HadoopViz: A MapReduce Framework for Extensible Visualization of Big Spatial Data

Ahmed Eldawy Mohamed F. Mokbel Christopher Jonathan
Department of Computer Science and Engineering, University of Minnesota
{eldawy,mokbel,cjonathan}@cs.umn.edu

***Abstract*—This paper introduces HadoopViz; a MapReduce-based framework for visualizing big spatial data. HadoopViz has three unique features that distinguish it from other techniques. (1) It exposes an extensible interface which allows users to define a new visualization types, e.g., scatter plot, road network, or heat map, by defining five abstract functions, without delving into the implementation details of the MapReduce algorithms. As it is open source, HadoopViz allows algorithm designers to focus on how the data should be visualized rather than performance or scalability issues. (2) HadoopViz is capable of generating big images with giga-pixel resolution by employing a three-phase technique, *partition-plot-merge*. (3) HadoopViz provides a *smoothing* functionality which can fuse nearby records together as the image is plotted. This makes it capable of generating more types of images with high quality as compared to existing work. Experimental results on real datasets of up to 14 Billion points show the extensibility, scalability, and efficiency of HadoopViz to handle different visualization types of spatial big data.**

I. Introduction

In recent years, there has been an explosion in the amounts of spatial data produced by several devices such as smart phones, space telescopes, medical devices, among others. For example, space telescopes generate up to 150 GB weekly spatial data [31], medical devices produce spatial images (X-rays) at a rate of 50 PB per year [12], a NASA archive of satellite earth images has more than 1 PB and increases daily by 25 GB [17], while there are 10 Million geotagged tweets issued from Twitter every day as 2% of the whole Twitter firehose [32]. Meanwhile, various applications and agencies need to process an unprecedented amount of spatial data. For example, the Blue Brain Project [21] studies the brain's architectural and functional principles through modeling brain neurons as spatial data [30]. Meteorologists study and simulate climate data through spatial analysis [13]. News reporters use geotagged tweets for event detection and analysis [27].

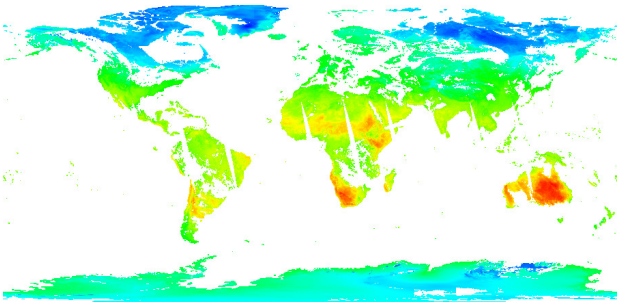
A major need for all these applications is the ability to visualize big spatial data by generating an image that provides a bird's-eye data view. Visualization is a very common tool that allows users to quickly spot interesting patterns which are very hard to detect otherwise. Examples of spatial data visualization include visualizing a world temperature heat map of NASA satellite data, a scatter plot of billions of tweets worldwide, a frequency heat map for Twitter data showing the hot spots of generated tweets, a road network for the whole world, or a network of brain neurons. In all these visualization examples, users should be able to zoom in and out in the generated image to get different resolutions of the whole data set.

This research is capitally supported by NSF grants IIS-0952977, IIS-1218168, IIS-1525953, CNS-1512877 and by AWS in Education Grant award

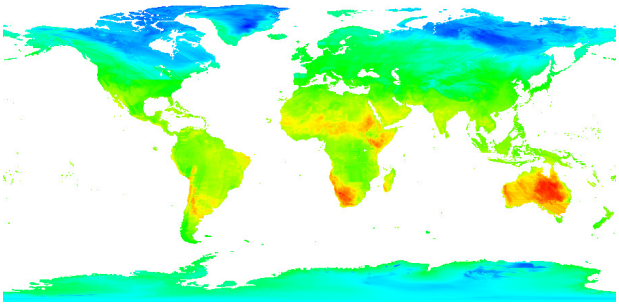
Figure 1(a) portrays an example of visualizing the heat map of world temperature in one month, with a total of 14 billion points. Traditional single-machine visualization techniques [7], [19], [22], [28] have limited performance, thus they take around 1 hour to visualize this data on a machine with 1TB of memory. GPUs can significantly speed up the processing [20], [23], yet they are still hindered by the limited memory resources of a single machine. Meanwhile, there exist three distributed algorithms for visualizing spatial data [9], [26], [33]. Two of them [26], [33] rely on a *pixel-level-partitioning* phase, which partitions and groups records by the pixel they affect, and then combines these records to calculate the color of that pixel, which takes 25 minutes on a 40-core cluster. The third technique [9], achieves better performance but it relies on an expensive preprocessing phase which limits its application. In general, these techniques suffer from three limitations: (a) They do not support a *smoothing* function to fuse nearby records together, which limits the image types they can generate. For example, Figure 1(a) contains white spots and strips due to missing values that need to be *smoothed* out. (b) The performance degrades with giga-pixel images due to the excessive number of pixels. (c) Each algorithm is tailored to a specific image type, e.g., satellite images [9], [26] or 3D triangles [33], and it cannot be used to visualize other kinds of big spatial data, e.g., scattered points, or road networks.

This paper presents HadoopViz, an extensible MapReduce-based framework for visualizing big spatial data. HadoopViz overcomes the limitations of existing systems as: (a) It applies a *smoothing* technique which allows it to produce more image types that require fusing nearby records together. For example, it produces the image in Figure 1(b) where missing values are smoothed out by interpolating nearby points. (b) It employs a three-phase approach, *partition-plot-merge*, where it automatically chooses an appropriate *partitioning* scheme to scale up to generate giga-pixel images. For example, it takes only 90 seconds to visualize the image in Figure 1(b). HadoopViz uses this approach to generate both *single-level* images with a fixed resolution, and *multi-level images* where users can zoom in and out. (c) It proposes a novel *visualization abstraction* which allows the same efficient core algorithms to be used with dozens of image types, such as scatter plot, road networks, or brain neurons. This allows users to focus only on designing how the desired image should look like, while HadoopViz is responsible of scaling it up to thousands of nodes.

Without HadoopViz, to equip a system with data visualization, one needs to implement an algorithm for visualizing satellite data [26], another algorithm for visualizing tweets [20], a third algorithm for heatmap visualization [9], and so on. This



(a) Takes one hour on a machine with 1TB memory, and 25 minutes on a 40-core cluster. All without the ability of recovering missing data



(b) Takes 90 seconds on HadoopViz running on a 40-core cluster and recovers missing data

Fig. 1. Temperature Heat Map of 14 Billion Points of NASA Satellite Data (Best viewed in colors)

is not practical as each technique has its own data structure and storage requirements. From a system point of view, the holistic and extensible approach of HadoopViz is very appealing and industry-friendly. One needs to realize it once in the system, then, a myriad of various forms of visualization types are immediately supported efficiently.

HadoopViz realizes this extensibility through five *abstract* functions where all visualization algorithms in HadoopViz are implemented using these functions, namely, `smooth`, `create-canvas`, `plot`, `merge`, and `write`. Once a user defines them, HadoopViz plugs these functions into its generic visualization algorithms to scale image generation to thousands of nodes and terabytes of data. (1) The optional `smooth` function can be used to fuse nearby records together, e.g., merge intersecting road segments or recover missing values in satellite data. (2) The `create-canvas` function initializes an empty drawing on which records are plotted, e.g., it initializes an in-memory image for drawing road network or a 2D histogram to create a heat map. (3) The `plot` function does the actual drawing of input records, e.g., it draws a line segment in a road network or updates the histogram according to a point location. (4) The `merge` function combines multiple canvases to form the final picture, e.g., it merges two images by blending pixel colors or merges two histograms by adding up the values in each entry. (5) The `write` function generates the final picture out of a canvas, e.g., it generates a colorful histogram out of a frequency map, or compresses a vector image into a standard vector image format.

This paper shows the extensibility of HadoopViz using six examples, *scatter plot*, *road network*, *frequency heat map*, *satellite heat maps*, *vectorized map*, and *countries borders*, all implemented using the five abstract functions. As HadoopViz is open source [14], users can refer to these case studies while providing their own image types, e.g., brain neurons or traffic data. In addition, users can reuse existing code or third party libraries in these abstract functions to scale them out. For example, we scale out the single-machine ImageMagick [15] package using HadoopViz which gives it a 48X performance boost. HadoopViz is extensively experimented with several real datasets including the world road network (165 million polylines), and NASA satellite data (14 billion points). HadoopViz efficient design allows it to visualize NASA dataset in 90 seconds. It also generates a video (composed of 72 frames) out of 1 trillion points in three hours on a 10-node cluster [1].

