HeatPipe: High Throughput, Low Latency Big Data Heatmap with Spark Streaming

Alexandre Perrot, Romain Bourqui, Nicolas Hanusse, David Auber

HeatPipe: High Throughput, Low Latency Big Data Heatmap with Spark Streaming

Alexandre Perrot, Romain Bourqui, Nicolas Hanusse, David Auber
Univ. Bordeaux, LaBRI, UMR 5800, F-33400 Talence, France
CNRS, LaBRI, UMR 5800, F-33400 Talence, France
{aperrot,bourqui,hanusse,auber}@labri.fr

Abstract

Heatmap visualization is a well-known type of visualization to alleviate the overplot problem of point visualization. As such, it is well suited to visualize Big Data. In order to tackle the velocity problem of Big Data, one has to leverage streaming computations. Recently, canopy clustering was shown to be well suited for Big Data heatmap visualization. In this article, we present how to design a streaming algorithm to compute canopy clustering using Apache Spark. This result is directly applicable to be included into a lambda architecture.

Keywords— **Big Data, Information Visualization, Heatmap, Lambda Architecture.**

1 Introduction

At the beginning of the 21st century, the generalization of broadband Internet access led to an increase in the amount of data generated and collected. This is the phenomenon we now know as "Big Data". The characteristics of Big Data are most often described as the "3 V" : Volume, Variety and Velocity. Volume refers to the sheer amount of data that needs to be stored to be later analyzed. Variety emphasizes the fact that the collected data is unstructured, from many different sources and of different types: social media posts containing images, links and text, geolocalized usage data or products reviews and sales, for example. Finally, Velocity refers both to the speed at which data is generated (expressed in events per second) and the fact that data is most of the time ephemeral, as it can quickly be obsolete or replaced by an updated version.

In the domain of visualization, the phenomenon of Big Data induces an increase of data to be visualized. However, the amount of data that can be displayed on a screen is inherently limited by its number of pixels. Trying to represent more data points on the screen than available pixels leads to a problem called overplotting, that is several data points will be represented in a single pixel without visual cues regarding the number of points it represents.

One of the solutions to alleviate the overplotting problem is to use a different kind of visualization that does not directly represent the points, but rather their density, called density visualization or heatmap. Heatmap visualizations are known to scale well in terms of perception, but their computing time on large datasets can be prohibitive.

In order to be able to process Big Data, new distributed systems have emerged. The Hadoop ecosystem has become the *de facto* standard for Big Data processing. Hadoop includes the HDFS distributed file system, enabling to store huge datafiles redundantly to avoid data loss. It also includes computing capabilities with the MapReduce framework. Such Big Data frameworks enable to process Tera and even Peta bytes of data with a powerful enough cluster. However, they rely on batch processing, i.e. processing all the data at once. The downside of this approach is that incoming data may wait a long time before being processed, so the last processed data is always outdated as soon as it is computed. Depending on the delay (hours, days or weeks), this can be problematic in situations where a decision has to be made rapidly to adapt to the current events.

This problem can be solved by using streaming computation frameworks. They enable to treat incoming data as it enters the system, always keeping the processed data up to date. However, some data processing algorithms require to iterate over the entire dataset. Such iteration is too time consuming when one wants to update the data as it arrives in the stream. Thus, batch and streaming computations are used simultaneously to build systems that can keep up to date information on incoming data. This is in essence the principle of the so-called *lambda architecture* described by Marz & Warren [10].

In this article, we expand on the *LIViD* batch solution [15] to create an algorithm for streaming multilevel heatmap computation. First, we present the state of the art in density visualization and big data processing frameworks. Then, we present the overview of our method before showing the algorithm. Finally, we give some insights relative to the implementation and performance of our solution.

2 Previous work

Heatmap visualizations are a good way to visualize point data to alleviate the overplot problem. They generally use *Kernel Density Estimation (KDE)* [14, 16] to compute a density function for the point set. Informally, KDE allows to spread the weight of each point across the screen using a kernel (generally Gaussian). In the general case (any kernel), computational complexity is $O(np)$, with n the number of points and p the number of pixels. Then, the density information of each pixel is mapped to a color using a color scale. Such visualizations are popular for geographical data [3, 4, 5] and graphs [17, 20]. However, the computational complexity of KDE makes it difficult to use on Big Data, even though it would solve overplot, which is a recurrent problem with large amounts of data.

