spatial RDDs accelerates the iterative execution to a greater extent.

5.3 Processing spatial join queries in spark

Spatial join, a computation and data intensive operation, incurs high data shuffle in Spark. Taking into account the spatial proximity of spatial objects, GeoSpark partitions Spatial RDDs in advance based on the objects' spatial locations in Spatial RDD layer (Section 4.3) and caches the Spatial RDDs. Therefore, GeoSpark join query algorithm re-uses the spatial partitioned RDDs (probably indexed as well) and avoids large scale data shuffle. Moreover, since GeoSpark skips the partitions which are guaranteed not to satisfy the join query predicate, it can significantly accelerate the overall spatial join processing.

API GeoSpark provides different Spatial SQL APIs for range join query and distance join query: (1) Given two geometry type columns, a range join query returns all spatial object pairs that satisfy a particular predicate such as "ST_Contains". The example below returns <taxi stop station, taxi trips pickup point> pairs of which the pickup point falls inside the taxi stop station. (2) A distance join query returns all possible spatial object pairs that are within a certain distance. The example below returns <taxi stop station, taxi trips pickup point> of which the pickup point is within 1 mile of the taxi stop station. The Scala/Java examples below perform the same operations but take as inputs two Spatial RDDs.

```
/* Spatial SQL Range join query */
SELECT *
FROM TaxiStopStations, TaxiTripTable
WHERE ST_Contains(TaxiStopStations.bound,
TaxiTripTable.pickuppoint)

/* Spatial SQL Distance join query */
SELECT *
FROM TaxiStopStation, TaxiTripTable
WHERE ST_Distance(TaxiStopStations.bound,
TaxiTripTable.pickuppoint) <= 1

/* Scala/Java RDD Range join query */
JoinQuery.SpatialJoinQuery(PickupPointRDD, StopStationRDD)
```

```
/* Scala/Java RDD Distance join query */
JoinQuery.DistanceJoinQuery(PickupPointRDD, StopStationRDD, 1)
```

***GSJoin* ALGORITHM** GeoSpark combines the approaches proposed by [42–44] to a new spatial range join algorithm, namely *GSJoin*, that re-uses spatial partitioned RDDs as well as their indexes. This algorithm, which takes two spatial partitioned RDDs A and B (i.e., TaxiStopStations and TaxiTripTable), consists of three steps as follows (see Algorithm 4 and Fig. 4).

- **Zip partitions** This step zips the partitions from A and B (TaxiStopStations and TaxiTripTable) according to their grid IDs. For instance, we merge Partition 1 from A and B to a bigger partition which has two sub-partitions. Both Partition 1 s from A and B have the spatial objects that fall inside Grid 1 (see Section 4.3). Partition 1 from A (TaxiStopStations) contains all taxi stop stations that locate in Grid 1 and Partition 1 from B (TaxiTripTable) contains all taxi trips that are picked up in Grid 1. Note that, the data in Partition1 from A is guaranteed to disjoint from other partitions (except 1) of B because they belong to totally different spatial regions (see Section 4.3). Thus, *GSJoin* does not waste time on checking other partitions from B with Partition 1 from A. This Zip operation applies to all partitions from A and B and produces an intermediate RDD called C.
- **Partition-level local join (no index)** This step runs a partition-level local range join on each partition of C. Each partition from C has two sub-partitions, one from A and one from B. If no indexes exist on both the sub-partitions, the local join will perform a nested loop join that traverses all possible pairs of spatial objects from the two sub-partitions and returns the qualified pairs. This costs $O(n^2)$ complexity on each C partition, where n is the number of objects in a sub-partition.
- **Partition-level local join (with index)** During the partition-level local join step, if an index exists on either one sub-partition (say, sub-partition from B is indexed), this local join will do an index-nested loop. It uses each object in the sub-partition from A as the query window to query the index of the sub-partition from B. This costs O(n*log(n)) complexity on each C partition, where n is the number of objects in a sub-partition. It is worth noting that this step also follows the Filter and Refine model which is similar to this part in Range query and Distance query (mentioned in Section 4.4). Each query

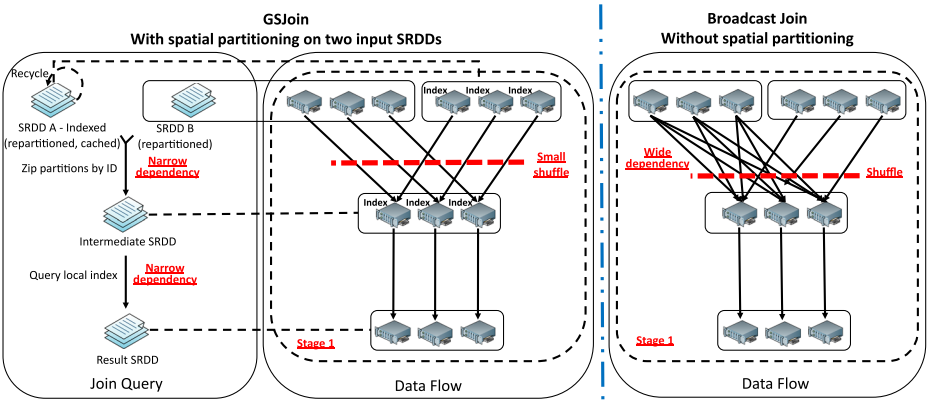


Fig. 4 Join query DAG and data flow

window needs to recheck the real shapes of candidate spatial objects which are obtained by scanning the index.

– **Remove duplicates** This step removes the duplicated spatial objects introduced by spatial partitioning. At that time, we duplicate the spatial objects that intersect with multiple grids and assign different grid IDs to these duplicates and this will lead to duplicated results eventually. Figure 5 illustrates an example. Since both Pa and Pb fall in Grid 1, 2, 3, and 4, the result "Pa intersects Pb" will be reported four times in the final join result set. In order to remove the duplicates, two methods are available. The first method is to use a "GroupBy" operation to collect all objects that intersect with Pa in this cluster then remove the duplicated ones. This method introduces a big data shuffle across the cluster because it needs to do a GroupBy on all results. We should always avoid unnecessary data shuffle. The second method is called "Reference point" [45]. The intuition of the reference point is to establish a rule that, if duplicated results appear, only report it once. When doing a partition-level local join, GeoSpark calculates the intersection of two intersecting objects (one from SRDD A and the other from B) and only reports the pair of objects when the reference point falls in the associated grid of this partition. To find this reference point, we first calculate the intersection shape of the two intersected objects. The X-coordinate/Y-coordinate of a reference point is the max X/Y of all coordinates of the intersection. In Fig. 5, we use the red point as the reference point and only report "Pa intersects Pb" when doing the local join on Partition 2. Note that, the reference point idea only works for Quad-Tree and KDB-Tree partitioning because their grids don't overlap with each other. R-Tree and other methods that produce overlapped grids have to use the first method because even the reference point can still appear in multiple grids.

Note that the *GSJoin* algorithm can query Spatial RDDs that are partitioned by any GeoSpark spatial partitioning method including Uniform grids, R-Tree grids, Quad-Tree grids, KDB-Tree grids and indexed by any of the indexing methods such as R-Tree index and Quad-Tree index. That is why we also implemented other partitioning methods and indexes in GeoSpark for benchmarking purpose.

Broadcast join algorithm Besides *GSJoin*, GeoSpark also provides a straightforward broadcast range join algorithm, which works well for small scale Spatial RDDs. When at least one of the two input Spatial RDDs is very small, this algorithm broadcasts the small Spatial RDD A to each partition of the other Spatial RDD B. Then, a partition-level local join (see *GSJoin*) happens on all partitions of B in parallel. The Broadcast join algorithm

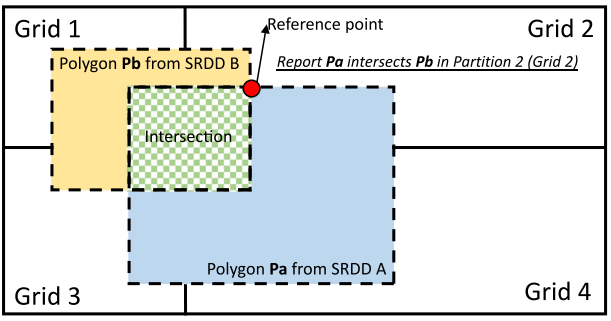


Fig. 5 Removing duplicates

has two important features: (1) it is faster than *GSJoin* for very small datasets because it does not require any spatial partitioning methods and duplicate removal. (2) it may lead to system failure or very long execution time for large datasets because it shuffles an entire SRDD A to each partition of SRDD B.

Distance join algorithm The distance join algorithms can be seen as extensions to the range join algorithms, *GSJoin* and Broadcast join. The only difference is that we add a distance buffer to all objects in either Spatial RDD A or B at the very beginning (even before spatial partitioning) to extend their boundaries. The extended spatial objects can be used in both range join algorithms.

