GeoSparkViz: A Scalable Geospatial Data Visualization Framework in the Apache Spark Ecosystem

Jia Yu
Arizona State University, Tempe AZ
jiayu2@asu.edu

Zongsi Zhang
Arizona State University, Tempe AZ
zzhan236@asu.edu

Mohamed Sarwat
Arizona State University, Tempe AZ
msarwat@asu.edu

ABSTRACT

Data Visualization allows users to summarize, analyze and reason about data. A map visualization tool first loads the designated geospatial data, processes the data and then applies the map visualization effect. Guaranteeing detailed and accurate geospatial map visualization (e.g., at multiple zoom levels) requires extremely high-resolution maps. Classic solutions suffer from limited computation resources and hence take a tremendous amount of time to generate maps for large-scale geospatial data.

The paper presents GeoSparkViz a large-scale geospatial map visualization framework. GeoSparkViz extends a cluster computing system (Apache Spark in our case) to provide native support for general cartographic design. The proposed system seamlessly integrates with a Spark-based spatial data management system, GeoSpark. It provides the data scientist a holistic system that allows her to perform data management and visualization on spatial data and reduces the overhead of loading the intermediate spatial data generated during the data management phase to the designated map visualization tool. GeoSparkViz also proposes a map tile data partitioning method that achieves load balancing for the map visualization workloads among all nodes in the cluster. Extensive experiments show that GeoSparkViz can generate a high-resolution (i.e., Gigapixel image) Heatmap of 1.7 billion OpenStreetMaps objects and 1.3 billion NYC taxi trips in ≈4 and 5 minutes on a four-node commodity cluster, respectively.

CCS CONCEPTS

• **Information systems** → *Spatial-temporal systems*; • **Computing methodologies** → *Distributed algorithms*; • **Human-centered computing** → *Visualization techniques*;

KEYWORDS

Distributed computation, Spatial visualization, Big spatial data

ACM Reference Format:
Jia Yu, Zongsi Zhang, and Mohamed Sarwat. 2018. GeoSparkViz: A Scalable Geospatial Data Visualization Framework in the Apache Spark Ecosystem. In *SSDBM '18: 30th International Conference on Scientific and Statistical Database Management, July 9–11, 2018, Bozen-Bolzano, Italy*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3221269.3223040

SSDBM '18, July 9–11, 2018, Bozen-Bolzano, Italy

ACM ISBN 978-1-4503-6505-5/18/07…$15.00
https://doi.org/10.1145/3221269.3223040

1 INTRODUCTION

Map Visualization allows users to summarize, analyze and reason about geospatial data. For example, a heat map of the New York City taxi trips helps the taxi company locate the hot pick-up and drop-off zones. Also, a scatter plot of the United States road network exposes isolated areas nationwide. A politician may utilize a Choropleth map to visualize the Twitter sentiment of each presidential candidate in each US county. To achieve that, a map visualization tool first loads the designated spatial data and then applies the map visualization effect, e.g., Heatmap, on such data. The system first rasterizes the spatial objects, then aggregates overlapped pixels, colorizes the map pixels, and finally renders an image for each spatial map tile.

Guaranteeing detailed and accurate geospatial map visualization (e.g., at multiple zoom levels) requires extremely high-resolution maps. Classic solutions suffer from limited computation resources and hence take a tremendous amount of time to generate maps for large-scale spatial data [19]. Moreover, the existing systems that decouple spatial data management and map visualization demand substantial overhead to connect the data processing engine to the map visualization tool. To remedy that, it is essential to combine both spatial data management and map visualization in the same cluster. For instance, SpatialHadoop can store the output of the spatial data management phase on HDFS then HadoopViz [8] starts generating the map visualization effect. However, the MapReduce approach suffers from large delays due to the need for storing intermediate results on HDFS. On the other hand, state-of-the-art Spark-based spatial data processing systems (e.g., GeoSpark [26], SIMBA [24], LocationSpark [22], SparkGIS [3]) can perform data management operations at scale but do not provide in-house support for geospatial map visualization. Hence, such systems lose the opportunity to connect and optimize both data management and map visualization together.

The paper presents GeoSparkViz [1] a large-scale geospatial map visualization framework. GeoSparkViz extends a massively parallelized cluster computing system (Apache Spark in our case) to provide native support for general cartographic design and seamlessly integrates with a Spark-based spatial data management system, GeoSpark [26]. Two benefits come as a byproduct of performing the data management and map visualization process in the same cluster: (1) It provides the data scientist a holistic system that allows her to perform data management and visualization on spatial data. That plug-and-play approach increases the usability of the system. (2) It reduces the overhead of loading the intermediate spatial data generated during the data management phase to

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

the designated map visualization tool. In addition, GeoSparkViz has the following main contributions:

- GeoSparkViz encapsulates the main steps of the geospatial map visualization process, e.g., rasterize spatial objects, aggregate pixels, into a set of massively parallelized RDD transformations in Apache Spark. Such RDD transformations provide out-of-the-box support for the user to generate a variety of map visualization effects, e.g., scatter plot and heat map, on RDDs. Furthermore, the user can customize these visualization transformations to specify user-supplied colors and even implement new RDD transformations to assemble a new map visualization effect. Most importantly, GeoSparkViz integrates the map visualization transformations with existing RDD transformations/actions (e.g., map, reduce, join, filter). To achieve that, the system leverages both the spatial data distribution and the geospatial map tile statistics to decide the sequence of operations in the Spark DAG.
- GeoSparkViz also proposes a map tile-aware data partitioning method that achieves load balancing for the map visualization workloads among all nodes in the cluster. To achieve that, the system draws a sample of the loaded spatial data to figure out its spatial distribution and hence generate non-uniform map tiles accordingly. The main goal is to partition the data once and use such partitioning for the entire visualization task in order to reduce the amount of data shuffled in the Spark cluster. Besides partitioning, the system also replicates a subset of the map pixels in order to avoid shuffling data during the map visualization phase.
- A full-fledged prototype of GeoSparkViz is implemented in Apache Spark. The paper presents an extensive experimental evaluation that compares and contrasts the performance of GeoSparkViz with state-of-the-art distributed map visualization systems over real large-scale spatial data extracted from the NYC taxi trip dataset, Open Street Maps and Topologically Integrated Geographic Encoding and Referencing (TIGER) project. The experiments show that GeoSparkViz can achieve up to 5 times faster run time performance than its counterparts for various map visualization workloads.

Given this outlook, the rest of the paper is organized as follows: Section 2 presents the related work. An overview of GeoSparkViz is given in Section 3. Section 4 illustrates the user interfaces in GeoSparkViz and explains how to extend default visualization effects. The main map visualization steps are described in Section 5. Sections 6 explains the map tile data partitioner. Several use case scenarios are given in Section 7. A comprehensive experimental evaluation is given in Section 8. Section 9 concludes the paper.

2 RELATED WORK

Geospatial map visualization: Many commercial map services such as Google Maps and MapBox allow users to visualize a small amount spatial data on a single machine. Other single machine solutions [5, 9, 14] let user visualize large-scale data by downsizing the spatial data (e.g. data sampling) but they are not able to provide high-quality images when the user wants more details.

Apache Spark System: Apache Spark [21] 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. The fault-tolerance and job scheduler in Spark rely on Directed Acyclic Graph (DAG). DAG in Apache Spark consists of vertexes and directed edges where each vertex represents an RDD and an edge represents the operation to be applied to an RDD. Every edge is directed from a source RDD and a destination RDD. A DAG is divided into multiple stages. Each stage contains a sequence of pipelined RDD transformations and the boundaries of stages are shuffle operations including Actions and some Transformations such as GroupByKey. All transformations in a stage are pipelined together and launched by the Spark DAG scheduler.

Scalable spatial data analytics systems: There exist efforts that aim at extending state-of-the-art parallel and distributed data systems as means to support massive-scale geospatial data processing. Parallel SECONDO [13], Hadoop-GIS [1], and SpatialHadoop [6] extend the Hadoop ecosystem to support global and local spatial indexing and to achieve efficient query processing over large-scale spatial data. SIMBA [24], LocationSpark [22], SparkGIS [3] and GeoSpark [26] extend Apache Spark to support SQL applications on Geospatial data types. Although the aforementioned systems can scale spatial data processing on a cluster, they do not provide support for map visualization.