[1]Please refer to the generated video at http://youtu.be/-mRRBMBtDa0

The rest of this paper is organized as follows: Section II highlights related work. Sections III and IV describe HadoopViz algorithms for generating single and multilevel images, respectively. Section V describes HadoopViz visualization abstraction. Section VI shows six visualization case studies using that abstraction. Section VII provides an experimental evaluation. Finally, Section VIII concludes the paper.

II. Related Work

This section discusses related work to HadoopViz from the following three different angles:

Big Spatial Data. The explosion in the amounts of spatial data has led to a plethora of research in big spatial data that either focus on specific problems (e.g., range query, spatial join [37], and kNN join [18]), or on building full-fledged systems for processing big spatial data, e.g., Hadoop-GIS [1], $\mathcal{MD}$-HBase [24], SciDB [29], and SpatialHadoop [10]. Unfortunately, none of these systems provide efficient visualization techniques for big spatial data.

Big Data Visualization. Many systems were designed to visualize non-spatial big data (e.g., [3], [16], [34]–[36]) by downsizing the data, using sampling or aggregation, and then visualizing the downsized data on a single machine as a chart or histogram. For example, Ermac [36] suggests injecting the visualization algorithms in the database engine so that sampling and aggregation are done early in the query plan. Similarly, M4 [16] rewrites SQL queries taking into account the limited size of the generated image to perform aggregation inside the database and return a small result size. *Bin-summarise-smooth* [35] downsizes the data by *binning* (partitioning), *summarizing* (aggregation), and *smoothing*, while the downsized data is visualized on a single machine. Unfortunately, all these techniques are designed for *non-spatial* data and do not apply for *spatial* data visualization.

Spatial Data Visualization. Major examples of visualizing spatial data include Google and Bing Maps, which allow users to interactively navigate through pre-generated static images. To generate similar images for user-defined data, existing techniques can be broadly categorized into: (a) *Single-machine techniques* [4], [7], [11], [19], [22], [28] that focus on defining how the generated image should look like per the application needs, while the performance is out of scope. MapD [20], [23] provides significant speedup to single machine algorithms by employing GPU, but it is still limited to the capabilities of

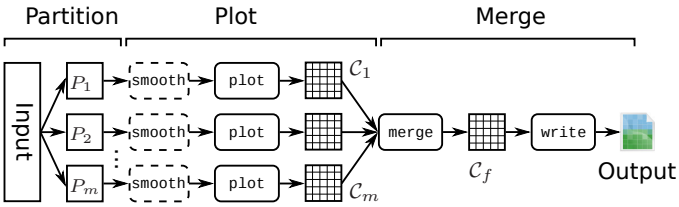


Fig. 2. Single level visualization algorithm

a single machine. (b) *Distributed techniques* [9], [26], [33] that use a cluster of distributed machines for better scalability. The pixel-level-partitioning technique [26], [33] is used to visualize 3D meshes and satellite data by partitioning records by the pixel they affect in the final image. SHAHED [9] uses a spatial partitioning technique which relies on an offline phase that indexes and cleans the data. Visualization in SHAHED has been revamped using HadoopViz so that it performs both partitioning and cleaning on the fly. Overall, there are three main drawbacks to these distributed systems. (1) Not efficient for generating big images with giga-pixel resolution. (2) Designed to support a specific image type and cannot directly apply to other image types. (3) They cannot produce images in which nearby records are fused together as they do not support a *smoothing* function.

HadoopViz. HadoopViz lies in the intersection of the above three areas, by providing a framework for *big spatial data visualization*. HadoopViz distinguishes itself form distributed spatial data visualization systems [9], [26], [33] in three main issues: (1) HadoopViz is an order of magnitude faster than existing techniques, which makes it more plausible for generating giga-pixel images over big data sets. (2) HadoopViz has an extensible design where users can plug in their own visualization logic by implementing five abstract functions. This allows HadoopViz to support new image types without changing the core visualization algorithms. (3) HadoopViz supports a user-defined *smoothing* function which expands the spectrum of supported image types to those which require nearby records to be fused together as the image is generated.

III. Single-Level Visualization

This section explains how HadoopViz visualizes satellite data as a single-level image, i.e., an image which shows all the details in one level. Sections V and VI generalize the described algorithms to other data types using the *visualization abstraction*. The inputs to this algorithm are an input file to visualize, the MBR of its contents, and the desired *ImageSize* in pixels, while the output is an image of the desired size which contains a temperature heat map, as in Figure 1(b).

Figure 2 gives an architectural view of the single-level visualization process in HadoopViz which follows a three phase approach where (1) the *partitioning* phase splits the input into m partitions, (2) the *plotting* phase plots a partial image for each partition, and (3) the *merging* phase combines the partial images into one final image. This section describes two algorithms that use this approach, *default-Hadoop partitioning* (Section III-A), and *spatial partitioning* (Section III-B). Section III-C describes how HadoopViz automatically chooses an appropriate algorithm for a given visualization problem.

Algorithm 1 Visualization using default Hadoop partitioning

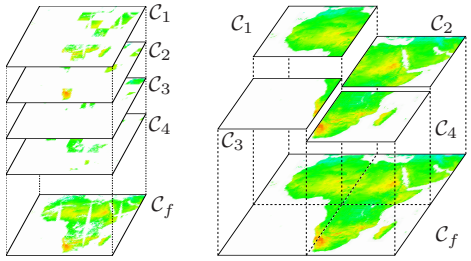
```
1:  function SingleLevelPlot(InFile, InMBR, ImageSize)
2:  // The input is already partitioned into m partitions P_1 to P_m
3:  // The Plotting Phase
4:  for each partition ⟨P_i, BR_i⟩ do
5:      Create a 2D matrix C_i of size ImageSize
6:      Update C_i according to each point p ∈ P_i
7:  end for
8:  // The Merging Phase
9:  Create a final matrix C_f with the desired ImageSize
10: For each reducer j, calculate C_j as the sum of all assigned matrices
11: One machine computes C_f as the sum of all C_j matrices
12: Generate an image by mapping each entry in C_f to a color
13: Write the generated image as the final output image
```

A. Default Hadoop Partitioning

This section describes the single-level visualization algorithm which uses the *default-Hadoop* partitioning. As shown in Algorithm 1, the algorithm assumes that the input is already loaded in HDFS, which splits the input into equi-sized blocks of 128MB each. This means that the *partitioning* phase has nothing to do.

The *plotting* phase in Lines 3-7 runs as a part of the *map* function where each mapper generates a partial image $\mathcal{C}_i$ for each partition P_i, $1 \leq i \leq m$. In Line 5, it initializes a matrix $\mathcal{C}_i$ with the same size as the desired *ImageSize* in pixels ($width \times height$), which acts as a *canvas* to plot records in P_i. Each entry contains two numbers, *sum* and *count*, which, respectively, contain the summation and count of all temperature values in the area covered by one pixel, both initialized to zeros. These two values are used together to *incrementally* compute the average temperature in each pixel. Line 6 scans the point in the assigned partition P_i and updates the matrix $\mathcal{C}_i$ according to its location and temperature. The point location determines the matrix entry to update, the temperature value is added to the *sum* entry, and the *count* entry is incremented by one. Finally, the mapper writes the matrix contents as an intermediate record to be processed by the next *merging* phase. These intermediate matrices are shuffled across the R reducers so that each reducer machine receives an average of m/R matrices.

In the *merging* phase, Lines 8-13 merge all intermediate matrices $\mathcal{C}_i$, in parallel, into one final matrix $\mathcal{C}_f$ and writes it as an output image. This phase runs in three steps, *partial merge*, *final merge*, and *write image*. The *partial merge* step runs locally in each machine where each reducer sums up all its assigned matrices into one final matrix $\mathcal{C}_j$ where $1 \leq j \leq R$. In the *final merge* step, a single machine reads back the R matrices and adds them up to a single final matrix $\mathcal{C}_f$. Figure 3(a) shows how this step *overlays* the intermediate matrices into one final matrix. Finally, the *write image* step in Line 13 computes the average temperature for each array position and generates the final image by coloring each pixel according to the average temperature of the corresponding array position. For example, the pixels can be colored with shades that range from blue to red between the lowest and highest temperatures, respectively. The image is finally written to disk as one file in a standard image format such as PNG.



(a) Overlay (b) Stitch

Fig. 3. Merging intermediate images

B. Spatial Partitioning

This section describes the single-level visualization algorithm using *spatial partitioning*. It differs from the previous algorithm in two parts, the *partitioning* phase uses a spatial partitioning, and the *merging* phase *stitches* intermediate images rather than *overlaying* them as shown in Figure 3(b). If the user requests a *smooth* image, as in Figure 1(b), then only this algorithm is applicable. This algorithm is implemented as a single MapReduce job as described below.

