GeoSparkViz in Action: A Data System with built-in support for Geospatial Visualization

Jia Yu [1], Anique Tahir [2], Mohamed Sarwat [3]
School of Computing, Infomatics, and Decision Systems Engineering
Arizona State University
699 S Mill Avenue, Tempe, AZ 85281
[1]jiayu2@asu.edu, [2]artahir@asu.edu, [3]msarwat@asu.edu

***Abstract*—Visualizing data on maps is deemed a powerful tool for data scientists to make sense of geospatial data. The geospatial map visualization (abbr. MapViz) process first loads the designated geospatial data, processes the data and then applies the map visualization effect. Guaranteeing detailed and accurate geospatial MapViz (e.g., at multiple zoom levels) requires extremely high-resolution maps. Classic solutions suffer from limited computation resources while scalable MapViz system architectures are not able to co-optimize the data management and visualization phases in the same system. This paper demonstrates GEOSPARKVIZ, a full-fledged system that allows the user to load, prepare, integrate and execute MapViz tasks in the same system. For demonstration purpose, we implemented a web interface using a node.js web server, Baidu echarts library, and MapBox on top of GEOSPARKVIZ to visually explore patterns in the New York City Taxi Trips dataset. The demonstration scenarios show how the data preparation and map visualization phases are combined in GEOSPARKVIZ.**

I. INTRODUCTION

Visualizing data on maps is deemed a powerful tool for data scientists to make sense of geospatial data. For example, a heat map of New York City taxi trips helps the NYC taxi and limousine company determine where the hot pick-up and drop-off locations are. A scatter plot of the worldwide road networks exposes isolated areas around the world. Another example is that a political campaign manager may leverage a choropleth map of Twitter sentiment analysis in each US county after each political debate. All aforementioned applications need to employ a scalable system that can visualize big spatial data.

A geospatial map visualization system faces many challenges. First, the massive scale of available spatial data hinders classic systems from generating geospatial maps at scale. In addition, the user needs to see visualized result quickly even for large-scale spatial analytics (e.g., spatial data mining, geospatial analysis). Many commercial map services such as Google Maps and MapBox only allow users to visualize a small amount of spatial data on a single machine. Recent solutions allow user visualize large-scale data by compressing or sampling the spatial data but they are not able to provide high-quality map images for the user (e.g., the multiple zoom levels, giga-pixels).

MapReduce-based spatial data visualization systems, e.g., [1], are scalable but these methods may still take long run time to generate high-quality maps. That happens due to the fact that MapReduce-based system, e.g., Hadoop, rely on HDFS to store intermediate results and do not provide an efficient way to process the data in memory. State-of-the-art big spatial data management systems can perform queries at scale [2] but do not provide in-house support for spatial map visualization. Furthermore, existing system architectures decouple the data processing and map visualization phases. For instance, the data scientist may use a data system for loading, processing and querying data, and a visualization tool, e.g., Tableau, for visualizing the map. The decoupled approach demands substantial overhead to connect the data management system to the map visualization tool.

The paper demonstrates GEOSPARKVIZ [4], a data system that provides built-in support for geospatial visualization (MapViz). The system performs the data processing and map visualization phases in the same data system, which leads to three main benefits: (1) It provides a holistic approach that allows the data scientist to load, process, query and visualize spatial data. That plug-and-play approach increases the usability of the system. (2) It reduces the overhead of loading the intermediate spatial data generated from the data processing phase to the designated map visualization tool. (3) It allows the system to co-optimize classic query operators (e.g., selection) and visualization operators side-by-side leading to a more interactive visualization.

We implemented a prototype of GEOSPARKVIZ in a Spark-based spatial data management system, GEOSPARK [5]. For demonstration purposes, we use the NYC Taxi Trips dataset. It contains NYC taxi trip records between January 2009 and December 2016. Each record includes pick-up and dropoff location coordinates, dates/times, trip distances, itemized fares, tip amount. In the demo, we visualize patterns in the spatial dataset and show how the data processing and map visualization phases are executed and co-optimized in GEOSPARKVIZ.