Scalable map visualization systems. There is a large body of research that builds upon parallel / distributed system approach to scale the visualization workflow [7, 8, 11, 12, 15, 18]. SHAHED [7] and HadoopViz [8] use MapReduce to parallelize the map image rendering pipelines such as scatter plot and heat map. However, SHAHED and HadoopViz are not able to co-optimize the map visualization process with other distributed query processing operations. In such cases, the user has to load, process and store the intermediate spatial data separately. GeoMesa [11] and Geotrellis [12] extend Apache Spark to manipulate spatial objects and pixels but they still cannot connect classic query processing operations and map visualization. MapD [15] leverages the Graphic Processing Unit (GPU) to parallelize and hence speed up queries and pixel manipulation in visualization but it is still limited to a single machine.

3 GEOSPARKVIZ ARCHITECTURE

As depicted in Figure 1, GeoSparkViz contains two main components: (1) *Map Visualization Pipeline* (2) *Map Tile Data Partitioner*. GeoSparkViz works in concert with the existing GeoSpark (Spatial RDD layer and Spatial Query Processing Layer) to deliver a one-stop in-memory cluster computation framework. The user is able to perform visual analytics (Scatter Plot, Heat Map, Choropleth Map) on his massive, yet interesting, geospatial data at a lower time cost without getting involved into the underlying implementation details.

3.1 Map Visualization Pipeline

GeoSparkViz breaks to pieces a general map visualization generation pipeline and parallelizes each piece, namely visualization step,

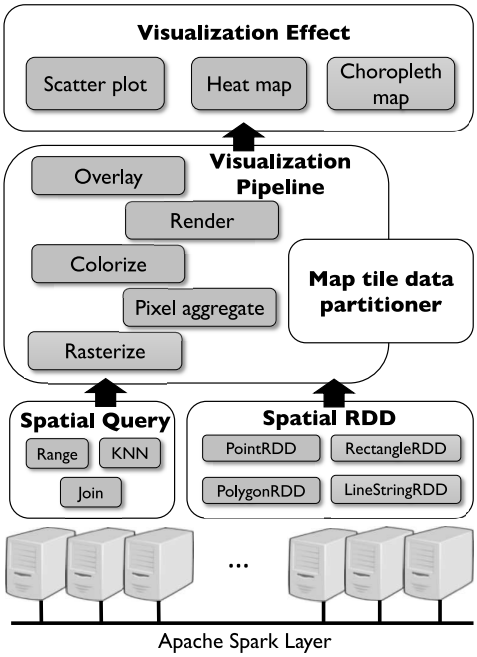


Figure 1: GeoSparkViz Overview

in a cluster system. These decoupled steps offer flexibility to both geospatial visualization experts and big data manipulators. On the one hand, the user can easily pick proper visualization steps in conjunction with GeoSpark Spatial RDD and spatial queries (explained later) to design new visual analytics pipeline which fits him best. Actually, the use case scenarios in Section 7 exactly follow this design mechanism. On the other hand, GeoSparkViz exposes these visualization steps to the user using extensible APIs. The user can easily override a step to implement new algorithms or rules. For instance, the user can rewrite the colorizing operator and put new coloring rules.

Step I: Rasterize spatial objects. This visualization step takes as input the massive datasets from various data sources and the designated image quality (in terms of pixel resolution). It then rasterizes/maps each spatial object (such as point, polygon and line string) to pixels (positions on the screen coordinate system) in parallel. Each pixel in the produced dataset carries an attribute called pixel weight which decides the color. To be precise, the output is a pixel-weight pair RDD.

Step II: Aggregate overlapped pixels. This step aggregates pixels that overlap with others to make sure each screen position has a deterministic weight. For each position on the screen, it enforces an aggregation strategy on the weights of all pixels which locate at this position to determine a final weight. GeoSparkViz accepts several aggregation strategies such as uniform, min, max, and average such that the finalized map can obtain different effects. In addition, this visualization step can apply classic image filters such as sharp, blur or diffusion to the pixels' weight (which decides the color). This step produces a pixel-weight pair RDD.

Step III: Colorize map pixels. This visualization step assigns colors to all pixels distributed in the cluster according to pixel weights by enforcing the given coloring rule. Hence, it generates a pixel-color pair RDD.

Step IV: Render map tile. This visualization step takes as input the distributed pixel-color RDD, uses this RDD to render map tiles, and finally generates a distributed map tile RDD.

Step V: Overlay multiple maps. This visualization step takes as input multiple distributed map tile datasets and overlays them one by one. For each map tile in a map tile dataset, this operator looks for the corresponding tiles from other datasets across the cluster and overlays them.

3.2 Map Tile-Aware Data Partitioner

The data partitioning component of the system uses a space partitioning method to repartition a given pixel dataset across the cluster. Pixels that fall inside a logical space partition go to the same physical data partition and stay at the same machine. Therefore, pixels on the same data partition can easily plot out a map tile of this partition when rendering.

The partitioner accommodates the need for visual analytics but also takes into account load balancing when processing skewed geospatial data. On one hand, it makes sure that each data partition contains a roughly similar number of pixels to avoid "stragglers" (a machine that takes much more inputs than others so that performs slowly). On the other hand, the logical space partition boundary of each data partition is derived from a map tile space partition of the final map so that data partitions that belong to the same tile space are able to be stitched together and produce the tile.

3.3 Spatial RDD and spatial query

GeoSparkViz is built upon GeoSpark[26, 27], which is equipped with an out-of-the-box Spatial Resilient Distributed Dataset (Spatial RDD) to provide support for spatial data types, indexes, and geometrical operations at scale. It also extends SparkSQL to offer Spatial SQL interface that follows SQL/MM-Part 3 standard [2].

GeoSparkViz harnesses the Spatial RDD and spatial query operators, including range query, range join query, distance join query and so on, from GeoSpark to process spatial data involved in a geospatial visual analytics workload. The black-boxed Spatial RDD and spatial query operators are able to exchange intermediate data with other GeoSparkViz components such as map visualization pipeline and map tile data partitioner immediately via memory.

4 GEOSPARKVIZ API

GeoSparkViz provides out-of-the-box support for three map visualization effects, scatter plot, heat map and choropleth map. These effects are presented to the user as Spark transformations to visualize the given Spatial RDD and spatial query results. In addition, GeoSparkViz is extensible so a user can simply extend a map visualization effect and embed her own visualization rule.

4.1 Create Spatial RDD and run spatial queries

GeoSparkViz takes as input any Spatial RDD and spatial queries then visualize them in the same cluster. Therefore, before using GeoSparkViz, the user should create corresponding RDDs for his spatial data and execute several spatial queries, as follows:

```
val mySpatialRDD = new SpatialRDD(sc,inputPath,FileFormat.WKT)
val queryResult = Query.SpatialRangeQuery(mySpatialRDD,queryWindow)
```

He can create a Spatial RDD and issue a spatial range query on the built RDD in just two lines of code. The user can either visualize *mySpatialRDD* or *queryResult* because they both are Spatial RDD.

4.2 Create map visualization effects

Users can select an instance from the three map visualization effects. GeoSparkViz produces the effects using the same map visualization pipeline but applies different rules such as rasterization rules, aggregation strategies, and colorizing rules. The pre-defined map visualization effects are as follows:

Scatter Plot.[2] A scatter plot graph uses the spatial objects' location coordinates to plot pixel points. Each pixel is rendered to the same uniform color. The pattern may reveal a relationship between spatial objects and show trends in the data.

Heat Map. This effect consists of standard steps in the map visualization pipeline. Different from the scatter plot, the heat map utilizes the aggregation strategies except Uniform to produce a colorful map. The color is decided by the weight of each pixel. The user can also add an image filter to observe special effects.

Choropleth Map. consists of many different areas, which are shaded or patterned in proportion to the measurement of the statistical metric being displayed on the map, such as population density in different countries. The Choropleth map provides an easy way to show the variability level of the metric within a region.