Algorithm 4 *GSJoin* algorithm for range join and distance join query

```
   Data: (repartitioned) SRDD A and (repartitioned) SRDD B
   Result: PairRDD in schema <Left object from A, right object from B>
   /* Step1: Zip partitions                                              */
1  foreach partition pair from SRDD A and B with the same grid ID i do
2  |  Merge two partitions to a bigger partition that has two sub-partitions;
3  Return the intermediate SRDD C;
   /* Step2: Run partition-level local join                              */
4  foreach partition P in the C do
5  |  foreach object O_A in the sub-partition from A do
6  |  |  if an index exists in the sub-partition from B then
   |  |  |  // Filter phase
7  |  |  |  Query the spatial index of this partition using the O_A's MBR;
   |  |  |  // Refine phase
8  |  |  |  Check the spatial relation using real shapes of O_A and candidate objects O_Bs;
   |  |  |  /* Step3: Remove duplicates                                  */
9  |  |  |  Report <O_A, O_B> pair only if the reference point of this pair is in P;
10 |  |  else
11 |  |  |  foreach object O_B in the sub-partition from B do
12 |  |  |  |  Check spatial relation between O_A and O_B;
   |  |  |  |  /* Step3: Remove duplicates                               */
13 |  |  |  |  Report <O_A, O_B> pair only if the reference point of this pair is in P;
14 Generate the result PairRDD;
```

Spark DAG The DAG and data flow of *GSJoin* are shown in Fig. 4. The join query introduces two transformations: zip partitions and partition-level local join. Both of them incur narrow dependencies. Therefore, they are pipelined to one stage with fast execution. On the contrary, the broadcast join algorithm (see the right part in Fig. 4) introduces two wide dependencies which lead to heavy network traffic and intensive computation. Until now, the effect of spatial partitioning is shown by degrading wide dependencies to narrow dependencies. For spatial analytics program that runs multiple join queries, the repartitioned A and B are cached into memory and recycled for each join query.

5.4 Spatial SQL query optimization

An important feature of GeoSpark is the Spatial SQL query optimizer. GeoSpark optimizer extends SparkSQL Catalyst optimizer to support Spatial SQL optimization. It takes as input the Spatial SQL statement written by the user and generates an efficient execution plan in terms of time cost. For advanced users who have the necessary Spark knowledge, they can use GeoSpark Scala /Java RDD APIs to achieve granular control over their applications (Fig. 6).

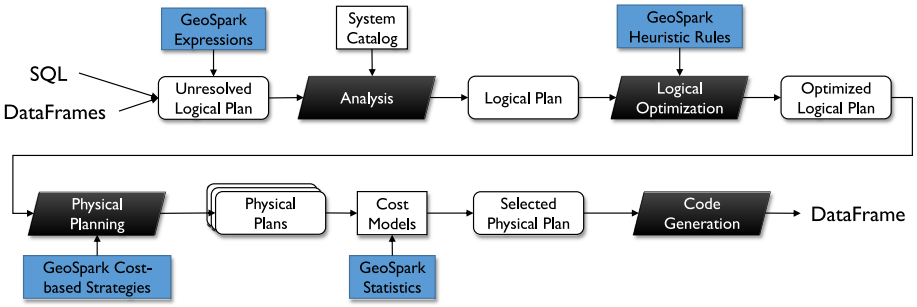


Fig. 6 Phases of query optimization in SparkSQL+GeoSpark. Black diamonds: phases; Rounded rectangles: Catalyst AST; Blue rectangles: GeoSpark's extension

5.4.1 Extend SparkSQL catalyst optimizer

SparkSQL Catalyst optimizer [22] runs its query optimization (aka., query planning) in four phases: Analysis, Logical Optimization, Physical Planning and Code Generation. After reading the entered SQL query, the SQL parser will generate the Abstract Syntax Tree (AST) and pass it to Catalyst. Each tree node in this AST is called Expression which is an operator such as Project, Sum and so on.

Analysis In this phase, Catalyst optimizer retrieves participating DataFrame information from Spark catalog and maps corresponding columns to the AST tree nodes. GeoSpark SQL API (e.g., ST_GeomFromWKT and ST_Contains) are written in Spark internal Expression format (not User Defined Function) such that Catalyst can easily understand the Spatial SQL functions and fuse them into the tree in this phase.

Logical optimization During this phase, Catalyst tries to apply some heuristic rules to the AST and transform the tree into a simplified version. Some common DBMS Heuristics-Based Optimization (HBO) are used here including constant folding, predicate pushdown, and projection pruning. To make this happen, Catalyst utilizes a mechanism called Pattern-Matching to capture a sub-tree of the AST according to the pattern description of a rule and then return a transformed new sub-tree.

Physical planning In this phase, Catalyst takes an optimized logical plan and applies Cost-Based Optimization (CBO) rules on it. Although Spark internally calls CBO rules as strategies, Spark still uses pattern matching mechanism to transform the AST based on the strategies. All current Spark strategies are only used to select join algorithms such as broadcast join, sort merge join and so on.

Code generation Catalyst uses the quasiquotes feature in Scala to generate Java bytecode for JVMs in this phase. In Spark, each Expression provides two implementations of its logic. One is written in native Scala and the other one is written in Scala quasiquotes format. The code generation details are not included in this paper since they are out of scope.

5.4.2 Heuristic rules for logical plans

GeoSpark optimizer adds several heuristic rules to Catalyst optimizer. During the Logical Optimization phase, Catalyst can optimize the AST that contains Spatial SQL Expressions.

This heuristics-based optimization (HBO) doesn't require any cost model. Three rules are currently available in GeoSpark optimizer.

Predicate pushdown The intuition of predicate pushdown is that some predicates of GeoSpark Spatial SQL queries can be "pushed" down closer to the data. In most cases, that is at the beginning part of an execution plan. The predicate pushdown can do range query filter on data before it is sent to other time-consuming operations such as a spatial join. Consider the SQL query below, a better plan is to use the range query filter to query TaxiStopStations.bound and only return the data within Manhattan then do the range join.

```
SELECT * FROM TaxiStopStations, TaxiTripTable
WHERE ST_Contains(TaxiStopStations.bound,
TaxiTripTable.pickuppoint)
AND ST_Contains(Manhattan, TaxiStopStations.bound)
```

Predicate merging Given a Spatial SQL query with multiple range query filters, it is better to merge the polygonal shapes of these filters into a single filter and run it against the underlying Spatial RDD/DataFrame. Consider the first SQL query below, it actually runs two range query filters, Manhattan's polygonal boundary and Queens' polygonal boundary, on the taxi trip pick up points. Apparently, this query should return nothing because of no intersection between Manhattan and Queens. A better logical plan should use the intersection of all ANDing filters as the new filter and run the single filter against the DataFrame (see Fig. 7). Of course, if the area of the intersection is 0, we don't even need to run this query. Consider the second query below, we can actually take the polygon union of these range query filters as a new filter instead of running the original filters separately.

```
SELECT * FROM TaxiTripTable
WHERE ST_Contains(Manhattan, TaxiTripTable.pickuppoint)
AND ST_Contains(Queens, TaxiTripTable.pickuppoint)
```

```
SELECT * FROM TaxiTripTable
WHERE ST_Contains(Manhattan, TaxiTripTable.pickuppoint)
OR ST_Contains(Queens, TaxiTripTable.pickuppoint)
```

This rule speeds up most spatial query cases and particularly favors the query on indexed Spatial DataFrames: querying an indexed DataFrame will first scan the index using one range filter (one sequential scan) and apply other range filters on the returned result (one partial table scan). Using a merged predicate only takes a single index scan.

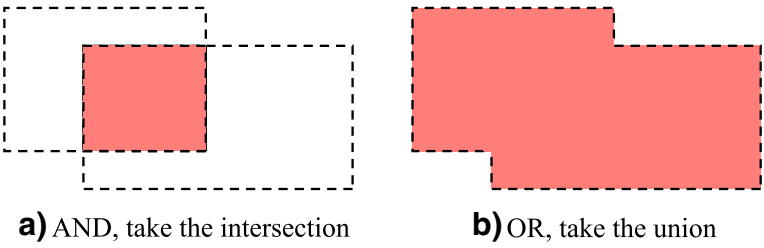


Fig. 7 Merge the range query predicates

Intersection query rewrite Calculating the intersection (a polygonal shape) between two spatial columns is quite common in spatial analytics (e.g., find the intersection between the habitats of lions and zebras). Consider the first two queries below, both of them require $O(N^2)$ times intersection calculation for running ST_Intersection on any possible <polygon, polygon> pairs between Lions.habitat and Zebras.habitat. This plan is prohibitively expensive because (1) the number of intersection calculations is tremendous (2) polygon intersection algorithms are orders of magnitude slower than intersects relation check (in other words, you can easily know whether two polygons intersect or not but finding the exact intersection shape is difficult). A better rewritten query is to first use ST_Intersects predicate to trigger a spatial join query and then perform the ST_Intersection only on returned <polygon, polygon> pairs that are guaranteed to have intersections. The spatial join query is much faster because it is optimized by GeoSpark. The last two queries below are the rewritten queries and their logical plans first run the ST_Intersects.

```
/* Without query rewrite */
SELECT ST_Intersection(Lions.habitat, Zebras.habitat)
FROM Lions, Zebras

SELECT *
FROM Lions, Zebras
WHERE ST_Intersection(Lions, Zebras)>10

/* With query rewrite */
SELECT ST_Intersection(Lions.habitat, Zebras.habitat)
FROM Lions, Zebras WHERE ST_Intersects(Lions.habitat,
Zebras.habitat);

SELECT *
FROM Lions, Zebras
WHERE ST_Intersection(Lions, Zebras)>10
AND ST_Intersects(Lions.habitat, Zebras.habitat)
```

5.4.3 Cost-based strategies for physical plans

GeoSpark provides Cost-Based strategies for physical plan optimization in addition to Spark's own join algorithm selection strategies. Physical planning only replaces the logical operator (tree node) with Spark's physical operator and has no change on the tree structure. In order to calculate the cost models, GeoSpark maintains its own catalog called GeoSpark statistics which stores the statistical information of the spatial columns in DataFrames. Two strategies are available in GeoSpark, index scan selection and spatial join algorithm selection.

