The VLDB Journal (2021) 30:237–258
https://doi.org/10.1007/s00778-020-00645-2

REGULAR PAPER

GeoSparkViz: a cluster computing system for visualizing massive-scale geospatial data

Jia Yu[1] · Mohamed Sarwat[2]

Received: 10 December 2019 / Revised: 24 June 2020 / Accepted: 27 September 2020 / Published online: 7 January 2021

Abstract
In the last decade, geospatial data which is extracted from GPS traces and satellites image has become ubiquitous. GeoVisual analytics, *abbr. GeoViz*, is the science of analytical reasoning assisted by geospatial map interfaces. GeoViz involves two phases: (1) spatial data processing: that loads spatial data and executes spatial queries to return the set of spatial objects to be visualized. (2) Map visualization: that applies a map visualization effect, e.g., Heatmap, on the spatial objects produced in the first phase. Existing GeoViz system architectures decouple these two phases, which lose the opportunity to co-optimize the data processing and map visualization phases in the same cluster. To remedy this, the paper presents GeoSparkViz, a full-fledged system that allows the user to load, process, integrate and execute GeoViz tasks on spatial data at scale. GeoSparkViz extends a state-of-the-art distributed data management system to provide native support for general geospatial map visualization. The system encapsulates the main steps of the map visualization process, e.g., pixelize spatial objects, pixel aggregation, and map tile rendering into a set of massively parallelized map building operators. This allows the system to co-optimize the spatial query operators and map building operators side by side. GeoSparkViz is also equipped with a GeoViz-aware spatial partitioning operator that achieves load balancing for GeoViz workloads among all nodes in the cluster. Experiments based on an implementation in Spark show that GeoSparkViz achieves up to an order of magnitude less data-to-visualization time than its counterparts when running visual analytics tasks over large-scale spatial data extracted from the NYC taxi dataset and OpenStreetMaps.

Keywords Distributed computation · Geospatial visualization · Big spatial data

1 Introduction

In the last decade, geospatial data which is extracted from GPS traces and satellite images has become ubiquitous. For example, NASA has released over 32 PB earth observation data captured by satellites and aircrafts [11]. By 2020, the number of GPS-installed smart phones worldwide will exceed 1.34 billion [29]. Mobile applications such as ride-sharing, geo-tagged social media and location-aware recommendation services generate massive-scale geospatial data. GeoVisual analytics, *abbr. GeoViz*, is the science of analytical reasoning assisted by geospatial map interfaces. For example, a GeoViz scatter plot of the New York City taxi trips visualizes the hot taxi pick-up locations (see Fig. 1). Also, a politician may utilize a GeoViz choropleth map to visualize the Twitter sentiment of each presidential candidate in each US county (see Fig. 1). GeoViz on big geospatial data involves two main phases:

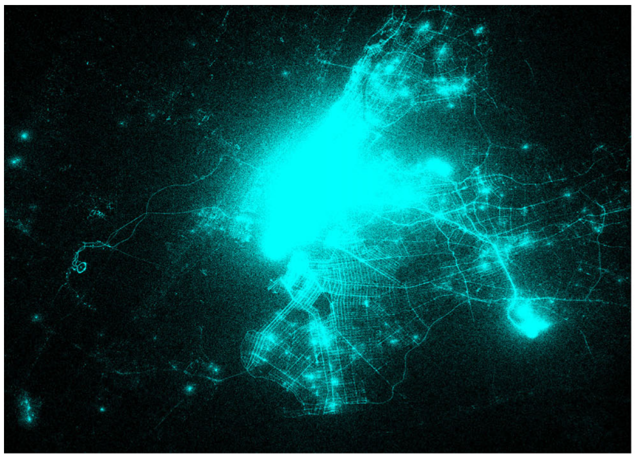
- **Phase I: Spatial Data Processing** In this phase, the system first loads the designated spatial data from distributed data systems. Based on the application, the system may then need to perform a data processing operation (e.g., spatial range query, spatial join) on the loaded spatial data to return the set of spatial objects to be visualized.
- **Phase II: Map Visualization** In this phase, the system applies the map visualization effect, e.g., Heatmap, on the spatial objects produced in Phase I. The system pixelizes

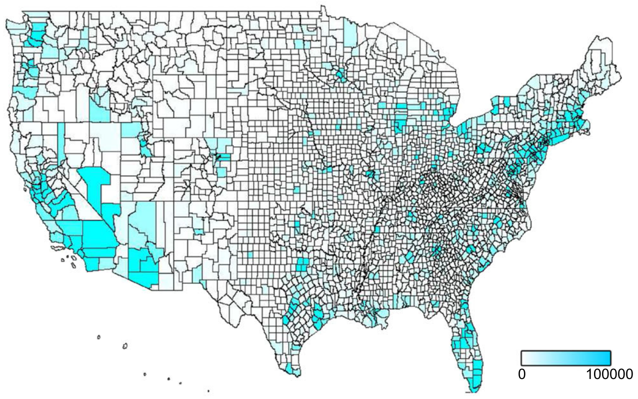
✉ Jia Yu
jia.yu1@wsu.edu

Mohamed Sarwat
msarwat@asu.edu

1 School of Electrical Engineering and Computer Science, Washington State University, 355 NE Spokane Street, Pullman, WA 99163, USA

2 School of Computing, Informatics, and Decision Systems Engineering, Arizona State University, 699 S Mill Avenue, Tempe, AZ 85281, USA


(a) NYC taxi trip scatter plot



(b) Tweets Choropleth map

Fig. 1 Geospatial visual analytics (GeoViz) examples

the spatial objects, calculates the intensity of each pixel and renders a geospatial map tile(s).

Existing GeoViz system architectures decouple the two phases and demand substantial overhead to connect the data management system to the map visualization tool because each phase treats the other as a black box (see Fig. 2). The user may spend additional coding effort to bridge them through a server program (with JDBC and ODBC) or RESTful APIs [19]. The intermediate data between the two phases may be very big and hence has to be dumped to disk files and re-loaded by visualization tools. For instance, the user may use a distributed spatial data system GeoSpark [40] to get a spatial query result which costs tens of GBs. Then he or she collects the on-disk result and asks Google Maps to create a heat map. On the computer cluster used in Sect. 8, this process took more than three hours and was still not over.

Such decoupled solutions are not adequate for large-scale GeoViz tasks due to the following challenges:

- **Challenge I: Scalable and fast map visualization** The massive scale of available spatial data hinders traditional spatial visualization techniques from portraying numerous objects on a single machine which has limited resources. Moreover, the users usually do not have much patience and would like to see map visualization as soon as possible regardless of the data scale.
- **Challenge II: High-resolution maps** A user would like to explore big spatial data on multi-scale maps (from the bird eye view to his point-of-interest) after processing the data [34]. Producing detailed spatial visualization (e.g., a high zoom level in Google Maps) requires extremely high-resolution maps. Traditional single machine solutions take tremendous processing time to serve users with qualified images.

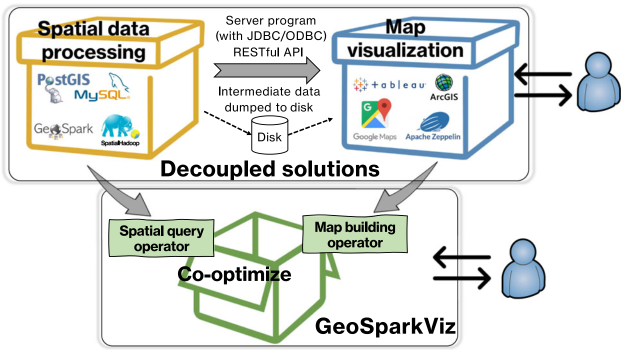


Fig. 2 Decoupled solution (each phase treats the other as a black box) versus GeoSparkViz (co-optimize two phases)

To remedy this, a hybrid system architecture that combines the spatial data processing and map visualization phases in the same distributed cluster can achieve more scalability. There has been a large body of research on scaling the geospatial map visualization phase using a massively parallelized cluster computing system. For example, SHAHED [9] and HadoopViz [10] produce high-resolution maps using MapReduce clusters. They can take as input the data generated by state-of-the-art large-scale spatial data management systems [3,8,36,40] installed in the same cluster. However, such a hybrid architecture that executes the two GeoViz phases sequentially loses the opportunity to co-optimize the data processing and map visualization phases. And the intermediate data still has to be dumped to disk. Hence, such an approach still exhibits limited scalability and high data-to-visualization time when executing a GeoViz task over massive-scale spatial data.

Table 1 Compare representative GeoViz systems (the full comparison is given in Sect. 2)

	Declarative GeoViz	Distributed	Accurate	Spatial Data Process.	Map Viz.
Traditional map viz. libraries (ArcGIS, Google Maps)	✗	✗	✓	✗	✓
Sampling/aggregation [4,12,24,33,38])	✗	✗	✗	✓	✓
Scalable spatial data systems [1,3,8,14,20,32,36,40]	✗	✓	✓	✓	✗
Scalable map visualization systems [9,10]	✗	✓	✓	✗	✓
Declarative viz libraries [27,28]	✓	✗	✓	✗	✓
GEOSPARKVIZ	✓	✓	✓	✓	✓

Recent research proposed to incorporate the visualization-awareness in a database system [15,35]. Such an approach allows users to define visualization workflows using SQL. However, existing Viz-aware data systems either do not provide native support for spatial data or do not address the problem of co-optimizing the data processing and visualization phases in a distributed and parallel data system.

The paper presents GEOSPARKVIZ [39], a cluster computing system for visualizing massive-scale geospatial data. It extends GEOSPARK [40], an in-memory data system for processing geospatial data at scale, to perform the spatial data processing and map visualization phases in the same cluster. Two benefits come as a byproduct of running the two phases of the GeoViz process in the same cluster: (1) It provides the user with a holistic system to perform the tasks in one place. The user no longer needs to write separate code for both phases and connect them via an additional program. (2) The intermediate data reside in the cluster's memory instead of dumped to disk. This reduces the overhead of data loading.

Most importantly, GEOSPARKVIZ has the following main contributions:

- GEOSPARKVIZ encapsulates the main steps of the geospatial map visualization process into a set of massively parallelized map building operators which allow user to declaratively generate a variety of map visualization effects, e.g., scatter plot, heat map, using GeoViz SQL. Moreover, given GeoViz SQL queries from users, GEOSPARKVIZ can parse the queries, run optimization to find efficient query plans which co-optimizes the spatial query operators (e.g., spatial join) and map building operators (e.g., pixelize) and execute the plans in the cluster.
- GEOSPARKVIZ also proposes a GeoViz-aware spatial partitioning operator that achieves load balancing for GeoViz workloads among all nodes in the cluster. It partitions the data only once for both data processing and map visualization phases and hence reduces the amount of data shuffled across the cluster nodes.
- A full-fledged prototype of GEOSPARKVIZ is implemented in GeoSpark (a Spark-based data system). The paper presents extensive experimental evaluation that studies the performance of GEOSPARKVIZ with state-of-the-art GeoViz system architectures. The experiments show that GEOSPARKVIZ can achieve close to one order of magnitude less data-to-visualization time than its counterparts for a variety of GeoViz workloads over several cluster settings.

The rest of the paper is organized as follows: Sect. 2 presents the related work. An overview of GEOSPARKVIZ is given in Sect. 3. Section 4 describes the declarative GeoViz language in GEOSPARKVIZ. The main map building operators are described in Sect. 5. Section 6 explains the GeoViz-Aware partitioning. Section 7 describes the GeoViz query execution plan and GeoViz optimization strategies. A comprehensive experimental evaluation is given in Sect. 8. Finally, Sect. 9 concludes the paper. Appendix A includes additional functions, types and examples.

2 Related work

Table 1 lists the representative work in visual analytics, map visualization and spatial data management.

