Large Interactive Visualization of Density Functions on Big Data Infrastructure

Alexandre Perrot, Romain Bourqui, Nicolas Hanusse, Frédéric Lalanne, David Auber

HAL Id: hal-01188721
https://hal.archives-ouvertes.fr/hal-01188721

Submitted on 17 Nov 2015

Large Interactive Visualization of Density Functions on Big Data Infrastructure

Alexandre Perrot   Romain Bourqui   Nicolas Hanusse   Frédéric Lalanne   David Auber

LaBRI, Univ. Bordeaux,* France

ABSTRACT

Point set visualization is required in lots of visualization techniques. Scatter plots as well as geographic heat-maps are straightforward examples. Data analysts are now well trained to use such visualization techniques. The availability of larger and larger datasets raises the need to make these techniques scale as fast as the data grows. The Big Data Infrastructure offers the possibility to scale horizontally. Designing point set visualization methods that fit into that new paradigm is thus a crucial challenge. In this paper, we present a complete architecture which fully fits into the Big Data paradigm and so enables interactive visualization of heatmaps at ultra-scale. A new distributed algorithm for multi-scale aggregation of point set is given and an adaptive GPU based method for kernel density estimation is proposed. A complete prototype working with Hadoop, HBase, Spark and WebGL has been implemented. We give a benchmark of our solution on a dataset having more than 2 billion points.

Index Terms:

Human-centered computing-Heat maps; Human-centered computing-Information visualization

1 INTRODUCTION

Point set visualization is without any doubt a key weapon in the arsenal of a data visualizer. The John Snow Map of deaths due to the 1854 London cholera epidemic is typically used to demonstrate the efficiency of point set visualization. By visually detecting a higher density of points (i.e., concentration of death) near a pump in the Soho district, John Snow was able to find that the source of the epidemic was infected water from that pump. The Figure 1.a shows a point set visualization of the John Snow dataset. If we zoom out the Figure 1.a (Fig. 1.c), one can see that due to point occlusion, it becomes harder to detect regions with high density of points. In [16], Kindlman et al. have proposed a method to evaluate the efficiency of a visualization technique. They formalize the fact that a change in the visualization must be proportional to a change in the data. According to the measures they introduce, standard point set visualization technique is not efficient. Indeed, even if the data does not change between Figure 1.a and Figure 1.c, the visualization varies significantly.

To solve this occlusion problem, many techniques have been set up. The core idea of all of them is first to count the number of points $c_{i,j}$ that are drawn at the same location i,j on the screen. Then one point $p_{i,j}$ is rendered for each unique location i,j and $c_{i,j}$ is used to determine the color of $p_{i,j}$. In the literature, the term *heatmap* is often used to describe that technique. More elaborate *heatmap* visualizations construct a continuous function by applying a convolution on $c_{i,j}$. This continuous function is called a density function. In the Figure 1.c, one can see that density function eases identification of dense regions on the map, even when there is plenty of point

---

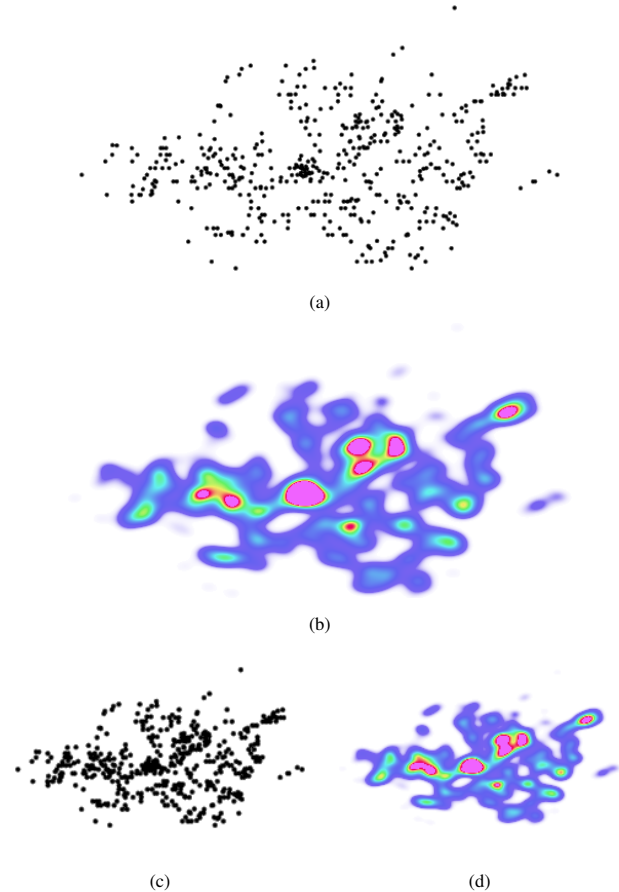*emails: {aperrot,bourqui,hanusse,flalanne,auber}@labri.fr

Figure 1: Point set visualization of the 1854 London cholera epidemic. (a) 512×512 plots of the original point set, (b) 512×512 visualization using a density function, (c) 256×256 rendering, due to occlusion it is difficult to detect dense regions, (d) 256×256 heatmap, even at low resolution, dense regions are easy to detect.

occlusion (Fig. 1.d). Thus, as discussed in [16], density function visualization helps to solve the data change vs visualization change problem illustrated by Figure 1.

Density functions have been used in a large number of applications. For instance, in [31, 6], it enables the analysis of eye-tracking data. Analysis of geolocalised data is also often done with density function visualization [14, 11, 19]. In this case, the heatmaps are displayed on top of a geographical map.

The main motivation of our research was to provide a fully working solution for standard Big Data Infrastructure (BDI). The Big Data ecosystem enables to process big datasets, but introduces some constraints. From an algorithmic perspective, it is preferable to scan the data a constant number of times. Moreover, the data

has to be stored in a distributed way. In order to bound latency, the entire dataset must not be transferred to the location where the visual analysis is done. A pre-computation can be done in order to allow aggregate or part of the dataset to be transferred on the user's computer. For all these reasons, visualization of Big Data is best done in a web browser. This enables visualizations to be stored on a server and sent on demand.