Recent research have proposed parallel implementations to address that scalability issue. Michailidis compared several multi-core implementations of KDE with different frameworks [13] and presented results of accelerating th KDE computation using a GPU and the CUDA framework [12]. As shown by Lukasik [9], the MPI framework can also be used to speed up the KDE process, either as a single-machine multi-core computation or a distributed process spanning multiple computers.

All those techniques still require to consider all the points when rendering the heatmap. This is not possible with Big Data, since the whole dataset might not fit on the client computer or may require too much time to be transferred through the network on a massively parallel architecture. A solution was recently proposed by Perrot et al. [15]. They designed a system which enables interactive heatmap visualization of billions of points on big data infrastructure. For that purpose, they leverage theoretical results on KDE [19], the canopy clustering algorithm [11] and the Hadoop platform to achieve horizontal scalabilty and thus push out the limit of heatmap visualization. However their technique only works with batch processing and therefore is not suitable for heatmap visualization of streamed data.

To our knowledge, only few results address the problem of heatmap visualization of streamed data [6,7,8]. Furthermore, they only consider complete datasets and generate dynamic visualization as a movie of past events.

2.1 Big Data and Lambda Architecture

Map Reduce [2] is the foundation of many Big Data computing frameworks. It allows to easily distribute complex computations by expressing them as a succession of *map* and *reduce* operations. These computations run on the entire dataset at once and can last hours to days, depending on its size. This is called batch computation, and the result is only accessible once the computation is finished.

Streaming computation enable to process incoming data as soon as it is ingested into the system. This enables to have a result accessible in almost real-time (i.e. with a very small delay). However, only simple computations can be efficiently executed in such a short amount of time. Two models for streaming computations emerged : *one-at-a-time* and *micro-batch*. The one-at-a-time model processes each record as soon as it arrives. This model gives the best latency between arrival and end of computation, but is unable to cope with large amounts of incoming data. On the other hand, the *micro-batch* computation model allows a high data throughput by processing data in small batches. Micro-batches are started at regular intervals called *micro-batch interval*. Each batch thus processes data received since the start of the previous micro-batch. The micro-batch interval typically varies from a few milliseconds to several seconds.

The general concept of *Lambda Architecture* has been introduced by Marz & Warren [10]. The Lambda Architecture balances the long running time and latency of batch computation with streaming computations. In this architecture, data is being ingested by two parallel layers : the batch layer and the speed layer. The lambda architecture rationale is that queries on big datasets cannot be efficiently computed in real-time over the entire dataset. Thus, to be able to answer specific queries, some intermediate data must be computed. This intermediate data is called a "view" and will enable to answer the desired query in a reasonable time. Different queries may need different views to be computed. The system maintains two types of views : the *batch views* and the *realtime view*. The batch views are computed over the entire dataset collected up to the start of the last batch computation. The real-time views are computed only using data that has been collected since the last batch. This way it is possible to keep the system updated using the streaming computations. Answering a query over the whole dataset requires to get the appropriate data from both types of views and merge the result depending on the query. The part of the system responsible for querying the correct views and merging the results to create the response sent to the user is called th *serving layer*. In a streaming environment, input data is constantly being ingested into the system. The rate of data arrival is the primary metric for performance, rather than the total data size to process, as used in the batch environment.

In the *Lambda Architecture*, any incoming data will first be treated by the speed layer, before being processed by the batch layer. The purpose of the speed layer is to give views upon the newest data as fast as possible. To be able to process incoming data, it is often needed to resort to approximations or to relax some constraints. The views generated by the speed layer are only temporary. Data will eventually be processed using a more accurate algorithm in the batch layer. In the speed layer, there is a tradeoff between speed and accuracy of the views.

3 Heatmap Speed Layer

As described above, Perrot et al. [15] have proposed an efficient solution to implement a batch layer for heatmap visualization. Their solution uses a multilevel aggregation of the data to store weighted abstract point set with increasing levels-of-detail into the serving layer (i.e. the views). These abstract weighted point sets enable to provide in constant time a heatmap visualization to the end user. To organize the levels of details, the so-called *tile pyramid*, as used by online mapping systems, is used. Computation time as well as the insertion of the pyramid into the serving layer require several minutes with dataset having 1 billion of points. However, after that preprocessing, this organization leads to only transfer the necessary parts of the abstract point set through the network and thus provide realtime exploration experience to the end user.

In the following, we propose a speed layer that can be used together with the Perrot et al [15] batch layer to build a lambda architecture which enables realtime exploration experience on streamed data. More precisely, the speed layer will maintain a partial abstract pyramid which will be merged on the fly with the batch layer pyramid by the serving layer. It enables to provide an up to date tile (abstract point-set subset) to any query of an end user.