II. SYSTEM OVERVIEW

Figure 1 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 MapViz language using a set of predefined MapViz effects (scatter plot, heat map, choropleth map,...), as follows: The user specifies the input spatial data attribute in the `SELECT` clause. The user can specify the input data table in the `FROM` clause.

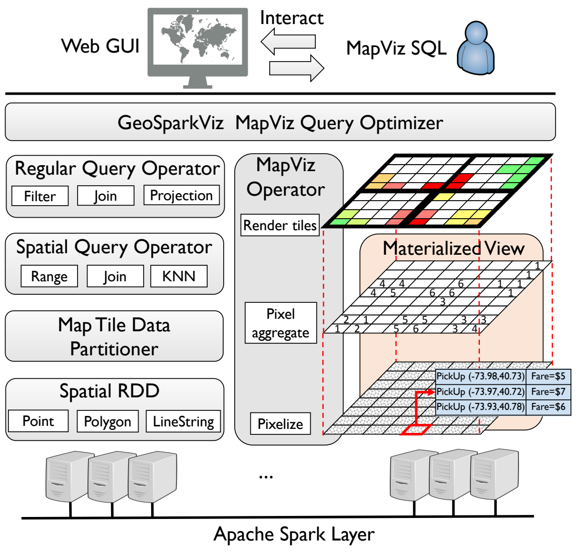


Fig. 1: GEOSPARKVIZ Overview

The input table(s) must consist of at least one spatial attribute (e.g., point, polygon). The `WHERE` clause can support classic spatial query predicates (e.g., necessary for the data processing phase) such as `ST_CONTAINS` and relational predicates such as `=, >, <`. The subset of the geospatial data that satisfies the query predicate will then be visualized using the MapViz effect stated in the `SELECT` clause. Moreover, the user can also pass non-spatial attributes along with spatial objects to the MapViz function. The syntax is as follows:

```
SELECT  [MapViz name]([Dataset].[Attributes])
FROM    [Spatial Dataset]
WHERE   [Where clause]
```

The system then processes the MapViz SQL query and returns the final map tiles / pixels to the user. To achieve that, GEOSPARKVIZ consists of the following components:

Visualization Operators: GEOSPARKVIZ breaks down the map visualization pipeline into a sequence of visualization operators (namely, Pixelize, Pixel Aggregate, and Render). Pixelize transforms spatial objects to pixels and groups the objects by pixels (a spatial object may be pixelized to multiple pixels and a pixel may be associated with multiple objects). Pixel Aggregate aggregates the selected attribute of grouped objects (e.g., Average(FareAmount)) and the aggregated values become the weights of pixels. Render determines the pixel colors based on the pixel weights. The system parallelizes the execution of each operator among the cluster nodes. In addition, GEOSPARKVIZ exposes the visualization operators to the user through the declarative MapViz language. The user can easily declare a new map visualization effect in a SQL statement. For instance, the user can define new coloring rules and pixel aggregation rules.

Materialized Views: For a given MapViz SQL query, GEOSPARKVIZ can materialize two levels of database views, cache them in memory and manage these views through the system catalog. The MapViz optimizer will then utilize the materialized views to speed up the upcoming MapViz queries if possible. The two levels of materialized views are: (1) Pixel view: generated by running the Pixelize operator. The schema of this view is ⟨`pixel, ListOf(SpatialObject)`⟩. Each spatial object may have many attributes such as coordinates, fare amount, trip distance, and so on. (2) Pixel aggregate view: that is generated using the Pixel aggregate operator. The schema of this view is ⟨`pixel, weight`⟩. The weight is the aggregated value of an attribute selected by the user. The user may choose to manually create and materialize the aforementioned views. The user can specify the materialization level, as follows: 0 - None; 1 - PixelViewOnly; 2 - PixelAggregateViewOnly; 3 - All.