To plot an effect for Spatial RDD, the user first needs to create an instance of map visualization effect and then call *Visualize* function to plot a Spatial RDD. The APIs are as follows:

```
val myScatterPlot = new ScatterPlot (resolutionX, resolutionY, Rules)
val mapTileRDD = myScatterPlot.Visualize(mySpatialRDD)
```

As shown above, the *Rules* parameter is a configuration file which specifies several visualization rules, including rasterization rules, aggregation strategies, and colorizing rules (explained in Section 5). In the file, the user can also specify that use non-spatial attributes to decide the colors of the map (see Initial weight in Section 5.1). The *Visualize* function produces a MapTile RDD which consists of map tiles (each partition is a map tile). The user can either persist this image to permanent storage such as disks / HDFS / Amazon S3 or overlay this MapTile RDD with other MapTile RDD (see Overlay in Section 5.5).

4.3 Define a custom map visualization effect

Besides the pre-defined map visualization effects and limited visualization rules, GeoSparkViz users are free to extend the visualization effect and assemble their custom map visualization effects from scratch. To achieve that, GeoSparkViz provides an abstract class called *BaseEffect*, which has four functions, as follows.

```
abstract class BaseEffect {
protected <Pixel,weight>RDD Rasterize (<Spatial object> RDD) {...}
protected <Pixel,weight>RDD PixelAggregate(<Pixel,weight>RDD) {...}
protected <Pixel,color>RDD Colorize(<Pixel,weight>RDD) {...}
protected <TileID,MapTile>RDD Render(<Pixel,color>RDD) {...}
```

[2]Demo video: http://www.public.asu.edu/~jiayu2/video/geosparkviz.mp4

Algorithm 1: Step I: Rasterize

```
Input: <Spatial object> RDD
Output: <Pixel, weight> pair RDD
Function Map(spatial object O)
    weight = O's non-spatial attribute;
    switch Rasterizing rule do
        case "outline-only" do
            Decompose O into line segments;
            Find all pixels covered by line segments;
            return <Pixel coordinate, weight>;
        case "filling area" do
            Find all pixels that are within the polygon boundary;
            return <pixel coordinate, weight>;
```

```
public <TileID,MapTile>RDD Visualize(<Spatial object> RDD) {
    Rasterize();
    PixelAggregate();
    Colorize();
    Render();
    }
}
```

As shown above, each function is a step in the map visualization pipeline (explained in Section 5). Each function in the *BaseEffect* already has its default logic. The user only needs to extend this abstract class and override the function he wants to customize. The user can write any code inside the new function as long as the input and output follow the specified formats in the function requirement. The *Visualize* function will then call the functions in the given order to assemble the pipeline. For example, the user wants to plot a heat map of land surface temperature but he wants to inject a customized colorizing strategy: if a region is colder than 0 celsius degree, mark it as blue otherwise red. He only needs to override the Colorize step and put his own logic:

$$Color = \begin{cases} Blue & weight < 0 \\ Red & weight >= 0 \end{cases}$$

5 MAP VISUALIZATION PIPELINE

This section details GeoSparkViz map visualization pipeline and highlights the algorithms. GeoSparkViz encapsulates each step in the general map visualization pipeline and implements the corresponding logic in a Spark RDD Transformation. The user is not aware of the underlying details and only focuses on designing the analytics workload. Moreover, the user is able to override any visualization step to support new algorithms (e.g., new coloring rules).

5.1 Rasterize

This step takes as input a Spatial RDD which contains numerous spatial objects and the designated map pixel resolution then rasterizes the spatial objects to pixels in parallel. GeoSparkViz is able to accept various data sources such as persistent storage (e.g., AWS S3 or HDFS) or intermediate data from spatial query operators. Other visualization steps manipulate these pixels and eventually plot them out on map tile images. To generate a map image, spatial objects have to be rasterized to corresponding pixels. Like a point object, each pixel also possesses a coordinate/position on the screen coordinate system. Different from the spatial coordinate

system, the pixel's x coordinate and y coordinate have to be integers. The number of pixels on this map is determined by the given map resolution. Algorithm 1 depicts the detail of this operator.

$$X = \frac{longitude - (-180)}{180 - (-180)} * width = \frac{longitude + 180}{360} * width \quad (1)$$

$$Y = \frac{latitude - (-90)}{90 - (-90)} * height = \frac{latitude + 90}{180} * height \quad (2)$$

As shown in Algorithm1, this operator packages the rasterization into a single *Map* operation.GeoSparkViz adopts two rasterization rules: (1) outline-only (Figure 2a): only marks pixels covered by spatial objects' outlines; (2) filling area (Figure 2b): marks all pixels covered by spatial objects.

Outline-only For the outline-only rule, GeoSparkViz first decomposes the shape of spatial objects (points, polygons and line strings) into line segments. It is worth noting that a point is just a special line segment that has the same starting and ending vertexes. For vertexes, we can easily project it to a pixel using Equation 1 and 2. Width and height that define the map resolution are in the unit of pixels. After transforming the starting and ending vertexes of a line segment, we take the renowned Bresenham's line algorithm to decide the pixel trace which is approximately close to the line. Its basic idea is: from the starting vertex X position to the ending vertex X position, this line algorithm increases the X by 1 and takes the integer Y which is closest to the ideal (fractional) Y. Each pixel covered by a certain object is turned to a <Pixel, weight> pair in the <Pixel, weight> RDD, the result of the *Map* operation. The weight indicates an aggregation value at this pixel coordinate and it decides the color of this pixel. The initial value of the weight is explained below.

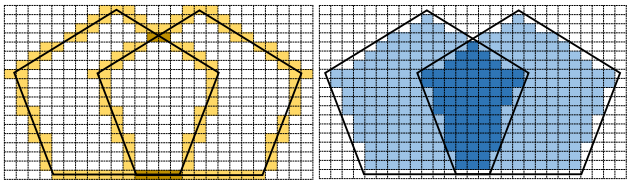
Filling area If the user chooses the filling area rasterization rule (only valid for polygons), we need to mark all pixels covered by polygons. For each polygon in the dataset, GeoSparkViz first transforms all vertexes to pixels, then locates pixels that fall inside the rasterized polygonal boundary. All covered pixels are added to the intermediate dataset in the format of <Pixel, weight> pairs.

Initial weight A spatial object can be rasterized to many pixels. It is also possible that the rasterize step produces many overlapped pixels (explained in the next section). The weight of each pixel is used to indicate the aggregation value. The initial weight in the <Pixel, weight> pair RDD has two possible choices depending on the visualization effects. If the user just wants to plot a map based on the spatial distribution of the dataset such as a Scatter plot/Heat map of geo-tagged tweets, GeoSparkViz will set the initial weight of all elements to a uniform value. If the user wants to plot a non-spatial attribute associated with spatial objects, GeoSparkViz will use the non-spatial attribute as the initial weight of each Pixel. For example, in a scatter plot of land surface temperatures, each geo-tagged thermal sensor has a temperature observation. GeoSparkViz will use each observed temperature as the initial weight.

Spark execution The Rasterize step runs a Spark *Map* operation on the input Spatial RDD. In Spark, it is called RDD Transformation with narrow dependency. It takes as input a single RDD (<spatial object> RDD) and transforms each partition of this RDD to a partition with pixels by using Algorithm 1. These new partitions become the <pixel, weight> RDD. During the execution period, Algorithm 1 happens on each partition of the input RDD at the same time. The output RDD holds the same RDD structure of the input and can be directly used in the next operation. This step has no data shuffle and will be pipelined together with many other no-shuffle operations by Spark DAG scheduler.

Algorithm 2: Step II: Pixel aggregate

```
Input: <Pixel, weight> pair RDD
Output: <Pixel, weight> pair RDD
Function MapPartition(a data partition P of the input dataset)
    Create an empty <Pixel, weight> HashMap HM;
    foreach <pixel, weight> pair in P do
        Find the current weight of this pixel in HM if it exists;
        // Aggregation stratgey: min, max, count,
           average, uniform
        Aggregate the weight;
    Create an empty <Pixel, weight> List L;
    foreach <pixel, weight> pair Px in P do
        nbPx's new weight =0;
        foreach neighbor pixel nbPx within image filter radius do
            nbPx's new weight = nbPx's new weight + Px's
              weight*Px's impact factor on nbPx;
        Store <nbPx, new weight> in L
    return L;
```



(a) Outline - Raster **(b) Filling area - Raster**

Figure 2: Rasterize vector objects to pixels

5.2 Pixel aggregate

The rasterize step may produce many overlapped pixels that are located at the same position on the screen coordinate system. This is because (1) some spatial objects overlap/intersect each other by nature (2) the resolution of the final map is low so that many objects overlap/intersect each other at this resolution. Since each position on the map should only be associated with one pixel and display the color of this pixel, GeoSparkViz should aggregate the weight of the overlapped pixels and determine the final weight of this pixel.