3.1 Pyramid building

The process of canopy clustering [11], described in Algo.1, reduces the number of points by selecting representatives. Given an aggregation distance d, it guarantees that no point is further than d from a representative and no two representative are closer than d. Furthermore, it guarantees a bounded maximal number of representative in relation with the chosen d, as shown in [15].

Algorithm 1: The canopy clustering algorithm.

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

The canopy clustering algorithm can be adapted for streaming computations. The general idea of a streaming version is as follows : whenever a new point enters the system, check if a representative exists and merge the point or create a new representative accordingly. In the present context of multiscale visualization, this operation must be done for every level of detail intended to be used in the visualization. In the tiles where no representative for this point exist, the point itself must be inserted. When a representative does exist, its weight must be updated.

To be able to process a high number of points in a streaming fashion, two obstacles must be lifted to increase the parallelism of the computation. The first obstacle is that the *LIViD* batch algorithm processes the levels of detail in a bottom-up fashion. This is because the amount of data to process is huge, and each subsequent aggregation reduces it, making the next level less expensive to compute. In a streaming environment, the number of points per iteration is much smaller and we want to minimize the computation time. Processing levels one after the other would mean a higher latency. Thus, it is necessary to treat each level independently, so as to increase the level of parallelism.

The second obstacle is that choosing representatives is dependant on the order of arrival of the points. This is not a problem when the points are processed sequentially. However, this is not a viable option if one wants to process Big Data in a streaming fashion. The batch algorithm alleviates this in a classical way, by partitioning space in independent regions, before solving conflicts between the regions. In order to increase the parallelism for the streaming version, we will need to further partition space using the *tile pyramid* to group points. In the following, we explain how we designed the streaming algorithm for higher parallelism.

4 Algorithm

Our streaming canopy clustering algorithm is composed of two steps, both designed to be scalable for high throughput. First, incoming points are duplicated for each level and grouped according to the tiles they belong to. This enables to process each tile of each level independently. Then, representatives are elected inside each tile using the sequential canopy clustering algorithm.

This gives a pyramid of tiles similar to the output of the *LIViD* batch process. The first difference is that conflicts have not been resolved across tiles, to allow faster parallel processing. Thus, the total number of points generated at this step should be slightly higher than what would have been generated by the batch algorithm. However, it cannot exceed the allowed number of representatives per tile. The second difference is that there is no hierarchical relationship between the representatives anymore, since the different levels are processed independently. The set of tiles computed this way is an approximation of the result of the batch algorithm, it produces more points and thus more data transfer on the network. It is thus a more faithful representation of the original point set at the expanse of bigger data transfer, which can degrade the user experience. Fig. 1 illustrates the process over the course of 3 micro-batches.

The second step of the streaming canopy clustering is to merge the tiles obtained for the current micro-batch with the current state obtained by previous micro-batches. The merge operation is a new canopy clustering pass, where