The *partitioning* phase runs in the map function and uses the SpatialHadoop partitioner [8], [10] to break down the input space into *disjoint* cells and assign each record to all overlapping cells. Notice that the *pixel-level partitioning* technique, employed in previous work [26], [33], is a special case of this phase where it applies a uniform grid equal to the desired *ImageSize*.

The *plotting* phase runs in the *reduce* function where all records in each partition are grouped together and visualized to generate one *partial* image. First, it applies the 2D interpolation techniques employed by SHAHED [9] to recover missing values. Unlike SHAHED, this function is applied on-the-fly which gives more flexibility to apply different smoothing functions. Then, the *plotting* phase initializes a matrix $\mathcal{C}_i$, similar to the one described in the *default-Hadoop* algorithm. However, the size of this matrix is calculated as $width = ImageSize.width\frac{BR_i.width}{InMBR.width}$ and $height = ImageSize.Height\frac{BR_i.height}{InMBR.height}$, where $ImageSize$ is the desired image size in pixels and $InMBR$ is the minimum bounding rectangle (MBR) of the input space. Finally, the reduce function scans all records in the given partition and updates the sum and $count$ of the array entries as described in the *default-Hadoop* algorithm.

The *merging* phase merges the intermediate matrices $\mathcal{C}_i$ into one big matrix by stitching them together according to their locations in the final image, as shown in Figure 3(b). Similar to the *default-Hadoop* algorithm, this phase runs in three steps, *partial merge*, *final merge*, and *write image*. The *partial merge* step runs in the *reduce-commit* function, where each reducer $j \in [1, R]$ creates a matrix $\mathcal{C}_j$ equal to the desired image size and adds all intermediate matrices to it. Each matrix $\mathcal{C}_i$ is added to a position in $\mathcal{C}_j$ according to the bounding rectangle BR_i of its corresponding partition P_i. The *final merge* step runs on a single machine in the *job-commit* function, where it reads back the R matrices written the previous step and adds them up into one matrix $\mathcal{C}_f$. Finally the *write image* step converts the final array to an image as described earlier and writes the resulting image to the output as the final answer.

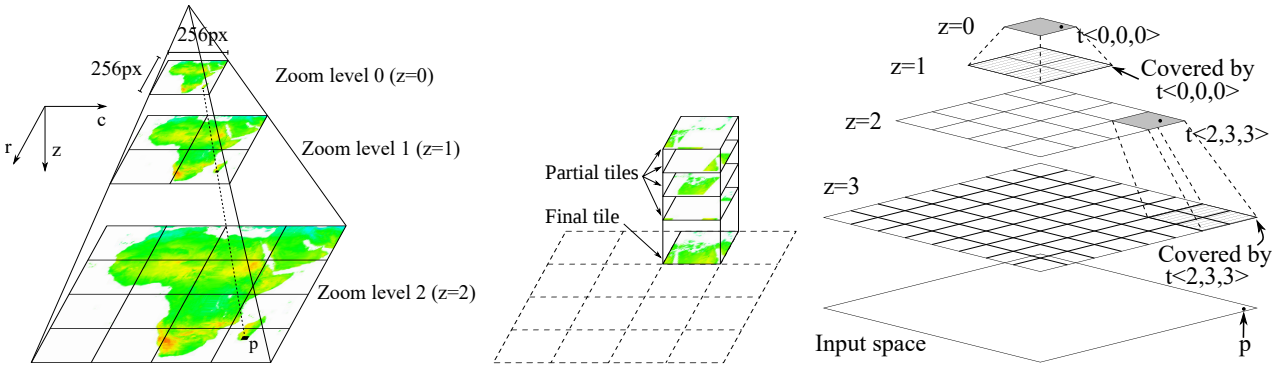
C. Discussion

If the user needs to generate a *smooth* image, then only the *spatial partitioning* algorithm is applicable. However, if no smoothing is required, both techniques are applicable and HadoopViz has to decide which one to use. By contrasting the two techniques, we find that there is a tradeoff between the *partitioning* and *merging* phases, where the first algorithm uses a *zero-overhead* partitioning phase, and an expensive *overlay merging* phase, while the second one pays an overhead in *spatial partitioning* but saves with the more efficient *stitching* technique in the *merging* phase. By intuition, the *default-Hadoop* algorithm is useful as long as the *plotting* phase can reduce the size of the input partitions by generating *smaller* partial images. This condition holds if the desired *ImageSize* in bytes is smaller than the size of one input partition, which is equal to HDFS block capacity. Otherwise, if the *ImageSize* is larger than an HDFS block, the *spatial partitioning* would be more efficient as it partitions the input, without significantly increasing its size, while ensuring that each partition is visualized into a much smaller partial image.

To prove the above condition analytically, we measure the overhead of the *partitioning* and *merging* phases. We ignore the overhead of the *plotting* phase and writing the output since they are the same for both algorithms. In the *default-Hadoop* algorithm, the *partitioning* phase has a zero overhead, and the *merging* phase processes m partial images, each with a size equal to the desired *ImageSize*, which makes its total cost equal to $m \times ImageSize$. In the *spatial* partitioning algorithm, the cost of *partitioning* phase is equal to the input size, as it scans the input once, and the cost of the *merging* phase is equal to the size of the desired image because partial images are disjoint and they collectively cover the desired image size. HadoopViz decides to use the spatial partitioning when it produces a less estimated cost, i.e., $InputSize + ImageSize < m \times ImageSize$. By rearranging terms and substituting $InputSize = m \times BlockSize$, the inequality becomes $(m-1)ImageSize > m.BlockSize$. Given that $m \gg 1$ for large inputs, the condition becomes $ImageSize > BlockSize$ as mentioned above.

IV. Multilevel Visualization

This section presents HadoopViz algorithm for generating gigapixel multilevel images where users can zoom in/out to see more/less details in the generated image. Similar to Section III, we focus on the case study of visualizing temperature data as a heat map while the next two sections show how HadoopViz is generalized to a wider range of images. Figure 4(a) gives an example of a three-level image, also called a *pyramid*, containing 1, 4, and 16 *image tiles* in three zoom levels, each of a fixed default size 256× 256 pixels. Google and Bing Maps use this type of images where the web interface allows users to navigate through offline-generated image tiles. This section shows how HadoopViz generates those tiles efficiently for custom datasets, while the same Google Maps APIs are used to view them. In addition to the input dataset and its MBR (*InMBR*), the user specifies a range of zoom levels to generate $[z_{\min}, z_{\max}]$, which actually decides the image size, by knowing the number of 256× 256 pixels tiles in each level. The output is *a set* of images, one for each tile in the



(a) A sample three-level image (b) Merging phase (c) Coarse-grained pyramid partitioning

Fig. 4. Multilevel Image Visualization

desired range of zoom levels along with an HTML file that uses Google Maps APIs to navigate these tiles.

One way to generate such multilevel images is to use our single-level visualization algorithm, described in Section III, to generate each tile of 256× 256 pixels. However, this solution is not practical even for a 10-level image as it contains millions of tiles. Another way is to generate one big image at the highest resolution and chop it down into tiles but this is not practical either as the size of that single big image could be hundreds of gigabytes which does not fit on most commodity machines. HadoopViz uses a smart algorithm which generates all the tiles efficiently in one MapReduce job by taking into account the structure of the pyramid.

The main idea of the multilevel visualization algorithm is to partition the data cross machines and plot each record to *all* overlapping tiles in the pyramid. The tiles that are partially generated by multiple partitions are merged to produce one final image for that tile. Similar to the single-level visualization algorithms, the choice of a *partitioning* technique plays an important role with multilevel visualization and affects the overall performance. This section describes two algorithms which use *default-Hadoop* partitioning (Section IV-A) and a novel *coarse-grained pyramid* partitioning (Section IV-B). After that, we show how HadoopViz combines these two algorithms to generate a user-requested image efficiently (Section IV-C).

A. Default-Hadoop Partitioning

In this section, we describe how to generate a multilevel image with the default partitioning scheme of Hadoop. The main idea is to allow each machine to plot a record to *all* overlapping pyramid tiles, and then to apply a *merging* phase which combines them to produce the final image for each tile. Since this algorithm uses the *default* partitioning scheme, the *partitioning* phase has a zero overhead.

The *plotting* phase runs in the *map* function and plots each record in the assigned partition P_i to all overlapping tiles in the pyramid. As shown in Figure 4(a), for a record $p \in P_i$, it finds all overlapping tiles in the pyramid in all zoom levels $[z_{min}, z_{max}]$, as at least one tile per zoom level. For each overlapping tile, it initializes a two-dimensional array of the fixed tile size, e.g., 256×256, updates the corresponding entries in the array, and caches that tile in memory in case other records in P_i overlap the same tile again. Once all records are plotted, all tiles, which are cached in memory, are written as intermediate key-value pairs, where the key is the tile ID and the value is the 2D array.