In order to gather overlapped and neighbor pixels to the same RDD partition to produce map tiles based on partitions, GeoSparkViz adopts a map tile data partitioner (see Section 6) to repartition the <pixel, weight> pair RDD from the rasterize step.

After repartitioning the pair RDD, the pixel aggregate step uses a *MapPartition* function to run a partition-wise algorithm (see Algorithm 2). For each partition in the <pixel, weight> pair RDD,

Algorithm 2 traverses all pixels in this partition and puts every pixel in a key-value hash map. The key is the pixel and the value is the current weight of this pixel. When traversing the pixels, if a pixel already exists in the hash map, Algorithm 2 will know this is an overlapped pixel and use an aggregation strategy to update the weight stored in the hash map.

GeoSparkViz provides several aggregation strategies to aggregate overlapped pixels: (1) Min: keep the minimum weight of pixels that locate at the same position (2) Max: similar to Min, but take the maximum weight (3) Average: similar to Min, but take the average weight (4) Count: instead of aggregating the weight, it only counts the number of pixels that overlap at the position (5) Uniform: it always assigns a fixed weight to the pixel's position no matter how many pixels overlap each other here. It is worth noting that, the first three strategies are used to visualize the non-spatial attribute (i.e., land surface temperature) of spatial objects on the map and the user will observe a colorful heat map. The last two strategies are used to visualize the spatial distribution of the objects in a Spatial RDD. The difference is that Count will show a heat map while Uniform prints a scatter plot which only has a single color.

Moreover, GeoSparkViz provides an optional function called image filter to add an extra visual effect on the final map tiles. An image filter, such as sharp, blur or diffusion, modifies the images by absorbing part of the available light and forcing longer exposure. For instance, images with Gaussian Blur show gradient colors. This is widely adopted in geospatial visual analytics. In digital photography, people normally use convolution matrixes to simulate the effects of filters. Its basic idea is, for a pixel in the image, add the colors from its local neighbors, weighted by the matrix, to this pixel. Each convolution matrix describes a (2*Radius+1) by (2*Radius+1) 2D array in which each individual element (Impact Factor) indicates how strong the center pixel's color impacts / is impacted by the corresponding neighbor pixel's new color. Figure 3 depicts the convolution matrix of Gaussian Blur (dark color means high impact and vice versa).

$7*10^{-6}$	$4*10^{-4}$	$2*10^{-3}$	$4*10^{-4}$	$7*10^{-6}$
$4*10^{-4}$	0.03	0.11	0.03	$4*10^{-4}$
$2*10^{-3}$	0.11	0.44	0.11	$2*10^{-3}$
$4*10^{-4}$	0.03	0.11	0.03	$4*10^{-4}$
$7*10^{-6}$	$4*10^{-4}$	$2*10^{-3}$	$4*10^{-4}$	$7*10^{-6}$

Figure 3: Gaussian Blur convolution matrix

After GeoSparkViz aggregates the weights of overlapped pixels, as an optional function, the image filter operator applies classic image filters to <pixel, weight> pair RDD so that the generated map tile images may obtain some special effects. Since the color of a pixel in GeoSparkViz is derived from the pixel's weight, this image filter works on pixels' weights directly. In other words, the weight of each pixel in the <pixel, weight> pair RDD is recomputed in accordance with a given convolution matrix in parallel. Eventually, each pixel's weight will be updated as follows:

$$Weight = \sum_{i=a\ pixel}^{Pixels\ in\ radius} i's\ IF * i's\ weight \tag{3}$$

As mentioned in the aggregation part, GeoSparkViz has repartitioned the <pixel, weight> pair RDD. Thus, neighbor pixels have been partitioned to the same partition. For a small portion of those pixels, the map tile data partitioner duplicates their neighbor pixels as buffer pixels in order to totally avoid the neighbor pixel search across the cluster.

In the repartitioned <pixel, weight> RDD, the algorithm loops over each pixel (denoted as cursor pixel) except buffer pixels and recalculates the pixel's weight according to its neighbor pixels' weights using Equation 3 (shown in Algorithm 2): for each pixel within its filter radius (decided by the convolution matrix), this algorithm sums up the products of this pixel's Impact Factor and weight (recall that the weights are stored in a hash map for constant retrieval time). Then it uses the sum as the new weight of the cursor pixel. All updated pixels' weights are stored in a new list.

Spark execution The Pixel aggregate step runs a Spark *MapPartition* operation on the input <pixel, weight> RDD. As a RDD Transformation with narrow dependency, it transforms each partition a single input RDD to a new partition by using Algorithm 2. This operation doesn't introduce any data shuffle across the cluster but GeoSparkViz needs to repartition the data before this step (see Section 6). This repartitioning yields a shuffle and Spark DAG scheduler then divides the Rasterize and Pixel aggregate to two separated stages.

5.3 Colorize

After finalizing the weight of each pixel, the colorize step decides a proper color for each pixel according to the weight. Colors become visible to the user when rendering the map. The relation between a color and a weight is defined by a mathematical equation (The weight needs to be normalized to [0, 255] RGB channel). GeoSparkViz plugs this equation into a *Map* function and executes it in parallel. The user can either provide his own equation or use GeoSparkViz default equation. Two equation types are common: (1) Linear equation (GeoSparkViz default). For example,

$$Color(R, G, B) = Color(255, 255, weight) \tag{4}$$

This equation will give the user a colorful image with white highlights. (2) Piecewise equation. For instance,

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

The user will see a three-color image by using this equation. After performing this operator, the weight in <Pixel, weight> pair RDD becomes an RGB color value.

Spark execution The Colorize step performs a Spark *Map* operation on the input <pixel, weight> RDD. Similar to the Rasterize and Pixel aggregate steps, the Colorize step transforms the input RDD to <pixel, color> RDD by running the algorithm above on each partition in parallel. This operation can be pipelined with the other no-shuffle operations as well.

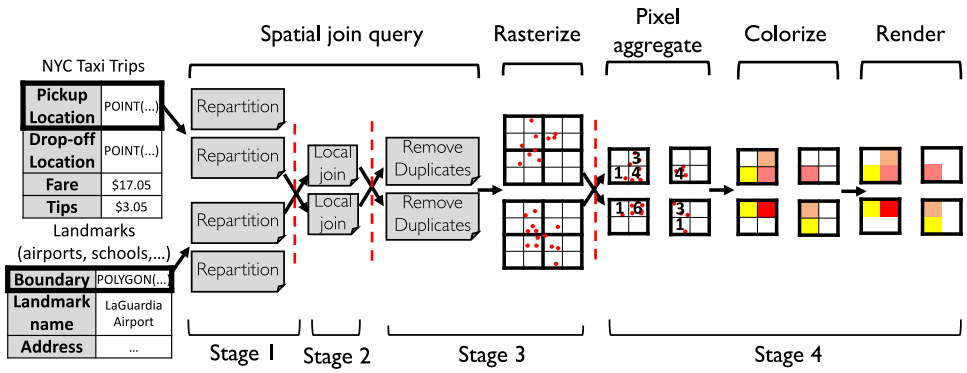


Figure 4: GeoSparkViz map visualization pipeline (in DAG)

Algorithm 3: Step IV: Render

Input: <Pixel, color> pair RDD
Output: <Tile ID, Image tile> pair RDD
1 **Function** MapPartition(*a partition P of the input dataset*)
2 Retrieve the tile ID ID;
3 Create a blank sub map tile $SubTile$ using the tile's resolution;
4 **foreach** *Pixel* Px *in* P **do**
5 Plot Px on $SubTile$ using Px's color;
6 **return** < ID, $SubTile$ >;
7 **Function** ReduceByKey($SubTile1$, $SubTile2$)
/* $SubTile1$ and $SubTile2$ have the same Tile ID */
8 Merge the two sub map tiles to one map tile $TILE$;
9 **return** < ID, $TILE$ >;

5.4 Render

This render step takes as input the <Pixel, color> RDD and generates a distributed map tile RDD. Algorithm 3 describes the details. Since we know pixels have been repartitioned by the map tile data partitioner according to their proximity, it is easy to generate a map tile on each data partition via a simple *Map* operation. The new map tile of each data partition has a map tile ID and some map tiles, called sub-tiles, from different data partitions may have the same tile ID because the data in these partitions is a portion of the same map tile (see Section 6). To recover those chopped map tiles, this step uses the tile ID as the key and executes a *ReduceByKey* on data partitions. This groups sub-tiles that have the same tile ID together then merges them into one map tile. Eventually, all chopped map tiles are recovered in parallel and the user can persist the generated distributed <Tile ID, map tile> RDD to external storage devices. This *ReduceByKey* only leads to a small scale data shuffle because only a few map tiles are chopped by the map tile data partitioner. After persisting the data, the map tiles can be stitched together according to their map tile ID if necessary.