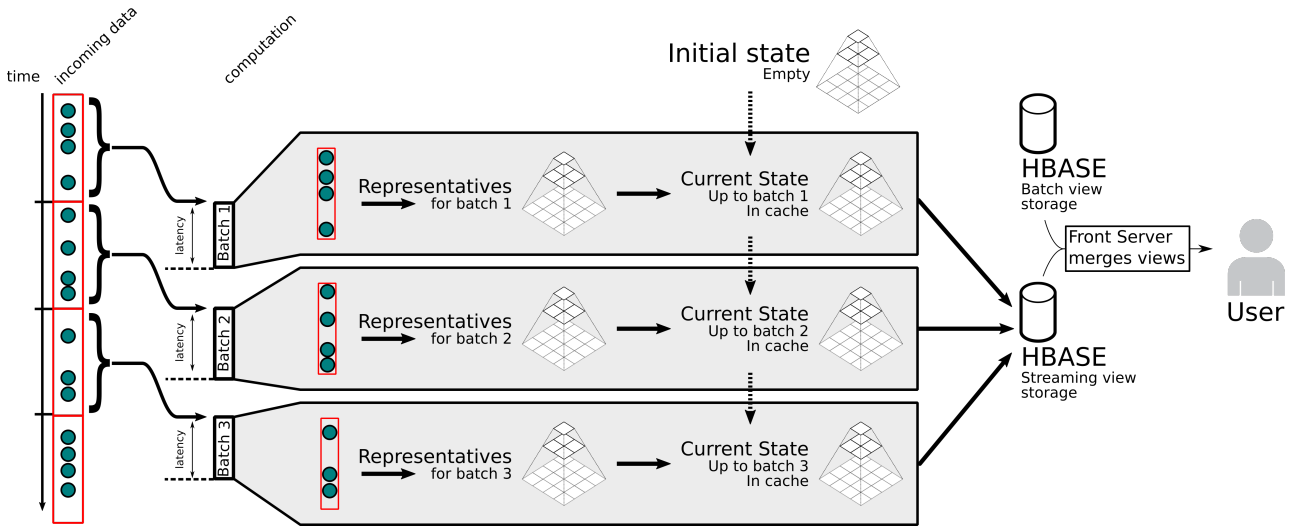

Figure 1: Illustration of the streaming canopy clustering. Incoming data is divided into micro-batches. Each micro-batch builds a tile pyramid with its representatives and merges it into the current state. This state is kept in the Spark streaming cache for quicker access for the streaming process and is stored in HBase for querying from the visualization.

the previous state is considered as existing representatives and the representatives of the current micro-batch as new points. This state is also a tile pyramid and represents an approximation of the density function of all the points ingested by previous micro-batches. Only tiles which have changed need to be updated. This gives a tile pyramid for the incoming points in an incremental fashion.

Finally, the serving layer of the *lambda architecture* can merge the batch view and the real-time view simply by concatenating the tiles. This gives a number of points for the final visualization that is twice the amount fixed in the batch layer.

events/s	interval	events/int.	mean total delay
6000	5s	30000	2.552s
5000	2s	10000	1.526s
10000	2s	20000	1.511s

Table 1: Several stability test of our system using different parameters. The mean total delay is the time between the start of a micro-batch and its completion. The system is considered stable if it is less than the micro-batch interval.

5 Implementation

For our implementation, we used the Spark Streaming library [18]. This library implements the micro-batch streaming model that we used to design the streaming canopy clustering. Furthermore, it is readily included in Spark and uses the same mechanisms for data caching, data parallelism and fault tolerance. The 1.6 version includes the method *mapWithState*, which enables to maintain state across micro-batches and update it using data from each micro-batch. The state information is stored as a RDD, thus benefiting from the standard replication, distribution and caching capabilities of Spark. When updating the state with new records, computations are started only for records that need to be updated. Finally, the tiles are stored in HBase for easy access from the serving layer. Only the updated tiles need to be reinserted into HBase in each micro-batch.

The main drawback of this approach is its stateful nature. The whole current state for the pyramid must be stored. The size of this state can become significant if every tile has received data. However, as in the batch implementation, it is inherently bounded due to the use of canopy. Furthermore, in the *Lambda Architecture* paradigm, the state of the speed layer is not designed to be kept forever. When the new data has been ingested by the batch layer, the state of the speed layer can be flushed. Thus, the growth of the state is limited by the speed of the batch layer.

In our implementation, we used the internal capabilities of Spark to store the current state. The storage used by the *mapWithState* function is in memory, so the resources of the cluster should be scaled to be able to hold it. This solution gives a lower latency, since Spark is responsible for colocating the current state and the update data. If in-memory storage for the whole state is not possible, storage in HBase could be used as an alternative. This solution is less memory intensive, but will induce higher latencies for data ingestion, due to database query latencies.