2.1 Sampling/aggregation-based techniques

Existing sampling techniques only visualize a fraction of the original dataset to achieve scalability interactivity. Random sampling and stratified sampling are two widely used approaches that only pick the most representative spatial objects. Nano Cube[17] and Hashed Cube[6] maintain compressed aggregates of the spatial data to scale the GeoViz process. RS-Tree [33] augments the R-tree data structure to retrieve just a sample of the spatial data that lie within the query range. ScalaR [4] and Tabula [38] store precomputed multiple resolution aggregates/samples of the data using a database system. VAS [24] and POIsam [12] propose online sampling techniques to yield more accurate visualization results for geospatial data. CloudBerry system [31] pre-fetches and caches subsets of query results to support interactive visualization on large geospatial data but it does not completely address the scalability issue. Several research projects, such as IncVisage [25], Vizdom [5] and imMens [18], propose to incorporate sampling techniques in database systems to bolster interactive general-purpose data visualization. Also, existing work comes up with several features (e.g., zoom consistency and spatial proximity [15,21,26]), which enable selecting a representative subset of the spatial objects (to be visualized on the map) from a big spatial dataset. For example, the zoom consistency feature [15,26] ensures that spatial objects appear on a higher zoom level should still appear on a lower zoom level of the map. The visibility constraint requires that the number of selected objects should not be larger than a certain number. The proximity constraint states that selected objects should not be too close to each other. Sampling/aggregation-based GeoViz approaches, though scale to big data, only generate geospatial map visualization on a subset of the data. They may reveal the global data distribution but are not capable of showing detailed local maps when users zoom to their points-of-interest.

Most importantly, GeoSparkViz performs accurate multi-scale visualization that generates the map tiles on the entire dataset instead of a subset. Our system and the existing sampling/aggregation methods are complementary to each other. Many sampling approaches such as RS-Tree [33], VAS [24] and POIsam [12] can work in conjunction with GeoSparkViz to further improve the scalability and interactivity on large-scale map visualization.

2.2 Scalable spatial data management 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 [20], Hadoop-GIS [1] and SpatialHadoop [8] extend the Hadoop ecosystem to support global and local spatial indexing and to achieve efficient query processing over large-scale spatial data. Although the Hadoop [13]-based spatial data systems can scale out the workload, they show poor performance due to the disk-based computation model and inefficient job scheduling in Hadoop. Apache Spark [30] is an in-memory cluster computing system that outperforms Hadoop. It provides a novel data abstraction called Resilient Distributed Datasets (RDDs) that are collections of objects partitioned across a cluster of machines. GeoMesa [14], SIMBA [36], LocationSpark [32], SparkGIS [3] and GeoSpark [40] extend Apache Spark to support SQL applications on Geospatial data types. Although the aforementioned systems can scale the spatial data processing phase on a cluster, they do not support the map visualization.

2.3 Scalable map visualization systems

There is a large body of research that builds upon parallel/distributed system approaches to scale the visualization workflow [9,10,16,22]. SHAHED [9] and HadoopViz [10] 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. Geotrellis [16] extend Apache Spark to manipulate spatial objects and pixels but they still cannot co-optimize classic query processing operations (necessary for data processing) and map visualization. MapD [22] leverages the Graphic Processing Unit (GPU) to parallelize and hence speed up query and pixel manipulation in visualization but it is still limited to a single machine.

2.4 Declarative visualization

Precise visual encoding and grammar logic are necessary components in defining data analytics graphics. Vega-Lite [27] and Reactive Vega [28] provide a visualization algebra to define customized visualization effects. However, they only optimize the visualization pipeline and cannot co-optimize database query operators with the visualization operators. CVL [15] proposes a cartographic visualization language, which generates cartography with a set of given constraints. Ermac [35] proposes a vision for a Data Visualization Management System (DVMS) that can translate visualization workflows into SQL. Nonetheless, Ermac is not designed to handle geospatial data. Similar to CVL and Ermac, GeoSparkViz supports declarative visualization. However, none of the above tackle the problem of executing GeoViz tasks in a distributed and parallelized data system. This may preclude the overall scalability of the GeoViz application.

3 GeoSparkViz overview

Figure 3 gives an overview of GeoSparkViz. The system assumes that the spatial dataset is partitioned and distributed among the cluster nodes. The user interacts with GeoSparkViz using a declarative SQL-like GeoViz language. The system processes the GeoViz task and returns the final map tiles to the user. To achieve this, GeoSparkViz consists of the following components:

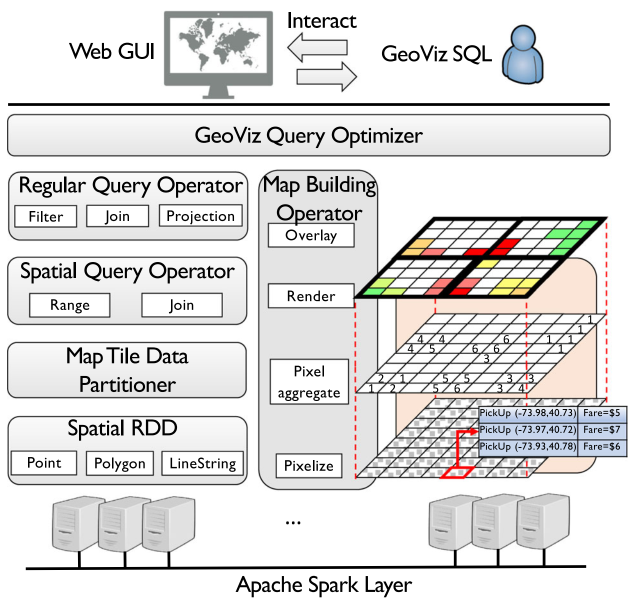


Fig. 3 GeoSparkViz overview

3.1 Spatial RDD and spatial query operators

GeoSparkViz builds upon GeoSpark [40] which extends Apache Spark to process geospatial data at scale. GeoSpark Spatial RDD is the input data source of GeoSparkViz and can be loaded from various spatial data files including Shapefiles, GeoJSON and Well-Known-Text. Spatial RDDs are in-memory distributed datasets that intuitively extend traditional RDDs for accommodating spatial objects. The data in Spatial RDD could be spatial partitioned (i.e., R-Tree) and serialized to a compact in-memory format. GeoSpark also provides efficient algorithms to run spatial range query and join query in the cluster. **Visualization (map building) Operators:** GeoSparkViz breaks down the map visualization pipeline into a sequence of query operators (namely, Pixelize, Pixel Aggregate and Render). The system parallelizes the execution of each operator among the cluster nodes. Furthermore, GeoSparkViz seamlessly integrates the map visualization operators with classic database query operators (e.g., range, join) used for the spatial data processing phase. Furthermore, GeoSparkViz exposes the map building operators to the user through the declarative GeoViz language. The user can easily declare a new map visualization effect in an SQL statement. For instance, the user can define new coloring rules and pixel aggregation rules.

3.2 GeoViz-aware spatial partitioner

GeoSparkViz employs a partitioner operator that fragments a given pixel dataset across the cluster. The partitioner accommodates map visual constraints and also balances the load among the cluster nodes when processing skewed geospatial data. On the one hand, it makes sure that each data partition contains roughly the same number of pixels to avoid “stragglers” (i.e., a machine that takes many more inputs than others and hence performs slowly). On the other hand, pixels are partitioned by their spatial proximity (i.e., pixels in the same data partition are from the same map tile). This way, the generated tile images can be easily stitched back together.

3.3 GeoViz optimizer

The optimizer takes as input one or more GeoViz queries, runs heuristic and cost-based optimization on the query execution plan, and figures out an efficient plan that co-optimizes the map building operators and spatial query operators (i.e., used for data processing). It introduces several heuristic optimization strategies: 1. single-query optimization (1) merge partitioner (2) push operators up/down; 2. cross-query optimization: merge repeated operators. For instance, if the GeoViz task eventually plots the results of multiple spatial range queries on the same dataset, the optimizer will decide to first map spatial objects to pixels and then execute spatial queries on pixels directly. This way, GeoSparkViz avoids redundant pixelization operators.

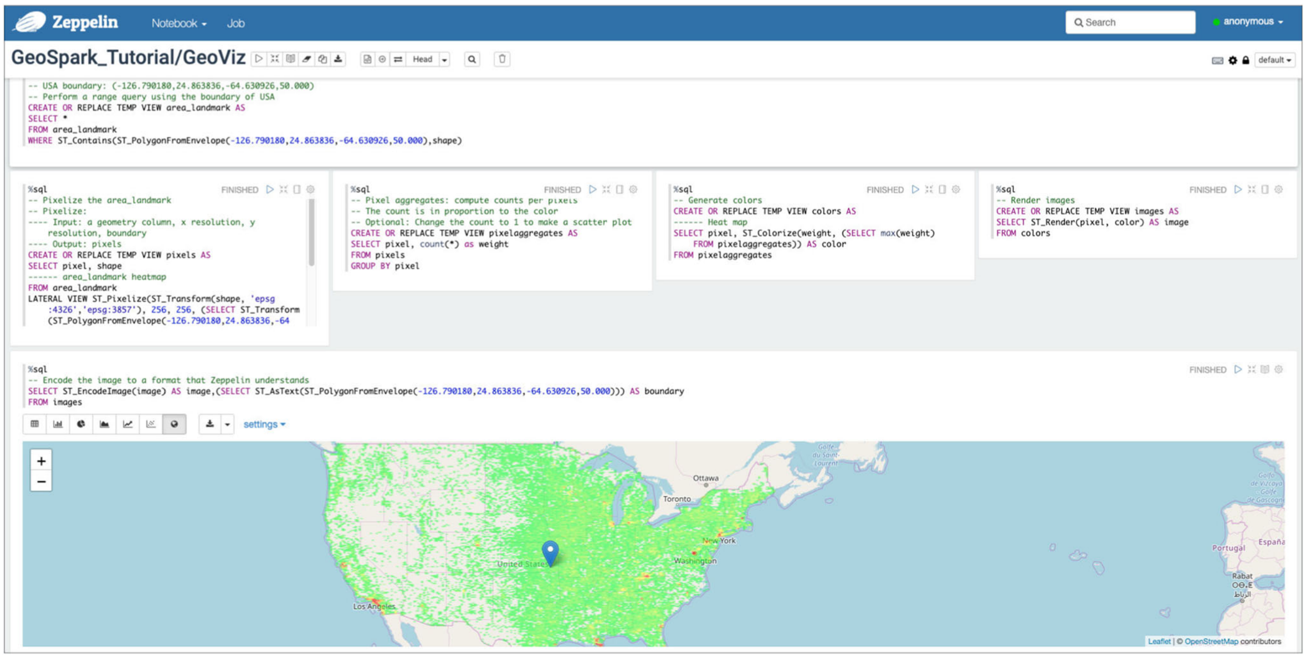
4 Declarative GeoViz

To combine both the map visualization and data processing phases, GeoSparkViz allows users to declaratively define a geospatial visual analytics (GeoViz) task using a SQL-like language (see Fig. 4). To achieve this, the system extends SQL to support GeoViz-related commands. Additional specification and examples can be found in “Appendix A”.

GeoViz SQL only works on structured data and thus extends the SQL interface of SparkSQL [2] which runs on top of structured RDDs in which every record (i.e., “row”) is structured to the same schema (same attributes), just like a regular database table. Similarly, Spatial RDDs in GeoSpark are also structured to the same table-like schema and hence are compatible with GeoSparkViz. The schema includes several attributes such as spatial attribute and pixel attribute. Although we use the term `Table` in this section, it is actually a structured `RDD`.