The *merging* phase runs in the *reduce* function and merges intermediate tiles to produce the final image for each tile. This step is necessary because the default Hadoop partitioner might assign overlapping records to different machines and each machine will generate a partial image for the same tile ID, as shown in Figure 4(b). The *merging* phase combines all those partial images by summing the corresponding entries into one final image, which is finally written to disk, as done in the single-level visualization algorithm. The image is written with a standard naming convention '`tile-z-c-v.png`', where z is the zoom level of the tile and (c, r) is the position of the tile in that zoom level.

B. Coarse-grained Pyramid Partitioning

The algorithm described above suffers from two main drawbacks when the pyramid is very large. First, the *plotting* phase incurs a huge memory footprint as it needs to keep all tiles of the whole pyramid in main-memory until all records are plotted. Second, the *merging* phase adds a huge processing overhead for merging all these tiles. This section describes how HadoopViz uses a novel *coarse-grained pyramid* partitioning technique to avoid the drawbacks of the *default* partitioning algorithm. In this technique, the input is partitioned according to pyramid tiles [2], which ensures that each machine generates a fixed number of tiles while totally avoiding the *merging* phase. While this can be done using a traditional *fine-grained* pyramid partitioning, which creates one partition per-tile, HadoopViz uses the *coarse-grained* partitioning which reduces the partitioning overhead by creating a fewer number of partitions while ensuring the correct generation of all image tiles. This algorithm runs in two phases only, *partition* and *plot*, which are implemented in one MapReduce job.

The *partitioning* phase runs in the *map* function and it uses the *coarse-grained pyramid* partitioning which assigns each record p to *select* tiles. This technique reduces the partitioning overhead by creating partitions for tiles only in levels that are multiples of a system parameter k, which controls the *partitioning granularity*. At one extreme, setting $k = 1$ is similar to using a *fine-grained* partitioning where it creates one partition per tile. On the other extreme, setting $k > z_{\max}$ generates only one partition at the top of the pyramid which contains all input records. Figure 4(c) shows an example with

$k = 2$ where it assigns a record p to only two partitions at levels $z = 0$ and $z = 2$. The machine that processes each partition will be responsible of generating the tile images in up-to k levels rooted at the assigned partition tile.

The *plotting* phase runs in the *reduce* function and it takes all the records in one partition, which corresponds to a tile t_i, and plots these records to all pyramid tiles under the tile t with at most k levels. For example, in Figure 4(c), the partition at tile $t = \langle 0, 0, 0 \rangle$ generates the five tiles at zoom levels 0 and 1. Once all records are plotted, an image is generated for each tile and all images are written to the output. No *merging* phase is required for this algorithm since each tile is generated by at most one machine.

C. Discussion

The default-Hadoop partitioning and pyramid-partitioning algorithms complement each other in generating a multilevel image of an arbitrary range of zoom levels, where the default-Hadoop partitioning is used to generate the top levels, while the pyramid partitioning is used to generate the remaining deeper levels. For the top levels, the *default-Hadoop* algorithm avoids the overhead of partitioning while the overheads of the *plot* and *merge* phases are minimal due to the small pyramid size. On the other hand, the *pyramid partitioning* algorithm would perform poorly in top levels as each tile will contain a huge number of records, e.g., the top tile overlaps all input records as it covers the whole input space. In deeper levels, the algorithms change roles as the *default-Hadoop* algorithm suffers from the overhead of the *plot* and *merge* phases, while the *pyramid* partitioning algorithm overcomes those limitations. Therefore, HadoopViz defines a threshold level z_θ, where levels $z < z_\theta$ are generated using default-partitioning, while other levels $z \geq z_\theta$ are generated using pyramid-partitioning.

To find the value of z_θ analytically, we compare the estimated cost of the two algorithms for a specific zoom level z in the pyramid and find the threshold level at which *pyramid* partitioning starts to give a lower cost. For the *default-Hadoop* partitioning algorithm, the cost of the *partitioning* phase is zero, while the cost of the *merging* phase is $m.4^z.TileSize$, where m is the number of partitions, 4^z is the number of tiles at level z, and *TileSize* is the fixed size of one tile. For the *pyramid* partitioning algorithm, the amortized cost of the *partitioning* phase for one level is $InputSize/k$, because the whole input is replicated once for each consecutive k levels, while there is a zero overhead of the *merging* phase. To find z_θ, we find the range of zoom levels where pyramid partitioning gives a less estimated cost, that is $InputSize/k < m.4^z.TileSize$. By rearranging the terms and separating z, it becomes $z \geq \frac{1}{2}\lg(\frac{B}{k.TileSize})$, i.e., $z_\theta = \lceil \frac{1}{2}\lg(\frac{B}{k.TileSize}) \rceil$.

V. Visualization Abstraction

HadoopViz is an extensible framework that supports a myriad of visualization procedures for various image types. In this section, we show how the single-level and multilevel visualization algorithms described in Sections III and IV are generalized to handle a wide range of image types, e.g., scatter plot, road network, frequency heat map, and vectorized map. To support one more image type within HadoopViz framework, the user needs to define five abstract functions, namely, `smooth`, `create-canvas`, `plot`, `merge`, and `write`. The goal is to make the designers of visualization algorithms worry free from the scalability and detailed implementation of their algorithms. So, algorithm designers only think about the visualization logic, while HadoopViz is responsible on scaling up that logic by employing thousands of computing nodes within a MapReduce environment. For example, ScatterDice [11] is a well known visualization system that is used to visualize multidimensional data using scatter plot. As HadoopViz supports scatter plot, among others, it can complement ScatterDice by scaling out its techniques to generate giga-pixel images of petabytes of data. HadoopViz can similarly scale out other visualization packages such as VisIt [6] or ImageMagick [15].

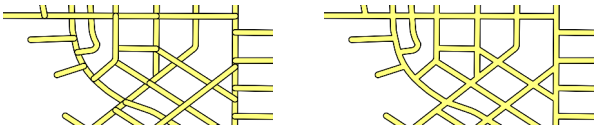
Algorithm 2 The abstract single-level algorithm

```
1:  // The Partitioning Phase
2:  Use spatial partitioning to create m partitions
3:  // The Plotting Phase
4:  for each partition ⟨P_i, BR_i⟩ do
5:      Apply smooth(P_i)
6:      C_i ← create-canvas(ImageSize BR_i / InMBR)
7:      for each p ∈ P_i, plot(p, C_i)
8:  end for
9:  // The Merging Phase
10: C_f ← create-canvas(ImageSize)
11: for each intermediate canvas C_i, merge(C_f, C_i)
12: write(C_f, outFile)
```

Algorithm 2 gives the pseudo-code of the *abstract* spatial-partitioning single-level visualization algorithm where the five abstract functions are used as building blocks. Any user-defined implementations for these functions can be directly plugged into this algorithm to generate a single-level image using HadoopViz. In the *partitioning* phase, Line 2 partitions the input spatially into m partitions, each partition i is defined by a bounding rectangle BR_i and a set of records P_i. In the *plotting* phase, Line 5 applies the `smooth` abstract function on each partition i to smooth its records. Line 6 in Algorithm 2 calls the `create-canvas` function to initialize a partial image $\mathcal{C}_i$, for each partition i. Line 7 calls the `plot` function to plot each record $p \in P_i$ on that partial image $\mathcal{C}_i$. In the *merging* phase, Line 10 calls `create-canvas` to initialize the final image canvas $\mathcal{C}_f$. After that, Line 11 calls `merge` successively on partial canvases to merge them into the final canvas. At the end of Algorithm 2, Line 12 uses the `write` function to write the final canvas $\mathcal{C}_f$ to the output as an image. We omit the abstract pseudo code of other algorithms due to limited the space, while interested readers can refer to the source code of HadoopViz [14]. In the rest of this section, we describe the five abstract functions and show how they can differ according to the visualization type. The next section gives six case studies of how these functions are implemented in real scenarios.

A. The Smooth abstract function

This is an optional preparatory function, where the input is the set of records that need to be visualized. The output is another set of records that represent a smoothed (or cleaned) version of the input by fusing nearby records together to produce a better looking image. HadoopViz tests for the existence of this function to decide whether to go for *spatial* or *default* partitioning. In addition, the plotting phase calls this



(a) Not smoothed (b) Smoothed

Fig. 5. Smoothing of road segments

function to smooth records in each partition before they are visualized. In the example of satellite data described earlier, the `smooth` function applies an interpolation technique to estimate missing temperatures as shown in Figure 1. Figure 5 gives another example of smoothing the visualization of a road network, where the `smooth` function merges intersecting road segments. If no smoothing is applied, road segments will be crossed, giving a not so accurate visualization (Figure 5(a)). Having this logic in a user-defined abstract function allows users to easily inject a more sophisticated logic. For example, if more attributes are available, the `smooth` function can correctly overlap (not merge) road segments at different levels such as roads and bridges. It is important to note that the `smooth` function has to be applied on the input data rather than the generated image because it can process all input attributes that probably disappear in the final image. One limitation in the current design is that it cannot smooth records across different partitions which we can support in the future.