The contribution of this article is a complete fully usable architecture for heatmap visualization at large scale. All the constraints mentioned above have been taken into account to allow its direct integration into modern Big Data Infrastructure. To sum up, for a point set of size n stored within a distributed system of k nodes, there is a preprocessing in time $O(\frac{n \times \log(n)}{k} \times i)$ when the points are uniformly spread on the plane, where i is the number of levels of detail desired. It enables to guarantee the approximation of the density function and the quality of visualization. Such a preprocessing step guarantees a constant size message transfer for any interaction event (zoom in/out or pan). Then, the client computes the visualization in $O(p)$ time, with p the number of pixels on the screen. In a theoretical model of computation, the interaction with our system is in constant time. In practice, our experiments show the relevance of our proposition.

The paper is organized as follows. First, we give an overview of the most related work. Second, we present an overview of the architecture we have set up. We then provide the algorithm details. We conclude by a discussion supported by the benchmark we have done with datasets having more than a billion points.

2 PREVIOUS WORK

2.1 Classical density visualization

The main technique used in the literature to compute the density function of a point set is called *Kernel Density Estimation* (KDE) [28]. Introduced independently by Rosenblatt [27] and Parzen [26], KDE was first used to estimate the density function of a random variable. Using a kernel function, one can generate a continuous function from a discrete point set. Follow the equation of the KDE and the equation of the Gaussian kernel, the most commonly used kernel:

$$KDE(x) = \frac{1}{n}\sum_{i=1}^{n} w_i f_\sigma(dist(x,x_i)) \qquad f_\sigma(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{x^2}{2\sigma^2}}$$

Unformally, a point of weight w spreads a weight $wf_\sigma(d)$ for any location at distance d. The bigger σ is, the further a point will spread its weight. Visually, this represents the amount of "smoothing" of the density function. In the Figure 2 one can see the density function computed using the black points as the original point set. In this Figure, we used a gaussian kernel.

One can find lots of applications of kernel density estimation in the literature. The graph splatting technique [29] introduced by van Liere and de Leeuw uses it to visualize node density in a graph. In that paper, the density function is called a "splat field" and is computed using the OpenGL texture rendering engine and the accumulation buffer. In [17], Lampe and Hauser have proposed another way to compute Kernel Density Estimations with OpenGL, using shaders instead of textures. In both of these works, the density function is only an estimation obtained by limiting the distance at which a point contributes to the density function.

Based on those works, Zinsmaier and Brandes proposed a level-of-detail visualization for large graphs [36]. In that work, the KDE technique is used to show nodes density. The sources and targets of edges are then moved to the nearest local maximum of the node density function using a hill-climbing algorithm. A similar idea has been used for edge bundling with Kernel Density Estimation by Hurter et al. [15].

When using KDE to visualize density function, one needs to compute the density value for every pixel of the visualization. The

complexity of this operation is $O(n * p)$, with n being the number of points in the dataset and p the number of pixels in the visualization. For instance, computing the exact density function of 1 million points on a 2 mega pixels screen (1920x1080) requires 2 trillion operations. This complexity issue makes KDE a candidate for parallel computing. In addition to the GPU techniques and approximation method cited earlier, Michailidis and Margaritis [25, 24] compared KDE implementations with different parallel computing frameworks and Łukasik [20] compared parallel implementations with MPI. Such methods are essential to enabling interactive visualization on large point sets.

2.2 Enabling density visualization for big datasets

In the following we present methods that enable to reduce the size of the original point set before starting the evaluation of the density. These techniques can be used as a pre process of techniques presented above.

Sampling: Sampling is a technique that removes data points from the dataset in order to reduce its size. It is successfully used to reduce overplot in scatterplots. Mayorga and Gleicher [21] used it in conjunction with KDE to help visualize density in cluttered scatterplots. Chen et al. [4] also used an adaptive hierarchical resampling scheme to abstract multi-class scatterplots.

Clustering: Contrary to sampling, clustering merges points, instead of removing them, and represents the merged data by a new point. Clustering is often repeated to produce a hierarchical representation of the data called a cluster tree [1, 2, 8, 10]. One can select an antichain on this tree to serve as dataset abstraction. The selection can depend on view parameters such as zoom and position, enabling interactive visualization.

Binned Aggregation : In ImMens [19], Liu et al. avoid the cost of the KDE computation by using binned aggregation. This process groups data points into predefined "bins" by partitioning the data space. The bin in which a data point is grouped does not depend on other points. It can thus be computed in constant time. This technique has the advantage to be fast to compute and can easily be computed in parallel, since the binning process is independent for each point. This technique can be considered as a case of geometrical clustering. The disadvantage of this technique is that it creates edge effects at the border of the bins. Li et al. [18] applied KDE to binned points to produce a multilevel heatmap visualization.

Out of core visualization: In order to scale up to bigger datasets, it is essential to use out of core architectures. This kind of architecture is useful for multilevel visualization, as demonstrated by Hadwiger et al. [13]. Moreover, the Map-reduce framework is useful for distributing existing algorithms [30]. Using a specialized system, it enables to process spatial datasets and render images of heatmaps [9]. Similarly, Meier et al. [23] have designed an application where the data is stored on a server and delivered on the fly. There are no pre-rendered images and point density function are computed on the client side. Every datapoint is stored with a tile-based index for every zoom-level, enabling fast retrieval of a portion of the map. However, as the authors themselves emphasize, their method is not efficient for large datasets.

None of these methods provide guarantees both toward big data constraints and quality of visualization for large datasets. In the following we propose a technique that shares similar ideas and that enables to scale to datasets with billions of elements, while conserving guarantees toward the quality of the visualization.