Spark execution The Render step performs a Spark *MapPartition* operation and a *ReduceByKey* operation on the input <pixel, color> RDD. In the first operation, for each input RDD partition, it transforms all pixels in this partition to a map tile image. In other words, it materializes the map tiles shown in the third level of Figure 5a. In the second operation, it transforms the sub-tiles to their original map tiles shown in the first level of Figure 5a. From the DAG's perspective, both operations are RDD Transformation with narrow dependencies. However, the latter operation introduces a small data shuffle which is the stage boundary because it merges some partitions together.

5.5 Overlay

The user may also need to overlay multiple map layers such as transport map layer or county boundary layer on top of the base map image for analytics purposes. For instance, the user may want to overlay a taxi trip pick up point heat map with the area landmarks in New York City to understand why some regions attract much more taxis. This overlay step takes as input two <Tile ID, map tile> RDDs and overlays them one by one in the order specified by the user. This step first leverages *ReduceByKey* operation on two input RDDs (a front map RDD and a back map RDD) with the tile ID as the key so that two map tiles from two RDDs are shuffled together across the cluster. Then this step merges together the two tiles which all describe the same area of the overall map. During this process, the overlay step makes sure that the map tile from the front image dataset stands in the front. This step generates a new <Tile ID, map tile> dataset which can be persisted to external storage devices or continue to overlay with another <Tile ID, map tile> dataset.

Spark execution The Overlay step performs a Spark *ReduceByKey* operation on the two input <Tile ID, map tile> RDDs and produces a single <Tile ID, map tile> RDDs. In Spark, this is a typical RDD Transformation with wide dependency and leads to a tremendous data shuffle as well as the stage boundary.

5.6 Overall Directed Acyclic Graph (DAG)

As we mentioned in Section 2, the job scheduler in Spark relies on the Directed Acyclic Graph (DAG). Any RDDs as well as their Actions and Transformations are elements in a DAG, which eventually affects the Spark application's performance in terms of execution time, memory consumption and network pressure.

Figure 4 illustrates a common map visualization pipeline in GeoSparkViz: the user issues a spatial join query on taxi pickup

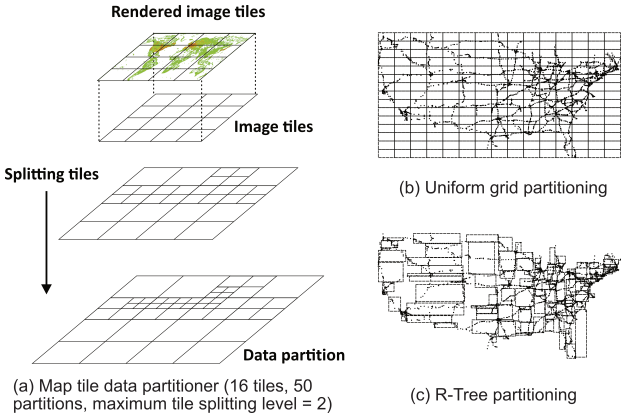


Figure 5: Spatial partitioning approaches

points and area landmarks' polygonal boundaries to select the taxi trips picked up in some area landmarks. The selected taxi trip pickup points are plotted on a heat map using Rasterize, Pixel aggregate, Colorize and Render steps. Each step shown in Figure 4 is a transformation which produces a new RDD. The produced RDD is a vertex in the DAG and each step is an edge. GeoSpark [26] runs a spatial join query in three transformations: repartition, local join (map), and duplicates removal. Two data shuffles occur after repartitioning and local joining, respectively. This leads to the boundaries of DAG Stage 1 and 2.

GeoSparkViz completes the map visualization pipeline using four transformations: Rasterize (*Map* operation), Pixel aggregate (*Map* operation), Colorize (*Map* operation), Render (*Map* operation). A data shuffle occurs after Rasterize step because of the map tile data partitioner (explained in Section 6). This leads to the boundary between Stage 3 and Stage 4. It is worth noting that, Stage 3 includes the transformations from GeoSpark Spatial Join Query and GeoSparkViz map visualization. This means these two transformations are pipelined together and run together such that there is no need to transfer the intermediate RDD across the cluster. Moreover, although Stage 4 contains several transformations, these transformations (all *Map* operations) can be pipelined together and executed faster.

6 MAP TILE-AWARE DATA PARTITIONER

Existing spatial partitioning approaches, such as R-Tree and Quad-Tree, exhibit good performance when executing spatial queries for the data preparation phase [24, 26]. However, these approaches do not consider the fact that the final output of the map visualization task will be eventually presented on a set of uniform map tile images (see Figure 5c). In other words, existing spatial partitioning techniques ignore the map tile boundaries and hence are not able to optimize the visualization operators that process pixels and produce map tiles. On the other hand, partitioning the workload based on the uniform map tiles demands less partitioning overhead. That also avoids the tedious process of recovering the map tiles to be visualized using existing map visualization tools. However, the uniform grid partitioning approach cannot handle the spatial data skewness and hence fails at balancing the workload among the cluster nodes (see Figure 5b).

The map tile data partitioner proposed by GeoSparkViz takes as input a set of pixels and finally returns the tile boundaries of determined data partitions. Each input pixels possesses a tile ID that indicates the uniform map tile where this pixel lies in. While enforcing the spatial proximity constraint, pixels assigned to the same partition should also belong to the same map tile image. In other words, all pixels in a data partition should have the same map tile ID. To determine the partitions, the partitioner employs a three-step algorithm:

Step I: Spatial Data Sampling: This step draws a random sample from the input spatial dataset and uses it as a representative set in order to diminish the data scale. Geometrical boundaries of every finalized data partition will be applied again to the entire dataset and make sure all pixels are assigned to partitions.

Step II: Tile-aware Data Partitioning: As shown in Figure 5, this step first splits the space into n^2 uniform map tiles (where n is the number of segments on longitude/latitude) which represent the initial geometrical boundaries for data partitions. Starting from the initial tiles, the partitioner repartitions each tile in a Top-down fashion. Similar to a Quad-Tree, the partitioning algorithm recursively splits a full tile quadrant space into four sub-tiles if a tile still contains too many pixels. As the splitting goes on, tile boundaries become more and more non-uniform, but load balanced. When the splitting stops (reach the maximum tile splitting level L, given by the user), the leaf level sub-tiles become the geometrical boundaries for the physical data partitions (see the last level in Figure 5). Eventually, this step builds a small quad-tree for each tile.

Step III: Physical Partitioning: This step passes the partition structure (Figure 5) stored in the master machine to all machines in the cluster. For every pixel, the map tile data partitioner first decides the uniform map tile that it belongs to. Then, this step searches the corresponding Quad-tree in a top-down fashion and stops at a sub-tile boundary that fully covers the pixels. If the search algorithm stops at a leaf-level sub-tile, the pixel is assigned to the corresponding partition. If the search stops at a non-leaf sub-tile (i.e., given a large polygon as input), the pixel is assigned to all leaf-level sub-tiles under this non-leaf sub-tile. Eventually, pixels or pixels that fall in the same leaf-level sub-tiles are physically located in the same cluster node.

Note that the three steps run sequentially and each step runs on a distributed pixel RDD in parallel.