B. The Create-Canvas abstract function

This function creates and initializes an appropriate in-memory data structure that will be used to create the requested image. The input to the `create-canvas` abstract function is the required image resolution in terms of $width$ and $height$ (e.g., in pixels). The output is an initialized canvas of the given resolution. If the desired image is a *raster* image, the canvas typically consists of a two-dimensional matrix as one per pixel. If the desired image is a *vector* image, it contains a vectorized representation of objects shapes. The `create-canvas` function is used in both the *plotting* and *merging* phases. In the *plotting* phase, HadoopViz calls this function to initialize the partial images used to plot each partition. In the *merging* phase, it is used to prepare the final image on which all partial images will be merged. For example, when visualizing a road network, it returns an in-memory *blank* image of the given size, while for a heat map, it returns a frequency map represented as a 2D array of numbers initialized to zeros.

C. The Plot abstract function

The `plot` function is called for each record in the input data set. It takes as input a canvas, previously created using `create-canvas`, and a record. Then, it plots the input record on the canvas. The *plotting* phase calls this function for each record in the partition to draw the partial images. The `plot` function uses a suitable algorithm to draw the record on the canvas. For example, when visualizing a road network, the Bresenham mid-point algorithm [5] is used to draw a line on the image. When visualizing a heat map, this function updates the two-dimensional histogram (created by the `create-canvas` function) based on the point location. To generate a vector image, it simplifies the record shape and represents its geometry as a vector. In general, this function can call any third party visualization package, e.g., VisIt [6] or ImageMagick [15], which allows HadoopViz to easily reuse and scale out existing visualization packages.

D. The Merge abstract function

The input to the `merge` function is two partial canvases, while the output is one final canvas that is composed by combining the two input partial layers. The *merging* phase calls this function successively on a set of layers to merge them into one. If the partial layers are disjoint, i.e., each one covers a different part of the image, merging them is straightforward as each pixel in the final image has only one value in one canvas. In case the partial layers cover overlapping areas, the same pixel in the final image may have more than one value. In this case, the `merge` function needs to decide how these values are combined together to determine the final value of that pixel. For example, if the canvases are raster images, two pixels are merged by taking the average of each color component in the two pixels. In case of generating a heat map, two entries in the histogram are merged together by adding up the corresponding values as each one represents a partial count.

E. The Write abstract function

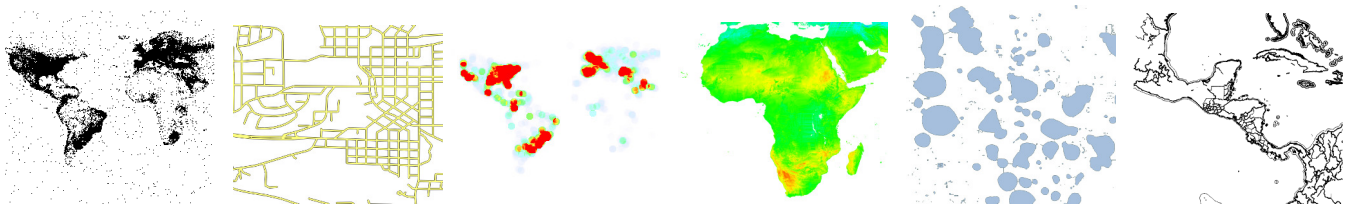
The `write` function writes the final canvas (i.e., image), computed by the `merge` function, to the output in a standard image format (e.g., PNG or SVG). This abstract function allows developers to use any custom representation for canvases which might contain additional metadata, and generate the image as a final step using the `write` function. For example, while generating a heat map, the canvas stores the frequencies as integers while the `write` function transforms them into colors and writes a PNG image. In the case of generating vector images, the canvas contains geometric representation of shapes and the `write` function encodes them in a standard Scalable Vector Graphics (SVG) format.

VI. Case Studies

This section describes six case studies of how to define a visualization type by implementing the five abstract functions described in Section V. These case studies are carefully selected to cover different aspects of the visualization process. Notice that all case studies described using the abstract functions can be used to generate both single and multilevel images. Case studies I and II give an example of non-aggregate visualization, where records are directly plotted, with and without a smoothing function. Case studies III and IV give examples of aggregate-based visualization, where records are aggregated before plotted, with and without a smoothing function. Case study V gives an example of generating a vector image with a smoothing function. Finally, case study VI shows how to reuse and scale out an existing visualization package which is used as a black box.

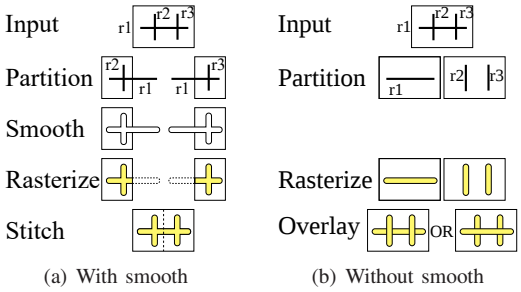
A. Case Study I: Scatter Plot

In *scatter* plot, the input is a set of points, e.g., geotagged tweets, and each point is plotted as a pixel in the final image as shown in Figure 6(a). To make HadoopViz support such images for both single and multi-level images, we need to define its five abstract functions as follows: (1) The `smooth`



(a) I. Scatter plot (b) II. Road network (c) III. Heat map (d) IV. Satellite data (e) V. Vectorized map (f) VI. Countries borders

Fig. 6. Six case studies all implemented in HadoopViz via the five abstract functions



(a) With smooth (b) Without smooth

Fig. 7. Road network visualization

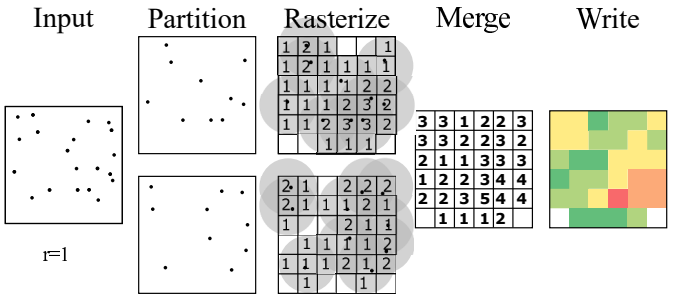


Fig. 8. Steps of frequency heat map (Best viewed in colors)

function is not required for scatter plot, as there are no records that need to be smoothed together. (2) The `create-canvas` function takes an image size in pixels and returns a blank in-memory image with the given size initialized with a transparent background. (3) The `plot` function takes an in-memory image and an input record r, and it projects the input point on the image and colors the corresponding pixel in black. (4) The `merge` function takes two in-memory images and merges them together. To merge two images, the top-left corner of the first image is projected to a pixel in the other one as done in the `plot` function. Then, the first image is painted on the second one at the projected location. Since each image is initialized with a transparent background, empty spaces in one image will reveal the contents of the other image. (5) The `write` function encodes the in-memory image in a standard image format (e.g., PNG) and writes it to disk.

B. Case Study II: Road Network

In road network visualization, the input is a set of road segments, each represented as a line, and the desired output is an image similar to the one illustrated in Figure 6(b). To support such images in HadoopViz, we need to define its five abstract functions as follows: (1) The `smooth` function takes a set of road segments, applies a *buffer* operation to each line segment to give it some thickness, and then applies a *union* operation to all resulting polygons to merge intersecting road segments. Specifying a `smooth` function enforces HadoopViz to use a spatial partitioning as shown in Figure 7(a). (2) The `create-canvas` function is exactly the same as in *scatter plot*, which returns an in-memory image of the provided size. (3) The `plot` function reads the result of the union operation returned by the `smooth` function, and draws each polygon in the dataset onto the image. The polygon is first projected from input space to image space, then, the interior of the polygon is filled with yellow while the boundaries are stroked in black. As spatial partitioning is used, some records might be replicated to two overlapping partitions such as r_1 in Figure 7(a). Each replica is merged with a different set of road segments and is plotted to two different partial images. However, it is automatically clipped at partition boundaries when plotted (denoted by dotted lines). (4) The `merge` function is exactly the same as in the scatter plot where one image is painted on the other according to its location. Notice that in this case, it will always *stitch* partial images together because they are all disjoint. As in Figure 7(a), the two images are clipped at the stitch line, hence, putting them side-by-side generates the correct picture without worrying about any overlaps. (5) The `write` function is exactly the same as in the *scatter plot*.