3 PROCESS OVERVIEW

The interactive visual exploration of density functions at large scale requires using distributed storage and distributed computation as well as GPU intensive computation. In this section we present an overview of our technique (see Figure 3). The technical details for each step are given in the following sections.
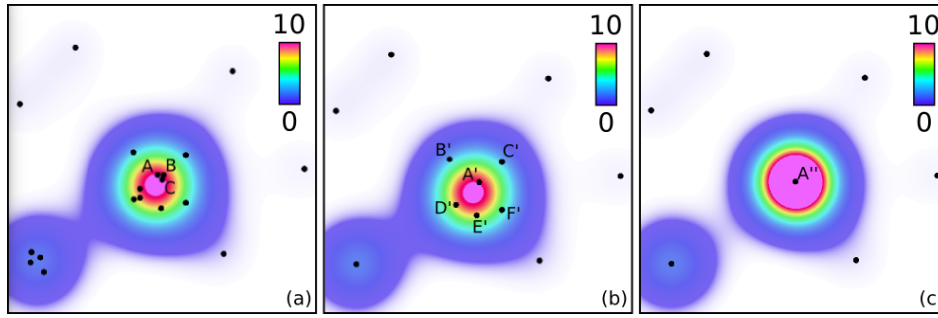
Figure 2: On image (a), the density function visualization of the original point set. The black dots represent the data points. The density has been computed using Kernel Density Estimation with a Gaussian kernel. On image (b), the density function visualization of the corresponding point set for *Approximate Kernel Density Estimation*. The closest points have been merged into a single point whose weight is the sum of the weights of the merged points. For instance, points A, B and C have been merged into A'. The weight of A' is the sum of the weights of A, B and C, which is 3. The differences between the two generated pictures are almost invisible to the naked eye. Using the SSIM image comparison algorithm [32], we obtain a similarity score of 0.959, which corresponds to a "good" rating, according to Zinner et al. [35]. Image (c) shows the result of another merging pass. Points B' through F' have been merged into A'', whose weight is now 10. Differences in the density visualization become visible because the grouping distance is high in respect to the screen space and to the σ used for the KDE.
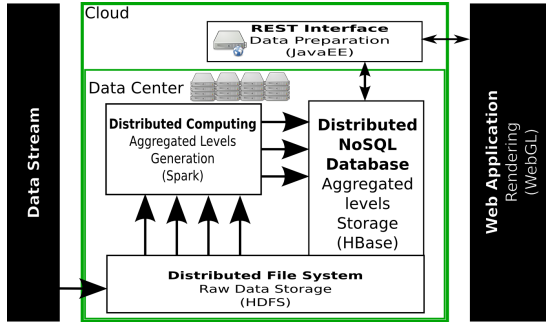

Figure 3: Overview of the system's different steps. Each step is presented with its role and the technology we used. The data stream is first stored in a distributed file system before aggregation. The aggregation is realized on a distributed computing cluster. The result of the aggregation step is stored in a distributed NoSQL database. A front server provides access to the stored aggregation via a REST interface. The client web application requests tiles from the server and render them using an adaptive rendering algorithm. Thanks to the aggregation step, the amount of data transferred is bounded.

Our solution is based on the use of a multi-scale approximation of density functions. For that purpose, we compute several levels. The $level_{max}$ is the entire point set and $level_{i-1}$ is obtained by merging points of $level_i$. Thus, the more aggregation steps we do, the lowest the number of points in the final level, $level_0$. This multi-scale approach enables us to guarantee the quality and processing time of our approximate density function visualization.

To work at large scale all the data is stored into a distributed file system and we designed a distributed algorithm to automatically generate aggregated levels. Combining data distribution and distributed computing enables to run computations directly on the computers containing the data. Even if they are aggregated, the generated levels can be very large, thus we also store them into a distributed filesystem through a NoSQL database. Distributed storage as well as distributed computation are the key features that enable our solution to scale to large datasets.

Once the levels have been computed, the next step is to compute and display the approximate density function on the screen. In our solution that stage is achieved on the client side. That operation requires computation of the density functions for each pixel on the

screen and mapping that density function to a color scheme. It necessitates to transfer data from the distributed storage to the client side. To prevent the transfer of an entire level to the client, every level is split into tiles, which are then stored into a highly accessible distributed database. Only requested tiles are transferred to the client. To get a similar data size per tile the size of tiles are adapted according to the level of aggregation (i.e. the more abstract a level is, the fewer tiles it has). That property guarantees the size of packets transferred on the network and thus smooth interaction according to network bandwidth.

On the client side, we use a Least Recently Used cache system to store tiles of point set transferred during the exploration. For the computation of the density function and the color mapping, we designed a multiscale approximation algorithm that can be fully implemented on the GPU. One of our requirements was to have a lightweight client interface. Therefore, all the design choices for our algorithm have been made to ensure the possibility to implement it on standard recent web browsers.

4   GENERATION OF LEVELS OF AGGREGATION

Starting from the original dataset, we compute several abstractions of the dataset using the Approximate Kernel Density Estimation introduced by Zheng et al. [34].

Let D be a square subspace of $\mathbb{R}^2$. Given the kernel f_σ and P the original point set, an *Approximate Kernel Density Estimation* (AKDE) Q is a weighted point set, such that the KDE on Q approximates the KDE on P. The distance between P and Q is defined as follows :

$$\|KDE_P - KDE_Q\| = max_{x \in D}|KDE_P(x) - KDE_Q(x)|$$

Q is said to be an ε-approximation of P if the distance between P and Q is less than ε. We also want $|Q|$ to be smaller than $|P|$, so that computing the density function on Q is less time consuming than on P.

This technique enables to reduce the number of points for a level of detail, while generating an almost identical density function. Figure 2 shows density function visualization for a point set and two approximations. We use the *Grouping Selection (GS)* technique, presented by Zheng et al. in [34] to generate our AKDE. The *GS* technique groups points closer than a threshold distance d and generates a set of weighted points. According to the authors, the error using this technique is bounded by d/σ, with d the grouping distance and σ the bandwidth (representing the standard deviation) of

the kernel used for KDE. Therefore, we can guarantee and control how our visualization approximates the original point set.

Finding points on a surface distant of at least d is related to the circle packing problem [12]. Given d and a finite square of area S, the number of circles with diameter d that can fit inside this square without intersections is bounded, naively by $\frac{S}{d^2}$. This is the same as determining points distant of at least d. Therefore, the number of points generated by the *GS* technique is bounded, independently of the size of the starting point set. Using results on the circle packing problem, it is possible to determine a grouping distance that ensures a specific upper bound on the size of the point set generated by *GS*.