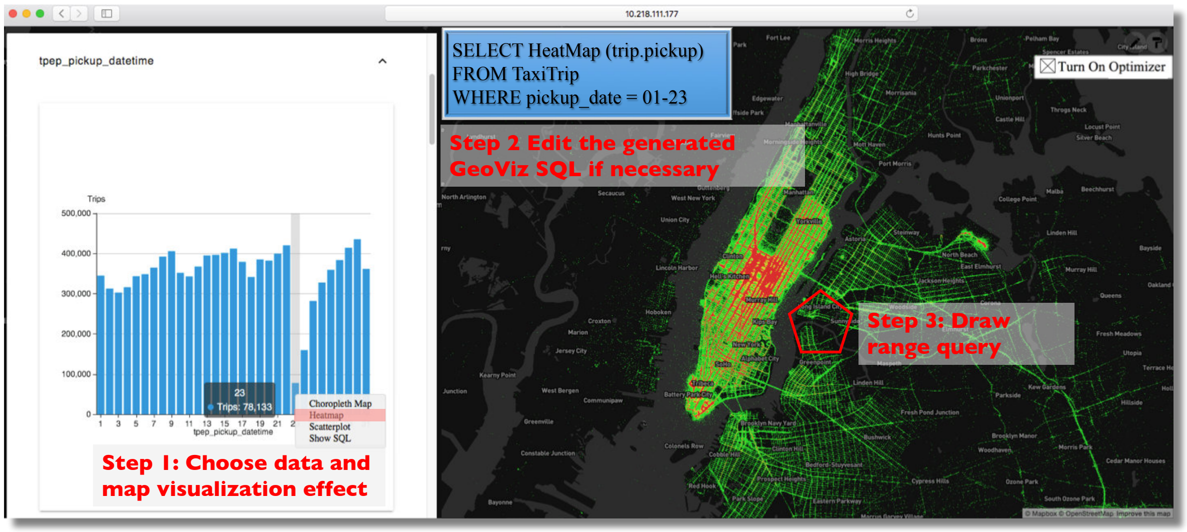
```
CREATE MapViz VIEW [Name] AS [MapViz query]
  WITH materialization_level = [0 | 1 | 2 | 3]
```

Map Tile Data Partitioner: GEOSPARKVIZ employs a partitioner operator that partitions a given spatial dataset across the cluster. Spatial objects that fall inside a logical map tile boundary go to the same physical data partition and stay at the same machine. Therefore, the system can easily render a map tile using pixels from the same data partition. The partitioner accommodates map visual constraints and also balances the load among cluster nodes when processing skewed spatial data. On one hand, it ensures that each data partition contains roughly the same number of spatial objects and pixels to achieve load balancing. On the other hand, the logical space partition boundary of each data partition is derived from a map tile space partition of the final map. That way, GEOSPARKVIZ can easily stitch the data partitions that belong to the same tile together and render map tiles efficiently.

Query Optimizer: The optimizer takes as input a MapViz query, utilizes the cached materialized views and figures out an execution plan that co-optimizes the map visualization operators and query operators. A MapViz query with `CREATE MapViz VIEW` clause always follows the straightforward decoupled plan (data preparation then map visualization) because it needs to materialize all two levels of views for future use. For upcoming queries, the optimizer will try to utilize the materialized views as many times as possible such that produce a more efficient execution plan (see Figure 4). For instance, if the MapViz query works on the same spatial region but uses a different `WHERE` clause, the optimizer will decide to utilize the cached Pixel view and skip the Pixel operator. If the query uses the spatial query predicate `ST_WITHIN` to require a partially different spatial area, the optimizer will produce a straightforward plan that goes down to the raw spatial data but retrieves the missing raw spatial data and still utilizes the cached materialized view to render the map tiles. In other words, unless the target spatial region is totally different from the materialized views, the optimizer will decide not to go down to the raw data. Otherwise, it will instruct the system to perform data processing and map visualization from scratch.