GeoSpark statistics GeoSpark maintains a set of statistical information which can be used for building cost models (e..g, calculate range query selectivity). To collect the information, the user needs to explicitly call GeoSpark Analyze() function. For example, GeoSparkSession.Analyze(dataFrameA, column1) asks GeoSpark to collect the statistics from the column 1 of DataFrameA. Currently, GeoSpark only stores the global MBR and the count of the specified column. The Analyze() function may take some time to finish

because collecting the statistics of the given column runs a Reduce function. For instance, to calculate the global MBR, GeoSpark will reduce the MBR of each tuple in this column to a big MBR. To minimize the analyzing overhead, GeoSpark also aggregates the count in the same Reduce function. However, although Analyze() takes time, GeoSpark only executes it once and doesn't need to update it because RDDs are immutable.

Index scan selection Given a SQL range query, if a participating DataFrame column is indexed by R-Tree or Quad-Tree, the optimizer will decide whether GeoSpark should use it or not. Using spatial index in a spatial range query is not always the best choice. For instance, for polygon and line string data, GeoSpark has to perform a refine phase on the results returned by index scanning in order to guarantee the accuracy. If a range query is not selective and returns a large fraction of the raw data, the overall time of the refine phase and index scan (filter phase) may even be longer than a simple table scan. To make a decision, GeoSpark first calculates the selectivity of the given range query as follows:

$$Selectivity = \frac{Query\ window's\ area}{Global\ MBR's\ area}$$

If the selectivity is less than 1%, GeoSpark will consider the query as a highly selective query and use the built spatial index on each partition of the DataFrame.

Spatial join algorithm selection GeoSpark optimizer can adaptively choose a proper join algorithm, *GSJoin* or Broadcast Join. As we mentioned in Section 5.3, Broadcast Join is good for small datasets but *GSJoin* is much more scalable. Therefore, GeoSpark query optimizer defines a join strategy as follows: If the count of one input DataFrame of the join query is smaller than a system-tuned number of rows (i.e., 1000 rows), the Broadcast join algorithm will be used and this DataFrame will be broadcasted; otherwise, *GSJoin* algorithm will be used.

6 Application use cases

6.1 Application 1: region heat map

Assume that a data scientist in a New York City Taxi Company would like to visualize the taxi trip pickup point distribution in the Manhattan area on a map. He can first call GeoSpark `Range Query` (SQL interface) to return the taxi trip pickup points in Manhattan and then call GeoSpark Heat Map to plot the distribution of these pickup points. The SQL statement issues a range query using "ST_Contains" predicate. The Heat Map API takes as input the resolution of the desired map and its "visualize" function takes a Spatial RDD, the result of the range query, and generates the corresponding heat map. In the generated heat map (Fig. 8), red colored area means that lots of taxi trips were picked up in this area. The scientist finds that a significant number of taxi trips started from the Manhattan area. The pseudo code is given in Fig. 9.

The heat map visualization application uses the following steps to visualize a Spatial RDD:

Step I: **Rasterize Spatial Objects.** To plot a spatial object on a raster image, the object has to be rasterized to the corresponding pixels so that the map generator can

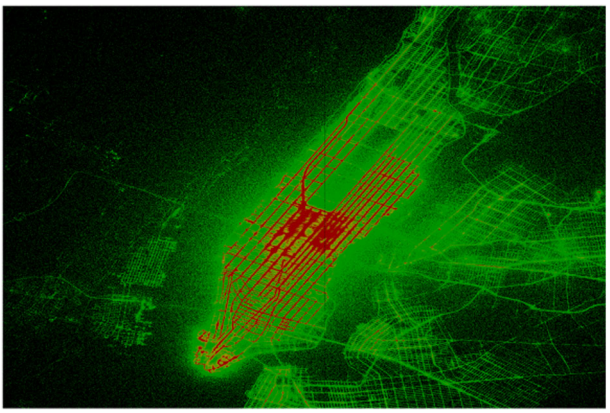


Fig. 8 Region Heat Map: taxi trip pickup points in Manhattan

plot out the pixels. A spatial object can be rasterized to many pixels and a pixel can be mapped by many spatial objects. This viz operator takes as input the spatial RDD and the designated map resolution. It then rasterizes each spatial object in the Spatial RDD to the map pixels in parallel. Each map pixel is mapped by the object's outline (Fig. 10a) or filling area (Fig. 10b). A pixel has its own coordinate (X, Y) which stands for its position on the final image. The viz layer performs the rasterization viz operator using a Map function and outputs a "Pixel" RDD which only contains pixels.

Step II: **Count by Pixels** Since a single pixel may be mapped by many spatial objects, this viz operator is to CountBy pixels according to their pixel coordinates. On the other hand, this layer is equipped with a distributed visualization mechanism to generate high resolution (billion pixels) maps: this layer repartitions the Pixel RDD using Uniform grids (see Fig. 2a). The intuition is that we want to produce map image tiles [46] in parallel instead of outputting a high resolution map on the master machine which will crash the master's JVM directly. The map tiles can be directly loaded by map service software such as Google Maps, MapBox and ArcGIS. After repartitioning the pixels using uniform grids (map tile boundaries), GeoSpark runs a local CountByKey (the keys are pixel coordinates) in each partition because nearby pixels are put in the same partition. This transforms the Pixel RDD to a PairRDD in <Pixel, Count> format. GeoSpark will render a partial image on each Pixel RDD partition in Operator III.

Step III: **Render Map Images.** This operator first runs a Map function in parallel. Each pixel in <Pixel, Count> RDD is assigned a color according to its count in this

```
var manhanttanDataFrame = spark.sql
("SELECT TaxiTripTable.pickuppoint FROM TaxiTripTable
WHERE ST_Contains(Manhattan, TaxiTripTable.pickuppoint)")
var vizEffect = new HeatMap(resolutionX, resolutionY)
vizEffect.Visualize(manhanttanDataFrame.toRDD)
```

Fig. 9 Region Heat Map: Range Query + Heat Map

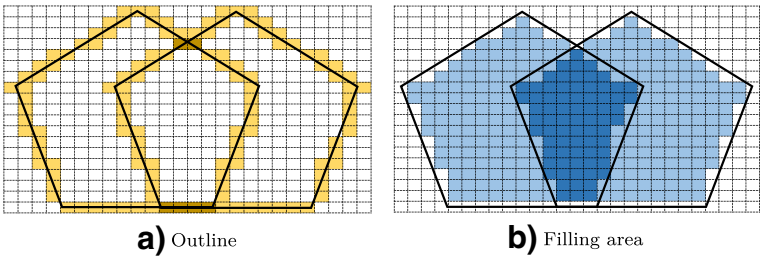


Fig. 10 Rasterize spatial objects to pixels

viz operator. The relation between a color and a count is defined by a user-supplied mathematical function such as Piecewise function. For instance,

$$Color = \begin{cases} Yellow & Count \in [0, 100) \\ Pink & Count \in [100, 200) \\ Red & Count \in [200, 255] \end{cases}$$

The user will see a three-colored image by using this function. After determining the colors, this operator renders a map image on each partition of the <Pixel, Color> RDD in parallel. The image of each partition is a map tile [46]. This viz operator finally generates a <ID, map image tile> RDD for the user, where ID is the tile ID. Moreover, this layer can stitch <ID, partial image> RDD to a single big image if needed.

Step IV: **Overlay Multiple Maps.** The user may also need spatial references such as landmarks or country boundaries to locate his POI when he views a map. This viz operator takes as input multiple <ID, partial image> RDDs and overlays them one by one in the order specified by the user. During the execution of this operator, this layer replaces (or mixes) the background pixel color with the front pixel color in parallel using a Map function.