To justify the use of a smoothing function, Figure 7(b) shows how the visualization process would work if no `smooth` function is provided. First, HadoopViz would use the default Hadoop partitioning which causes overlapping records to be in two different partitions. The `plot` function can still apply the *buffer* operation to each segment but not the *merge* operation because overlapping records are in two different machines. The `merge` function would overlay partitions instead of stitching them. Depending on which image goes on top, there are two possible final images which are both incorrect.

C. Case Study III: Frequency Heat Map

In this case study, the input is a set of points, e.g., geotagged tweets, and the output is a colored map, e.g., Figure 6(c), where dense areas are colored in red and sparse areas are colored in blue. To support such kind of visualization in HadoopViz, we need to define the five abstract functions as follows: (1) The `smooth` function is not needed for such image type. (2) The `create-canvas` function creates a two-dimensional array of integers (called, *frequency map*), where the value in each entry represents the number of records around it; all initialized with zeros. (3) The `plot` function, takes one point, projects it to the frequency map, and increments all entries within a predefined radius ρ to denote that the point is within the range of those pixels. As more points are plotted,

entries in the frequency map will denote the density of points in each pixel, as shown in Figure 8. (4) The **merge** function takes two frequency maps, and merges them together by adding up corresponding entries in both. (5) The **write** function takes one frequency map and converts it to an image by coloring corresponding pixels in the image according to the density in the frequency map. First, it normalizes densities to the range $[0, 1]$, and then it calculates the color of each pixel by making a linear combination between the two colors, blue and red. Finally, the created image is written to the output in the standard PNG format.

D. Case Study IV: Satellite Data

In this case study, the input is a set of temperature readings, each associated with a geolocation, and the output is a temperature heat map, like the one in Figure 6(d), where the color represents the average temeprature of the underlying region. In addition, some regions do not contain any values due to clouds or satellite mis-alignment leaving some blind spots [9]. The values in those uncovered areas need to be estimated using a two-dimensional interpolation technique. To support such image type in HadoopViz, we need to define the five abstract functions as follows: (1) The **smooth** function takes a set of points in a region and *recovers* missing points using a two-dimensional interpolation function in a way similar to SHAHED [9]. Although HadoopViz uses the same technique as in SHAHED, it applies the **smooth** function on-the-fly allowing users to easily inject a better smoothing function. (2) The **create-canvas** function initializes a two-dimensional array of the input size where each entry contains two numbers, *sum* and *count*, used together to compute the average temperature incrementally. (3) The **plot** function projects a point onto the 2D array, and updates both *sum* and *count* in the array according to the temperature value of the point. (4) The **merge** function is very similar to that of the frequency heat map but it adds up both *sum* and *count* in the merged layers. (5) The **write** function starts by calculating the average temperature in each entry as $avg = sum/count$. Then, we use the same **write** function of the frequency heat maps.

E. Case Study V: Vectorized Map

This case study shows how to create a vector image that represents a map, such as Google Maps or Bing Maps. Many recent applications on smart phones and on the web prefer vector images over raster images due to their smaller size and nice rendering. For simplicity, this case study explains how to plot a map of lakes as shown in Figure 6(e), however, the technique can be easily expanded to plot more map objects through more sophisticated implementations of the **plot** function. To generate a vector image for lakes in HadoopViz, we define the abstract functions as follows: (1) In the **smooth** abstraction function, we use a *map simplification* algorithm which reduces the amounts of details in the polygons according to the resolution of the final image. The goal of this function is to reduce the generated image size by removing very fine details that will be hardly noticed by users according to the image size. If a multilevel image is generated, the **smooth** function will be called once for each zoom level so that it keeps more details in deeper levels. This function also removes very small lakes which are too small to plot in the image. (2) The **create-canvas** abstract function initializes an empty list of polygons. (3) The **plot** function adds the polygon representation of the lake geometry to the list of polygons in the canvas. (4) The **merge** function, simply, merges the two lists of polygons in the two canvases into one list in the output canvas. (5) The **write** function encodes the canvas contents as a standard SVG image and writes it to the output.

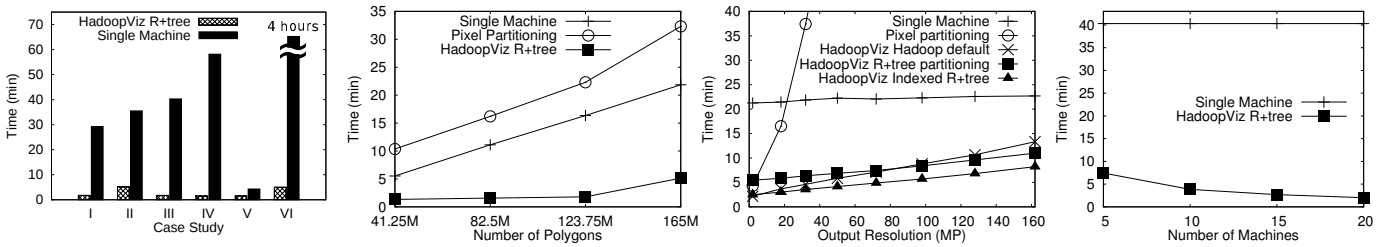
F. Case Study VI: Parallel ImageMagick

ImageMagick [15] is a popular open source package that can produce and edit images. However, it is single-machine and does not support any multi-node parallelization functionality. This case study shows how to use the binaries of ImageMagick as a blackbox and utilize the extensibility of HadoopViz to seamlessly parallelize it. This allows users to visualize extremely large datasets using ImageMagick that it cannot handle otherwise. For example, this technique speeds up the visualization of a 130GB file from four hours on a single machine, to five minutes using HadoopViz. In this case study, the input is a set of straight line segments that represent the administrative borders in the whole world (e.g., countries and cities) which need to be visualized as shown in Figure 6(f). To visualize the input as lines in the final image, we define the five functions as follows: (1) No **smooth** function is needed. (2) The **create-canvas** function spawns an ImageMagick process in the background and sends it a '`viewbox`' command to initialize an image with the desired size. (3) The **plot** function projects a line from the input to the image space, and send the ImageMagick process a '`draw line`' command with the projected boundaries. As HadoopViz needs to transfer canvases from *mappers* to *reducers*, it cannot simply move a running process. So, to transfer a canvas, we close the ImageMagick instance and transfer the generated image across network. (4) The **merge** function uses the '`draw image`' ImageMagick command to draw one image onto the other image. (5) The **write** function closes the ImageMagick process of the final canvas, retrieves the image created by that instance, and writes it to the output as a file.

VII. Experiments

This section provides an experimental evaluation for HadoopViz to show its extensibility and scalability. All HadoopViz experiments are conducted on a cluster of 20 nodes of Apache Hadoop running on Ubuntu 10.04.4 machines with Java 1.7. Each machine has an Intel(R) Xeon E5472 processor with 4 cores @3 GHz, 8GB of memory and a 250GB hard disk. The HDFS block size is 128 MB. All single machine experiments run on a machine with 12 cores of Intel(R) Haswell E5-2680v3 CPU @ 2.50GHz and 1TB memory. In all experiments, we use total execution time as the main performance metric of our experiments.

For the input data, we use three real datasets extracted from OpenStreetMap [25], namely, `nodes`, `ways`, and `lakes`, in addition to one satellite dataset from NASA called `nasa`. The `nodes` dataset (1.7 billion points) is used for case studies I and III, the `ways` dataset (165 million polylines) for case studies II and VI, the `lakes` dataset (8.4 million polygons) for case study V, and the `nasa` dataset (14 billion points) for case study IV. For experiments repeatability, OpenStreetMap and NASA



(a) Case studies (b) Input size - case study II (c) Image size - case study I (d) Cluster size - case study III

Fig. 9. Single Level Image Performance

datasets are made available at the two following links, respectively. http://spatialhadoop.cs.umn.edu/datasets.html#osm - http://e4ftl01.cr.usgs.gov/MOLA/MYD11A1.005/.

A. Single-Level Visualization

For single-level image visualization, we consider seven different algorithms where each is applied in suitable experiments. (1) A *single-machine* algorithm which loads the whole dataset into main-memory, smooths records in memory, scans records and plots them to an in-memory image, and finally writes that image to the output. (2) Distributed *pixel-level-partitioning* [26], [33] which is implemented in HadoopViz using a uniform grid partitioning with a grid size equal to image size. (3, 4, 5) HadoopViz with default Hadoop partitioning, grid, and R+-tree partitioning. (6, 7) HadoopViz with grid-based and R+-tree-based indexes which utilize an existing index to avoid the spatial partitioning phase.

