HAL
open science

Distributed Graph Layout with Spark

Antoine Hinge, David Auber

To cite this version:

Antoine Hinge, David Auber. Distributed Graph Layout with Spark. Information Visualisation (iV), 2015 19th International Conference on, Jul 2015, Barcelone, Spain. pp.271–276. hal-01187421

HAL Id: hal-01187421

https://hal.science/hal-01187421v1

Submitted on 26 Aug 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed Graph Layout with Spark

Hinge, Antoine
LaBRI
Université de Bordeaux
Bordeaux, France
antoine.hinge@labri.fr

Auber, David
LaBRI
Université de Bordeaux
Bordeaux, France
david.auber@labri.fr

Abstract

This paper presents a novel way to draw very large graphs, especially those too big to fit the memory of a single computer. This new method takes advantage of the recent progress in distributed computing, notably using the Apache MapReduce library called Spark. Our implementation of a force-directed graph drawing algorithm and the way to compute repulsive forces in MapReduce are exhibited. We demonstrate the horizontal scalability of this algorithm and show layouts computed on a Hadoop cluster with our method.

Keywords

Graph Drawing; MapReduce; Hadoop; GraphX; Spark;

1 Introduction

With the advent of social networks and connected object, networks have a more prominent place than ever. As a result, graphs representing those networks have grown exponentially in size. It is increasingly relevant to be able to visualize those graphs to better comprehend the data structure and help human operators analyze data.

Many force-directed algorithms have been developped to draw undirected graphs. The classical approach is iterative: an attractive and repulsive force are applied to each vertex at each step, computing its displacement [7, 12]. Relying on a physical model, like the spring-electrical model, those algorithms try to reach the energy minimum by iterating until the layout stabilizes. Effectively, they are unable to draw large graphs as they need an all-pair distance computation, which quickly becomes expensive as graphs become larger.

The multi-level approach developped in GRIP [8] and FM^3 [9, 10] overcomes this limitation by introducing a multi-level scheme. A graph filtration is created where each child graph is a summary built from its parent graph. A force-directed algorithm is applied on the child-most graph, which then gives the initial layout of its parent graph. Each level is drawn this way until the parent-most graph, where we get the final layout. This method ensures that the initial layout is already close to the energy minimum before applying the force-directed algorithm, thus accelerating convergence. In FM^3, the multipole approach circumvents the computation of repulsive forces for every vertex pair. Graphs which where too large for regular force-directed schemes like Fruchterman and Reingold [7] can be drawn in a matter of seconds.

New implementations of the force-directed algorithm have recently been developped by taking advantage of the Graphics Processing Unit (GPU) available in most laptops [6, 3]. In Frishman [6], a multi-level approach is presented using general purpose computation on the GPU. These methods greatly accelerate computation.

Two solutions are commonly mentionned when faced with important volume of data to analyze: High Performance Computing (HPC) or MapReduce [4]. The HPC approach relies on multi-core architectures using massive parallelism. This method has proven itself to be effective but requires data to be wholly available on a computation cluster, which is usually not where data is stored. As a

result, a transfer has to be done to the HPC cluster to perform analysis. Within the Big Data context, where graphs are huge and datasets usually need precomputation, transfering data to the HPC cluster is a serious hindrance. To bypass this obstacle, we chose to explore the graph drawing problem using the MapReduce paradigm, where data is processed close to where it is stored. Each computer in the cluster does computations on the fraction of data stored on its disks. Those computations are coordinated and summarized by a master node. In this paradigm, we avoid transfers to a dedicated HPC cluster each time an analysis is needed. This paradigm also ensures a high degree of parallelism and horizontal scalability, meaning that if storage or computation capacity are needed, a new machine can be added to an existing cluster without having to replace any existing computer.

Spark [17] is a MapReduce framework particularly suitable to iterative algorithms. Contrary to the classical MapReduce framework [15, 4], it ensures reusability of datasets computed during execution, without having to necessarily store them on disk. Using this library, users do not need to recompute variables or reload them from disk, as is generally the case with Hadoop MapReduce. Therefore, the implementation of force-directed graph drawing algorithm is facilitated in this environment.