6.2 Application 2: spatial aggregation

A data scientist in NYC Taxi Company would like to further investigate the distribution of taxi pickup points because he makes an interesting observation from the Region Heat Map: the distribution is very unbalanced. New York City has many taxi zones and each zone may have very different demographic information. Therefore, this time, the scientist wants to know the pickup points distribution per taxi zone. This spatial aggregation can be easily completed by GeoSpark: He first uses `Range Join Query` (SQL interface) to find the taxi trips that are picked up in each taxi zone then calculates the count per taxi zone. The result of this aggregation can be directly visualized by the GeoSpark Choropleth Map. The corresponding pseudo code is available in Fig. 11. The SQL statement issues a range join query since the inputs of "ST_Contains" are from two datasets. The "COUNT()" function counts the taxi trip pickup points per taxi zone. Each row in the join query result is in the form of "TaxiZoneShape,Count". The Choropleth Map API takes as input the map resolution and its "Visualize" function can plot the given join query result. Figure 12a is the generated Choropleth Map. A red colored zone means that more taxi trips were picked up in that zone. According to the Choropleth map, the scientist finds that the hottest zones are

```
var aggregatedDataframe = spark.sql
("SELECT TaxiZone.boundary, COUNT(TaxiTripTable.pickuppoint)
FROM TaxiZone,TaxiTripTable
WHERE ST_Contains(TaxiZone.boundary, TaxiTripTable.pickuppoint)
GROUP BY TaxiZone.boundary")
var vizEffect = new ChoroplethMap(resolutionX, resolutionY)
vizEffect.Visualize(aggregatedDataframe.toRDD)
```

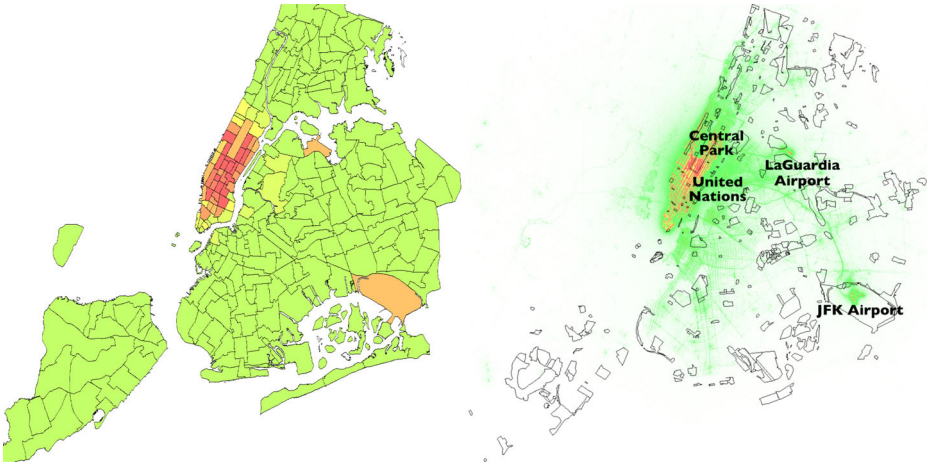
Fig. 11 Spatial Aggregation: Range Join Query + Choropleth Map

in Manhattan but there are two zones which are in orange colors but far from Manhattan. Then, he realizes these two zones are La Guardia Airport and JFK Airport, respectively.

6.3 Application 3: spatial co-location pattern mining

After finding many pickup points in La Guardia Airport and JFK Airport, the data scientist in the NYC Taxi Company makes a guess that the taxi pickup points are co-located with the New York area landmarks such as airports, museums, hospitals, colleges and so on. In other words, many taxi trips start from area landmarks (see Fig. 12b). He wants to use a quantitative metric to measure the degree of the co-location pattern. This procedure is called spatial co-location pattern mining.

Spatial co-location pattern mining is defined by two kinds of spatial objects that are often located in a neighborhood relationship. Ripley's K function [47] is commonly used in judging co-location. It usually executes multiple times and forms a 2 dimensional curve for observation. To obtain Ripley's K in each iteration, we need to calculate the adjacency matrix of two types of spatial objects given an updated distance restriction. To obtain the adjacency matrix, a time-consuming distance join is necessary. The user can use GeoSpark



a) Spatial Aggregation: Trip pickup points aggregated by taxi zones **b)** Spatial Co-location: Taxi trips co-locate with area landmarks

Fig. 12 Visualized spatial analytics

```
val PickupPoints = new PointRDD(sparkContext,DataPath1,FileType.CSV)
val AreaLandmarks = ShapefileReader.readToGeometryRDD(DataPath2)

PickupPoints.spatialPartitioning(GridType.KDBTREE)
PickupPoints.buildIndex(IndexTYpe.QuadTree)
PickupPoints.indexedRDD.cache()

for (i <- 1 to 10) {
currentDistance = beginDistance + i*distanceIncrement
AreaLandmarks.spatialPartitioning(PickupPoints.getPartitioner,
  currentDistance)
var adjacencyMatrixCount = JoinQuery.DistanceJoinQueryFlat (
  sparkContext, PickupPoints, AreaLandmarks, currentDistance).count
println(RipleyK(coefficient, adjacencyMatrixCount))}
```

Fig. 13 Spatial co-location pattern mining: iterative distance join query

RDD APIs to assemble this application[2] and migrate the adjacency matrix computation to a Spark cluster.

A snippet of the application source code is given in Fig. 13. The user first needs to create two Spatial RDDs, PickupPoints from a CSV file and AreaLandmarks from an ESRI shapefile. Then he should run spatial partitioning and build local index on the larger Spatial RDD (PickupPoints RDD is much larger in this case) and cache the processed Spatial RDD. Since Ripley's K requires many iterations (say, 10 iterations) with a changing distance, the user should write a for-loop. In each loop, he just needs to call `DistanceJoinQuery` API to join PickupPoints (cached) and AreaLandmarks. `DistanceJoinQuery` takes as input two Spatial RDDs and a distance and returns the spatial objects pairs that are located within the distance limitation. Although the distance is changing in each loop, the cached PickupPoints can be quickly loaded from memory to save plenty of time. The experiment verifies this conclusion.

7 Experiments

This section presents a comprehensive experiment analysis that experimentally evaluates the performance of GeoSpark and other spatial data processing systems. We compare four main systems:

- GeoSpark: we use GeoSpark 1.0.1, the latest release. It includes all functions needed in this experiment. We also open GeoSpark customized serializer to improve the performance.
- Simba [23]: we use Simba's latest GitHub repository which supports Spark 2.1. By default, Simba automatically opens Spark Kryo serializer.
- Magellan [24]: Magellan 1.0.6, the latest version, is used in our experiment. By default, Magellan automatically opens Spark Kryo serializer.
- SpatialHadoop [9]: we use SpatialHadoop 2.4.2 in the main GitHub repository in the experiments.

[2]Runnable example: https://github.com/jiayuasu/GeoSparkTemplateProject

Table 2 Dataset description

Dataset	Size	Description
OSMpostal	1.4 GB	171 thousand polygons, all postal areas on the planet (from Open Street Map)
TIGERedges	62 GB	72.7 million line strings, all streets in US. Each street is a line string which consists of multiple line segments
OSMobject	90 GB	263 million polygons, all spatial objects on the planet (from Open Street Map)
NYCtaxi	180 GB	1.3 billion points, New York City Yellow Taxi trip information

Datasets Table 2 summarizes four real spatial datasets used in the experiments, described as follows:

- OSMpostal [48]: contains 171 thousand polygons extracted from Open Street Maps. These polygons are the boundaries of all postal code areas on the planet.
- TIGERedges [49]: contains 72.7 million line strings provided by United States Census Bureau TIGER project. These line strings are all streets in the United States. Each of them consists of many line segments.
- OSMobject [50]: includes the polygonal boundaries of all spatial objects on the planet (from Open Street Maps). There are 263 million polygons in this dataset.
- NYCtaxi [37]: The dataset contains the detailed records of over 1.1 billion individual taxi trips in the city from January 2009 through December 2016. Each record includes pick-up and drop-off dates/times, pick-up and drop-off precise location coordinates, trip distances, itemized fares, and payment methods. But we only use the pick-up points in our experiment.

Workload We run the following spatial queries on the evaluated systems. Distance query and distance join results are not stated in the evaluation because they follow similar algorithms as the range query and the range join query, respectively.

- Spatial Range query: We run spatial range queries on NYCtaxi, OSMobject and TIGERedges with different query selectivities. Simba can only execute range queries on points and polygons without index. Its local index construction fails on points and polygons due to extremely high memory utilization. Magellan does not support accurate queries on polygons and line strings.
- Spatial KNN query: We test the spatial KNN query by varying K from 1 to 1000 on NYCtaxi, OSMobject and TIGERedges datasets. Only GeoSpark and SpatialHadoop support KNN query. Simba offers a KNN API but doesn't return any results.
- Spatial Range join query ($\bowtie$): We run OSMpostal $\bowtie$ NYCtaxi, OSMpostal $\bowtie$ OSMobject, OSMpostal $\bowtie$ TIGERedges. Magellan only supports the first case (polygons $\bowtie$ points). Simba only offers distance join between points and points (explained later).

Cluster settings We conduct the experiments on a cluster which has one master node and four worker nodes. Each machine has an Intel Xeon E5-2687WV4 CPU (12 cores, 3.0 GHz per core), 100 GB memory, and 4 TB HDD. We also install Apache Hadoop 2.6 and Apache Spark 2.1.1. We assign 10 GB memory to the Spark driver program that runs on the master machine, which is quite enough to handle any necessary global computation.

Performance metrics We use two main metrics to measure the performance of the evaluated systems: (1) Execution time: It stands for the total run time the system takes to execute a given job. (2) Peak execution memory: that represents the highest execution memory used by the system when running a given job.

7.1 Performance of range query

This section evaluates the performance of all four systems on NYCtaxi points, OSMobject polygons and TIGERedges linestrings (see Figs. 16, 17, and 18). We vary the range query selectivity factor as follows: (1) we use the entire dataset boundary as the largest query window.(2) then, we reduce the window size to its $\frac{1}{4}$th and generate many rectangular windows in this size but at random locations within the dataset boundary. (3) we keep reducing the window size until we get four different query selectivity factors (window size): $\frac{1}{64}$*boundary (SELE*1), $\frac{1}{16}$*boundary, $\frac{1}{4}$*boundary, boundary (SELE*64).