4.1 Map building operators

GeoSparkViz breaks down the map visualization pipeline to several declarative map building operators: pixelize, pixel aggregate and render. The reason is twofold: (1) flexibility: the users can easily customize each operator to achieve various map visualization effects using the declarative language. (2) optimization opportunity: when the user has a GeoViz



```
-- USA boundary: (-126.790180,24.863836,-64.630926,50.000)
-- Perform a range query using the boundary of USA
CREATE OR REPLACE TEMP VIEW area_landmark AS
SELECT *
FROM area_landmark
WHERE ST_Contains(ST_PolygonFromEnvelope(-126.790180,24.863836,-64.630926,50.000),shape)
```

```
%sql
-- Pixelize the area_landmark
-- Pixelize:
---- Input: a geometry column, x resolution, y
   resolution, boundary
---- Output: pixels
CREATE OR REPLACE TEMP VIEW pixels AS
SELECT pixel, shape
------ area_landmark heatmap
FROM area_landmark
LATERAL VIEW ST_Pixelize(ST_Transform(shape, 'epsg
   :4326','epsg:3857'), 256, 256, (SELECT ST_Transform
   (ST_PolygonFromEnvelope(-126.790180,24.863836,-64
```

```
%sql
-- Pixel aggregates: compute counts per pixels
-- The count is in proportion to the color
-- Optional: Change the count to 1 to make a scatter plot
CREATE OR REPLACE TEMP VIEW pixelaggregates AS
SELECT pixel, count(*) as weight
FROM pixels
GROUP BY pixel
```

```
%sql
-- Generate colors
CREATE OR REPLACE TEMP VIEW colors AS
------ Heat map
SELECT pixel, ST_Colorize(weight, (SELECT max(weight)
    FROM pixelaggregates)) AS color
FROM pixelaggregates
```

```
%sql
-- Render images
CREATE OR REPLACE TEMP VIEW images AS
SELECT ST_Render(pixel, color) AS image
FROM colors
```

```
%sql
-- Encode the image to a format that Zeppelin understands
SELECT ST_EncodeImage(image) AS image,(SELECT ST_AsText(ST_PolygonFromEnvelope(-126.790180,24.863836,-64.630926,50.000))) AS boundary
FROM images
```

Fig. 4 GeoSparkViz interactive GeoViz SQL notebook (deeply integrated with Apache Zeppelin)

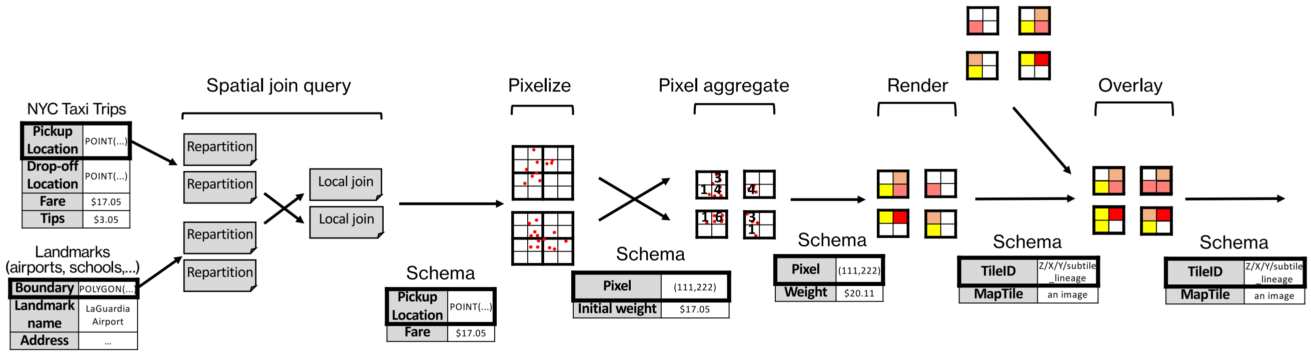


Fig. 5 Data flow of the straightforward GeoViz query execution plan

task which consists of map building operators and spatial query operators, the GeoViz optimizer may find a better query plan in terms of query execution time by re-organizing operators (see Sect. 7).

The map building operators must be executed in a sequential order: `Pixelize − > Pixel aggregate − > Render − > Overlay (optional)`. In the GeoViz SQL, GeoSparkViz provides specific SQL functions for each map building operator. Figure 5 shows an example of the output table schema of each operator.

Pixelize GeoSparkViz offers ST_Pixelize SQL function for the pixelize operator (see Sect. 5.1).

```
SELECT Pixel, [Observation / 1] AS Initial_weight
FROM   [Spatial_dataset]
LATERAL VIEW  ST_Pixelize([Spatial_attribute],
  [width], [height]) AS Pixel
```

– **Syntax** The user should use the SQL `LATERAL VIEW` syntax. ST_Pixelize function should be placed in the `LATERAL VIEW` clause because it generates one or more output rows per input row. The `FROM` clause can directly accept any table that consists of at least one spatial attribute.
– **Semantics** In ST_Pixelize, the user should specify a spatial data attribute (e.g., point, polygon) which is to be visualized, and the resolution of the desired map (in the format of width * height, the unit is pixel). Depending on the application, the user may opt to use a spatial observation attribute or "1" as the initial weight attribute in the `SELECT` clause. In Fig. 5, "fare" attribute is used as the initial weight.
– **Output table schema** The schema consists of two attributes, a pixel attribute and a numerical attribute

which is the initial weight. For every single spatial object, ST_Pixelize may generate one or more pixels.

Pixel aggregate The user can freely aggregate the overlapped pixels to produce the weight for each pixel (see Sect. 5.2).

```
SELECT Pixel, [aggregate](Initial_weight) AS weight
FROM   [Pixel_view]
GROUP BY  Pixel
```

- **Syntax** The user can use the original SQL GroupBy syntax to group records by the pixel attribute and apply an aggregate function for each group. The `FROM` clause should use the output table of Pixelize operator. No specialized function is needed in this operator.
- **Semantics** Several SQL aggregation functions (count, min, max, avg) are supported. They should aggregate the initial weight attribute. The aggregated weight value determines the actual color of the associated pixel.
- **Output table schema** The schema consists of a pixel attribute and a numerical attribute which is the weight of this pixel. The pixel attribute is in a Pixel type introduced by GeoSparkViz (see "Appendix A.1")

Render The rendering process (see Sect. 5.3) can be completed by two SQL functions in sequence:

(1) ST_Colorize: given the weight of a pixel, this function produces a color which is usually in proportion to the weight. The user can also write her own user-defined colorize function to replace ST_Colorize as long as it follows the same output format as this function.

```
SELECT Pixel, ST_Colorize(weight, [Max]) as Color
FROM   [Pixel_aggregate_view]
```

- **Syntax** The user can use ST_Colorize as a regular SQL function and put in the `SELECT` clause. The `FROM` clause should use the output table of the Pixel aggregate operator.
- **Semantics** ST_Colorize takes as input the weight and a max weight. The Max weight is used to normalize the weight to a particular RGB color range such that the function can produce a meaningful color. For example, the user can set the max = 40 (°C) for a temperature heatmap so any place on the map which has temperature $>= 40$ will be plotted as red.
- **Output table schema** The schema consists of a pixel-type attribute and an RGB color (in integer type).

(2) ST_Render: this function produces map tile images based on pixels.

```
SELECT Tile_ID, ST_Render(Pixel, Color) as MapTile
FROM   [Pixel_Colorize_view]
GROUPBY ST_TileID(Pixel) AS Tile_ID
```

- **Syntax** The user can use the original SQL GroupBy syntax to group pixels by their map tile id and apply ST_Render to each group of pixels. The `FROM` clause uses the output table from the colorize step. The ST_TileID function in the `GROUPBY` clause returns the tile ID of a given pixel. If the user just wants to get a single image of the entire dataset, he or she can abandon the `GROUPBY` clause and simply apply the ST_Render to all pixels in the pixel view to a single image (explained in "Appendix A").
- **Semantics** ST_Render is an aggregation function. It takes as input a group of pixels and their colors and produces a map tile based on a group of pixels.
- **Output table schema** The schema consists of two attributes, a tile id attribute and a map tile attribute. The map tile attribute is in Image type (explained in "Appendix A").

Overlay After getting a map, the user may opt to overlay it with another map (see Sect. 5.4). GeoSparkViz offers a ST_Overlay function to conduct this operation.

```
SELECT m1.Tile_ID, ST_Overlay(m1.MapTile, m2.MapTile)
  as MapTile
FROM   [Map tile view 1] m1, [Map tile view 2] m2
WHERE m1.Tile_ID = m2.Tile_ID
```

- **Syntax** The user should perform the Overlay operator using the SQL inner join syntax. The `FROM` clause should have two map tile views, each of which is an output table of a Render operator. The query joins two map tile views together by their tild IDs. Then the user can use ST_Overlay as a regular SQL function to overlay one tile over another tile.
- **Semantics** The ST_Overlay function takes as input two map tile attributes, both in Image type. The tile in the first attribute will be the front image and the one in the second attribute will be the back image. Both tiles have the same tile ID.
- **Output table schema** The schema consists of two attributes, a tile ID attribute and a map tile attribute. The tiles in the map tile attribute are the overlay images of two map tile images.

4.2 Assemble GeoViz queries

GeoSparkViz users can easily define a new map visualization effect using declarative map building operators. The users can also combine them together to a nested SQL query. For example, given a set of New York City Taxi trips records, the user can build a scatterplot of the taxi trip pickup points as follows (Fig. 1):

```
/* Render: Render tiles */
SELECT Tile_ID, ST_Render(pixel, color)
FROM {
  /* Render: Colorize pixels */
  SELECT pixel, ST_Colorize(weight, MAX) as color
6      FROM {
7        /* Pixel aggregate */
8        SELECT pixel, 1
9        FROM {
10         /* Pixelize */
11         SELECT pixel, 1
12         FROM   NYCtaxi taxi
13         LATERAL VIEW  ST_Pixelize(taxi.pickup,
14           16384, 16384) AS pixel }
15       GROUPBY pixel } }
16   GROUPBY ST_TileID(pixel) AS Tile_ID
```

Lines 10–14 of the query above is the first sub-query which uses ST_Pixelize to conduct the pixelization operator and it assumes the resolution of the final map is 16,384 * 16,384 (OSM Zoom Level 6 [23]). The query depicted by Lines 7–9 and 15 performs the pixel aggregate operator. Lines 4–6 describes the third query which generates colors for pixels. Lines 1–3 and 16 show the tile rendering.

4.3 Integrate map building operators with spatial query operators

In GeoSparkViz, spatial query operators can also process pixels. It is possible for the users to interleave spatial query operators and map building operators. GeoViz optimizer will optimize the query plan (explained in Sect. 7).

4.3.1 With spatial range query

The system user may need to filter the spatial dataset in the data processing phase before applying the map visualization effect. This can be achieved through the well-known spatial range query, which returns a subset of the input spatial objects or pixels that lie within a given query window.

Given a GeoViz query, the system will generate different GeoViz query execution plans which interleave the execution of the range query operator as well as the pixelize and pixel aggregate map building operators in different orders (see Fig. 6a), depending on the situation. The variety of query execution plans allows GeoSparkViz to co-optimize both phases of the GeoViz query (explained in Sect. 7). An example is as follows:

Example 1 Given the scatterplot of taxi trip pickup points in Sect. 4.2 , the user may only want to see the Manhattan area of NYC. The original query plan will be the leftmost one in Fig. 6a which is (1) retrieve New York City taxi trip pickup points that are within Manhattan area (2) render the data to the GeoViz effect. In GeoSparkViz, a re-written plan can be expressed by the GeoViz SQL statement as follows:

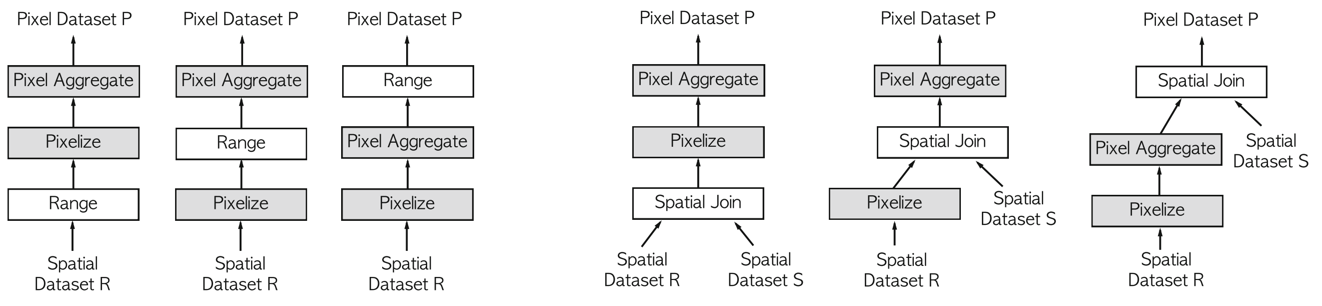
```
SELECT Tile_ID, ST_Render(view.pixel, view.color)
FROM   NYCtaxi_ScatterPlot view
WHERE  ST_WITHIN(view.pixel, ManhattanBound)
GROUPBY ST_TileID(view.pixel) as Tile_ID
```

GeoViz Query 1 declares a spatial range query on the taxi records to find those trips that lie within Manhattan using the ST_WITHIN function. Instead of acting on the original taxi records, the query actually uses a GeoViz view called `NYCtaxi_ScatterPlot` which returns a map scatter plot of Manhattan taxi trips. This view is a query which includes the GeoViz SQL functions, ST_Pixelize, Pixel aggregate and ST_Colorize (Lines 4–15 of the example in Sect. 4.2). The produced map tiles follow the Open Street Map zoom level 6 specification [23]). The ST_Render function generates map tiles on pixels that belong to the same tile and ST_TileID produces the tile ID for each pixel.