Figure 9(a) compares the performance of HadoopViz with R+-tree partitioning to the single-machine algorithm, as they are both applied to the six case studies described in Section VI to visualize a 32 mega-pixels image. This experiment shows an order of magnitude speedup of HadoopViz as compared to traditional single machine algorithms. It also shows the flexibility and efficiency of HadoopViz when used with different image types using the single level algorithm. For example, it visualizes 14 Billion points of NASA data in a 90 seconds. Case study V runs relatively faster as it operates on the much smaller `lakes` dataset. The figure also shows the power of HadoopViz as it provides a 48X speedup of the single-machine ImageMagick visualization package.

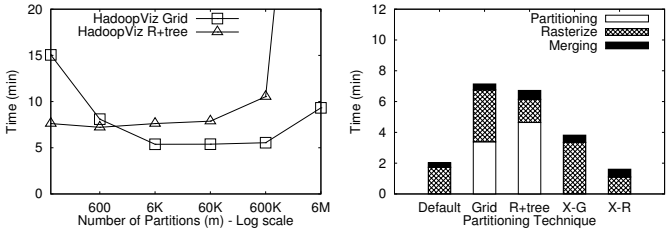
In Figure 9(b), we change the input size of the road network, by sampling at 25%, 50% and 75%, while fixing the image size at 32 mega-pixels. HadoopViz outperforms both single machine and pixel-level-partitioning algorithms for all input sizes. The performance of pixel-level-partitioning is very poor as it has to process 32 Million partitions, as one per pixel. At this image size, pixel-level-partitioning is even slower than a single machine which relies on a huge main memory of 1TB.

Figure 9(c) gives the effect of changing the desired image size from 2 to 160 mega-pixels. This experiment runs on the scatter plot case study as it does not contain a `smooth` function which allows for the use of default Hadoop partitioning. This figure shows clearly that pixel-level partitioning is only useful with a small image sizes of less than 20 mega-pixels. The single-machine algorithm is slightly affected by the image size as it performs all processing in main-memory after the file is loaded from disk. On the other hand, all techniques in HadoopViz scale very well with the generated image size making it useful for generating both small and big images. As described in Section III, the default Hadoop partitioning performs better with small images while spatial R+-tree partitioning performs better with big images. This experimentally justifies the decision made in the partitioning phase where it switches from default Hadoop partitioning to spatial partitioning when image size grows larger than a block size. Although this condition holds at the points at 50 and 72 mega-pixels, while the performance crossover happens at 80 mega-pixels, the difference is very small and both techniques perform very similar. This experiment also employs the indexed R+-tree technique which skips the spatial partitioning phase by utilizing an existing R+-tree index constructed using SpatialHadoop [10]. As shown, this technique outperforms all other techniques as it gets the good performance of R+-tree partitioning without having to pay the overhead of partitioning.

Figure 9(d) shows how HadoopViz scales out with cluster size when visualizing a heat map as compared to a single machine algorithm. This figure also shows that HadoopViz scales out very nicely with cluster size as it parallelizes all of the three phases of the algorithm. Even with five machine, it outperforms the single-machine algorithm.

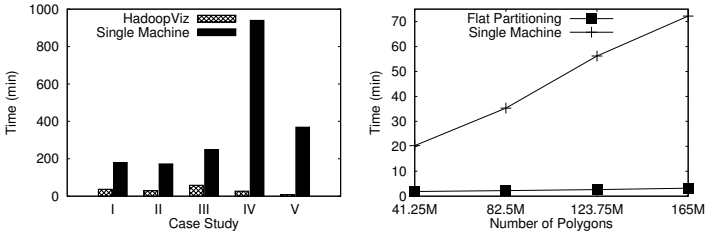
Figure 10(a) gives the effect of tuning number of partitions m on HadoopViz running with grid and R+-tree partitioning. We cannot compare with default Hadoop partitioning as number of partitions is automatically calculated by HDFS according to input size and HDFS block capacity. We change number of partitions from 60 to 6 Million to cover the spectrum of all values which are all shown on a log scale. Grid partitioning performs poorly on both extremes where it suffers from load imbalance at $m = 60$ and 600, and huge processing overhead at $m = 6$ Million. On the other hand, R+-tree-based partitioning is very stable when m changes from 60 to 600K, then the performance suddenly drops at $m = 6$ Million. The reason of this huge drop is that the partitioning phase has to search 6M rectangles for each input record to find overlapping partitions. This incurs a huge overhead even with an optimized in-memory R+-tree index with $\log n$ search time, as opposed to constant time in uniform grid partitioning.

Figure 10(b) breaks down the time of single-level plot in HadoopViz into the three phases. In this experiment, we skip the `smooth` function to be able to apply default Hadoop partitioning. The indexed R+-tree and indexed grid techniques are denoted X-G and X-R, respectively. At the small image size generated in this experiment (4 MegaPixels), the default Hadoop partitioning performs very well where the plotting phase accounts for most of the time. On the other hand, both



(a) Number of Partitions (m) (b) Performance breakdown

Fig. 10. Single-level image tuning with case study II



(a) Case studies (b) Input size - case study II

Fig. 11. Multilevel Image Performance

spatial partitioning techniques, grid and R+-tree, are much slower due to the overhead of the partitioning phase. Although the grid partitions very quickly, it performs poorly in the plotting phase due to load imbalance. Existing indexes save the partitioning time of both techniques making R+-tree much better than grid. To conclude, we recommend the use of R+-tree if the `smooth` and `plot` functions are complex to achieve a better load balance. Otherwise, a grid partitioning can be used when these functions are simple as it saves in the partitioning time. If no smooth function is required, default partitioning is good enough.

B. Multilevel Visualization

In multilevel visualization, we compare the performance of three techniques. (1) A single-level machine algorithm which builds the whole pyramid in main-memory and then dumps it to disk as one image per tile. (2) HadoopViz with default-Hadoop partitioning. (3) HadoopViz with pyramid partitioning.

Figure 11(a) compares the performance of a traditional single-machine algorithm to HadoopViz for visualizing a multilevel image. In this experiment, we generate a pyramid of 11 levels, which resembles a 70 giga-pixel image at the highest level. The experiment shows up-to two orders of magnitude speedup of HadoopViz as compared to a single machine algorithm. The speedup is much higher compared to single-level experiments, in Figure 9(a), due to the huge output size which is written to a single disk in single-machine experiment, as opposed to the HDFS in HadoopViz. This experiment also shows the great power and flexibility of HadoopViz as these multilevel images are created using the same five abstract functions that were used to implement the single-level images. In other words, users do not need to do any additional effort, other than defining the five functions, to generate multilevel images.

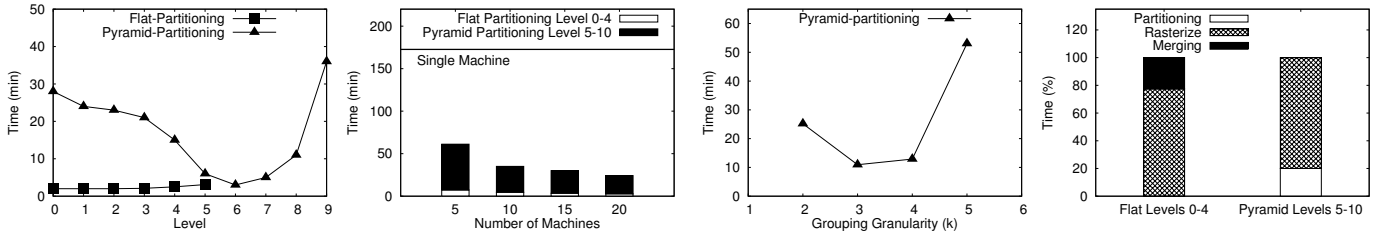
Figure 11(b) gives the performance of generating a pyramid of six levels using default partitioning and single-machine algorithms. As the input size increases from 41M polygons to 165M polygons, the performance of the single machine drops as it needs to read and parse the whole input file. On the other hand, HadoopViz scales very well as the scanning of the input is done in parallel using the MapReduce framework.

Figure 12(a) gives the performance of both *default* and *pyramid* partitioning algorithms in HadoopViz while generating each level in the pyramid. This experiment confirms our earlier discussion that default-partitioning performs better at top levels while pyramid-partitioning performs better at deeper levels. At top-levels, default partitioning performs better as it avoids the overhead of partitioning while the cost of merging is still low. On the other hand, pyramid partitioning performs poorly due to the load imbalance as the top levels contain only a few tiles. At deeper levels, the performance of default partitioning drops due to the huge number of tiles that need to be shuffled over network and then merged. In this experiment, it failed at levels six and higher with an out-of-memory exception due to the huge pyramid size that has to be stored in each machine. On the other hand, pyramid partitioning performs better as it partitions the input into thousands of tiles improving the load balance. The running time increases again at deeper levels due to the exponential increase in the size of the output which cannot be avoided. According to the system configuration and the analysis made in Section IV-C, HadoopViz should use pyramid-partitioning for levels five and higher. Although default-Hadoop partitioning algorithm performs better at level five, the difference is slight and both algorithms perform well.