7.1.1 Impact of GeoSpark local indexing

For the NYCtaxi point dataset, the fastest GeoSpark method is GeoSpark range query without index (Fig. 14). We created different versions of GeoSpark, as follows: (1) NoIndex: GeoSpark running with no local indexes built in each SRDD partition. (2) RTree: GeoSpark running with R-Tree index built on each local SRDD partition. RTree-Search represents the R-tree search time, whereas RTree-Build denotes the R-Tree construction time. (3) QuadTree: Similar to RTree but with Quad-Tree index stored in each local SRDD partition instead. The NoIndex version of GeoSpark has similar range query search time as the indexed version of GeoSpark but the indexed (i.e., RTree or QuadTree) range query needs extra time to build the index. This is because the NYCtaxi point data is too simple

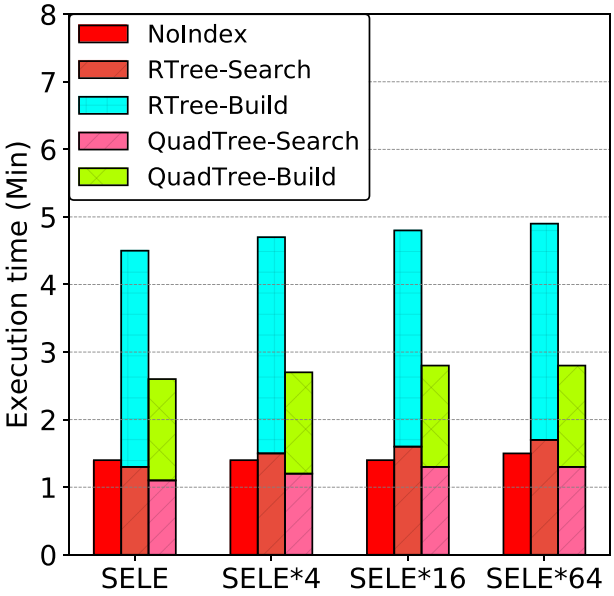


Fig. 14 The impact of GeoSpark local index on NYCtaxi range query

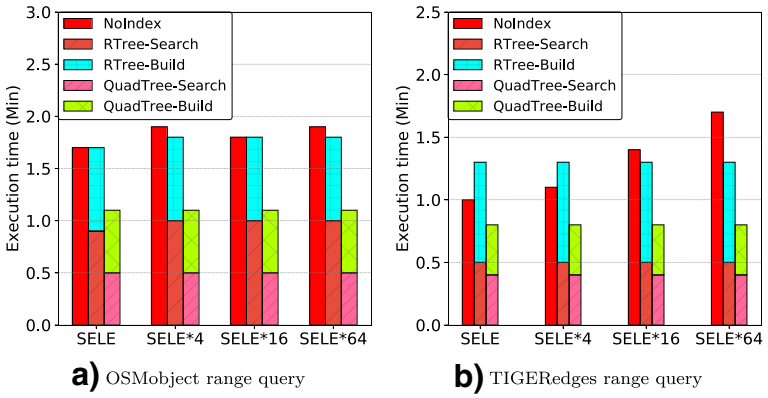


Fig. 15 The impact of GeoSpark local index on complex shapes range query

and using index to prune data does not save much computation time. The RTree version even has around 5% longer search time than the NoIndex version because GeoSpark follows the Filter and Refine model: after searching local indexes using MBRs, GeoSpark rechecks the spatial relation using real shapes of query window and spatial objects in order to guarantee the query accuracy (the window can be a very complex polygon rather than a rectangle). As it turns out in Fig. 15a and b (OSMobject and TIGERedges), GeoSpark RTree leads to 2 times less search time than the NoIndex version. GeoSpark QuadTree exhibits 4 times less search time than the NoIndex version. This makes sense because the tested polygon data and line string data have very complex shapes. For instance, a building's polygonal boundary (OSMobject) and a street's shape (TIGERedges) may have more than 20 coordinates. A local index in GeoSpark can prune lots of useless data to save the computation time while the regular range query needs to check all complex shapes in each partition across the cluster.

7.1.2 Comparing different systems

We show the range query execution times of the four systems in Figs. 16a, 17a and 18a. Simba can run range query on NYCtaxi points and OSMobject polygons without index. We do not use Simba R-Tree index range query because it always runs out of memory on these datasets even though the tested cluster has 400GB memory. Magellan and SpatialHadoop do not use indexes in processing a range query by default. We do not show the results for Magellan on OSMobject polygons and TIGERedges linestrings because it only uses MBRs and hence cannot return accurate query results. We use the most optimized GeoSpark in the comparison: GeoSpark range query without index in Fig. 16a and QuadTree in Figs. 17a and 18a.

For NYCtaxi point data (Fig. 16a), GeoSpark shows the least execution time. Furthermore, its execution time is almost constant on all query selectivities because it finishes the range query almost right after loading the data. On highly selective queries (ie.e., SELE*1), GeoSpark has similar execution time with Simba and Magellan. Moreover, the execution times of Simba, Magellan and SpatialHadoop increase with the growth of the query window size. On SELE*4, *16 and *64, Simba and Magellan are 2-3 times slower than GeoSpark.

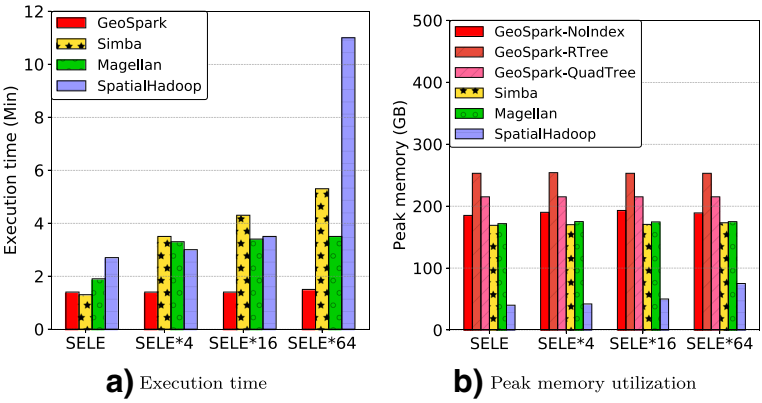


Fig. 16 Range query with different selectivity (SELE) on NYCtaxi points

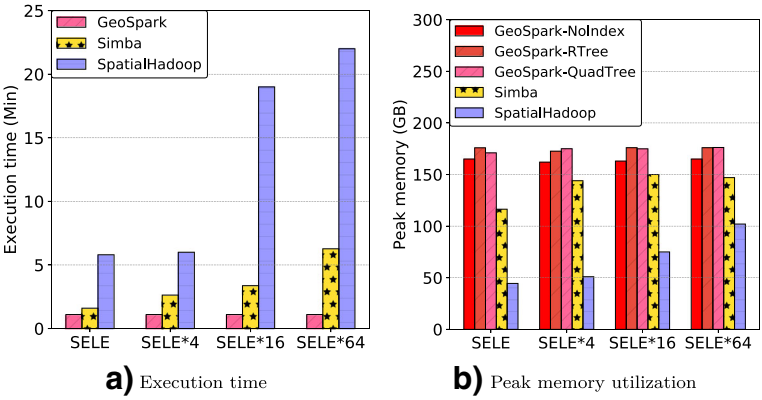


Fig. 17 Range query with different selectivity (SELE) on OSMobject polygons

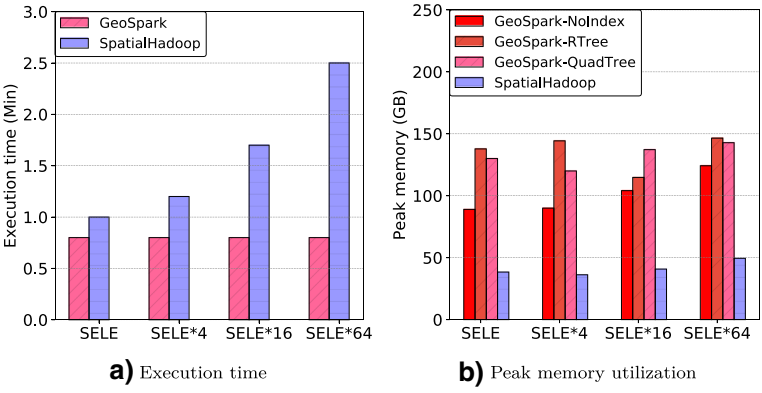


Fig. 18 Range query with different selectivity (SELE) on TIGERedges line strings

SpatialHadoop is 2 times slower than GeoSpark on SELE*1 and 10 times slower than GeoSpark on SELE*64.

For OSMobject polygons and TIGERedges linestrings (Figs. 17a and 18a), GeoSpark still has the shortest execution time. On OSMobject polygons, the QuadTree version of GeoSpark is around 2-3 times faster than Simba and 20 times faster than SpatialHadoop.

GeoSpark outperforms its counterparts for the following reasons: (1) on polygon and line string data, GeoSpark's Quad-Tree local index can speed up the query by pruning a large amount of data. (2) GeoSpark serializer overrides the default Spark generic serializer to use our own spatial data serialization logic.