4.3.2 With spatial join query

The data scientist may need to integrate multiple datasets in the spatial data processing phase of the GeoViz task. To achieve this, a spatial join operator takes as input two spatial datasets R and S as well as a spatial join predicate and returns every pair of objects in R and S that satisfy the spatial predicate.

The system can interleave the execution of the spatial join operator as well as the pixelize and pixel aggregate map building operators resulting in different GeoViz query execution plans (see Fig. 6b). An example of a GeoViz query that combines spatial join and map building operators is given below.



(a) Combining MapViz with Spatial Range **(b)** Combining MapViz with Spatial Join

Fig. 6 Combining map building operators with spatial query operators

Example 2 Given a set of NYC taxi trips and a set of area landmarks, the query finds all taxi trips with pickup location lying within the area landmarks such as airports, cemeteries, parks, schools, and visualizes the join result using a heat map visualization effect. The original query plan will be the leftmost one in Fig. 6b. The optimized query plan can be expressed by the GeoViz SQL statement as follows:

```
SELECT Tile_ID, ST_Render (view.pixel,
                          view.pixel.color)
FROM    NYCtaxi_HeatMap view, AreaLandmarks
                        arealm
WHERE   ST_WITHIN(view.pixel, arealm.boundary)
GROUPBY ST_TileID(view.pixel) as Tile_ID
```

The view NYCtaxi_HeatMap which returns a heat map of all trip pick points in NYC is a query which includes the GeoViz SQL functions, ST_Pixelize, Pixel aggregate and ST_Colorize.

5 Map building operators

GEOSPARKVIZ supports four main map building operators: (1) Pixelize (Sect. 5.1), (2) Pixel Aggregate (Sect. 5.2) and (3) Render (Sect. 5.3) (4) Overlay (optional). Each map building operator works as a single step in the map tile image generation pipeline (`Pixelize − > Pixel aggregate − > Render − > Overlay`) and parallelizes the corresponding logic over the geospatial dataset(s) distributed among the cluster nodes.

Note that, although this section introduces the implementation details including input/output RDD formats of these operators, the user only interacts with GEOSPARKVIZ via GeoViz SQL queries (see Sect. 4) and is not aware of the internals.

5.1 Pixelize

To generate a map image, this operator pixelizes spatial objects 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.

Input format This operator takes as input (1) a Spatial RDD (schema: <spatial object, weight>) which contains numerous spatial objects (2) the designated map pixel resolution. Each spatial object carries an additional attribute as the weight (loaded from the original data source, such as Shapefile). A Spatial RDD can be from various data sources such as persistent storage (e.g., AWS S3 or HDFS) or outputs of spatial query operators.

Output format This operator outputs a <Pixel, weight> RDD. Other visualization operators manipulate these pixels and eventually plot them out on map tile images.

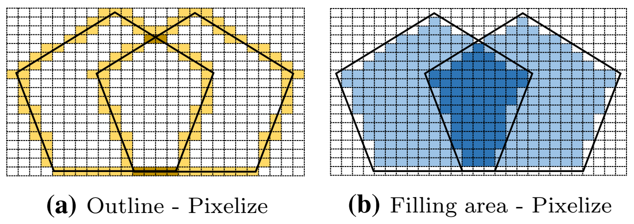


Fig. 7 Pixelize vector objects to pixels

Algorithm Algorithm 1 depicts the detail of this operator. This operator packages the pixelization into a single distributed *Map* operation. GEOSPARKVIZ adopts two pixelization rules: (1) outline-only (Fig. 7): only marks pixels covered by spatial objects' outlines; (2) filling area (Fig. 7): marks all pixels covered by spatial objects.

Outline-only 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 Eqs. 1 and 2.

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

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

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 will be used to determine the pixel color. The initial value of the weight is explained below.

Filling area If the user chooses the filling area pixelization rule (only valid for polygons), we need to mark all pixels covered by polygons. For each polygon in the input RDD, GEOSPARKVIZ first transforms all vertexes to pixels, then locates pixels that fall inside the pixelized polygonal boundary. All covered pixels are added to the output <Pixel, weight> RDD.

Weight A spatial object needs to be pixelized to one or many pixels which eventually have colors. The weight of this object indicates the color of these pixels (explained in Sect. 5.3). For example, in a heatmap, pixels with a higher weight may show red while those with a lower weight show green color. The user can decide the initial value of the weight in the <Pixel, weight> RDD depending on the visualization effects. If he or she just wants to plot a map based on the spa-

```
Input: <Spatial object, weight> RDD, map resolution
        (width*height)
Output: <Pixel, weight> RDD
Function Map(spatial object O)
    /* weight is a spatial observation or 1
       */
    switch Pixelization rule do
        case "outline-only"
            Decompose O into line segments;
            Find all pixels covered by line segments;
            return <Pixel coordinate, weight>;
        end
        case "filling area"
            Find all pixels that are within the polygon
            boundary;
            return <pixel coordinate, weight>;
        end
    endsw
```

Algorithm 1: Operator I: Pixelize

```
Input: <Pixel, weight> RDD partitioned by the GeoViz-aware
        spatial partitioner
Output: <Pixel, weight> RDD
Function MapPartition(a data partition P of the input
RDD)
    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 and update the weight;
        If the pixel does not exist, add it and its weight to the
        $HM$;
    end
    return $HM$;
```

Algorithm 2: Operator II: Pixel aggregate

tial distribution of the dataset such as a Scatter plot/Heat map of geo-tagged tweets, he can set the initial weight of all spatial objects as 1. If the user wants to plot spatial observations associated with spatial objects, he or she can use the spatial observation as the initial weight of each Pixel. For example, in a shapefile of land surface temperatures, each record has a location and a temperature observation. The user can use observed temperatures as the initial weights for locations and plot a heat map.

Cost Given $N_{objects}$ input spatial object and P data partitions, the cost of the pixelize operator $C_{Pixelize} = objects\ per\ partition$, which is equivalent to $\frac{N_{objects}}{P}$

5.2 Pixel aggregate

The pixelize operator 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 relatively 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 aggregates the weight of the overlapped pixels and determine the final weight of this pixel.

Input format This operator takes as input a <pixel, weight> RDD which is re-partitioned by the GeoViz-aware spatial partitioner (see Sect. 6). In the re-partitioned RDD, pixels in the same RDD partition are neighbors on the actual map tile image.

Output format This operator outputs a <pixel, weight> RDD. In this RDD, no pixels have the same screen coordinates with others.

Algorithm The pixel aggregate operator uses a *MapPartition* function to run a partition-wise algorithm (see Algorithm 2). For each partition in the <pixel, weight> 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 allows several aggregation strategies to aggregate overlapped pixels (see Pixel aggregate in Sect. 4.1): (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 should be used to visualize the spatial observations (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.

Cost Given N_{pixels} input pixels and P data partitions, the cost of the pixel aggregate operator $C_{Pixel\ aggregate}$ is equal to the number of pixels per partition ($\frac{N_{pixels}}{P}$).

5.3 Render

After aggregating pixels, the render operator decides a proper color for each pixel according to the weight. Then it produces a colorful map tile image for each RDD partition.

Input format The render operator takes as input the `<Pixel, Weight>` RDD and a max weight (optional)

Output format This operator generates a distributed map tile RDD in the format of `<TileID, Map Tile>`. The map

```
Input: <Pixel, weight> RDD, max weight
Output: <Tile ID, Map tile> RDD
// Colorize
Function MapPartition(a data partition P of the input RDD)
    Create an empty <pixel, color> list $L$;
    Optional photo filter: Update weights using the convolution matrix;
    foreach Pixel $Px$ in $P$ do
        Decide $Px$'s color according to its weight;
        Add $< Px, Color >$ to $L$;
    end
    return $< Px, Color > L$;
// Render
Function MapPartition(a data partition P of the $< Px, Color >$ RDD)
    Get the tile ID $ID$;
    Create a blank image canvas $IMG$ using the tile's resolution;
    foreach Pixel $Px$ in $P$ do
        Plot $Px$ on $IMG$ using $Px$'s color;
    end
    return $< ID, IMG >$;
```

Algorithm 3: Operator III: Render operator

tiles in this RDD compose the complete map at the given map resolution.

Algorithm Algorithm 3 describes the two steps in the Render operator: colorize and render.

Colorize The system accepts a user-defined function or uses a default equation to decide the pixel color, which is based on pixel weights. The weight can be pre-normalized to domain [0,255]. GeoSparkViz plugs the color equation into a *MapPartition* function to perform *Colorize* step. For each pixel in this partition, it calculates the color and plots the pixel on a blank image canvas. Two equation types are common: (1) Linear equation (GeoSparkViz default). For example,

$$Color(R, G, B) = Color(255, 255, weight) \tag{3}$$

This equation will give the user an image with diverse colors. (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.

In the colorize step, the user can opt to run an optional image processing filter function which applies classic photo filters such as sharp, blur (used in heat map) or diffusion to the map tile image in order to deliver some special effects. Its basic idea is, for a pixel in the image, add the weight values from its neighbor pixels, controlled by a convolution matrix, to this pixel. Each convolution matrix describes a 3*3 matrix in which each individual element indicates how strong the center pixel's color is affected by the corresponding neighbor pixel's color. The new weight value of a pixel is $\sum_{i=a\ pixel}^{Neighbors} i's\ impact * i's\ weight$. To achieve this, this operator first puts all < pixel, weight > pairs in a HashMap. Before calculating the color for a pixel, it first fetching the weights of neighbor pixels and updates the weight of this pixel.

Render The operator then uses another *MapPartition* function to put pixels in each partition to an image and generates a $TileID$, $Maptile$ RDD. Each The newly generated image of each data partition has a TileID. Each map tile is a Java BufferedImage object which stores pixel colors in a compact byte array. Note that some images generated using different data partitions may have the same TileID since the data in these partitions represents a portion of the same map tile (see Sect. 6). A tile image browser can handle this easily. Eventually, all map tiles are rendered in parallel and the system can then return the generated `<TileID, Map Tile>` to the user.

Cost Given N_{pixels} pixels and P data partitions, the cost C_{Render} of the render operator is equal to $\frac{N_{pixels}}{P} * 2$ because there are two *MapPartition* functions used. The cost of applying photo filters does not affect it because fetching 3*3 values from a HashMap is O(1) complexity.

5.4 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.

Input format As an optional operator, the overlay operator takes as input two <Tile ID, Map Tile> RDDs and overlay them one by one in the order specified by the user.

Output format This operator outputs a unified <Tile ID, Map Tile> RDD.

Algorithm This operator first leverages *ReduceByKey* operation on two input RDDs (a front image RDD and a back image RDD) with the tile ID as the key so that two map tiles from two RDDs are shuffled together across the cluster. Then this operator merges together the two tiles which all describe the same area of the overall image. During this process, the overlay operator makes sure that the map tile from the front image RDD stands in the front. This operator generates a new <Tile ID, Map Tile> RDD which can be persisted to external storage devices or continue to overlay with another <Tile ID, Map Tile> RDD

Cost The cost of Overlay operator includes two parts:

$$C_{Overlay} = C_{shuffle\ tiles} + C_{merge\ tiles} \tag{4}$$

Shuffling tiles needs to send N_{pixel} pixels so that its cost can be estimated as N_{pixel}. The output RDD of the shuffling step is actually repartitioned according to map tile ID. Assume the pixels are uniformly distributed, there will be same number of pixels belong to every map tile ID. Merging tiles re-plots pixels that are from two sets but have the same ID to one tile. Therefore, the cost of merging tiles $C_{merge\ tiles}$ is equal to the number pixels in the merged tiles ($\frac{N_{pixel}}{T}$).