We implement the *GS* technique by using the canopy clustering algorithm [22]. This algorithm groups points that are at distance at most d from each other. The resulting group is called a *canopy*. Every canopy has a *center* point. A point can start a new canopy if it is further apart from the other centers than d. If a point is closer than d from a center, then it is in its canopy. Each canopy can be represented by its center with a weight equal to the sum of weights of points in the canopy. While the original algorithm uses two different distances, we use a simplified version where those distances are equal. This version of the algorithm is presented in algorithm 1.

Data: A set of points, a threshold distance d
Result: The set of canopies
canopies = empty set;
for *every point p in the set* **do**
    for *every canopy c in canopies* **do**
        if *distance(p,c) ¡ d* **then**
            go to next point;
    add p to canopies;
return canopies

Algorithm 1: The canopy clustering algorithm. The distance to the current point is checked for every canopy. This result in a complexity of $O(n^2)$ for the pseudo code presented here. A better complexity can be achieved using a spatial index data structure, such as a Quad Tree or a KDTree. With those data structures, the complexity is $O(n * \log(|canopies|))$.

4.1 Canopies of level i and the threshold distance d_i

In the following, we build a hierarchical set of canopy centers such that $level_0 \supseteq level_1 \ldots \supseteq level_i$. d_0 is the threshold distance of $level_0$ defined so that $n_0 = |level_0|$ is the maximal number of points (and message size) to transmit to the lightweight client. Depending on the context, we aim at upper bounding n_0 by a small constant (from 100 to 1000). To provide this guarantee, it is easy to show that taking any $d_0 \geq \sqrt{\frac{2A}{n_0\sqrt{3}}}$ is enough assuming that A corresponds to the number of pixels of the screen.

The levels of detail of the heatmap correspond to zoom levels in the visualization. Each zoom level corresponds to a 2^i zoom factor, that is the distance between pixels for the same pair of points seen on the screen is doubled each time we zoom in. The surface of a zoom level on screen is therefore $A_i = A_0/4^i$. Thus, taking $d_i = d_{i-1}/2$ guarantees to have at most n_0 visible points among $n_i \leq \min(n, 4^i n_0)$.

In order to get an efficient and quick computation, we start by computing $level_i$, the initial threshold d_i can be chosen arbitrarily depending on the maximal precision we want. The set of canopy centers of $level_i$ will then be used as the entry of the computation of $level_{i-1}$ taking $d_{i-1} = 2d_i$.
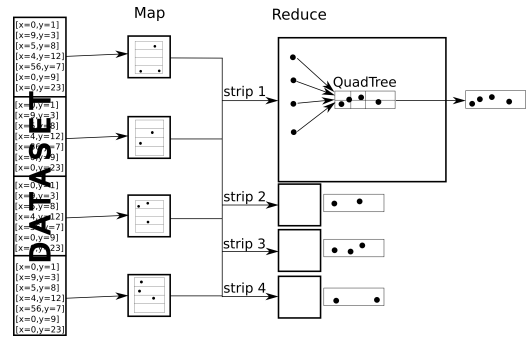

Figure 4: Process of the first Map Reduce job. The map phase assigns a strip to each point, and the reduce phase executes the canopy clustering locally.

4.2 Distributing the algorithm

Introduced in 2004 by Dean et al. [7], MapReduce is now the *de facto* standard framework for distributing Big Data computations. It draws from the functional programming concepts of *map* and *reduce* to provide an easy way to distribute computations on a platform of k nodes. The framework takes care of the synchronization and network aspects and leaves the actual algorithm development to the user. Unformally, massive data of size n is stored in parts of size $O(n/k)$ on k nodes. The map function applies in parallel in time $O(n/k)$, scanning the data and computing intermediate results. These results are sent to nodes, called *reducers*, in charge of combining the intermediate results to get the global result. The usage of MapReduce is based on units of work called jobs. A job is a succession of a Map and a Reduce. Note that the same node can play both mapper and reducer roles.

The smaller the input of the reducers, the quicker the global computation. Ideally, we aim at getting a theoretical time complexity of $O(n/k)$ but this turns out to be difficult whenever a well-known algorithm is adapted into a MapReduce implementation.

The Mahout big data algorithm library[1] provides a MapReduce implementation for the canopy clustering. In this implementation, the dataset is randomly partitioned between the mappers. Each mapper runs a local canopy clustering on its subset of the data. The resulting canopies are then transferred to a single reducer. This reducer applies another local canopy clustering. This implementation is well suited in cases where the set of canopies resulting from the mappers is small enough to fit on a single machine. However, due to the necessity to generate very abstract levels as well as detailed ones, our system must be able to cope with a very large (although bounded) number of canopies. Therefore, we could not use the Mahout implementation of the canopy clustering for our system. We thus designed a new implementation. Our implementation of the canopy clustering no longer uses a single reducer and thus, our approach can scale horizontally.

The random partitioning used in Mahout can lead to calculate the distance between points that are very far apart. We propose to use a more specialized data partition guaranteeing that the n_i points are scanned at most twice. We start from a partition of data into strips that cuts the dataset along one of the dimensions. Let $h = 4d$ be the height of a strip, if two points are in the same strip, their distance along one of the dimensions is at most h. This reduces the distance calculations to a more useful subset. In both of our jobs, the mappers will be responsible for the data partitioning and the reducers will execute the effective clustering process. One trick is that a reducer is in charge of a random set of strips. Even if some strips contain much less than n_i/h points, the reducer will receive at
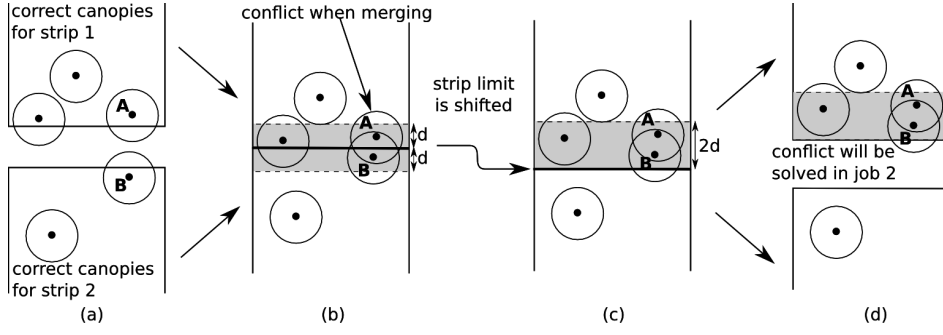
---