According to range query's DAG (see Fig. 3), Spark performs RDD transformations to produce the result RDD and internally calls the serializer. GeoSpark serializer directly tells Spark how to understand and serialize the spatial data quickly while the default Spark generic serializer wastes some time on understanding complex spatial objects. Although the GeoSpark serializer is faster than the Spark default serializer, it is worth noting that in order to support heterogeneous spatial objects in a single Spatial RDD, GeoSpark customized serializer costs some extra bytes to specify the geometry type per spatial object besides the coordinates (see Section 4.5): (1) 3 extra bytes on point objects (15% additional memory footprint) (2) 2+n extra bytes on polygons or line strings, where n is the number of coordinates (7% addition memory footprint, if n = 10). Our experiment verifies the theoretical values. Figures 16b, 17b and 18b illustrate the peak memory utilization of different systems. Since the range query processing algorithm is similar on evaluated systems (a filter operation on all data partitions), the peak memory utilization is roughly equal to the input data memory footprint. Compared to Simba and Magellan, GeoSpark costs around 10% additional memory on points and polygons (Figs. 16b, 17b). GeoSpark Quad-Tree index range query costs 13% additional memory and R-Tree index range query costs 30% additional memory. This is because a tree index takes 10%-40% additional space to store the tree node information [35, 36]. In general, R-Tree consumes more space because it needs to store MBRs and extra child node information while Quad-Tree always has 4 child nodes. SpatialHadoop always has 2-3 times less memory utilization because Hadoop-based systems don't utilize memory much and all intermediate data is put on the disk.

7.2 Performance of K nearest neighbors (KNN) query

This section studies the performance of GeoSpark and SpatialHadoop. SpatialHadoop does not use an index for KNN queries by default. Simba offers a KNN API but the source code does not return any results on the tested datasets. We vary K to take the values of 1, 10, 100 and 1000 and randomly pick several query points within the dataset boundaries.

Since the KNN query shows similar performance trends as the range query, we only put the results of KNN on NYCtaxi points and OSMobject polygons. R-Tree [14, 41] data structure supports KNN because it uses MBR to represent tree node boundaries.

As depicted in Figs. 19a and 20a, on NYCtaxi points, GeoSpark NoIndex is $\tilde{2}0$ times faster than SpatialHadoop and GeoSpark RTree is $\tilde{6}$ times faster than SpatialHadoop. On OSMobject polygons, GeoSpark NoIndex is 5 times faster than SpatialHadoop and the RTree version of GeoSpark is 7 times faster than SpatialHadoop in terms of total execution time including the index building time. On point data, GeoSpark NoIndex shows similar search time performance as the indexed version (explained in Section 7.1).

The execution time of a KNN query in GeoSpark and SpatialHadoop remains constant with different values of K. That happens because the value of K is very small in contrast to the input data and the majority of the time is spent on processing the input data.

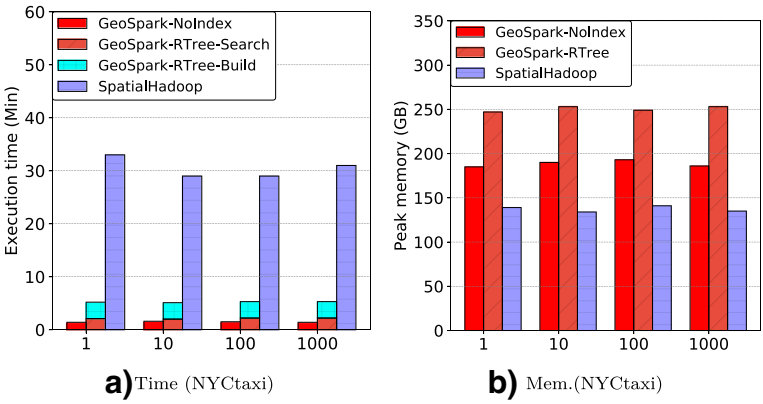


Fig. 19 KNN query with different K on NYCtaxi points

The peak memory utilization of GeoSpark is $\tilde{2}$ times higher than SpatialHadoop because Spark stores intermediate data in main memory. It is also worth noting that, although GeoSpark's peak memory utilization remains constant in processing range and KNN queries, SpatialHadoop's peak memory utilization of KNN queries is 3 times larger than that of range queries because SpatialHadoop needs to sort the candidate objects across the cluster, which leads to a data shuffle across the network. In practice, heavy network data transfer increases memory utilization.

7.3 Performance of range join query

This section evaluates the range join query performance of compared systems when joining the following datasets: (a) OSMpostal $\bowtie$ NYCtaxi, (b) OSMpostal $\bowtie$ OSMobject, (c) OSMpostal $\bowtie$ TIGERedges.

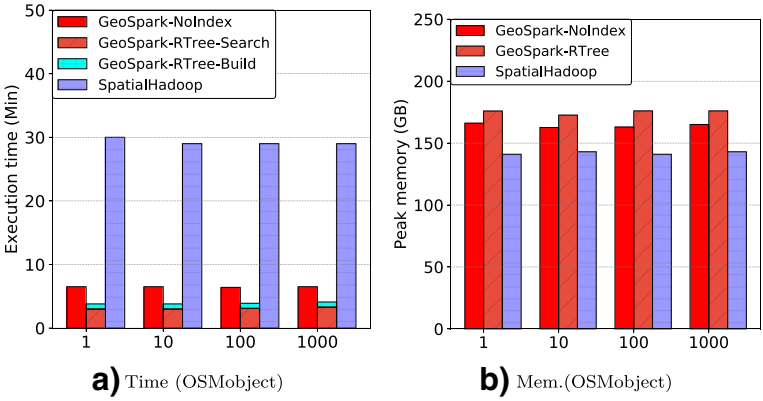


Fig. 20 KNN query with different K on OSMobject polygons

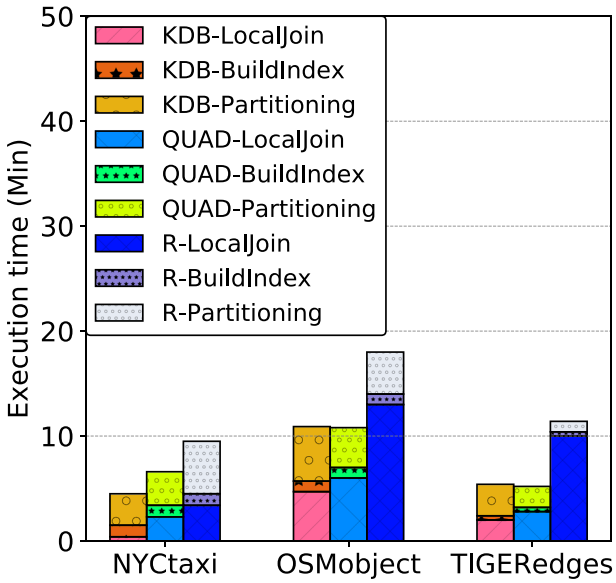


Fig. 21 The impact of spatial partitioning on range join in GeoSpark

7.3.1 Impact of spatial partitioning

As depicted in Fig. 21, GeoSpark KDB-Tree partitioning method exhibits the shortest local join time (KDB-LocalJoin) on all join queries. Quad-Tree partitioning local join time (QUAD-LocalJoin) is 1.5 times slower than KDB-LocalJoin. R-Tree partitioning local join time (R-LocalJoin) is around 2 times slower than KDB-Tree partitioning. This is because KDB-Tree partitioning generates more load-balanced grid cells. For instance, on OSMpostal polygons $\bowtie$ NYCtaxi points, during the spatial partitioning step (see the DAG and data flow of GSJoin in Fig. 4): (1) Shuffled serialized data across the cluster by GeoSpark are 12.9GB (KDB), 13.2GB (QUAD) and 13.4GB (R) (2) The min, median and max shuffled SRDD partition sizes are 4MB-6.4MB-8.8MB (KDB), 1MB-3.2MB-10.4MB (QUAD), 5MB-9.3MB-103MB (R). Obviously, the partition size of KBD-Tree partitioning is more balanced. Quad-Tree method is not as balanced as KDB-Tree. R-Tree partitioning has an overflow data partition, which is much larger than other partitions because R-Tree does not usually cover the entire space.

According to the example given above, it makes sense that GeoSpark KDB-Tree spatial partitioning has the least local join time and R-Tree partitioning has the slowest local join speed. Another factor that slows down R-Tree partitioning local join is the additional data shuffle step resulting from removing duplicates among overlapped partitions (see Section 5.3).