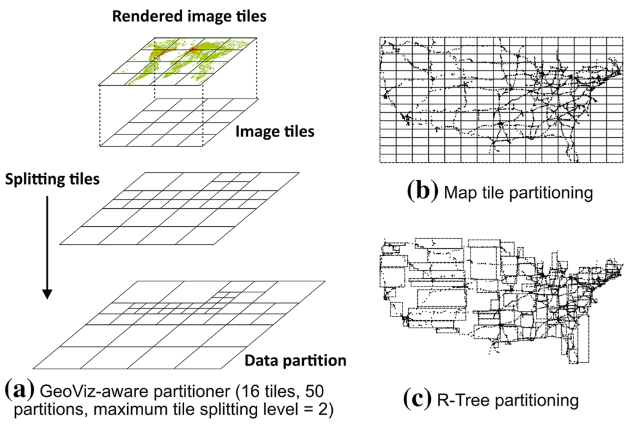

Rendered image tiles
Image tiles
Splitting tiles
Data partition
(a) GeoViz-aware partitioner (16 tiles, 50 partitions, maximum tile splitting level = 2)
(b) Map tile partitioning
(c) R-Tree partitioning

Fig. 8 Spatial partitioning approaches

6 GeoViz-aware spatial partitioner

A good spatial data partitioning method should (1) partition spatial objects by their spatial proximity while still (2) ensure the load balance: each partition contains roughly the same number of spatial objects [7,40]. Moreover, because GeoSparkViz wants to partitions data *once* for both the data processing and map visualization phases to reduce the amount of data shuffling in the cluster (explained in Sect. 7), the data partitioner should also (3) follow map visualization constraint: the produced images can be used as uniform map tiles easily.

Existing spatial partitioning approaches, such as R-Tree and Quad-Tree, exhibit good performance when executing spatial queries for the data processing phase [36,40]. However, these approaches do not consider the map visualization constraint (see Fig. 8c). In other words, existing spatial partitioning techniques are not able to be used in the visualization operators that process pixels and produce map tiles. On the other hand, partitioning the workload based on the uniform map tile boundaries avoids the tedious process of recovering the map tiles. However, the tile-based partitioning approach cannot handle the spatial data skewness and hence fails at balancing the workload among the cluster nodes (see Fig. 8b).

Input format The GeoViz-aware spatial partitioner proposed by GeoSparkViz takes as input a SpatialRDD or a <pixel, weight> RDD (see Sect. 5).

Output format This partitioner outputs a repartitioned SpatialRDD or a repartitioned <pixel, weight> RDD. Each spatial object (or pixel) internally possesses a tile ID that indicates the uniform map tile where this object lies in. While enforcing the spatial proximity constraint, spatial objects assigned to the same partition should also belong to the same map tile image. In other words, all geospatial objects in a data partition should have the same map tile ID.

Algorithm 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 objects are assigned to partitions.

Step II: Tile-aware Data Partitioning As shown in Fig. 8, this step first splits the space into uniform map tiles, 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 objects. As the splitting goes on, tile boundaries become more and more non-uniform, but load balanced. When the splitting totally 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 Fig. 8).

Step III: Physical Partitioning This step passes the partition structure (Fig. 8) stored in the master machine to all machines in the cluster. For every spatial object or pixel, GeoViz 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 spatial object. If the search algorithm stops at a leaf-level sub-tile, the object 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 object is assigned to all leaf-level sub-tiles under this non-leaf sub-tile. Eventually, objects or pixels that fall in the same leaf-level sub-tiles are physically located in the same cluster node.

Cost Note that the three steps run sequentially and each step runs on a distributed spatial dataset (i.e., geospatial objects or pixels) in parallel. Therefore, the overall cost for GeoSparkViz GeoViz-aware spatial partitioner is:

$$C_{Partitioner} = C_{Sampling} + C_{CalcPartition} + C_{RepartitionAll}$$

Consider a set of geospatial objects or pixels with N$_{objects}$/N$_{pixels}$ elements in total and P data partitions, sampling ratio s, the cost of the parallel spatial sampling step $C_{Sampling}$, which performs a local scan per partition, is $\frac{N}{P}$. The cost $C_{CalcPartition}$ of deciding the tile-aware spatial partitions on the master machine is equivalent to scanning the sample once is $s * N$. For a geospatial object or pixel in a partition, Step III searches the corresponding Quad-tree to locate the relevant data partition. Assuming that every map tile is splitted until the max tile splitting level L (L = 3 by default) and there are 4^L sub-tiles under each tile, the search cost is: C$_{LocatePart} == log_4(4^L) = L$

Given that each partition contains $\frac{N}{P}$ objects/pixels, the cost of assigning partition ID and shuffling all records across the cluster is as follows (N can be either N$_{objects}$ or N$_{pixels}$):

$$\begin{aligned} C_{RepartitionAll} &= \frac{N}{P} * C_{LocatePart} + C_{shuffle} \\ &= \frac{N}{P} * L + N \end{aligned}$$

7 GeoViz optimizer

A GeoViz query execution plan involves several operators: (1) map building operators: perform pixelization, pixel aggregation and map tile rendering. (2) GeoViz-aware spatial partitioner: repartitions geospatial data and pixels across the cluster nodes based upon spatial proximity and visualization constraints. (3) spatial query operators: process spatial queries on the input dataset.

The optimizer takes as input a GeoViz query, written in SQL, parses it and finds the straightforward GeoViz execution plan which performs the two GeoViz phases (i.e., data processing and map visualization) separately in a serial fashion. Then, the optimizer enumerates a set of candidate plans by interleaving map building operators and spatial query operators and picks the one which minimizes the total run time of the GeoViz task.

7.1 Estimate the intermediate data size

An important aspect that GeoSparkViz takes into account is the intermediate data size produced by each operator. Reducing the intermediate data size leads to less data passed between different GeoViz operators and hence reduces the amount of data shuffle across the network. Note that estimating the intermediate data size is difficult since operators in the GeoViz pipeline do not only deal with complex spatial objects (e.g., polygons, points, and line strings), but also manipulate pixels and map tiles. Suppose the GeoViz task is to visualize a set of uniformly distributed rectangular spatial objects (size: $N_{objects}$).

Pixelize operator The number of pixels (N_{pixels}) produced by the pixelize operator may not be the same as the number of input spatial objects $N_{objects}$. When the map resolution R is high, each rectangle object will be pixelized to a single pixel and the human eye cannot identify the shape using the low-resolution map. On the other hand, in case the resolution is high, a rectangle object may be assigned to multiple pixels and the human eye can hence identify detailed shapes on the final map. Therefore, the estimated output data size is in proportion to R. We propose a parameter called Scale Up Factor (denoted as SF_{up}). It indicates how many times the pixelize operator multiplies its data. Thus, the estimated output data size is:

$$N_{pixels} = N_{objects} * SF_{up} \tag{5}$$

By default, we assume that

$$SF_{up} = 2^{zoom\ level}$$

where zoom level is a parameter defined by OpenStreetMap [23]:

$$Zoomlevel = log_4(\frac{R}{256 * 256})[23]$$

This is because spatial objects in the big datasets used by our experiments and in practice are at the road/block level and usually appear as a few pixels if the zoom level is low (i.e., level 2). However, the users can change the parameter if their scenarios differ from our assumption. For example, if all objects in the input dataset are points, SF_{up} can be set as 1.

Pixel aggregate operator This operator shrinks the intermediate data size when the designated map resolution is low. This is due to the fact that many overlapped pixels are aggregated into a single pixel and the size of the output pixels is never more than the resolution R. In case the resolution is much higher than the produced pixels N_{pixels} from the pixelize operator, the pixel aggregate operator may not reduce the size of the intermediate results. If we assume input pixels are uniformly distributed, the number of output pixels after the aggregation is:

$$N_{pixels} = min(N_{pixels}, R) \tag{6}$$

Spatial query operators The spatial range query operator filters the input data size using a range query window so that it also reduces the intermediate data size. Spatial join operator may prune less intermediate data since it consists of many range windows. GeoSparkViz estimates the intermediate data size using the classic spatial query selectivity factor estimation. Thus, suppose N$_{window}$ query windows have the same area Area$_{window}$, query operators'

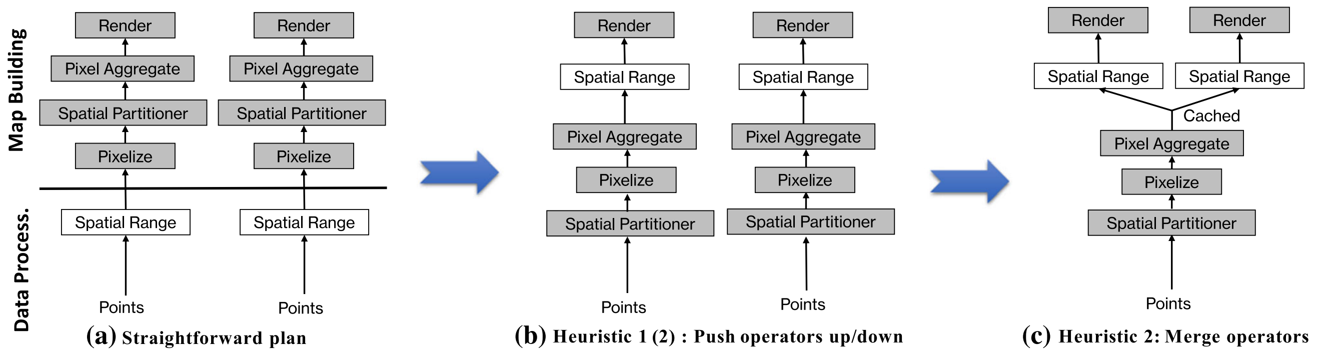


Fig. 9 (Multi-) GeoViz Query 1: Multi-Range+map building

impact on the intermediate data size can be estimated by adding up the selectivity of each query window: $N_{output} = Query\ selectivity * N = \frac{Area_{window}}{Dataset\ area} * N_{window} * N$, N can be either $N_{objects}$ or N_{pixels}. If there are too many query windows (e.g., in a spatial join query), the optimizer samples the windows and estimates the selectivity.

7.2 Query optimization

To conduct the query optimization, the optimizer enumerates a set of candidate plans based upon heuristic optimization strategies (Heuristic Based Optimization). Then the optimizer calculates the time cost of each candidate GeoViz execution plan using the cost models provided in Sects. 5 and 6 and selects the plan with the minimum execution time cost (Cost-Based Optimization).

7.2.1 Heuristic-based optimization

The GeoViz optimizer generates a set of candidate query plans by directly applying several heuristic-based optimization strategies to the original query plan obtained from the query parser.

The user may submit a GeoViz application which involves multiple GeoViz tasks (e.g., prints customized maps for different regions of a big dataset). Therefore, the heuristic-based optimization runs in two steps:

Step I: Single-query heuristic strategy No matter how many GeoViz queries exist in one application, the optimizer first applies heuristics to each GeoViz query: *(1) merge partitioners:* if there are multiple spatial partitioning operators in the plan, the optimizer only keeps the plan's first partitioner which is the GeoViz-aware spatial partitioner. This can avoid unnecessary data shuffles. *(2) push operators up/down:* some operators may reduce the cost if they are put at difference places in the plan. For example, spatial query operators can reduce intermediate data size (e.g., Spatial Range and Spatial Join) and hence reduce the cost of their subsequent operators. The optimizer will try to move them to difference places of the query plan and thus yield a number of new candidate plans.

Step II: Cross-query heuristic strategy If a map building operator or a spatial partitioner appears multiple times across several queries in the application, the optimizer will merge the repeated operators to avoid redundant computation. Note that, the optimizer merges the same map building operator only if the operator's input sub-tree remains the same in all queries. In other words, this operator should have the same input data and parameters in all queries of this batch (see Fig. 9(c) and Case 1 in Sect. 7.3). The optimizer will cache the output of the merged operator after the first round execution of this operator. This persists the output in memory and hence allows it to be re-used by subsequent operators.

7.2.2 Cost-based optimization

After applying the heuristic strategies, GeoSparkViz optimizer produces candidate execution plans. The optimizer then calculates the time cost of each candidate query plan using the cost models provided in Sects. 5 and 6 and finds the plan with the minimum cost.

Cost of spatial range query operator Given $N_{objects}$ input spatial objects or N_{pixels} input pixels, each partition has $\frac{N}{P}$ records, where P is the number of partitions. The time cost of a spatial range query can be estimated as the time of scanning one partition ($C_{range} = \frac{N}{P}$). If the input data is partitioned by the spatial partitioner, the query may be faster because some partitions can be pruned if their boundaries do not intersect the query window. This is useful when the computing resources are scarce. However, since the query runs in a cluster, the cost is dominated by scanning the remaining partitions (as known as the "stragglers"): $C_{range} = \frac{N}{P}$.

Cost of spatial join query operator Given $N_{objects}$ input spatial objects or N_{pixels} input pixels plus N_{window}, without considering indexes, the cost of a spatial join query operator is equivalent to a local nested loop join on P data partitions in parallel. Thus, its cost is ($\frac{N*N_{window}}{P^2}$), N can be either $N_{objects}$ or N_{pixels}.