In this paper, we will present how to write a graph drawing algorithm with the MapReduce paradigm and its implementation using the Spark library. We will demonstrate how, from collected data stored in a Hadoop cluster, we can compute a layout for graphs of millions of edges. To this end, we will exhibit the challenges that MapReduce represent for graph drawing algorithms. We will go into details concerning the implementation of graph drawing algorithms and whether or not they are directly applicable in a MapReduce paradigm.

We then present an overview of the implemented algorithm. We demonstrate the exact functionning of this algorithm in the MapReduce paradigm. Each step is detailed and we demonstrate that this algorithm ensure horizontal scalability and a high degree parallelism.

Graph layouts obtained with this method are presented and compared with other force-directed graph drawing methods like FM^3. We demonstrate how the various visualization are equivalent. In conclusion, we will summarize the various results presented here and propose idea to continue our work and improve our algorithm.

2 Graph Drawing with MapReduce

We demonstrate that classical graph drawing algorithms are impossible to implement without modifications in the MapReduce paradigm.

2.1 Spark environment

In a force directed algorithm, attractive and repulsive forces are computed and applied iteratively to obtain a new layout. Spark [17] offers a MapReduce environment where it is easy to create iterative algorithms. Hence using this library is appropriate for a distributed force-directed graph drawing algorithm.

GraphX [16] is a Spark library especially developped for implementing algorithm on graphs and propose a ready-to-use interface, for example to create a graph from an file of its edges. This library provides implementation of classical MapReduce design patterns in the case of graphs: joins of vertex attributes, a message-passing function along the edges of the graph (as used in PageRank) and many others.

2.2 Complete force-directed algorithm

Our first idea was to search for the energy minimum using a force-directed algorithm, as seen in Fruchterman and Reingold [7]. This method has been proven effective with a very small number of vertices, where every repulsion force between two vertices can be computed, producing interesting layouts.

Because it requires each vertex in the graph to know the position of all other vertices, this task is not possible to carry out efficiently in MapReduce, where data is stored in several computer of the Hadoop cluster. Doing the cartesian product of vertex attributes (in our case the position of each vertex) is not feasible, especially when dealing with very large datasets. It requires a $O(V^2)$ complexity in memory, V being the number of vertex. For those reasons, this approach was not further investigated in MapReduce.

2.3 Multi-level approach

To compensate size limitations in the force-directed algorithm, the multi-level approach, as presented in GRIP [8]

or FM^3 [9], creates the hierarchy using a sequential algorithm. A hierarchical filtration of graphs is created where each rougher graph is a topological summary of its parent graph. At each new level, less edges and vertices are left, which makes force-directed algorithms more efficient on those smaller graphs. Drawing the rougher graph gives the initial layout for its parent graph, which in turn reduces the time needed to converge with the Fruchterman Reingold algorithm for this particular parent graph.

In FM^3, every vertex is added in a buffer. One vertex is sampled in this buffer and every other vertex at distance 2 or less in the graph is collapsed in a solar system, effectively removing them from the buffer. Another point is then chosen in the buffer. This process is iterated until the vertex buffer is empty. While this strategy is very efficient in a single core architecture where computation is sequential, parallelism is needed in MapReduce. In Frishman [6], most of the computation is done on the GPU but the filtration step is done on the Central Processing Unit. Whether this filtration can be obtained in a non-sequential way remains an open question.

Even if such a hierarchy was available, the full force-directed algorithm is still applied to the whole graph when computing the final layout, on the last level. For this reason, a distributed force-directed algorithm is a necessary step to draw large graphs in MapReduce.

3 Algorithm overview

In this section, we describe the implementation of our force-directed algorithm. The way the algorithm computes attractive forces is described as well as the problem faced with repulsive forces. This problem leads to an approximate solution to compute repulsive forces in MapReduce.

3.1 Force directed algorithm

Force directed algorithms are iterative methods where attractive and repulsive forces are applied to each vertex at each step, computing its displacement. Those algorithm try to reach the energy minimum by iterating over the vertex displacement.