[1] http://mahout.apache.org

Figure 5: An example of the data partition in the second job, focusing on the border of two strips. Image (a) shows the result of the first job. When the strips are merged, as shown on image (b), it creates a zone called "conflict zone", whose height is $2d$. This zone is colored in grey. In this zone, two points coming from the two strips can be closer than d. Image (c) shows the partitioning used in the second job. In order to be able to process the entirety of the conflict zone in a single strip, the strip limit is shifted downwards by d. As illustrated on image (d), the conflict zone is then totally contained in the upper strip. The conflict can be detected and solved. One of the points will be removed from the canopy set during the second job.

most $O(n_i/k)$ whenever the data distribution is close to be uniform.

First job: In the first job, the mappers receive points from the dataset. For each point, the index of the strip containing it is computed. Then the mapper emits a key value pair for each point, where the key is the strip index and the value is the point itself. Every record with the same key is then transmitted to the same reducer. Each reducer thus has every point inside a given strip. The canopy clustering is then executed inside each strip. We use a QuadTree to reduce the complexity, as suggested in algorithm 1. The pseudo code for the Map and Reduce phases of the job can be seen in algorithm 2. The Figure 4 shows how data is partitioned, transferred and assigned to reducers. After this process, the points are marked according to the result of the clustering.

Map *point*:
    emit (strip(point), point);

Reduce *(strip, points)*:
    canopies = new QuadTree;
    for *every point p in points* **do**
        s = elements in canopies with dist(p) less than d;
        if *s is empty* **then**
            add p to canopies;
    for *c in canopies* **do**
        emit c;

Algorithm 2: First job pseudo code for the Map and Reduce functions. The Map computes the strip for each point. The Reduce executes the canopy clustering on each strip.

Second job: Once the first job is completed, the canopy clustering has been executed inside of each strip. However, as shown on Figure 5, when the strips are put back together, a point A close to the border of its strip can be closer than d from a point B in the neighbouring strip. In this case, both A and B are closer than d from the limit of the strips. We can define the zone which contains every such point. This zone is located at the border of adjacent strips and extends to a distance d from either side of the border. We call this zone "conflict zone".

The second Map Reduce job of our implementation aims at executing the canopy clustering on the conflict zone. We retain the idea of strips for data partitioning. However, the strip limit is shifted downwards by d, as shown on Figure 5. This way, the conflict zone is now contained in a single strip. This will enable to execute the canopy clustering on this area inside of each reducer.

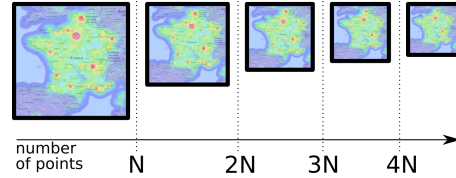This time, it is not necessary to execute a full canopy clustering

Figure 6: As the data grows, the resolution of the KDE diminishes to keep a constant rendering time.

on the strip. Canopies outside the conflict zone are already valid. Only canopies inside the conflict zone need to be treated. Using the QuadTree, we check if a canopy is already closer than d. If not, we add this canopy to the QuadTree.

The data partition technique used in this implementation can induce load-balancing issues if the data points distribution is not uniform enough. It is then possible for a single strip to contain many more points than the others. However, when applying this algorithm recursively to generate the levels, only the first one can suffer from this issue. Once the algorithm has been executed a first time, the point set has a bounded size. This property is conserved when dividing it into strips, thus guaranteeing a maximum number of points per strip.

5 KDE RENDERING

The final step of our system is the rendering of the density function. This part happens on the client side after the transfer of the visible tiles. Due to the complexity of KDE calculation, approximations of the computation by limiting the kernel size have been developed in previous works [17, 36]. Here, we compute an exact KDE to be able to interactively change the kernel's size and type.

We use a simple technique that enables us to guarantee a constant rendering time on a given client at the expanse of KDE precision. The main idea of our technique is to adapt the rendering resolution according to the power of the GPU. The KDE is rendered on a smaller resolution when there are too many points, ensuring that the total number of operations needed is constant. The resulting smaller resolution KDE is then stretched to the original screen's resolution.

We introduce a threshold N of points that can be rendered at the original screen resolution in a small enough time to maintain satisfactory framerate. When we need to render more points than N, we will render the KDE on a smaller resolution. We define $m = \lceil \frac{n}{N} \rceil$, with n the number of points to render. Instinctively, m measures the difference of magnitude between the data to render and the ac-

ceptable amount. We want to render a smaller KDE on $p' = \frac{p}{m}$ pixels. In order to keep the screen's aspect ratio, we take $w' = \frac{w}{\sqrt{m}}$ and $h' = \frac{h}{\sqrt{m}}$, with w and h respectively the width and height of the screen. We thus have $p' = w' * h'$. This interrelation is illustrated in Figure 6.

Rendering the n points costs $p' * n = \frac{p*n}{\lceil \frac{n}{N} \rceil}$. Using the properties of the ceiling function, it is easy to show that $\frac{p*n}{\lceil \frac{n}{N} \rceil} \leq p * N$, so the complexity of the smaller KDE rendering is $O(p * N)$. This only depends on the display resolution and the threshold constant. Considering that these values only depend on the client computer, one can consider that, for a given computer, our rendering algorithm runs in $O(1)$.

6 IMPLEMENTATION DETAILS

The techniques and algorithms we describe here could be implemented using several technologies. We provide here an exhaustive list of the technologies we used. This constitutes a working set for a proof of concept. Figure 3 shows the complete system.