7 USE CASE SCENARIO

This section uses three real use cases to illustrate the map visualization effects offered by GeoSparkViz, as follows:

USA Mainland Rails Scatter Plot GeoSparkViz is able to plot Spatial RDDs, PointRDD, RectangleRDD, PolygonRDD and LineStringRDD, which cover most spatial objects, to scatter plots. A scatter plot consists of four map visualization steps, rasterizing, pixel aggregate, colorizing and rendering. As described in Figure 6, GeoSparkViz scatter plot effect plots all rails within USA mainland to a rail network map with different zoom levels. This rail dataset published by US Census Bureau contains 181 million line strings and each line string object represents a complete real rail

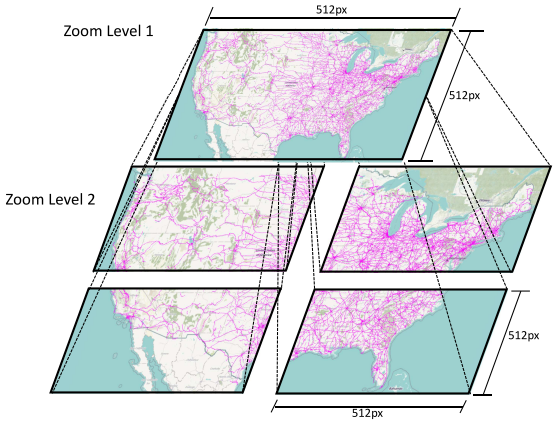


Figure 6: USA mainland rails scatter plot

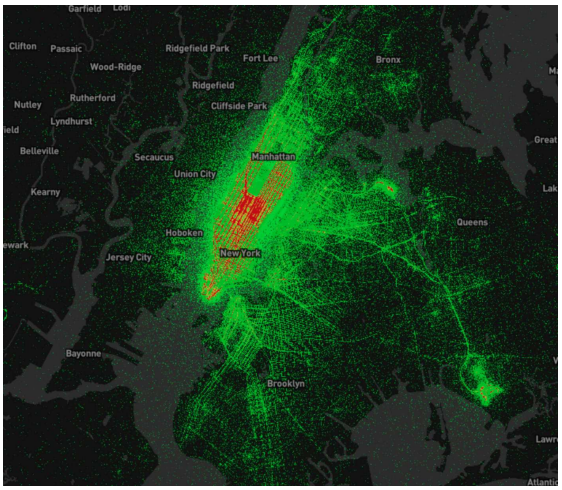
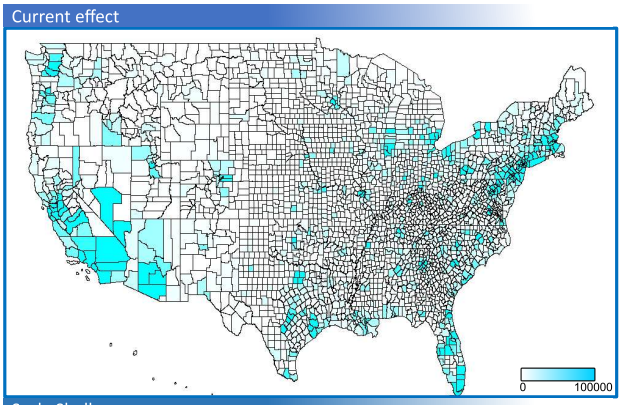


Figure 7: Taxi trips pickup point heat map

that consists of multiple line segments. GeoSparkViz loads this dataset into a LineStringRDD and easily customizes/generates this scatter plot. The user can ask GeoSparkViz to plot maps for several different zoom levels [17].

NYC Taxi Trips Heat Map GeoSparkViz heat map effect is produced by the same map visualization pipeline. Figure 7 shows a heat map of a 260 Gigabytes dataset of NYC Yellow Cab and Green Taxi trips, *abbr.* NYC Taxi [16]. The dataset contains detailed records of over 1.1 billion individual taxi trips in the city from January 2009 through June 2015. Each record includes pick-up and drop-off dates/times, pick-up and drop-off precise location coordinates, trip distances, itemized fares, payment method, and travel distance. GeoSparkViz represents the pickup and drop-off locations as a PointRDD and passes it to the heat map effect.

USA Tweets Distribution Choropleth Map A choropleth map provides an easy way to show the variability level of the metric within a region. One way to get the raw data of a choropleth map is: (1) Perform a spatial join query between regions and



Scala Shell

```
scala> :paste
val sc = new SparkContext ( ) ;
val tweets = new PointRDD ( sc , InputPath1 ,WKT) ;
val USAcounties = new PolygonRDD ( sc , InputPath2 , GeoJSON ) ;
val joinResult = JoinQuery.SpatialJoinQueryCountByKey(tweets,USAcounties);
val visOperator = new ChoroplethMap ( ResolutionX , ResolutionY , DatasetBoundary ) ;
visOperator . Visualzie( sc , joinResult , OutputPath ) ;
```

Figure 8: Create a choropleth map in Spark scala shell

POIs to get the result set with this schema: <Region, POIs list>. (2) Count the POIs in each region to get a result set with this schema: <Region, Count>. For a plotted geospatial region, the count of POIs within this region represents the object density of filling pixels. Therefore, the color in each plotted region is in proportion to the number of POIs in this region. The darker color means more POIs and vice versa.

GeoSparkViz choropleth map effect leverages the same map visualization pipeline in the heat map and uses an extra overlaying step to add region boundaries. This effect takes as input a spatial join query result, between any Spatial RDD (e.g., PointRDD) and a PolygonRDD, and directly generates the choropleth map. Figure 8 shows a choropleth map that reflects USA mainland Twitter tweets distribution in county unit. This Twitter dataset includes 10 million geo-tagged tweets that span the entire USA mainland. The filling color of a plotted USA county (region) is in proportion to the number of tweets in this county. GeoSparkViz treats USA county boundary outlines as a PolygonRDD and the Twitter dataset as a PointRDD, calls GeoSpark APIs to join these two Spatial RDD together and visualizes the spatial join query result to a map. The user can easily build this map from scratch by typing in the code (Figure 8) in Spark Scala shell and view the generated map in browser.

8 EXPERIMENTS

In this section, we conduct a comprehensive experimental evaluation of GeoSparkViz. We use six real spatial datasets in the experiments (see Table 1): (1) *TIGER Area Landmarks*: 130,000 polygonal boundaries of all area landmarks (i.e., hospitals, airports) collected by U.S. Census Bureau TIGER project. (2) *OpenStreetMap Postal Area Dataset*: 170,000 polygonal boundaries of postal areas (major cities) on the planet. Each polygon in this dataset is represented by 10 or more vertexes. (3) *TIGER Roads*: includes the shapes

Dataset	Records	Size	Description
TIGER Area Landmarks	130 thousand	140 MB	Polygonal Boundaries of area landmarks in US
OSM Postal Codes	171 thousand	1.4 GB	Polygonal Boundaries of postal areas (major cities) in the world
TIGER Roads	20 million	7.7 GB	Line string shapes of all roads in the world
TIGER Edges	73 million	23 GB	Line string shapes of all rivers, roads, rails in US
NYC Taxi	1.3 billion	219 GB	New York City taxi trip pickup points
OSM Point	1.7 billion	63 GB	All points in the world

Table 1: Test datasets

of 20 million roads in US. Each road is represented in the format of a line string which is internally composed of many connected line segments. (4) *TIGER Edges*: contains the shapes of 73 million edges (i.e., roads, rivers, rails) in US. Each edge shape is represented by a line string which has connected line segments. (5) *New York Taxi* [25]: contains 1.3 billion New York City taxi trip records from January 2009 through December 2016. Each record includes pick-up and drop-off dates/times, pick-up and drop-off location coordinates, trip distances, itemized fares, and payment method. But we only use the pickup point coordinates in the experiments. (6) *OpenStreetMap Point*: contains 1.7 billion spatial points on the planet, e.g., boundary vertices of attractions and road traces.

Cluster settings. We conduct the experiments on a cluster which has one master node and two 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, Apache Spark 2.11, HadoopVIz 2.4, and GeoSparkViz 1.0.