In the MapReduce paradigm, the same scheme is applied, as seen in Algorithm 1. Each step, repulsive and attractive forces are computed. Those forces are added to the current position of vertices in the layout. A new layout is generated this way, closer to the energy minimum. The layout is initialized at random. The number of iteration is set as $10 \log V$, with V the number of vertices.

Data: Graph
Result: Graph Layout
Initialization;
for *Fixed number of iterations* **do**
  Computing attractive forces;
  Computing repulsive forces;
  Applying forces to the layout;
end

Algorithm 1: Spark force-directed algorithm

For each vertex pair connected by an edge, an attractive force is computed. A spring attractive force is used. The direction is set as the edge direction and the norm is expressed as $f_{att}(d_{att}) = \frac{(d_{att} - d_0)}{k^2}$ where d_{att} is the distance between the pair of vertices. Two parameters are set: d_0 a nominal distance parameter and a parameter k. The parameter d_0 is the optimal distance for attractive forces, meaning that neighbors at distance d_0 are not subject to the corresponding attractive force.

Repulsive forces are computed between each pair of vertex. Those forces ensure that the layout is spread out. The formula used for repulsive forces is the one presented in Fruchterman and Reingold[7]. Namely, a repulsive force is a vector going from one vertex to the other. Its norm is computed with $f_{rep}(d_{rep}) = \frac{k}{d_{rep}^2}$, where d_{rep} is the distance between the pair of vertices. The parameter k, the same parameter as the one used to compute attractive forces, is used to harmonize attractive and repulsive forces.

3.2 Attractive forces

Attractive forces are computed using the edges and position of each vertex. For any given edge, the position of its source vertex is mapped with the Id of its corresponding destination vertex. Those informations are then reduced using the destination vertex Id to compute the attractive forces for each vertex. This is summarized in Figure 1.

Using this scheme, messages passed on the cluster are

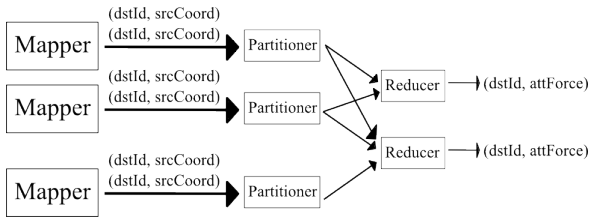

Figure 1: MapReduce implementation of attractive forces

bounded by the number of edges in the graph. This ensures horizontal scalability.

3.3 Approximate repulsive forces

Repulsive forces are computed between all vertex pairs. Computing the all-pair repulsive forces is not possible in a MapReduce environment, due to data partitionning, as seen in section 2. An approximate repulsive force computation must be used to compute repulsive forces in MapReduce.

The approximate force is computed from a summary of the layout, inspired by GEM [5]. This generalization is embodied by barycenters, which create local vertex clusters. Inside those clusters, local repulsive forces are generated to ensure that the graph is as spread out as possible.

With n barycenters, each vertex only needs to know the position of up to n elements. It reduces the memory needed to $O(n \cdot N)$ instead of $O(N^2)$. Using barycenters, the horizontal scalability is ensured.

4 Center and centroid repulsive forces

To overcome the all-pair repulsive forces memory issue, barycenters are used. Two different types of structures are employed: the center of the layout and local barycenters. These structures are described as well as the way repulsive forces are computed. We also describe mechanisms (expansion and reduction) used to keep the centroids relevant even when the layout is updated.

4.1 Center and centroids

We call centroid a local barycenter that summarize the layout around its position. Each centroid is composed of its coordinates in the layout, the current number of vertices associated to it and the maximal distance to any associated vertex. Vertices in the graph are associated to their closest centroid. A centroid array stores each centroid currently in the layout. To initialize this array, some vertex are randomly sampled from the layout. Their position gives the starting position of each centroid in the initial centroid array.

The layout center represents the mean of coordinates from every vertex in the layout.

Both structures generate repulsive forces: the layout center ensures that the graph is as spread out as possible while centroids fulfill the same purpose at a local level.