7.3.2 Comparing different systems

Magellan only supports OSMpostal $\bowtie$ NYCtaxi (polygons, points) using Z-Curve spatial partitioning. Simba only supports distance join between points and points. In order to make Simba work with OSMpostal $\bowtie$ NYCtaxi, we take the central point of each postal area

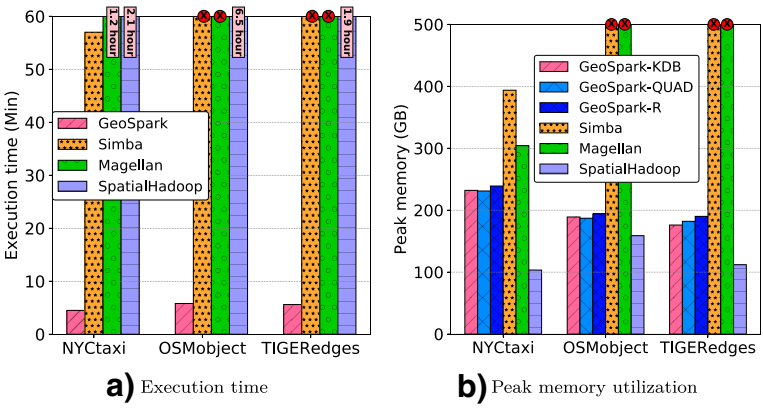


Fig. 22 Range join query performance on datasets ⋈ OSMpostal

in OSMpostal to produce a point dataset and use the average radius of OSMpostal polygon as distance to run the distance join. By default, Simba uses R-Tree spatial partitioning and builds local R-Tree indexes on the smaller dataset of the join query. SpatialHadoop uses Quad-Tree spatial partitioning by default and it also builds Quad-Tree local index on NYCtaxi, OSMobject and TIGERedges. GeoSpark uses three different spatial partitioning methods, KDB-Tree (KDB), Quad-Tree (QUAD), and R-Tree (R) (these partitioning methods are not GeoSpark local indexes). Based on the performance of GeoSpark range query and KNN query, we build GeoSpark Quad-Tree index on NYCtaxi, OSMobject and TIGERedges by default. The experimental result is shown in Fig. 22.

In Fig. 22a, we compare GeoSpark KDB-Tree partitioning join query with Quad-Tree local index, Simba, Magellan and SpatialHadoop. As we can see from this figure, in comparison with GeoSpark, Simba is around 10 times slower (OSMpostal ⋈ NYCtaxi), Magellan is 15 times slower (OSMpostal ⋈ NYCtaxi) and SpatialHadoop is more than 25 times slower on all join query scenarios. The execution time contains all parts in a join query including spatial partitioning, local index building and local join. A red cross means that this join data type is not supported by this system.

Simba exhibits higher join time for the following reasons: (1) it uses R-Tree partitioning. As explained above, R-Tree partitioning is not as balanced as KDB-Tree partitioning. (2) GeoSpark customized serializer tells Spark how to serialize spatial data exactly. Simba uses the default Spark kryo serializer which produces many unnecessary intermediate data when serializing and shuffling spatial data. (3) Simba's join algorithm shuffles lots of data across the cluster. Based on our test, Simba shuffles around 70GB data while GeoSpark only shuffles around 13GB.

GeoSpark outperforms Magellan because (1) Magellan uses Z-Curve spatial partitioning which leads to many overlapped partitions [16]. Using Z-Curve to index spatial objects loses lots of spatial proximity information. Thus, this forces the system to put lots of spatial objects that are impossible to intersect together. This wastes a significant amount of execution time and introduces a huge network data transfer. Based on our test, Magellan reads 623GB data through the network during the join query execution. (2) Magellan doesn't support any spatial tree index like R-Tree or Quad-Tree. (3) Magellan doesn't have customized serializer like GeoSpark.

SpatialHadoop is slow because SpatialHadoop has to put intermediate data on the disk and this becomes even worse on spatial join because the spatial partitioning part in the join query shuffles lots of data across the cluster which stresses memory as well as disk.

The peak memory utilization is given in Fig. 22b. GeoSpark has the lowest peak memory and Simba has 1.7 times higher peak memory utilization. The peak memory used by Simba is close to the upper limitation of our cluster, almost 400GB. That means if we increase the input data size, Simba will crash. Magellan has 1.3 times higher peak memory utilization than GeoSpark. SpatialHadoop still has the lowest peak memory which is around 1 - 2 times less than GeoSpark.

7.4 Performance of application use cases

This section evaluates the three use cases presented earlier in Section 6. We use the same GeoSpark source code in Figs. 9, 11 and 13 to test the performance: (1) App1: Region heat map: we first run a range query on NYCtaxi to only keep Manhattan region pick-up points then produce a map with OSM L6 zoom level [51] which has 4096 map tiles and 268 million pixels. (2) App2: Spatial aggregation: we perform a range join between NYCtaxi points and NYC taxi zones (published along with [37], 264 taxi polygonal zones in NYC) then count the taxi trip pickup points in each zone. (3) App3: Spatial co-location pattern mining: we cache the spatial-partitioned Spatial RDD and its local index and iterate co-location pattern mining 10 times. To be precise, we also write the applications using the compared Hadoop-based and Spark-based systems and test their performance.

It is worth noting that SpatialHadoop is able to plot heat maps (used in App1) and range join query (used in App2). SpatialHadoop uses Quad-Tree spatial partitioning by default and it also builds Quad-Tree local index on NYCtaxi. It doesn't support distance join query used in App3. Magellan only supports App2 (NYC taxi zones $\bowtie$ NYCtaxi (polygons, points) using Z-Curve spatial partitioning). It doesn't support distance join query in App3. Simba only supports distance join between points and points (used in App3) but it doesn't support NYC taxi zones $\bowtie$ NYCtaxi (polygons, points) (used in App2). By default, Simba uses R-Tree spatial partitioning and builds local R-Tree indexes on the smaller dataset of the join query. GeoSpark uses KDB-Tree spatial partitioning and builds Quad-Tree index on NYCtaxi. The experimental result is shown in Table 3. An X means that this join type is not supported by this system.

As it turns out in Table 3, region heat map takes GeoSpark 7 minutes because it needs to repartition the pixels across the cluster following the uniform grids. Its peak memory is

Table 3 Performance of GeoSpark applications (min is for execution time, GB is for peak memory)

Application	GeoSpark	Simba	Magellan	SpatialHadoop
Region heat map	7 min	X	X	42min
	193GB	X	X	65GB
Spatial aggregation	17 min	X	20min	41min
	234GB	X	307GB	105GB
Spatial coLocation	1st iter., 8.5min	Crashed	X	X
	10 iter., 10.5min execution, 240GB cached, 101GB	Crashed	X	X

dominated by the range query part because the repartitioning part only shuffles the Manhattan region data which is smaller than the input Spatial RDD. SpatialHadoop runs 4 times slower than GeoSpark because it puts intermediate on disk. However, its memory utilization is much less than GeoSpark.

Regarding App2 spatial aggregation using NYCtaxizone $\bowtie$ NYCtaxi, the execution time of GeoSpark is ~17 minutes, which is 2 times longer than that on OSMpostal $\bowtie$ NYCtaxi because there are 10 times more taxi zones than OSMpostal postal areas in New York City. In addition, spatial aggregation executes a CountBy operation to count the pickup point per taxi zone and this leads to data shuffle across the cluster. GeoSpark is faster than Magellan because of KDB-Tree partitioning and Quad-Tree index. Its memory consumption is lower than Magellan due to the help of GeoSpark customized serializer. SpatialHadoop is still around 2 times slower than other systems but its peak memory is lower.

As depicted in Table 3, GeoSpark and Simba support App3 Co-location pattern mining because they support distance joins. However, Simba takes a very long time to run and eventually crashes because of memory overflow in distance join query. GeoSpark is the only system can properly handle this application and finish it in a timely manner. Recalled that we run 10 iterations using GeoSpark in this application. The first iteration of the co-location pattern mining algorithm takes 8.5 minutes and all 9 others take 2.3 minutes. The first iteration takes 30 times more time than the other iterations. This happens because GeoSpark caches the spatial RDD and the corresponding local indexes. Hence, the upcoming iteration directly reads data from the memory cache, which saves a lot of time.

8 Conclusion and future work

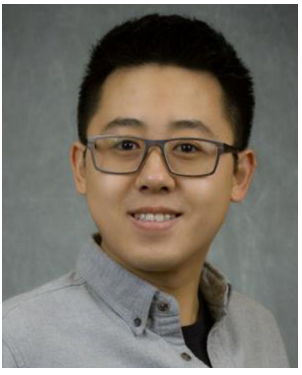
The paper describes the anatomy of GeoSpark, an in-memory cluster computing framework for processing large-scale spatial data. GeoSpark provides Spatial SQL and Spatial RDD APIs for Apache Spark programmers to easily develop spatial analysis applications. Moreover, the system provides native support for spatial data partitioning, indexing, query processing, data visualization in Apache Spark to efficiently analyze spatial data at scale. Extensive experiments show that GeoSpark outperforms Spark-based systems such as Simba and Magellan up to one order of magnitude and Hadoop-based system such as SpatialHadoop up to two orders of magnitude. The release of GeoSpark stimulated the database community to work on spatial extensions to Spark. We expect that more researchers and practitioners will contribute to GeoSpark code base to support new spatial data analysis applications.