In the domain of Big Data, Hadoop is a well established standard and covers the majority of our needs. We use the Hadoop Distributed File System (HDFS) as raw data repository before processing. There are several implementations of the Map Reduce paradigm on top of Hadoop, via its resource manager YARN (Yet Another Resource Negotiator). The main one is Hadoop MapReduce2 (HMR2), which mainly uses disk based I/O between the computations. We used another Map Reduce implementation called Spark, which is also built on top of YARN and HDFS. Spark favors in-memory computation instead of the disk-based I/O used by HMR2. Spark is based on the principle of Resilient Distributed Datasets (RDDs) [33] to distribute computations closest to the data. In addition to the Map and Reduce operations, Spark provides several other operations optimized for distributed computing. This flexibility allows Spark to be easier to use than HMR2. In practice, we noted speed gains of order 100 from using Spark instead of HMR2.

The size of the data generated by our clustering algorithm demands a NoSQL distributed database. We chose to use the Hadoop database, HBase, which is an open source implementation of Google's BigTable [3] database system.

The link to the client application is made using a front server implemented using a JavaEE servlet. The task of this front server is to request data from HBase, and transform it into a format understandable by the client. This way, the client application only needs to know the front server location and does not directly contact the big data cluster.

We implemented the client application to run on modern browsers using HTML5 and WebGL for rendering. As a threshold for the KDE rendering, we use the variable $MAX\_FRAGMENT\_UNIFORM\_VECTORS$, which is the maximum number of variables that can be sent to the fragment shader.

7 BENCHMARKS

The visualization of density functions on top of geographical maps (Geographical heatmaps) is frequently done by datavisualizers. To run this benchmark, a complete prototype for that specific application has been implemented. It uses the technologies presented in the section 6, as well as Google Maps.

To enable the use of our system on low bandwidth network, the number of points per tile n_0 (cf. sec 4.1) has been set to 1000. This parameter enables to transfer only a few kilobytes (maximum 15KB in all our tests) on the network during interactions.

Infrastructure: The clustering benchmark has been run on our lab infrastructure composed of 16 computers (non virtualized). Each computer has 64GB of RAM, 2x6 hyperthreaded cores at
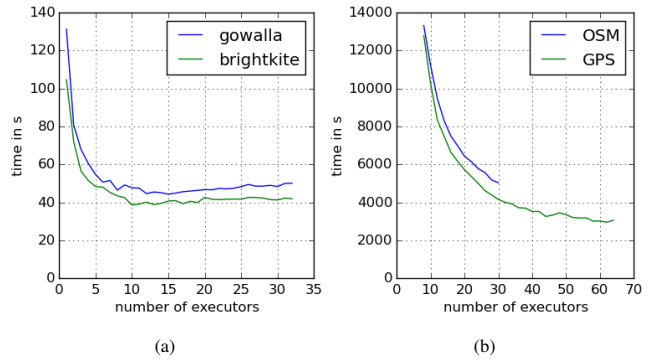

Figure 7: (a): Benchmark for the brightkite and gowalla datasets. Each executor has 4 cores. The optimal number of executors is attained around 10 because of the small size of those datasets. (b): Benchmark for the OSM and GPS datasets. Each executor has 2 cores. The benchmark for the OSM dataset was stopped at 30 executors due to memory constraints. This benchmark shows the scalability of our clustering algorithm.

2.1GHz and 2 hard drives of 1 TB each. The computers are linked by a 1Gb/sec network infrastructure. During program execution, one computer serves as the master node and 15 as slave computing nodes. The rendering benchmark was conducted with an NVIDIA 970M graphics card for the GPU rendering and a Core i7-4710HQ for the CPU rendering.

Datasets: We used four free datasets with sizes of different orders of magnitude, two small and two large. We give the datasets size in terms of records and in terms of unique positions, since the computational complexity of a density function depends on the number of different positions. The smallest one is the Brightkite dataset (4.7M records and 693K different positions). This dataset [5] has been built by recording the positions of each Brightkite user between April 2008 and October 2010. Heatmaps of this dataset have been made by Liu et al. [19] and Li et al. [18]. A view of this dataset in our application can be seen on Figure 8. The Gowalla dataset [5] (6.4M records and 1.2M different positions) is similar to the Brightkite dataset but for Gowalla's users between february 2009 and october 2010. The OpenStreetMap (OSM) dataset[2] (2.2B records and 1.7B different positions), this one represents every point of interest registered in the OpenStreetMap database. On Figure 8, one can grasp the scale of this dataset. Finally, the biggest dataset we used is a collection of GPS traces registered in the OpenStreetMap database (2.7B records and 2.2B different positions). The OSM and GPS datasets have very different point distributions, with OSM being more evenly spread than GPS. This made the OSM dataset more demanding in terms of memory needed for the clustering process, despite its lower number of unique positions.

Clustering benchmark: For each dataset, 21 levels of detail were generated. In HBase, the storage of these 21 levels necessitates 381MB for the brightkite dataset, 668MB for the gowalla dataset, 609GB for the OpenStreetMap dataset and 321GB for the GPS dataset. Unsurprisingly, there is an additional cost due to the storage of the entire level hierarchy. The difference in storage size between OSM and GPS is due to their different point distribution, leading to quicker aggregation. The figure 7 presents the results obtained on our lab infrastructure. We measured the scalability of our implementation using Spark executors. In Spark, executors are computation workers, enabling a finer granularity than computers. This enables to test finely the behaviour of the program