III. DEMONSTRATION SCENARIO

We implemented a client side web applications on the top of GEOSPARKVIZ to visually explore the NYC taxi trips. All needed datasets, such as taxi trips, have been pre-loaded into the cluster. The client application consists of two parts. The first part is implemented as a node.js web server that takes in a request for visualization and

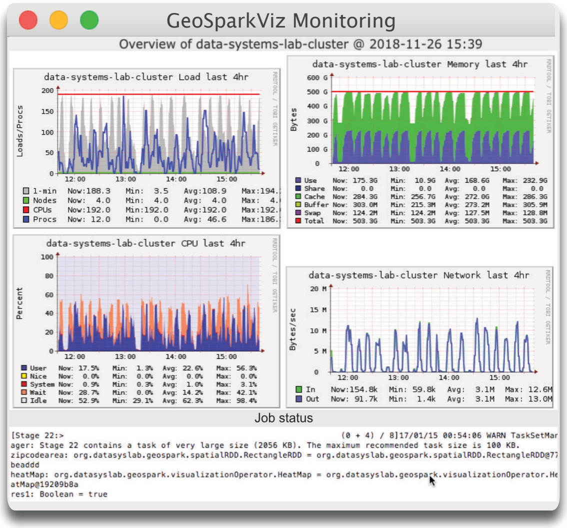

SELECT HeatMap (trip.pickup)
FROM TaxiTrip
WHERE pickup_date = 01-23

Step 2 Edit the generated GeoViz SQL if necessary

Step 3: Draw range query

Step 1: Choose data and map visualization effect

Fig. 2: Taxi pickup points heat map on a snowy day



GeoSparkViz Monitoring
Overview of data-systems-lab-cluster @ 2018-11-26 15:39
data-systems-lab-cluster Load last 4hr
data-systems-lab-cluster Memory last 4hr
data-systems-lab-cluster CPU last 4hr
data-systems-lab-cluster Network last 4hr
Job status

```
[Stage 22:>                                         (0 + 4) / 8]17/01/15 00:54:06 WARN TaskSetMan
ager: Stage 22 contains a task of very large size (2056 KB). The maximum recommended task size is 100 KB.
zipcodearea: org.datasyslab.geospark.spatialRDD.RectangleRDD = org.datasyslab.geospark.spatialRDD.RectangleRDD@77
beaddd
heatMap: org.datasyslab.geospark.visualizationOperator.HeatMap = org.datasyslab.geospark.visualizationOperator.He
atMap@19209b8a
res1: Boolean = true
```

Fig. 3: GeoSparkViz backend monitoring panel

executes a MapViz SQL query in GeoSparkViz to create the map tile(s) raster image(s). The second part is a web GUI that gives the demo attendee several map visualization options. In addition, the demo attendee can monitor the backend cluster status via the monitoring panel (see Figure 3).

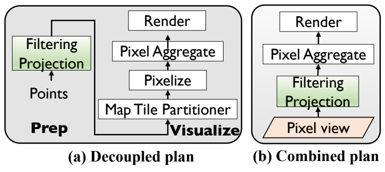


Fig. 4: Visualizing range queries

The web GUI has two parts: (1) Left Panel: shows a set of histograms that represent statistics describing the demonstrated dataset (e.g., the number of taxi trips that happened on each day). To generate the histograms, we use Baidu echarts library. The demo attendee can choose any of the histograms to start with. (2) Geospatial Map: depicts a map of the target spatial area and the user-selected MapViz effect (i.e., MapViz SQL query) on the map. During the demo, the audience will fully interact with the histograms on the left panel and the map. We use MapBox to load the map tiles generated by GeoSparkViz and stitch them to the earth background. The demo attendee can also uncheck ”Turn On Optimizer” button to experience the speed without the help of materialized MapViz views and the optimizer. We give two demonstration scenarios, described below:

Scenario I: Drop in Taxi Pickups on January 23rd 2016: In this scenario, the demo attendee selects the Number of Trips per Day histogram. As given in Figure 2, the demo audience can first create a heat map on the taxi trips in 2016 with `CREATE MapViz VIEW` clause to understand the overall trend. Then he right clicks on any bar in the histogram bar chart and ask GeoSparkViz for a heat map on the selected taxi pickup points (Step 1). The demo attendee will also see the MapViz SQL query used by GeoSparkViz to generate the heatmap as shown in the text box of Figure 2. The generated MapViz SQL consists of classic SQL operators (e.g., Selection predicates) as well as a HeatMap operator.The demo attendee can manually edit the generated SQL. GeoSparkViz will process the edited MapViz SQL and generate a new map visualization. The attendee can easily recognize that the number of taxi trips dramatically decreased on January 23rd 2016 and started to slowly pick up again on the 24th and 25th. According to the weather report, NYC had a severe snow storm on that day. As shown in the heat map, Manhattan (as opposed to other areas in NYC) Taxi commute is still active on January 23rd 2016 in spite of the blizzard.

```
SELECT HeatMap(Trips.pickup)
FROM Trips
WHERE pickup_date = 01-23-2016
    AND ST_Contains(TimeSquare, pickup)
```

To further investigate the traffic on the day of January

23^{rd} 2016, the demo attendee can draw a spatial range query polygon on any geographical area of choice (Step 3 in Figure 2) on the map. This will ask the system to give more details about the geographical distribution of taxi trips in the selected area on that day. For the first MapViz query with `CREATE MapViz VIEW`, GeoSparkViz uses the decoupled plan in Figure 4. Other MapViz queries produced the interaction follows the combined plan Figure 4b which leverages the materialized pixel view.

Scenario II: Who is paying more tips in NYC? The NYC taxi and limousine company divides the New York City area into 260 polygonal zones and each taxi zone refers to a street block such as Times Square. A data scientist may study the average tip percentage in each taxi zone and use a Choropleth Map to plot these zones (the color of a zone is in proportion to the average tip percentage in this zone). In other words, the demo attendee can explore the pick up locations where customers pay more tips to taxi drivers.

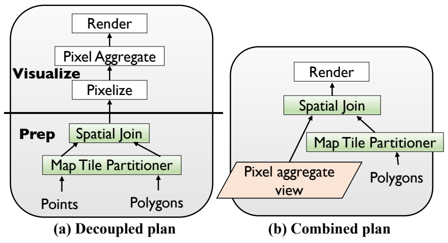
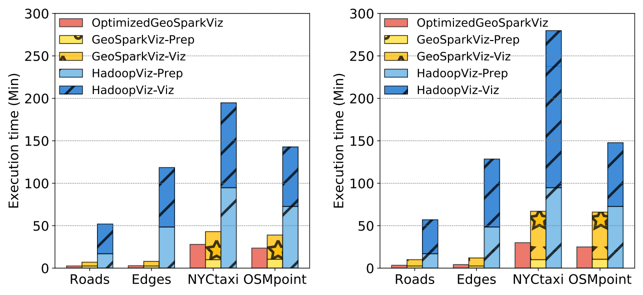


Fig. 5: Visualizing join queries

This MapViz query still utilizes the MapViz views materialized in Scenario I. The demo attendee will then write the following MapViz SQL statement:

```
SELECT ChoroplethMap(Zones.geom,
           average(tip_amount/total_amount))
FROM   Trips, Zones
WHERE  ST_Contains(Zones.geom, Trips.pickup)
       AND pickup_date = 2016
```