References

1. NRC (2001) Committee on the science of climate change, climate change science: an analysis of some key questions, National Academies Press, Washington
2. Zeng N, Dickinson RE, Zeng X (1996) Climatic impact of amazon Deforestation?A mechanistic model study. Journal of Climate 9:859–883
3. Chen C, Burton M, Greenberger E, Dmitrieva J (1999) Population migration and the variation of dopamine D4 receptor (DRD4) allele frequencies around the globe. Evol Hum Behav 20(5):309–324

4. Woodworth PL, Menéndez M, Gehrels WR (2011) Evidence for century-timescale acceleration in mean sea levels and for recent changes in extreme sea levels. Surv Geophys 32(4-5):603–618
5. Dhar S, Varshney U (2011) Challenges and business models for mobile location-based services and advertising. Commun ACM 54(5):121–128
6. PostGIS Postgis. http://postgis.net/
7. Open Geospatial Consortium. http://www.opengeospatial.org/
8. Aji A, Wang F, Vo H, Lee R, Liu Q, Zhang X, Saltz JH (2013) Hadoop-GIS: a high performance spatial data warehousing system over MapReduce. Proc Int Conf on Very Large Data Bases, VLDB 6(11):1009–1020
9. Eldawy A, Mokbel MF (2015) Spatialhadoop: a mapreduce framework for spatial data. In: Proceedings of the IEEE International Conference on Data Engineering, ICDE, pp 1352–1363
10. Zaharia M, Chowdhury M, Das T, Dave A, Ma J, McCauly M, Franklin MJ, Shenker S, Stoica I (2012) 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, pp 15–28
11. Ashworth M (2016) Information technology – database languages – sql multimedia and application packages – part 3: Spatial, standard, International organization for standardization, Geneva, Switzerland
12. Pagel B-U, Six H-W, Toben H, Widmayer P (1993) Towards an analysis of range query performance in spatial data structures. In: Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART symposium on Principles of Database Systems PODS '93
13. Patel JM, DeWitt DJ (1996) Partition based spatial-merge join. In: Proceedings of the ACM international conference on management of data, SIGMOD, pp 259–270
14. Guttman A (1984) R-trees: a dynamic index structure for spatial searching. In: Proceedings of the ACM international conference on management of data, SIGMOD, pp 47–57
15. Samet H (1984) The quadtree and related hierarchical data structures. ACM Comput Surv (CSUR) 16(2):187–260
16. Eldawy A, Alarabi L, Mokbel MF (2015) Spatial partitioning techniques in spatial hadoop. Proc Int Conf on Very Large Data Bases, VLDB 8(12):1602–1605
17. Eldawy A, Mokbel MF, Jonathan C (2016) Hadoopviz: A mapreduce framework for extensible visualization of big spatial data. In: Proceedings of the IEEE International Conference on Data Engineering, ICDE, pp 601–612
18. Eldawy A (2014) Pigeon: a spatial mapreduce language. In: IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014, pp 1242–1245
19. Lu J, Guting RH (2012) Parallel secondo: boosting database engines with Hadoop. In: International conference on parallel and distributed systems, pp 738 –743
20. Vo H, Aji A, Wang F (2014) SATO: a spatial data partitioning framework for scalable query processing. In: Proceedings of the ACM international conference on advances in geographic information systems, ACM SIGSPATIAL, pp 545–548
21. Thusoo A, Sen JS, Jain N, Shao Z, Chakka P, Anthony S, Liu H, Wyckoff P, Murthy R (2009) Hive: a warehousing solution over a Map-Reduce framework. In: Proceedings of the International Conference on Very Large Data Bases, VLDB, pp 1626–1629
22. Armbrust M, Xin RS, Lian C, Huai Y, Liu D, Bradley JK, Meng X, Kaftan T, Franklin MJ, Ghodsi A, Zaharia M (2015) Spark SQL: relational data processing in spark. In: Proceedings of the ACM international conference on management of data, SIGMOD, pp 1383–1394
23. Xie D, Li F, Yao B, Li G, Zhou L, Guo M (2016) Simba: efficient in-memory spatial analytics. In: Proceedings of the ACM international conference on management of data, SIGMOD
24. Sriharsha R Geospatial analytics using spark. https://github.com/harsha2010/magellan
25. You S, Zhang J, Gruenwald L (2015) Large-scale spatial join query processing in cloud. In: Proceedings of the IEEE International Conference on Data Engineering Workshop, ICDEW, pp 34–41
26. Hughes NJ, Annex A, Eichelberger CN, Fox A, Hulbert A, Ronquest M (2015) Geomesa: a distributed architecture for spatio-temporal fusion. In: SPIE defense+ security, pp 94730F–94730F, International society for optics and photonics
27. Finkel RA, Bentley JL (1974) Quad trees a data structure for retrieval on composite keys. Acta informatica 4(1):1–9
28. Herring JR (2006) Opengis implementation specification for geographic information-simple feature access-part 2: Sql option, Open Geospatial Consortium Inc
29. Apache Accumulo. https://accumulo.apache.org/
30. Hunt P, Konar M, Junqueira FP, Reed B (2010) Zookeeper: Wait-free coordination for internet-scale systems. In: USENIX annual technical conference, Boston, MA, USA June 23-25

31. Butler H, Daly M, Doyle A, Gillies S, Schaub T, Schmidt C (2014) Geojson, Electronic. http://geojson.org
32. Perry M, Herring J (2012) Ogc geosparql-a geographic query language for rdf data, OGC Implementation Standard Sept
33. Group H et al (2014) Hierarchical data format version 5
34. ESRI E (1998) Shapefile technical description, an ESRI white paper
35. Yu J, Sarwat M (2016) Two birds, one stone: A fast, yet lightweight, indexing scheme for modern database systems. Proc Int Conf on Very Large Data Bases, VLDB 10(4):385–396
36. Yu J, Sarwat M (2017) Indexing the pickup and drop-off locations of NYC taxi trips in postgresql - lessons from the road. In: Proceedings of the international symposium on advances in spatial and temporal databases, SSTD, pp 145–162
37. Taxi NYC, Commission L Nyc tlc trip data. http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml
38. Robinson JT (1981) The k-d-b-tree: a search structure for large multidimensional dynamic indexes. In: Proceedings of the 1981 ACM SIGMOD international conference on management of data, Ann Arbor, Michigan, April 29 - May 1, 1981, pp 10–18
39. Opyrchal L, Prakash A (1999) Efficient object serialization in java. In: Proceedings of the 19th IEEE international conference on distributed computing systems workshops on electronic commerce and web-based applications/middleware, 1999, IEEE, pp 96–101
40. Cao P, Wang Z (2004) Efficient top-k query calculation in distributed networks. In: Proceedings of the twenty-third annual ACM symposium on principles of distributed computing, PODC 2004, St. John's, Newfoundland, Canada, July 25-28, 2004, pp 206–215
41. Roussopoulos N, Kelley S, Vincent F (1995) Nearest neighbor queries. In: Proceedings of the ACM international conference on management of data, SIGMOD, pp 71–79
42. Zhou X, Abel DJ, Truffet D (1998) Data partitioning for parallel spatial join processing. Geoinformatica 2(2):175–204
43. Luo G, Naughton JF, Ellmann CJ (2002) A non-blocking parallel spatial join algorithm
44. Zhang S, Han J, Liu Z, Wang K, Xu Z (2009) SJMR: parallelizing spatial join with mapreduce on clusters. In: Proceedings of the 2009 IEEE international conference on cluster computing, August 31 - September 4, 2009, New Orleans, Louisiana, USA, pp 1–8
45. Dittrich J, Seeger B (2000) Data redundancy and duplicate detection in spatial join processing. In: Proceedings of the 16th international conference on data engineering, San Diego, California, USA, February 28 - March 3, 2000, pp 535–546
46. Consortium OG (2010) Opengis web map tile service implementation standard, tech. rep., Tech. Rep. OGC 07-057r7. In: Masó J, Pomakis K, Julià N (eds) Open Geospatial Consortium. Available at http://portal.opengeospatial.org/files
47. Ripley BD (2005) Spatial statistics, vol 575, Wiley, New York
48. Haklay MM, Weber P (2008) Openstreetmap: User-generated street maps. IEEE Pervasive Computing 7(4):12–18
49. TIGER data. https://www.census.gov/geo/maps-data/data/tiger.html
50. OpenStreetMap. http://www.openstreetmap.org/
51. OpenStreetMap. Open street map zoom levels. http://wiki.openstreetmap.org/wiki/Zoom_levels



Jia Yu is a PhD student at the Computer Science department, School of Computing, Informatics, and Decision Systems Engineering, Arizona State University, where he also is a member of Data Systems Lab. His research interests focus on database management systems (DBMS), spatialtemporal databases, distributed data computation/storage engine, data indexing, query optimization and data visualization. Jia is the main contributor of several open-sourced projects such as GeoSpark system and Hippo lightweight index.

Zongsi Zhang is a data engineer at Grab Ltd, Singapore. He obtained his Master degree in Computer Science from School of Computing, Informatics, and Decision Systems Engineering, Arizona State University. He used to be a member of Data Systems Lab. His works focus on high performance & distributed computation system and automation of data preprocess. Zongsi has some vital contributions to GeoSpark System.

Mohamed Sarwat is an Assistant Professor of Computer Science and the director of the Data Systems (DataSys) lab at Arizona State University (ASU). Mohamed is also an affiliate member of the Center for Assured and Scalable Data Engineering (CASCADE). Before joining ASU in August 2014, Mohamed obtained his MSc and PhD degrees in computer science from the University of Minnesota in 2011 and 2014, respectively. His research interest lies in the broad area of data management systems. Mohamed is a recipient of the University of Minnesota Doctoral Dissertation Fellowship. His research work has been recognized by the "Best Research Paper Award" in the IEEE 16th International Conference on Mobile Data Management (MDM 2015), the "Best Research Paper Award" in the 12th International Symposium on Spatial and Temporal Databases (SSTD 2011), and a ?Best of Conference? citation in the IEEE 28th International Conference on Data Engineering (ICDE 2012).