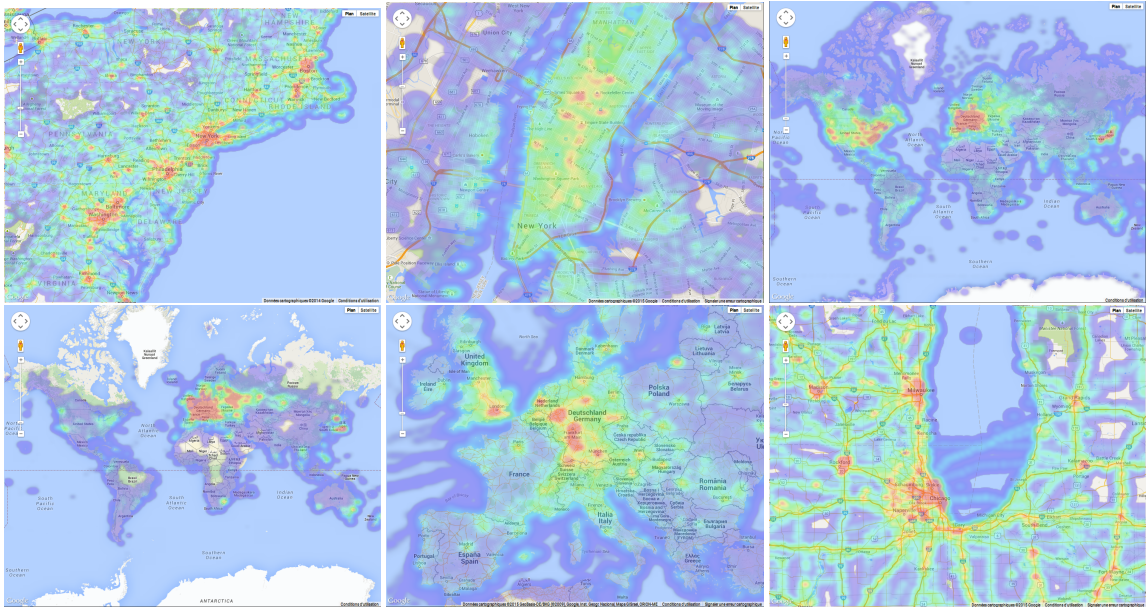---

[2]http://spatialhadoop.cs.umn.edu/index.html

Figure 8: Views of the four datasets, and the number of points displayed. Top row, from left to right: brightkite (1671), gowalla (1334), and OSM (2873). Bottom row: GPS dataset, global view (3185), Europe view (3218) and Chicago view (1468). On the GPS view, the high density in Europe is clearly highlighted. From all those view, it is clear that the density function is accurately represented using much fewer points. Furthermore, independently of the zoom level, the number of points displayed has the same magnitude.
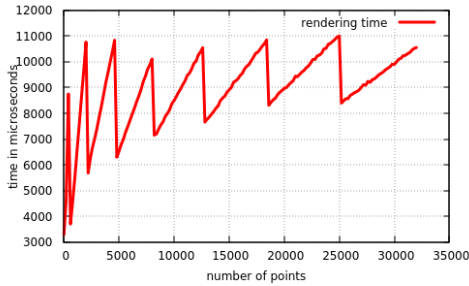

Figure 9: GPU rendering time on an NVIDIA 970M, up to 32000 points. In this case, $N = 512$. The sudden drops correspond to the changes of KDE resolution. One can see that the rendering is bounded by a constant. In practice, with $n_0 = 1000$, there is no more than 16000 points in a 1000*1000 visualization.

when the resources grow. Our benchmark demonstrates clearly that our approach is able to scale horizontally when treating really large datasets. For example, generating the levels for the GPS dataset took less than 45 minutes with 64 executors having 2 cores each.

Rendering benchmark: We now demonstrate that our technique is able to render KDEs with a large number of points in bounded time. Figure 9 shows the results of a benchmark with up to 32000 random points. The rendering time always stays below 11ms, which corresponds to more than 90fps. As expected, this technique maintains an acceptable rendering time by diminishing the KDE resolution when the number of points grow, as highlighted by the drops in Figure 9. Between the resolution changes, the rendering time scales linearly with the number of points.

Image Quality benchmark: To show the relevance and efficiency of the pipeline proposed here, we compare its results with a complete rendering of the entirety of the data. For this experiment, we chose the brightkite dataset because its small size enables local computations. We use a CPU rendering of this dataset as the ground truth and compare it to the levels of abstraction produced by our pipeline, rendered both on CPU and GPU. For this comparison, we use the SSIM image metric [32], which gives a score of similarity between two images. This metric is known to be closer to human perception than per-pixel error metrics. Figure 10 shows the resulting similarity scores. The quality is first limited by the clustering algorithm and then by the rendering algorithm. This creates a "bell" shape of optimal settings. From the angle of image quality, it is thus equivalent to render with more points on a smaller resolution or at full resolution with fewer points. This property demonstrates the validity of our approach, and opens the perspective to adapt the abstraction level to the device used.

8 CONCLUSION & FUTURE WORK

We presented a technique that enables interactive exploration of point sets at really large scale. We introduced a new distributed algorithm to compute a canopy clustering as well as a GPU based method to render density functions in constant time, supported by a theoretical justification. To validate our assumptions, a full implementation of the technique was made and a complete benchmark on large datasets has been run. All our results prove the efficiency of our approach in term of both horizontal scalability and quality of the produced visualization. Our technique is thus able to interactively explore sets of points of any size, the only limiting factor is the size of the Big Data Infrastructure.

In its current implementation, this system is not able to support incremental updates, however, the clustering algorithm used is fully suited for the task. Future works will be to extend that method for the visualization of dynamic set of points. New Big Data technologies, like Storm, must be investigated to produce efficient solutions.
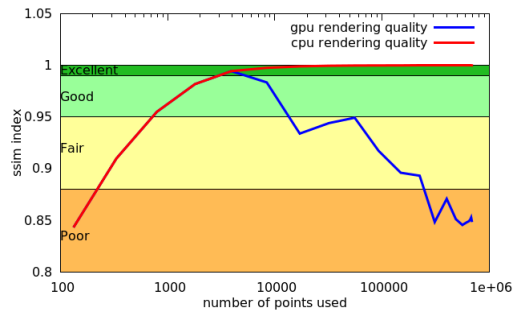
Figure 10: Comparison of the ground truth to the abstractions generated. The colored zones denote the user ratings proposed by Zinner et al. [35]: dark green for "Excellent", light green for "Good", yellow for "Fair" and orange for "Poor".

REFERENCES

[1] J. Abello, F. Van Ham, and N. Krishnan. Ask-graphview: A large scale graph visualization system. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):669–676, 2006.