Compared approaches. In order to carefully investigate the map visualization performance, we compare the following approaches on generating scatter plot and heat map: (1) GeoSparkViz: This approach is the full GeoSparkViz system which fully employs map visualization pipeline and map tile data partitioner. (2) HadoopViz: this approach is SpatialHadoop and its visualization extension, namely HadoopViz. Intermediate data in this approach is transferred through disk. By default, we use OpenStreetMap standard zoom level 6 [17] as the default map visualization setting for all compared approaches: it requires 4096 map tiles (256*256 pixels per tile), 268 million pixels in total. The maximum tile splitting level in GeoSparkViz map tile data partitioner is 3, which means each map tile is split at most 3 times.

8.1 Impact of Spatial Partitioning

In this section, we compare four different spatial data partitioning approaches, GeoSparkViz map tile data partitioning, uniform grid partitioning, Quad-Tree spatial partitioning and R-Tree partitioning. All these partitioning methods are implemented in GeoSparkViz. The workload directly performs the scatter plot effects on the entire spatial datasets. For GeoSparkViz partitioner, we also vary the maximum tile splitting level parameter (i.e., Level 1, 2, and 3).

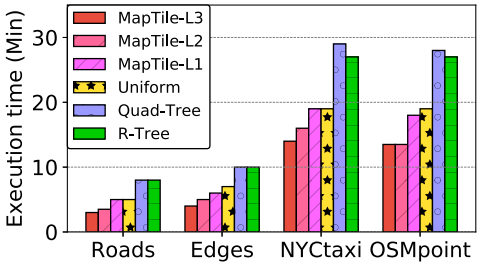


Figure 9: Performance of data partitioners

As shown in Figure 9, GeoSparkViz scatter plot with map tile data partitioner run 1.5X - 2X faster than the uniform grid partitioning method as expected. This is because the uniform grid partitioning approach does not balance the load among the cluster nodes. That degrades the performance more when the spatial dataset is very skewed. Moreover, a visualization task with larger GeoSparkViz max tile splitting level runs 15% faster than its variant with the lower splitting level. That happens since the map tile partitioner can produce more balanced data partitions when it keeps splitting tiles (until reaching the minimum data partition boundary). Furthermore, the Quad-Tree and R-Tree partitioning approaches are 50% - 70% slower than other partitioning methods because such partitioning methods do not consider the map tile sizes and hence the system has to add an extra step to recover the map tiles before rendering. The map tile recovery step assigns each pixel a TileID and groups pixels by their TileID. This step leads to an additional data shuffling operations to group pixels.

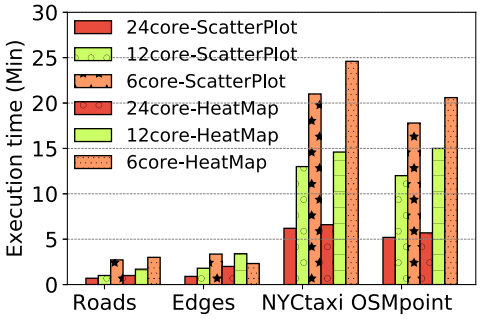


Figure 10: Scatter plot and heat map on different cores

8.2 Effect of Cluster Size

To demonstrate the scalability of GeoSparkViz, we evaluate its performance on different cluster settings. In the first part of the experiments, we vary the total number of CPU cores in a 2-node cluster to be 6 cores, 12 cores and 24 cores. In addition, we only change the cores registered in Apache Spark without changing the number of workers such that each CPU core setting still runs on the same 2-node cluster. In the second part of the experiments, we vary the number of nodes in the Spark cluster to be 1 node, 2 nodes,

3 nodes and 4 nodes. Every machine utilizes its full computation power, which consists of 12 CPU cores so the total CPU cores in the corresponding clusters are 12 cores, 24 cores, 36 cores and 48 cores. Note that we could not run experiments for the NYCtaxi and OSMpoint datasets on a single node due to their large sizes compared to the memory size on a single node.

As depicted in Figure 10, the run time cost of GeoSparkViz increases as we decrease the number of cores in the cluster. That makes sense due to the fact that a larger cluster (more CPU cores) can process more tasks in parallel. On the other hand, we run all experiments on four main datasets, Roads, Edges, NYCtaxi and OSMpoint (see Figure 10. The latter two datasets have over 1 billion

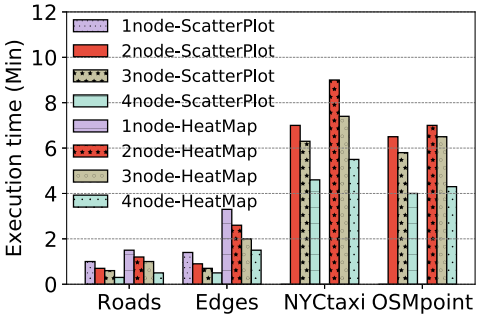


Figure 11: Scatter plot and heat map on different machines

records, which is about 30 times larger than Edges and 70 times larger than Roads. The experiments also show that the time spent on relatviely large-scale datasets (NYCtaxi and OSMpoint) is only close to an order of magnitude (instead of 30 -70 times) higher than that on smaller datasets. That makes sense because although the small datasets have much fewer records, their internal objects are line strings, which contain multiple line segments. Processing line strings including checking spatial query predicate and rasterization take more time due to their complex geometrical shapes.

As shown in Figure 11, the execution time decreases with the growth of the number of machines in the cluster. We observe the same pattern when running the experiments for both scatter plot and heat map visualization effects. However, the execution time exhibits a sub-linear correlation with the number of machines. For example, the execution time on the 4-node cluster is 1.5 times (not 2 times) less than that on the 2-node cluster. This is because adding more machines to the Spark cluster does not just simply increase the computation power but also introduces a side effect: more data shuffle read across the cluster machines. More data shuffled across the network may significantly slow down the performance. For example, based on the experiments, the NYCtaxi scatter plot generation on the 4-node cluster leads to approximately 7.2 GB data sent across the network. The same visualization effect running on the 2-node/3-node cluster only leads to 4.7 GB and 5.9 GB transferred across the network, respectively

8.3 Impact of Map Zoom Level

Figure 12 studies the impact of different map zoom levels on GeoSparkViz. We use OpenStreetMap standard zoom level as our criteria. Higher zoom level means that GeoSparkViz produces more map tiles. We use 256*256 pixel resolution for each map tile and vary the zoom level to be L2, L4 and L6. OSM zoom level 2 has 64 tiles, 1 million pixels; level 4 stands for 256 tiles, 16 million pixels; level 6 demands 4096 tiles, 256 million pixels. We produce scatter plots map visualization on four datasets.

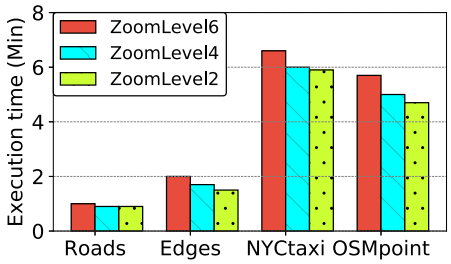


Figure 12: Impact of map zoom levels

As shown in Figure 12, the higher the zoom level, the more time GeoSparkViz takes to execute the map visualization. That makes sense because, with smaller zoom levels, GeoSparkViz only generates low-resolution maps. In that case, the rasterization, pixel aggregate, colorizing and rendering operators process fewer pixels.

8.4 Comparing GeoSparkViz and HadoopViz

Impact of the visualization effect. we first study the performance of the map visualization workload. We run both the Heatmap and Scatterplot visualization effects on GeoSparkViz and HadoopViz approaches. The experiment also involves four datasets with different scales.

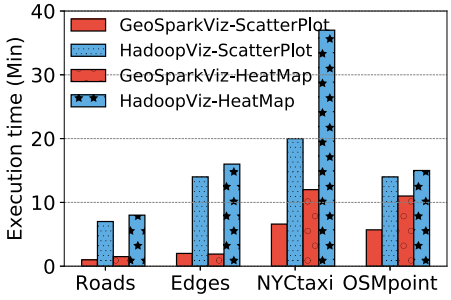


Figure 13: Performance of scatter plot and heat map

As shown in Figure 13, GeoSparkViz is 3-4 times faster than HadoopViz for generating scatter plot visualization. The result makes sense because: (1) The map tile data partitioner in GeoSparkViz is more load-balanced than the uniform partitioning adopted by HadoopViz. The uniform partitioning repartitions the space to uniform grids and does not take into account data skewness. (2) In contrast to GeoSparkViz, HadoopViz reads/writes intermediate data on disk. On small datasets like Roads and Edges, GeoSparkViz is around 8 times faster than HadoopViz because GeoSparkViz can process all intermediate data in memory.