As depicted in the MapViz SQL statement above, the demo attendee first runs a Spatial Join Query between zones and taxi trip pickup points to find the trips that lie within the area of each taxi zone and then calculates the average tip percentage. The result is passed to GeoSparkViz optimizer. After taking the SQL statement above, GeoSparkViz yields an efficient execution plan. The straightforward decoupled plan (see Figure 5) first completes the spatial join query, pixelize the polygons and then renders each map tile. It introduces two data shuffling operations, one for the taxi trip dataset and one for the zone dataset (distributed spatial join requires a spatial data partitioning). A data shuffle sends lots of data across the cluster and significantly increases the query and visualization latency. However, GeoSparkViz follows two optimization strategies that will be demonstrated to the demo attendee in the system back-end: (1) *Utilize materialized MapViz views*: This strategy allows the zone dataset to join pixels directly such that it utilizes the materialized Pixel aggregate view to reduce the data scale of the taxi trip datasets. Due to the small size of the aggregated pixel information, the scale of the data shuffles is decreased (see Figure 5). (2) *Skip unnecessary operators*: Similar to Scenario I, GeoSparkViz uses a map tile data partitioner to only partition the zone dataset. The other data shuffle introduced by partitioning taxi trip pickup points is skipped to decrease the MapViz latency.



(a) Scatter plot (b) Heat map

Fig. 6: Performance of MultiRange + MapViz

Experiments. We compare the optimized GeoSparkViz, regular GeoSparkViz and HadoopViz, the counterpart in Hadoop ecosystem on a 3-node cluster (1 master and 2 workers, each has a 12-core CPU and 100 GB memory). Four real spatial datasets are tested: (1) *TIGER Roads*: the line string shapes of 20 million roads in the US. (2) *TIGER Edges*: the line string shapes of 73 million edges (i.e., roads, rivers, rails) in the US. (3) *New York Taxi* [3] (4) *OpenStreetMap Point*: 1.7 billion spatial points on the planet. We issue a spatial range query and visualize its result to either scatter plot or heat map. The query is performed five times in this workload using five different spatial range predicates. Optimized GeoSparkViz executes the optimized plan depicted in Figure 4(b) and the straightforward plan depicted in Figure 4(a).

As shown in Figure 6 ("prep" - load, process and query time, "viz" - visualization time), optimized GeoSparkViz is 50% faster than regular GeoSparkViz and up to an order of magnitude faster than HadoopViz for generating scatter plot visualization. The results makes sense because: (1) The GeoSparkViz execution plan (see Figure 4) first pixelizes spatial objects to pixels and caches them into memory, and all spatial range predicates run directly on the cached pixel dataset. (2) The MapViz partitioner in GeoSparkViz is more load-balanced than the data partitioner in HadoopViz. The map tile method just repartitions the space to uniform grids and does not take into account data skewness. (3) In contrast to Spark-based systems, HadoopViz reads/writes intermediate data on disk. On small datasets like Roads and Edges, GeoSparkViz runs $8X$ faster than HadoopViz because, GeoSparkViz can process all intermediate data in memory.

References

[1] A. Eldawy, M. F. Mokbel, and C. Jonathan. Hadoopviz: A mapreduce framework for extensible visualization of big spatial data. In *ICDE*, pages 601–612, 2016.
[2] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo. Simba: Efficient in-memory spatial analytics. In *SIGMOD*, pages 1071–1085, 2016.
[3] J. Yu and M. Sarwat. Indexing the pickup and drop-off locations of NYC taxi trips in postgresql - lessons from the road. In *SSTD*, pages 145–162, 2017.
[4] J. Yu, Z. Zhang, and M. Sarwat. Geosparkviz: a scalable geospatial data visualization framework in the apache spark ecosystem. In *SSDBM*, pages 15:1–15:12, 2018.
[5] J. Yu, Z. Zhang, and M. Sarwat. Spatial data management in apache spark: The geospark perspective and beyond. *Geoinformatica*, 2018.