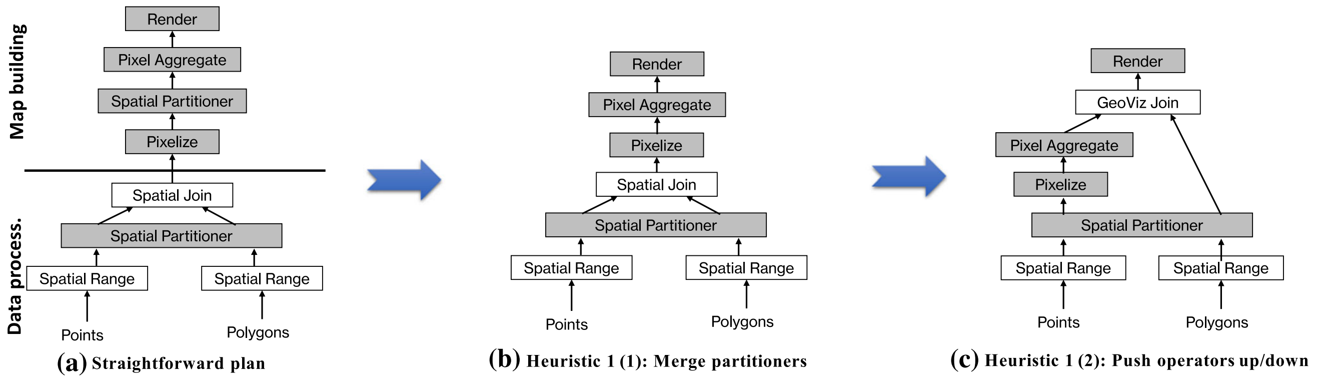


Fig. 10 GeoViz Query 3: Range+Join+map building

We take the leftmost plan in Fig. 6a as an example to illustrate how to compute the cost of a GeoVIz query plan. The complete plan includes the following operators: `Spatial Range -> Pixelize -> Partitioner -> Pixel aggregate -> Render`. The corresponding cost is:

$$Cost = C_{range} + C_{pixelize} + C_{partitioner} + C_{pixel\ aggregate} + C_{render} \quad (7)$$

We substitute the costs with their actual values and obtain the following equation:

$$Cost = \frac{N}{P} + \frac{N_{range}}{P} + \left(\frac{N_{pixelize} * L}{P} + N_{pixelize}\right) + \frac{N_{pixelize}}{P} + \frac{N_{pixel\ aggregate} * 2}{P} \quad (8)$$

where N is the size of the input dataset and N with subscripts are the output sizes of the corresponding operators.

7.3 Case study

In this section, we study two typical GeoViz workloads to show the optimized query execution plans over the straightforward execution plans.

Case 1: Multiple (range + map building) Given a dataset about attractions around the world, the data scientist wants to create scatter plots or heat maps for several regions he or she is interested in. This workload includes multiple GeoViz Query 1, range query + map visualization. Each spatial range query has a different query window. The straightforward execution plan that runs the data processing and visualization in series is given in Fig. 9a. If GeoSparkViz follows the straightforward execution plan, several operators will be executed multiple times: pixelize, pixel aggregate and spatial partitioner. However, GeoSparkViz pushes the spatial range query operator to a place after the pixelize aggregate operator and finds that a cross-query operator merging optimization can be applied here. The optimizer decides to pixelize spatial objects **only once** and then executes spatial range queries **multiple times**. Moreover, the optimizer caches the output of the pixel aggregate operator.

Case 2: Range + spatial join + map building Assume that the data scientist wants to visualize a heat map of Manhattan taxi trips that were picked up in Manhattan landmark areas, e.g., parks and schools.

Example 3 Given a set of NYC taxi trip pickup points and a set of area landmarks, the query finds all Taxi trips with pickup location lying within Manhattan and the area landmarks that also lie within Manhattan. The query applies spatial join between Manhattan Taxi trips and the Manhattan landmarks and visualizes the join result using a heat map visualization effect. The corresponding SQL is as follows:

```
SELECT  ST_Render (view.pixel, view.color)
FROM    NYCtaxi_HeatMap view, AreaLandmarks A
WHERE   ST_WITHIN(view.pixel, ManhattanBound),
        ST_WITHIN(A.boundary, ManhattanBound),
        ST_WITHIN(view.pixel, A.boundary);
GROUPBY ST_TileID(view.pixel, 6)
```

The non-optimized execution plan (see Fig. 10) first performs range queries on New York City taxi trip pickup points and US landmark area boundaries using Manhattan region as the query window, respectively. Then, it joins the results of the two range queries and passes the join result to the map visualization operators, which in turn generate the map image. However, as shown in Figs. 5 and 10, the straightforward plan exhibits two time-consuming data shuffling operations introduced by spatial data partitioners.

On the other hand, the optimized plan picked by GeoSparkViz first applies the spatial range predicate on the two datasets to reduce the data scale. Then, it performs the pixelize operator on the point object dataset (NYC Manhattan taxi trips), then runs the pixel aggregate operator on the overlapped pixels to reduce the intermediate data size. Instead of directly pixelizing the polygon dataset (landmark area boundaries), which leads to large-scale intermediate data, the optimizer joins the city boundaries with pixels. Moreover, since both the spatial join operator and the pixel aggregate operator run on the same partitioned data, the two spatial partitioning operators are merged and placed at the beginning of the plan (i.e., pushed down). In addition, the duplicate removal step is skipped by the optimizer because it does not affect the visualization effect.

8 Experiments

In this section, we conduct a comprehensive experimental evaluation of GeoSparkViz.

Objectives We study the performance of GeoSparkViz from the following aspects: (1) overall map building speed (Sect. 8.1). (2) the effectiveness of GeoViz-aware spatial partitioner (Sect. 8.1) (3) the effectiveness of GeoViz query optimization (Sects. 8.2, 8.3 and 8.4) (4) system scalability on different cluster sizes and map resolutions (Sects. 8.5, 8.6). Our main evaluation metric is the execution time.

Datasets We use six real spatial datasets in the experiments (see Table 2 and Fig. 11): (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 of 20 million roads in US. Each road is represented in the format of a line string which is internally composed by 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* [37]: 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, SpatialHadoop 2.4 (+ visualization extension called HadoopViz), GeoSpark 0.9 (+ visualization extension), and GeoSparkViz in this cluster.

Compared approaches In order to carefully investigate the visual analytics 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 spatial query operators, map building operators, GeoViz-aware Partitioner, and optimizer. Given a test scenario, GeoSparkViz optimizes the execution plan as far as possible. (2) GeoSparkViz*: This approach is the full GeoSparkViz but it only tries a part of the heuristic strategies. The detailed explanation is put in corresponding sections. (3) SparkViz: this approach is GeoSpark and its visualization extension. It leverages Apache Spark to transfer intermediate data via memory but runs a GeoViz in two separate phases (data processing SparkViz-Proc and map visualization SparkViz-Viz) without any optimization. It uses regular spatial partitioning method in data processing and map tile partitioning in map visualization. (4) HadoopViz: this approach is SpatialHadoop and its visualization extension, namely HadoopViz. It also runs a GeoViz in two separate phases (data processing HadoopViz-Proc and map visualization HadoopViz-Viz) without any optimization. Intermediate data in this approach is transferred through disk.

GeoViz Query Workload To analyze the performance of GeoSparkViz, we perform experiments on three main GeoViz query workloads. The **Map building only** workload only runs the GeoViz-aware spatial partitioner and map building operators. GeoSparkViz does not run spatial queries and query optimization for this workload. The **MultiRange+map building** workload is similar to GeoViz Query 1, which issues a spatial range query and visualize its result to either scatter plot or heat map. The query is performed five times in this workload (see below) using five different spatial range predicates. GeoSparkViz uses the plan depicted in Fig. 9c and the other approaches use the straightforward plan. We use the following range query predicates: (1) Arizona state boundary for Roads (2) Arizona state boundary for Roads Edges (3) Manhattan island boundary for NYCtaxi (4) US mainland boundary for OSMpoint. The **Range+Join+map building** workload is similar to GeoViz Query 3. GeoSparkViz runs the plan depicted in Fig. 10c and the other approaches use the straightforward plan depicted in Fig. 10a. For this workload, we use the following range and join predicates: (1) AreaLandmarks in Arizona joined with Roads in Arizona, (2) AreaLandmarks in Arizona joined with Edges in Arizona (3) AreaLandmarks in Manhattan joined with NYCtaxi in Manhattan , and (4) OSMPostal in US mainland joins OSMpoint in US.

Default parameter settings By default, we use OpenStreetMap standard zoom level 6 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 GeoViz-aware spatial partitioner is 3, which means each map tile is split at most 3 times. Unless mentioned otherwise, SparkViz uses Quad-Tree partitioning in data processing phase and map tile partitioning for the map visualization phase. HadoopViz uses R-Tree data partitioning in the data processing phase

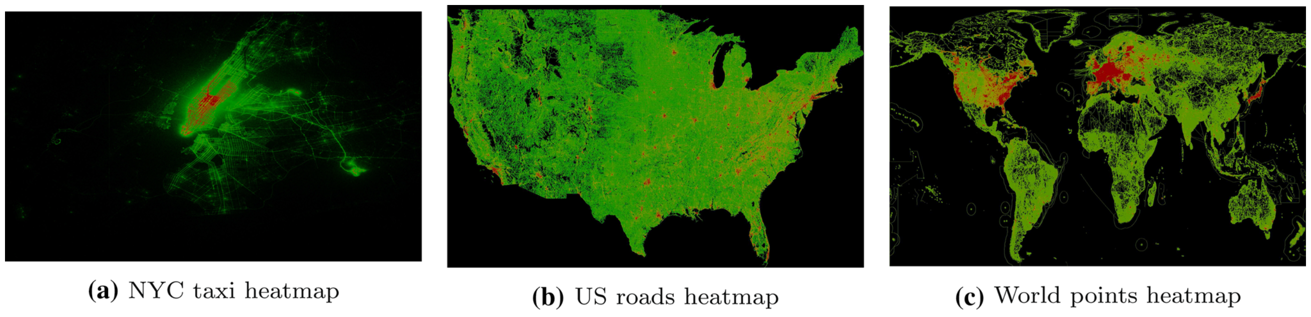


Fig. 11 Map visualization examples (best viewed in color)

Table 2 Test datasets

Dataset	Records	Description
Tiger landmarks	130 thousand	Polygonal boundaries of area landmarks in US
OSM postal codes	171 thousand	Polygonal boundaries of postal areas (major cities) in the world
TIGER roads	20 million	Line string shapes of all roads in the world
TIGER edges	73 million	Line string shapes of all rivers, roads, rails in US
NYC taxi	1.3 billion	New York City taxi trip pickup points
OSM point	1.7 billion	All points in the world

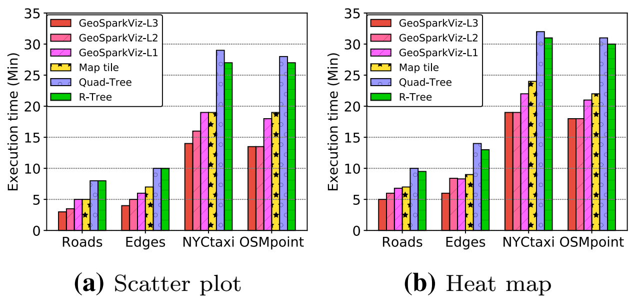


Fig. 12 Performance of GeoViz partitioner

and map tile partitioning in the map visualization phase. All compared approaches use the 64MB as initial data partition size.

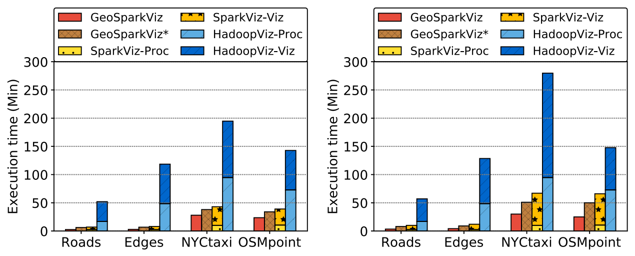
8.1 Impact of spatial partitioning

In this section, we study the performance of GEOSPARKVIZ map building and GeoViz-aware spatial partitioner. To this end, we compare four different spatial data partitioning approaches, GEOSPARKVIZ GeoViz-aware partitioning, map tile partitioning, Quad-Tree spatial partitioning and R-Tree partitioning. All these partitioning methods are implemented in GEOSPARKVIZ. The GeoViz workload used in this section contains only map visualization, which directly performs the visualization effect on the entire spatial datasets and produces scatter plots or heat maps. For GEOSPARKVIZ partitioner, we also vary the maximum tile splitting level parameter (i.e., Level 1, 2, and 3).