4.2 Centroid-repulsive forces

The center and centroids of the layout generate repulsive forces are applied to each vertex in the layout. The force is computed using the Fruchterman and Reingold formula, as described in Section 3.1. Instead of being computed between a pair of vertices, it is computed between centroids and vertices, using corresponding direction and distance, or between the center and vertices in the same manner.

To compute centroid-repulsive forces, the centroid array is broadcast to each vertex. Those values are mapped to compute the force between each centroid-vertex pair. They are then reduced over vertex identifiers to compute the global mean centroid-repulsive force for each vertex. Center-repulsive forces are computed in a similar manner.

4.3 Updating the centroids

The layout is updated at each step of the algorithm. To keep repulsive forces relevant, positions of the layout center and centroids are recomputed at each step. This way, the repulsive force applied to each vertex is always accurate.

Computing the mathematical mean of elements distributed in a Hadoop cluster is a well known MapReduce application [14]. Updating the position of centroids is an example of this design pattern. Centroids are updated by

mapping vertices positions and distance to associated centroid. Reducing by centroid identifier gives new centroids positions, number of associated vertices and maximal distance to a vertex, returning respectively the sum of positions, a count value and the max of centroid-vertex distances. Position of the center is updated the same way.

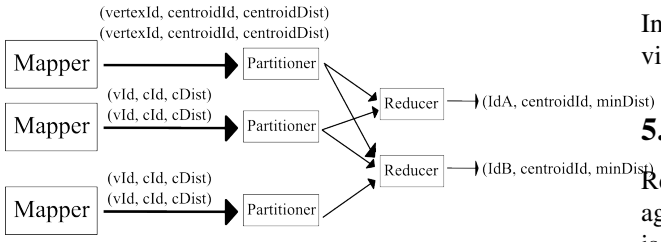

Figure 2: MapReduce implementation of vertex-centroid association

Centroid-vertex associations are also updated by broadcasting the centroid array and mapping the centroid-vertex distance. The reduce operation is done on the vertex identifiers, keeping only the minimum distance, and corresponding centroid identifier for a given vertex. This operation is illustrated in Figure 2.

4.4 Expanding and compressing centroids

Centroid-repulsive forces ensure that each vertex cluster, represented by a centroid, is spread out. To this end, the centroid array must be a good summary of the graph: under or overfitting must be avoided. To this purpose, two mechanisms are used. If a centroid is too specific, it is removed from the layout. If a centroid is too general, it is split in two. The second centroid is positioned closed to the first one, with added noise to determine its exact position.

To trigger these mechanisms, a criterion, called mass, is set to the number of vertices associated to a given centroid. To bound the number of centroids, a lower and upper mass are set to a percentage of the total number of vertices. The lower bound is set to 0.25% of the number of vertices, which ensures that there are no more than 400 centroids in the layout. The upper bound is set to 5% of the number of vertices, which ensures that there are no less than 20 centroids in the layout.

With a fixed maximal number of centroids, the memory complexity, as described in Section 3.3, is $O(N)$.

5 Algorithm implementation in Spark

In this section, we describe how we implemented the previous algorithm using Spark [17].

5.1 Data structure

Resilient Distributed Datasets (RDD) are the default storage method for very large objects in Spark. This structure is a collection of elements that is both fault-tolerant and facilitates parallel operations. Those objects can be created by parallelizing an existing Scala collection like an Array or a List. It can also be generated from an already existing data partition on Hadoop Distributed File System for example.

The GraphX library for Spark provides two RDD structures, an EdgeRDD and a VertexRDD. It also provides a identifier for vertices called VertexId.

The EdgeRDD structure is an RDD of all directed pair of vertex. It can be represented as a 2-column table, with the first column being VertexId of source vertex and the second column being the VertexId of destination vertex.

The VertexRDD is an RDD containing a VertexId and a vector of values. In our case, the VertexRDD is composed of the VertexID and the position in the layout, consisting of two Double coordinates for the x and y coordinates of the vertex.