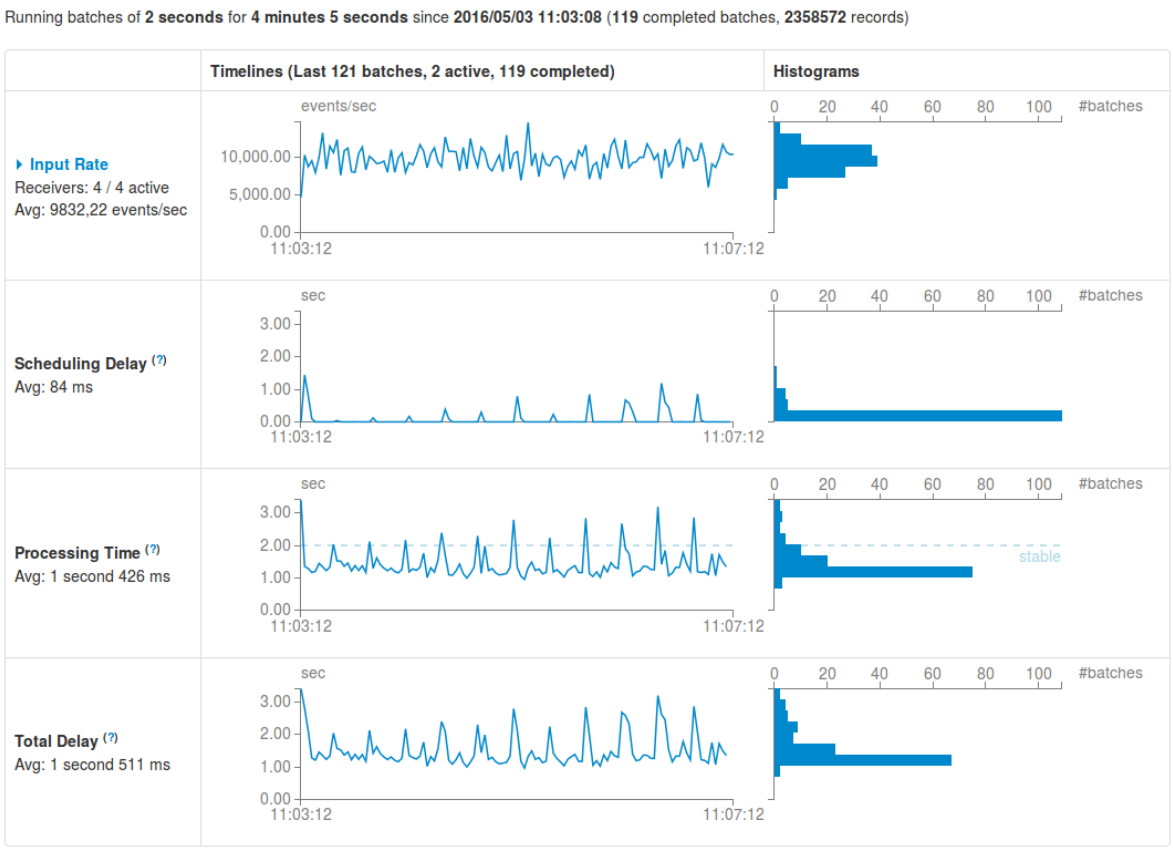

Figure 2: Report generated by Spark for a run with 8 4-core Spark executors. The input data comes from the brightkite dataset. At this point, the system has ingested an average of 9832 points per second for a little more than 4 minutes, for a total of around 2 million points. This running time allows to see significant statistics without overloading the report. Most notably the input rate, which describes the number of points received per second, and the processing time per batch. This shows that the system remains stable with an average processing time of 1.5s, as highlighted by the dashed line in the processing time curve. The regular spikes visible in the processing time are due to Spark regularly creating a backup save of the current state, to be able to handle node failures.

5.1 Performance

As stated before, the main parameter to measure the performance of a streaming data system is the amount of data it is able to ingest per second while staying "stable". In the case of a micro-batch model, the system is said to be stable when the average processing time per micro-batch is less than the micro-batch interval.

To test our implementation, we used the brightkite dataset [1], containing 4.5M geolocalized user check-ins, collected between april 2008 and october 2010. We simulated data ingestion at a several rates randomly oscillating around a fixed number of checkins per second. For example, at a rate of 10K checkins/s, the whole dataset is ingested in 7.5 minutes, so this is much more than the rate at which the dataset was generated, as it was collected over the course of two years.

As seen on Fig. 2, with our implementation, an 8 4-core workers cluster was able to sustain 10K incoming points per second with batch interval of 2 seconds. The average processing time was 1.5s, leaving some margin in case of any failure or sudden burst in the data arrival rate. Other benchmarks can be seen on Table 1.

Conclusion

In this article, we described a new technique for streaming multilevel aggregation using canopy clustering. This allows to visualize in real-time incoming Big Data as a heatmap. We provided details of implementation and performance results on Spark Streaming.

Acknowledgements

This research project has been powered by LSD, the LaBRI data platform funded by the "Nouvelle Aquitaine" French Region.

References

[1] 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.

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

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] C. Li, G. Baciu, and H. Yu. Streammap: Smooth dynamic visualization of high-density streaming points. *IEEE Transactions on Visualization and Computer Graphics*, 2017.

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

[10] N. Marz and J. Warren. *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications Co., 2015.

[11] 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.

[12] 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.

[13] 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.

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

[15] A. Perrot, R. Bourqui, N. Hanusse, F. Lalanne, and D. Auber. Large interactive visualization of density functions on big data infrastructure. In *Large Data Analysis and Visualization (LDAV), 2015 IEEE 5th Symposium on*, pages 99–106. IEEE, 2015.

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

[17] 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.

[18] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438. ACM, 2013.

[19] 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.

[20] 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.