As shown in Fig. 12, GEOSPARKVIZ GeoViz optimizer run 1.5X–2X faster than uniform map tile partitioning method as expected. This is because the map tile partitioning approach does not balance the load among the cluster nodes. This degrades the performance even more when the spatial dataset is very skewed. Moreover, a visualization task with larger GEOSPARKVIZ max tile splitting level run 15% faster than its variant with the lower splitting level. This happens since the GeoViz-aware 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 shuffling 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.

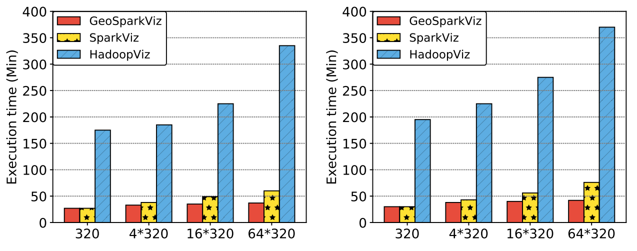
8.2 Performance of MultiRange+map building

In this section, we study the performance of the GeoViz query optimization by running the MultiRange+map building GeoViz query workload. We run the query on GEOSPARKVIZ, GEOSPARKVIZ*, SparkViz, and HadoopViz GeoViz approaches. The experiment also involves four datasets with different scale. GEOSPARKVIZ* only applies Heuristic Strategy 1 and does not merge operators across queries.



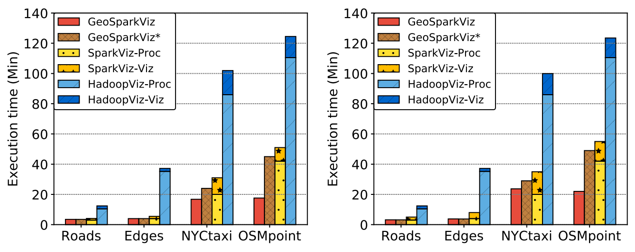
(a) Scatter plot **(b)** Heat map

Fig. 13 Performance of MultiRange+map building



(a) Scatter plot **(b)** Heat map

Fig. 15 Effect of the size of the query area (km^2)



(a) Scatter plot **(b)** Heat map

Fig. 14 Performance of Range+Join+map building

As shown in Fig. 13, compared to GeoSparkViz, SparkViz spends 50% more time and GeoSparkViz* costs around 20% more time, for generating scatter plots. GeoSparkViz is around an order of magnitude faster than HadoopViz.

The result makes sense: (1) GeoSparkViz* is faster than SparkViz because the GeoViz partitioner in GeoSparkViz is more load-balanced than the map tile partitioning adopted by SparkViz. Note that, GeoSparkViz* pixelizes spatial objects and renders tiles after the spatial range query. Because, after applying Heuristic Strategy 1 (Fig. 9b), GeoSparkViz* still thinks that "query first and pixelize later" exhibits a lower cost and thus goes back to this plan: `spatial range -> pixelize-> pixel aggregate -> render`. (2) GeoSparkViz is faster than GeoSparkViz* because it applies both Heuristic 1 and 2 and thus finds a better plan. Its execution plan (see Fig. 9c) first pixelizes spatial objects to pixels and caches them into memory, and all spatial range queries (with different predicates) run directly on the cached pixel dataset. (3) GeoSparkViz is faster than HadoopViz because it uses an efficient query plan and HadoopViz reads/writes intermediate data on disk.

When generating the heat map visualization effect for the MultiRange+map building workload, GeoSparkViz is around two times faster than SparkViz. Generating the heat map visualization effect takes 20%-50% 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 data partition.

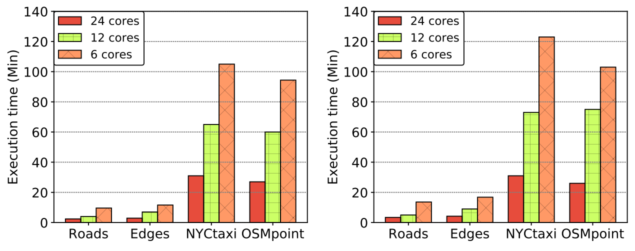
8.3 Performance of Range+Join+map building

Figure 14 studies the performance of the GeoViz query optimization by running Range+Join+map building GeoViz query workload in GeoSparkViz, GeoSparkViz*, SparkViz and HadoopViz on four different datasets. GeoSparkViz* tries Heuristic 1(1) “merge partitioners” (Fig. 10b) but does not perform Heuristic 1(2) “Push operators up/down”.

As it turns out from the figure, GeoSparkViz is around two times faster than SparkViz and an order of magnitude faster than HadoopViz on all data scales. This happens because (1) The “merge partitioner” strategy picked by GeoSparkViz and GeoSparkViz* (see Fig. 10b) only leads to a single data shuffling operation, which happens at the very beginning. Meanwhile, SparkViz and HadoopViz perform two data shuffling operations, which takes its toll on the overall data-to-visualization time. Based on the experiments, for the case of OSMpoint data, shuffling intermediate data across cluster in GeoSparkViz only sends around 1 GBytes of data across network. On the other hand, SparkViz transfers more than 5 GBytes of data during the shuffling phase. (2) on large-scale datasets NYCtaxi and OSMpoint, GeoSparkViz aggregates the pixels before performing the spatial join operator, and hence reduces the amount of intermediate data passed to the spatial join operator, which improves the performance w.r.t time. This is also the reason why GeoSparkViz runs 20–50% faster than GeoSparkViz* in this case. On small-scale datasets such as Roads and Edges, after trying the "Push operators up/down" strategy, GeoSparkViz still decides to use the plan in Fig. 10b instead of the one in Fig. 10c because of the cost. Thus, its execution time is similar to GeoSparkViz* in this case. Moreover, GeoSparkViz is around 6 times faster than HadoopViz. This happens due to the fact that HadoopViz performs two data shuffling steps and the shuffled data needs to be written/read to HDFS.

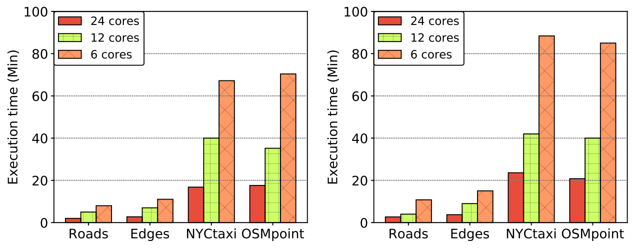
8.4 Impact of range query area

This section studies the effectiveness of GeoViz query optimization by varying the range query window size which affects the cost of query plans. We use NYCtaxi dataset



(a) Scatter plot (b) Heat map

Fig. 16 MultiRange+map building on different CPU cores

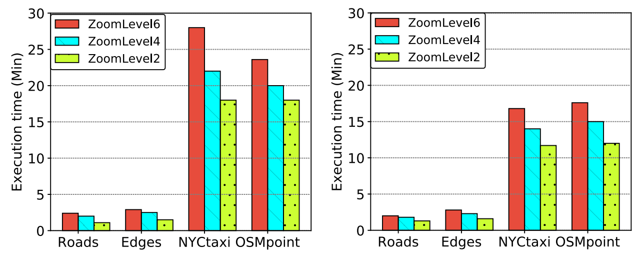


(a) Scatter plot (b) Heat map

Fig. 17 Range+Join+map building on different CPU cores

and MultiRange+map building GeoViz workload but vary the range query window area. The smallest query window area is a 320km^2 rectangle region in the center of New York City region. We keep multiplying this area by 4 and generate another three query windows, 4*320 km^2, 16*320 km^2, 64*320 km^2. Each test workload takes as input a range query window and GeoViz Query 1 in this workload repeats 5 times using this query window. We also compare GEOSPARKVIZ with SparkViz and HadoopViz.

As shown in Fig. 15, the execution time of all three compared approaches increases with the growth of query window area. However, SparkViz and HadoopViz cost more and more time on larger query area while the time cost of GEOSPARKVIZ increases slowly. On the largest query area, 64*320 km^2, GEOSPARKVIZ is around 50% faster than SparkViz and 10 times faster than HadoopViz. This makes sense because GEOSPARKVIZ optimizer decides to first pixelize spatial objects and cache aggregated pixels into memory. The rest 4 GeoViz Query 1 in this workload work on the cached pixels and plot qualified pixels directly. It is worth noting that, for the smallest query window 320 km^2, GEOSPARKVIZ achieves the same performance with SparkViz because GEOSPARKVIZ optimizer actually chooses the naive plan shown in Fig. 9, which is same with SparkViz and HadoopViz. The optimized plan is not good for very small query windows (only return a few qualified records) because it has to pixelize all objects no matter how many objects are needed for visualization. But GEOSPARKVIZ optimizer is able to figure out the fastest plan for the given scenario.



(a) Range + Viz (b) Range + Join + Viz

Fig. 18 Impact of map zoom level

8.5 Effect of parallelism

To demonstrate the parallel efficiency of GEOSPARKVIZ, we test its performance on different cluster settings. We vary the number of CPU cores to be 6 cores, 12 cores and 24 cores. In order to make sure there is enough memory for processing intermediate data, we only change the number of CPU cores registered in Apache Spark without changing the number of workers.

As depicted in Figs. 16 and 17, the time cost of GEOSPARKVIZ increases as we decrease the number of cores in the cluster. This makes sense due to the fact that a larger cluster (more CPU cores) can process more tasks in parallel.

On the other hand, we runs all experiments on four main datasets, Roads, Edges, NYCtaxi and OSMpoint (see Figs. 16, 17). The latter two datasets have over 1 billion records which are around 30 times larger than Edges and 70 times larger than Roads. The experiments also show that the time spent on 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. This 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 check spatial query predicate and pixelize takes more time due to their complex geometrical shape.

8.6 Impact of map zoom level

Figure 18 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 16 tiles, 1 million pixels; level 4 stands for 256 tiles, 16 million pixels; level 6 demands 4096 tiles, 268 million pixels. We run the two GeoViz workloads (i.e., MutliRange+map building and Range+Join+Viz) on these map levels and produce scatter plots map visualization. As shown in Fig. 18, the higher the zoom level, the more time GEOSPARKVIZ takes to execute the GeoViz query. This makes sense because, with smaller zoom levels, GeoSparkViz only generates low resolution map tiles. In this case, the pixlelize, pixel aggregate, and rendering operators process fewer pixels.

9 Conclusion and future work

In this paper, we presented GeoSparkViz, a cluster computing system for visualizing massive-scale geospatial data. The proposed approach pushes the geospatial map visualization functionality inside the core engine of a distributed/parallel data management system. The GeoSparkViz approach allows users to declaratively define geosaptial visual analytics (GeoViz) tasks. Thanks to its hybrid nature, the system adopts a GeoViz-aware spatial partitioning scheme and execution strategies that co-optimize the map visualization operations with spatial query operators. Experiments based on real spatial data show that GeoSparkViz can achieve up to one order of magnitude less data-to-visualization time compared to its counterparts. In the future, we plan to incorporate classic database query optimization strategies, e.g., materializing pixel aggregates, to reduce the data-to-visualization time even more. We also plan to integrate GeoSparkViz with recently developed declarative visualization libraries (e.g., Vega-Lite [27], Reactive Vega [28]) as well as sampling-based systems (e.g., ScalaR [4], RS-Tree [33]) to support interactive GeoViz operations (e.g., Zoom-In/out).

A Additional GeoViz SQL specification

This section includes an additional specification about GeoViz SQL. This is complementary to the content in Sect. 4. We also give more examples to demonstrate how to assemble map effects.

A.1 Type and function specification

GeoViz SQL allows declarative SQL-like queries over structured RDDs. Each RDD has a schema which consists of a number of attributes. Each attribute has a type in Spark.

A.1.1 Types

GeoSparkViz adds two new types of objects: pixels and image. This way, Spark can understand and manipulate data for maps. In addition, GeoSpark itself adds a new type in Spark called Geometry to represent geospatial data.

Geometry [40] This is a generic data type which internally represents a variety of spatial objects, such as points, line strings, and polygons. It has several fields such as coordinates.