When generating the heat map visualization effect (see Figure 13), GeoSparkViz is also around 3-4 times faster than Hadoop. Generating the heat map visualization effect takes 20% more time than generating a scatter plot because generating heat map effect needs to apply an image processing filter to colors in the Render operator and this leads to more local iterations on each partition.

Impact of the visualization window size. This section also studies the impact of varying the visualization window size. We use the NYCtaxi dataset but vary the spatial range window size to cover different area sizes in NYC. The smallest query window area is the center of New York City region. We keep multiplying this area by 4 and generate another three query windows, 4*Area, 16*Area, 64*Area. The biggest query window is the actual size of the entire New York City region.

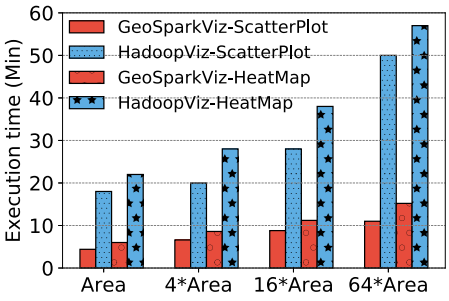


Figure 14: Impact of data scale

As shown in Figure 14, the execution time of the compared approaches increases with the growth of the data size. However, HadoopViz costs more time on larger query area while the time cost of GeoSparkViz increases slowly. On the largest query area, 64*Area, GeoSparkViz is around 5 times faster than HadoopViz. This makes sense because GeoSparkViz pipelines multiple transformations, Rasterize, Pixel aggregate, Colorize and Render, together and all intermediate data is transferred through memory. In addition, GeoSparkViz's map tile data partitioner is more load balanced than the data partitioner in HadoopViz which is the uniform grid data partitioner.

9 CONCLUSION

In this paper, we presented GeoSparkViz, a map visualization system for massive-scale spatial data. The proposed approach pushes the spatial map visualization functionality inside the core engine of Apache Spark. The system comes with a set of optimized out-of-the-box implementation of popular map visualization effects (e.g., Scatter Plat, Heat Map). That way, GeoSparkViz provides the data scientist a holistic system that allows her to load, prepare, integrate, visualize big spatial data in Spark. The experiments show that GeoSparkViz can generate a high-resolution (i.e., Gigapixel image) six zoom levels Heatmap of 1.7 billion Open Street Maps objects and 1.3 billion NYC taxi trips in ≈4 and 5 minutes on a four-node commodity cluster, respectively. In the future, we also plan to integrate GeoSparkViz with recently developed declarative visualization languages (e.g., [10, 20]) as well as sampling-based systems (e.g., ScalaR [4], RS-Tree [23]) to support interactive map visualization operations (e.g., Zoom-In/out).

10 ACKNOWLEDGMENT

This work is supported in part by the National Science Foundation (NSF) under Grant 1654861, the Salt River Project Agricultural Improvement and Power District (SRP), and the DOD-ARMY Training and Doctrine Command (TRADOC).

REFERENCES

[1] Aji, A., Wang, F., Vo, H., Lee, R., Liu, Q., Zhang, X., and Saltz, J. H. Hadoop-gis: A high performance spatial data warehousing system over mapreduce. *PVLDB 6*, 11 (2013), 1009–1020.
[2] Ashworth, M. Information technology – database languages – sql multimedia and application packages – part 3: Spatial. Standard, International Organization for Standardization, Geneva, Switzerland, 2016.
[3] Baig, F., Mehrotra, M., Vo, H., Wang, F., Saltz, J. H., and Kurç, T. M. Sparkgis: Efficient comparison and evaluation of algorithm results in tissue image analysis studies. In *Workshop on Biomedical Data Management and Graph Online Querying - VLDB* (2015), pp. 134–146.
[4] Battle, L., Stonebraker, M., and Chang, R. Dynamic reduction of query result sets for interactive visualizaton. In *Big Data* (2013), pp. 1–8.
[5] Cruz, I. F., Ganesh, V. R., Caletti, C., and Reddy, P. Giva: a semantic framework for geospatial and temporal data integration, visualization, and analytics. In *SIGSPATIAL* (2013), pp. 544–547.
[6] Eldawy, A., and Mokbel, M. F. Spatialhadoop: A mapreduce framework for spatial data. In *ICDE* (2015), pp. 1352–1363.
[7] Eldawy, A., Mokbel, M. F., Alharthi, S., Alzaidy, A., Tarek, K., and Ghani, S. Shahed: A mapreduce-based system for querying and visualizing spatio-temporal satellite data. In *ICDE* (2015), pp. 1585–1596.
[8] Eldawy, A., Mokbel, M. F., and Jonathan, C. Hadoopviz: A mapreduce framework for extensible visualization of big spatial data. In *ICDE* (2016), pp. 601–612.
[9] Elmqvist, N., Dragicevic, P., and Fekete, J.-D. Rolling the dice: Multidimensional visual exploration using scatterplot matrix navigation. *TVCG 14*, 6 (2008), 1539–1148.
[10] Heer, J., and Bostock, M. Declarative language design for interactive visualization. *TVCG 16*, 6 (2010), 1149–1156.
[11] Hughes, J. N., Annex, A., Eichelberger, C. N., Fox, A., Hulbert, A., and Ronquest, M. Geomesa: a distributed architecture for spatio-temporal fusion. In *SPIE Defense+ Security* (2015), pp. 94730F–94730F.
[12] Kini, A., and Emanuele, R. Geotrellis: Adding geospatial capabilities to spark, 2014.
[13] Lu, J., and Guting, R. H. Parallel Secondo: Boosting Database Engines with Hadoop. In *ICPADS* (2012), pp. 738 –743.
[14] Maciejewski, R., Rudolph, S., Hafen, R., Abusalah, A., Yakout, M., Ouzzani, M., Cleveland, W. S., Grannis, S. J., and Ebert, D. S. A visual analytics approach to understanding spatiotemporal hotspots. *TVCG 16*, 2 (2010), 205–220.
[15] Mostak, T. An overview of mapd (massively parallel database), 2013.
[16] NYC-TaxiTrip, 2009. New York City Taxi and Limousine Commission http://www.nyc.gov/html/tlc/html/about/trip_record_data.html.
[17] OpenStreetMap. Open Street Map wiki page: zoom levels and map tiles. http://wiki.openstreetmap.org/wiki/Zoom_levels, 2006.
[18] Sarwat, M. Interactive and scalable exploration of big spatial data - A data management perspective. In *MDM* (2015), pp. 263–270.
[19] Sarwat, M., and Nandi, A. On Designing GeoViz-Aware Database Systems âĂŞ Challenges and Opportunities. In *SSTD* (2017).
[20] Satyanarayan, A., Moritz, D., Wongsuphasawat, K., and Heer, J. Vega-lite: A grammar of interactive graphics. *TVCG 23*, 1 (2017), 341–350.
[21] Spark, A. Apache Spark homepage. http://spark.apache.org/, 2018.
[22] Tang, M., Yu, Y., Malluhi, Q. M., Ouzzani, M., and Aref, W. G. Locationspark: A distributed in-memory data management system for big spatial data. *PVLDB 9*, 13 (2016), 1565–1568.
[23] Wang, L., Christensen, R., Li, F., and Yi, K. Spatial online sampling and aggregation. In *PVLDB* (2015), vol. 9, pp. 84–95.
[24] Xie, D., Li, F., Yao, B., Li, G., Zhou, L., and Guo, M. Simba: Efficient in-memory spatial analytics. In *SIGMOD* (2016), pp. 1071–1085.
[25] Yu, J., and Sarwat, M. Indexing the pickup and drop-off locations of NYC taxi trips in postgresql - lessons from the road. In *SSTD* (2017), pp. 145–162.
[26] Yu, J., Wu, J., and Sarwat, M. Geospark: a cluster computing framework for processing large-scale spatial data. In *SIGSPATIAL* (2015), pp. 70:1–70:4.
[27] Yu, J., Wu, J., and Sarwat, M. A demonstration of geospark: A cluster computing framework for processing big spatial data. In *ICDE* (2016), pp. 1410–1413.