[2] D. Auber and F. Jourdan. Interactive refinement of multi-scale network clusterings. In *Ninth International Conference on Information Visualisation*, pages 703–709. IEEE, 2005.

[3] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[4] H. Chen, W. Chen, H. Mei, Z. Liu, K. Zhou, W. Chen, W. Gu, and K.-L. Ma. Visual abstraction and exploration of multi-class scatterplots. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):1683–1692, 2014.

[5] E. Cho, S. A. Myers, and J. Leskovec. Friendship and mobility: user movement in location-based social networks. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1082–1090. ACM, 2011.

[6] E. Cutrell and Z. Guan. What are you looking for?: an eye-tracking study of information usage in web search. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 407–416. ACM, 2007.

[7] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[8] J.-Y. Delort. Vizualizing large spatial datasets in interactive maps. In *2010 Second International Conference on Advanced Geographic Information Systems, Applications, and Services (GEOPROCESSING)*, pages 33–38, Feb. 2010.

[9] A. Eldawy, M. F. Mokbel, S. Alharthi, A. Alzaidy, K. Tarek, and S. Ghani. Shahed: A mapreduce-based system for querying and visualizing spatio-temporal satellite data. *IEEE 31st International Conference on Data Engineering (ICDE)*, 2015.

[10] N. Elmqvist and J.-D. Fekete. Hierarchical aggregation for information visualization: Overview, techniques, and design guidelines. *IEEE Transactions on Visualization and Computer Graphics*, 16(3):439–454, 2010.

[11] D. Fisher. Hotmap: Looking at geographic attention. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1184–1191, Nov. 2007.

[12] M. Goldberg. The packing of equal circles in a square. *Mathematics Magazine*, pages 24–30, 1970.

[13] M. Hadwiger, J. Beyer, W.-K. Jeong, and H. Pfister. Interactive volume exploration of petascale microscopy data streams using a visualization-driven virtual memory approach. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2285–2294, 2012.

[14] H. Hotta and M. Hagiwara. Online geovisualization with fast kernel density estimator. In *IEEE/WIC/ACM International Joint Conferences on Web Intelligence and Intelligent Agent Technologies, 2009. WI-IAT '09*, volume 1, pages 622–625, Sept. 2009.

[15] C. Hurter, O. Ersoy, and A. Telea. Graph bundling by kernel density estimation. In *Computer Graphics Forum*, volume 31, pages 865–874. Wiley Online Library, 2012.

[16] G. Kindlmann and C. Scheidegger. An algebraic process for visualization design. In *InfoVis*. IEEE, 2014.

[17] O. D. Lampe and H. Hauser. Interactive visualization of streaming data with kernel density estimation. In *Pacific Visualization Symposium (PacificVis), 2011 IEEE*, pages 171–178. IEEE, 2011.

[18] C. Li, G. Baciu, and Y. Han. Interactive visualization of high density streaming points with heat-map. In *Smart Computing (SMARTCOMP), 2014*, pages 145–149. IEEE, 2014.

[19] Z. Liu, B. Jiang, and J. Heer. imMens: Real-time visual querying of big data. In *Computer Graphics Forum*, volume 32, pages 421–430. Wiley Online Library, 2013.

[20] S. Łukasik. Parallel computing of kernel density estimates with mpi. In *Computational Science–ICCS*, pages 726–733. Springer, 2007.

[21] A. Mayorga and M. Gleicher. Splatterplots: Overcoming overdraw in scatter plots. *IEEE Transactions on Visualization and Computer Graphics*, 19(9):1526–1538, 2013.

[22] A. McCallum, K. Nigam, and L. H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 169–178. ACM, 2000.

[23] S. Meier, F. Heidmann, and A. Thom. Heattile, a new method for heatmap implementations for mobile web-based cartographic applications. In T. Bandrova, M. Konecny, and S. Zlatanova, editors, *Thematic Cartography for the Society*, Lecture Notes in Geoinformation and Cartography, pages 33–44. Springer, Jan. 2014.

[24] P. D. Michailidis and K. G. Margaritis. Accelerating kernel density estimation on the gpu using the cuda framework. *Applied Mathematical Sciences*, 7(30):1447–1476, 2013.

[25] P. D. Michailidis and K. G. Margaritis. Parallel computing of kernel density estimation with different multi-core programming models. In *21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 77–85. IEEE, 2013.

[26] E. Parzen. On estimation of a probability density function and mode. *The Annals of Mathematical Statistics*, 33(3):1065–1076, 09 1962.

[27] M. Rosenblatt. Remarks on some nonparametric estimates of a density function. *The Annals of Mathematical Statistics*, 27(3):832–837, 09 1956.

[28] B. W. Silverman. *Density estimation for statistics and data analysis*, volume 26. CRC press, 1986.

[29] R. Van Liere and W. De Leeuw. Graphsplatting: Visualizing graphs as continuous fields. *IEEE Transactions on Visualization and Computer Graphics*, 9(2):206–212, 2003.

[30] H. T. Vo, J. Bronson, B. Summa, J. L. D. Comba, J. Freire, B. Howe, V. Pascucci, and C. T. Silva. Parallel visualization on large clusters using mapreduce. In *IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, pages 81–88. IEEE, 2011.

[31] O. Špakov and D. Miniotas. Visualization of eye gaze data using heat maps. *Electronics and electrical engineering*, 2:55–58, 2007.

[32] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004.

[33] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[34] Y. Zheng, J. Jestes, J. M. Phillips, and F. Li. Quality and efficiency for kernel density estimates in large data. In *Proceedings of the 2013 international conference on Management of data*, pages 433–444. ACM, 2013.

[35] T. Zinner, O. Hohlfeld, O. Abboud, and T. Hoßfeld. Impact of frame rate and resolution on objective qoe metrics. In *International Workshop on Quality of Multimedia Experience*, pages 29–34. IEEE, 2010.

[36] M. Zinsmaier, U. Brandes, O. Deussen, and H. Strobelt. Interactive level-of-detail rendering of large graphs. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2486–2495, 2012.