Pixel This type extends the Geometry type to support pixels and hence spatial query operators can process it directly. It is used by several map building operators: Pixel, Pixel aggregate and Render. Besides the original fields in Geometry, it has several additional fields: (1) resolution (2) tile id. A pixel can be considered as a point object.

Image This type is a serializable wrapper of Java BufferedImage class and actually holds the map tile data. It provides serialization functions to BufferedImage. Each map tile in GeoSparkViz is an Image type object.

A.1.2 Functions

ST_TileId Each pixel in GeoSparkViz has several internal attributes. The tile ID of a pixel is used to partition the pixels properly.

```
SELECT ST_TileID(pixel)
FROM   Pixel_view
```

- **Input** The function takes as input a pixel attribute.
- **Output** It returns the tile ID of this pixel. The ID is a string type object.

ST_EncodeImage This function returns the base64 string representation of an image. This is a specific function for the server-client environment. For example, some client libraries such as Apache Zeppelin can directly display base64 images.

```
SELECT ST_EncodeImage(map_tile)
FROM   Map_tile_view
```

- **Input** The function takes as input an image attribute.
- **Output** It returns a base64 string of the image.

A.2 Additional GeoViz query examples

In this section, we provide more examples about how to assemble GeoViz queries. Another example, scatter plot of taxi trip pickup points, can be found in Sect. 4.2.

Spatial dataset We use the NYC taxi trip dataset mentioned in Fig. 5 as the running example in this section. The dataset is loaded into a structured Spatial RDD.

Heat map of taxi trip pick up points This shows a heat map of the distribution of pickup points of taxi trips. The color is in proportion to the density of pickup points. The max weight is 100 which means: if there are more than 100 trips picked up in a place, this place shows red color. The initial weight in Pixelize operator is 1 and the aggregation strategy is count().

```
/* Render: Render tiles */
SELECT Tile_ID, ST_Render(pixel, color)
FROM {
  /* Render: Colorize pixels */
  SELECT pixel, ST_Colorize(weight, 100) as color
  FROM {
    /* Pixel aggregate */
```

```
      SELECT pixel, count(*)
      FROM {
        /* Pixelize */
        SELECT pixel, 1
        FROM   NYCtaxi taxi
        LATERAL VIEW  ST_Pixelize(taxi.pickup,
          16384, 16384) AS pixel }
      GROUPBY pixel } }
  GROUPBY ST_TileID(pixel) AS Tile_ID
```

Single-image map of taxi trip pick up points This shows a heat map of the distribution of pickup points of taxi trips in a map image which does not have any tiles. Other parameters are the same as the previous one. This is similar to the queries shown in Fig. 4.

```
/* Render: Render tiles */
SELECT Tile_ID, ST_Render(pixel, color)
FROM {
  /* Render: Colorize pixels */
  SELECT pixel, ST_Colorize(weight, 100) as color
  FROM {
    /* Pixel aggregate */
    SELECT pixel, count(*)
    FROM {
      /* Pixelize */
      SELECT pixel, 1
      FROM   NYCtaxi taxi
      LATERAL VIEW  ST_Pixelize(taxi.pickup,
      16384, 16384) AS pixel
      }
    }
  }
```

Heat map of trip fare This shows a heat map of trip fare. If trips picked up from a place cost more money, this place will show a red color. The max weight is 30 which means: if trips from a place cost more than 30 dollars, this place will show a red color. The initial weight in the Pixelize operator is the "trip fare" attribute and the aggregation strategy is avg().

```
/* Render: Render tiles */
SELECT ST_Render(pixel, color) as map
FROM {
  /* Render: Colorize pixels */
  SELECT pixel, ST_Colorize(weight, 30) as color
  FROM {
    /* Pixel aggregate */
    SELECT pixel, avg(initial_weight) as weight
    FROM {
      /* Pixelize */
      SELECT pixel, trip_fare as initial_weight
      FROM   NYCtaxi taxi
      LATERAL VIEW  ST_Pixelize(taxi.pickup,
        16384, 16384) AS pixel }
      GROUPBY pixel } }
GROUPBY ST_TileID(pixel) AS Tile_ID
```

References

1. Aji, A., Wang, F., Vo, H., Lee, R., Liu, Q., Zhang, X., Saltz, J.H.: Hadoop-GIS: a high performance spatial data warehousing system over MapReduce. Proc. VLDB Endow. PVLDB **6**(11), 1009–1020 (2013)
2. Armbrust, M., Xin, R.S., Lian, C., Huai, Y., Liu, D., Bradley, J.K., Meng, X., Kaftan, T., Franklin, M.J., Ghodsi, A., Zaharia, M.: Spark SQL: relational data processing in spark. In: Sellis, T.K., Davidson, S.B., Ives, Z.G. (eds.) Proceedings of the ACM International Conference on Management of Data, SIGMOD, pp. 1383–1394. ACM (2015)
3. Baig, F., Mehrotra, M., Vo, H., Wang, F., Saltz, J.H., 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, pp. 134–146 (2015)
4. Battle, L., Stonebraker, M., Chang, R.: Dynamic reduction of query result sets for interactive visualizaton. In: *Proceedings of International Conference on Big Data, BigData*, pp. 1–8 (2013)
5. Crotty, A., Galakatos, A., Zgraggen, E., Binnig, C., Kraska, T.: Vizdom: interactive analytics through pen and touch. Proc. VLDB Endow. PVLDB **8**(12), 2024–2027 (2015)
6. de Lara Pahins, C.A., Stephens, S.A., Scheidegger, C., Comba, J.L.D.: Hashedcubes: simple, low memory, real-time visual exploration of big data. IEEE Trans. Vis. Comput. Graph. TVCG **23**(1), 671–680 (2017)
7. Eldawy, A., Alarabi, L., Mokbel, M.F.: Spatial partitioning techniques in spatial hadoop. Proc. VLDB Endow. PVLDB **8**(12), 1602–1605 (2015)
8. Eldawy, A., Mokbel, M.F.: A demonstration of spatialhadoop: an efficient mapreduce framework for spatial data. Proc. VLDB Endow. PVLDB **6**(12), 1230–1233 (2013)
9. Eldawy, A., Mokbel, M.F., Alharthi, S., Alzaidy, A., Tarek, K., Ghani, S.: Shahed: a mapreduce-based system for querying and visualizing spatio-temporal satellite data. In: Proceedings of the International Conference on Data Engineering, ICDE, pp. 1585–1596. IEEE (2015)
10. Eldawy, A., Mokbel, M.F., Jonathan, C.: Hadoopviz: a mapreduce framework for extensible visualization of big spatial data. In: Proceedings of the International Conference on Data Engineering, ICDE, pp. 601–612. IEEE (2016)
11. Earthdata Cloud Evolution. http://www.naturalearthdata.com/downloads/
12. Guo, T., Feng, K., Cong, G., Bao, Z.: Efficient selection of geospatial data on maps for interactive and visualized exploration. In: Proceedings of the ACM International Conference on Management of Data, SIGMOD, pp. 567–582. ACM (2018)
13. Apache Hadoop. http://hadoop.apache.org/
14. Hughes, J.N., Annex, A., Eichelberger, C.N., Fox, A., Hulbert, A., Ronquest, M.: Geomesa: a distributed architecture for spatio-temporal fusion. In: *SPIE Defense+Security*, pp. 94730F–94730F. International Society for Optics and Photonics (2015)
15. Kefaloukos, P.K., Salles, M.A.V., Zachariasen, M.: Declarative cartography: in-database map generalization of geospatial datasets. In: Proceedings of the International Conference on Data Engineering, ICDE, pp. 1024–1035. IEEE (2014)
16. Kini, A., Emanuele, R.: Geotrellis: adding geospatial capabilities to spark. Spark Summit (2014)
17. Lins, L., Klosowski, J.T., Scheidegger, C.: Nanocubes for real-time exploration of spatiotemporal datasets. IEEE Trans. Vis. Comput. Graph. TVCG **19**(12), 2456–2465 (2013)
18. Liu, Z., Jiang, B., Heer, J.: immens: real-time visual querying of big data. In: Computer Graphics Forum, vol. 32, pp. 421–430. Wiley Online Library (2013)
19. Apache Livy. https://livy.apache.org/
20. Lu, J., Güting, R.H.: Parallel secondo: boosting database engines with hadoop. In: International Conference on Parallel and Distributed Systems, ICPADS, pp. 738–743. IEEE (2012)
21. Mahdian, M., Schrijvers, O., Vassilvitskii, S.: Algorithmic cartography: placing points of interest and ads on maps. In: Proceedings of the International Conference on Knowledge Discovery and Data Mining, SIGKDD, pp. 755–764 (2015)

22. Mostak, T.: An overview of MAPD (massively parallel database). White paper, Massachusetts Institute of Technology (2013)
23. OpenStreetMap. Map Zoom Level. http://wiki.openstreetmap.org/wiki/Zoom_levels
24. Park, Y., Cafarella, M.J., Mozafari, B.: Visualization-aware sampling for very large databases. In: Proceedings of the International Conference on Data Engineering, ICDE, pp. 755–766. IEEE (2016)
25. Rahman, S., Aliakbarpour, M., Kong, H., Blais, E., Karahalios, K., Parameswaran, A.G., Rubinfeld, R.: I've seen "enough": incrementally improving visualizations to support rapid decision making. Proc. VLDB Endow. PVLDB **10**(11), 1262–1273 (2017)
26. Sarma, A.D., Lee, H., Gonzalez, H., Madhavan, J., Halevy, A.Y.: Efficient spatial sampling of large geographical tables. In: Proceedings of the ACM International Conference on Management of Data, SIGMOD, pp. 193–204 (2012)
27. Satyanarayan, A., Moritz, D., Wongsuphasawat, K., Heer, J.: Vega-lite: a grammar of interactive graphics. IEEE Trans. Vis. Comput. Graph. TVCG **23**(1), 341–350 (2017)
28. Satyanarayan, A., Russell, R., Hoffswell, J., Heer, J.: Reactive vega: a streaming dataflow architecture for declarative interactive visualization. IEEE Trans. Vis. Comput. Graph. TVCG **22**(1), 659–668 (2016)
29. Scarsella, A., Stofega, W.: Worldwide Smartphone Forecast 2020–2024. Technical report, International Data Corporation (IDC) (2020). https://www.idc.com/getdoc.jsp?containerId=US46135620
30. Apache Spark. http://spark.apache.org/
31. Su, S., An, M., Perry, V., Jia, J., Kim, T., Chen, T., Li, C.: Visually analyzing A billion tweets: an application for collaborative visual analytics on large high-resolution display. In: Proceedings of International Conference on Big Data, BigData, pp. 3597–3606 (2018)
32. Tang, M., Yu, Y., Malluhi, Q.M., Ouzzani, M., Aref, W.G.: LocationSpark: a distributed in-memory data management system for big spatial data. Proc. VLDB Endow. PVLDB **9**(13), 1565–1568 (2016)
33. Wang, L., Christensen, R., Li, F., Yi, K.: Spatial online sampling and aggregation. Proc. VLDB Endow. PVLDB **9**(3), 84–95 (2015)
34. Weibel, R., Dutton, G.: Generalising spatial data and dealing with multiple representations. Geograph. Inf. Syst. **1**, 125–155 (1999)
35. Wu, E., Battle, L., Madden, S.R.: The case for data visualization management systems. Proc. VLDB Endow. PVLDB **7**(10), 903–906 (2014)
36. Xie, D., Li, F., Yao, B., Li, G., Zhou, L., Guo, M.: Simba: efficient in-memory spatial analytics. In: Proceedings of the ACM International Conference on Management of Data, SIGMOD, pp. 1071–1085. ACM (2016)
37. Yu, J., Sarwat, M.: 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 (2017)
38. Yu, J., Sarwat, M.: Turbocharging geospatial visualization dashboards via a materialized sampling cube approach. In: Proceedings of the International Conference on Data Engineering, ICDE, pp. 1165–1176. IEEE (2020)
39. Yu, J., Zhang, Z., Sarwat, M.: Geosparkviz: a scalable geospatial data visualization framework in the apache spark ecosystem. In: Proceedings of the International Conference on Scientific and Statistical Database Management, SSDBM, pp. 15:1–15:12 (2018)
40. Yu, J., Zhang, Z., Sarwat, M.: Spatial data management in apache spark: the GeoSpark perspective and beyond. GeoInformatica **23**(1), 37–78 (2019)