Figure 12(b) shows the scalability of HadoopViz with cluster size where we increase the cluster size from 5 to 20 and measure the performance of generating an image with 11 levels with $z_\theta = 4$. This experiment shows a near perfect scale out for both algorithms due to the parallelization of all of the phases in both algorithms. It also shows a huge speedup over the baseline of single-machine performance. Deeper levels (5 to 10) take much more time due to the exponential increase in number of tiles.

In Figure 12(c), we change the grouping granularity (k) to verify its effect on the performance of the pyramid partitioning algorithm. This parameter was introduced in Section IV-B to control the trade-off between load balance and partitioning overhead by grouping multiple pyramid levels in one partition. As expected, a smaller value of k provides a poor performance as it produces too many partitions. On the other extreme, using a large value of k produces a few partitions hurting the load balance. Both values of 3 and 4 provide good performance as they achieve a good balance between load balance and partitioning overhead.

Figure 12(d) shows the percentage breakdown of the processing time of HadoopViz into the three phases. Default-partitioning algorithm spends most of its time in the plotting phase while the merging phase takes less time. The reason is that the plotting phase processes the whole input and produces partial pyramids of small sizes while the merging phase processes these partial pyramids in parallel to produce the final answer. On the other hand, the pyramid-partitioning algorithm finishes the partitioning phase quickly while most of the time is spent in the plotting phase which draws the final tiles directly and writes them to the output.



(a) Number of levels (b) Cluster size (c) Grouping granularity (k) (d) Phases

Fig. 12. Multilevel image performance tuning with case study II

VIII. Conclusion

In this paper, we presented HadoopViz; a MapReduce-based framework for visualizing big spatial data. HadoopViz can efficiently produce giga-pixel images for billions of input records. There are three main features in HadoopViz which make it unique to other visualization systems. (1) HadoopViz uses a *smoothing* technique to produce better looking images by fusing nearby records together. (2) HadoopViz can efficiently produce giga-pixel images by employing a three-phase technique, *partition-plot-merge*. This technique is applied to generate both single level and multilevel images. (3) It is extensible as it allows users to plug-in their own visualization logic by only implementing five abstract functions. HadoopViz takes these functions and plugs them in ready-made algorithms which allow the user-defined algorithms to automatically run on thousands of nodes. We use the defined abstraction to implement six visualization types, scatter plot, road network, frequency heat map, satellite heat map, vectorized map, and countries borders. We experimentally evaluate HadoopViz using real datasets on a cluster of 20 machines and show up-to two orders of magnitude speedup over existing techniques with an excellent scalability as it visualizes 14 Billion points in 90 seconds.

References

[1] A. Aji *et al.*, "Hadoop-GIS: A High Performance Spatial Data Warehousing System over MapReduce," in *VLDB*, 2013, pp. 1009–1020.

[2] W. G. Aref and H. Samet, "Efficient Processing of Window Queries in The Pyramid Data Structure," in *PODS*, apr 1990, pp. 265–272.

[3] L. Battle, M. Stonebraker, and R. Chang, "Dynamic Reduction of Query Result Sets for Interactive Visualization," in *BigData*, 2013, pp. 1–8.

[4] A. Bezerianos *et al.*, "GraphDice: A System for Exploring Multivariate Social Networks," *Comput. Graph. Forum*, vol. 29, no. 3, 2010.

[5] J. E. Bresenham, "Algorithm for Computer Control of a Digital Plotter," *IBM Systems journal*, vol. 4, no. 1, pp. 25–30, 1965.

[6] H. Childs *et al.*, "VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data," Jul 2011.

[7] I. F. Cruz *et al.*, "GIVA: a semantic framework for geospatial and temporal data integration, visualization, and analytics," in *SIGSPATIAL*, 2013, pp. 534–537.

[8] A. Eldawy, L. Alarabi, and M. F. Mokbel, "Spatial Partitioning Techniques in SpatialHadoop," in *PVLDB*, 2015, pp. 1602–1605.

[9] A. Eldawy *et al.*, "SHAHED: A MapReduce-based System for Querying and Visualizing Spatio-temporal Satellite Data," in *ICDE*, 2015.

[10] A. Eldawy and M. F. Mokbel, "SpatialHadoop: A MapReduce Framework for Spatial Data," in *ICDE*, 2015.

[11] N. Elmqvist *et al.*, "Rolling the Dice: Multidimensional Visual Exploration using Scatterplot Matrix Navigation," *TVCG*, vol. 14, no. 6, pp. 1539–1148, Nov 2008.

[12] "European XFEL: The Data Challenge," 2012, http://www.eiroforum.org/activities/scientific_highlights/201209_XFEL/index.html.

[13] J. Faghmous and V. Kumar, *Spatio-Temporal Data Mining for Climate Data: Advances, Challenges, and Opportunities*. Advances in Data Mining, Springer, 2013.

[14] (2015) HadoopViz Source Code. https://github.com/aseldawy/spatialhadoop2/tree/master/src/edu/umn/cs/spatialHadoop/visualization.

[15] "ImageMagick," http://www.imagemagick.org/.

[16] U. Jugel *et al.*, "M4: A Visualization-Oriented Time Series Data Aggregation," *PVLDB*, vol. 7, no. 10, pp. 797–808, 2014.

[17] (2015, Mar.) Land Process Distributed Active Archive Center. https://lpdaac.usgs.gov/about.

[18] W. Lu, Y. Shen, S. Chen, and B. C. Ooi, "Efficient Processing of k Nearest Neighbor Joins using MapReduce," *PVLDB*, 2012.

[19] R. Maciejewski, "A Visual Analytics Approach to Understanding Spatiotemporal Hotspots," *TCVG*, vol. 16, no. 2, pp. 205–220, March 2010.

[20] "MapD Twitter Demo," http://mapd.csail.mit.edu/tweetmap-desktop/.

[21] H. Markram, "The Blue Brain Project," *Nature Reviews Neuroscience*, vol. 7, no. 2, pp. 153–160, 2006.

[22] A. Middel, "A Framework for Visualizing Multivariate Geodata," in *Visualization of Large and Unstructured Data Sets*, 2007, pp. 13–22.

[23] "Todd Mostak. An Overview of MapD (Massively Parallel Database). Harvard Technical Report," http://geops.cga.harvard.edu/docs/mapd_overview.pdf.

[24] S. Nishimura *et al.*, "MD-HBase: Design and Implementation of an Elastic Data Infrastructure for Cloud-scale Location Services," *DAPD*, vol. 31, no. 2, pp. 289–319, 2013.

[25] "OpenStreetMaps," http://www.openstreetmap.org/.

[26] G. Planthaber, M. Stonebraker, and J. Frew, "EarthDB: Scalable Analysis of MODIS Data using SciDB," in *BIGSPATIAL*, 2012, pp. 11–19.

[27] J. Sankaranarayanan, H. Samet, B. E. Teitler, and M. D. L. J. Sperling, "TwitterStand: News in Tweets," in *SIGSPATIAL*, 2009, pp. 42–51.

[28] J. Song, R. Frank, P. L. Brantingham, and J. LeBeau, "Visualizing the spatial movement patterns of offenders," in *SIGSPATIAL*, 2012.

[29] M. Stonebraker, P. Brown, A. Poliakov, and S. Raman, "The Architecture of SciDB," in *SSDBM*, 2011, pp. 1–16.

[30] F. Tauheed *et al.*, "Accelerating Range Queries for Brain Simulations," in *ICDE*, 2012, pp. 941–952.

[31] "Telescope Hubbel site: Hubble Essentials: Quick Facts," http://hubblesite.org/the_telescope/hubble_essentials/quick_facts.php.

[32] "Twitter. The About webpage." https://about.twitter.com/company.

[33] H. T. Vo *et al.*, "Parallel Visualization on Large Clusters using MapReduce," in *LDAV*, 2011, pp. 81–88.

[34] R. Wesley, M. Eldridge, and P. T. Terlecki, "An Analytic Data Engine for Visualization in Tableau," in *SIGMOD*, 2011, pp. 1185–1194.

[35] H. Wickham, "Bin-summarise-smooth: a framework for visualising large data," had.co.nz, Tech. Rep., 2013.

[36] E. Wu, L. Battle, and S. R. Madden, "The case for data visualization management systems," *PVLDB*, vol. 7, no. 10, pp. 903–906, 2014.

[37] S. Zhang *et al.*, "SJMR: Parallelizing spatial join with MapReduce on clusters," in *CLUSTER*, 2009, pp. 1–8.