Reading from a text file, both the EdgeRDD, composed of undirected edges without loops, and the VertexRDD, composed of distinct vertices randomly initialized in the layout, are generated.

5.2 Attractive forces

To compute the attractive forces, the EdgeRDD is merged with the VertexRDD to obtain the correct force computation. More precisely, by using the join function, already implemented in the Spark environment, positions of the source vertex and destination vertex are obtained. Those informations are mapped in a PairRDD with the destination identifier as key and both positions of the source and

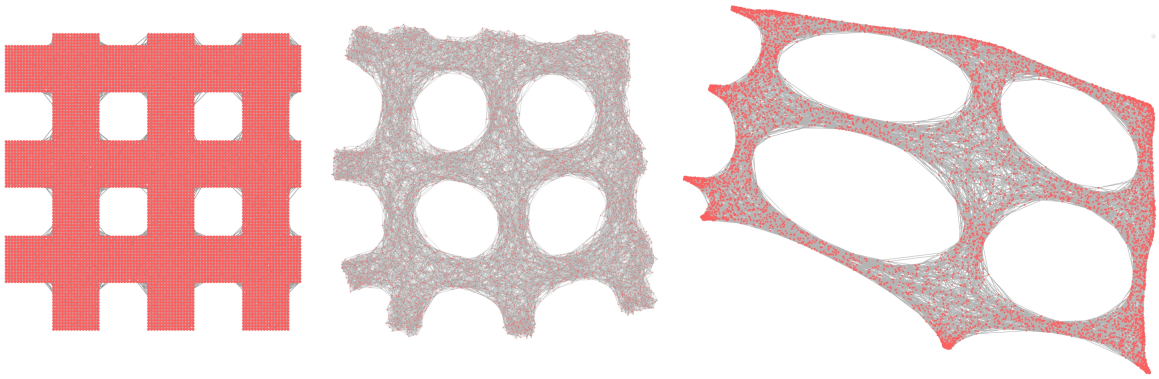

Figure 3: Layout on Tulip [2] for a generated dataset: *(Left)* Graph as generated *(Middle)* Layout generated by FM^3 *(Right)* Layout generated by our algorithm on a Hadoop cluster

the destination vertices as values. They are then reduced in order to compute the single attractive force applied to the destination vertex : using reduceByKey() each forces components are summend and a count value is generated. This way, the total mean attractive force is computed for each vertex.

This returns an attractive force RDD that is stored to be added to the current layout at the end of the step.

5.3 Centroid-repulsive forces

Before the fist iteration, the Array of centroid is initialized by taking vertices at random without replacement. This is done using the takeSample() Spark method on the VertexRDD containing the layout.

To compute the centroid-repulsive forces, the centroid Array is broadcast to each machine in the cluster using the SparkContext.broadcast() method. The VertexRDD is mapped and computes the repulsive forces, with the centroids of the centroid array. This returns a PairRDD where the key is the VertexId and the value is the current repulsive force. Those values are then reduced using the reduceByKey() method, which returns a centroid-repulsive force RDD. This RDD is stored to be added to the current layout at the end of the step.

The center-repulsive forces are computed in a similar manner, but only the center position is broadcast using SparkContext.broadcast().

5.4 Updating the centroids

Updating the centroids is done in the manner described in Section 4.3. Spark map() and reduceByKey() methods are used on the original VertexRDD. The SparkContext.broadcast() method is also used to send the centroid Array to each executor.

5.5 Expanding and compressing centroids

Since the centroids array contains few elements, this step is done on a single machine. To this end, this step is computed on the driver node.

5.6 Updating the layout

At the end of the algorithm, the attractive forces are computed and stored in an RDD. The repulsive forces (both centroid-repulsive forces and center-repulsive forces) are also stored in an RDD. To add these forces to the current layout, the innerZipJoin() method is used to compute a join efficiently: this method enables a very quick join operation for RDD sharing similar indexes. Attractive forces, center and centroid repulsive forces are applied on all vertices.

6 Results

We ran our graph drawing algorithm on generated datasets designed to have interesting layout properties as well as on a selection of large graphs from the SNAP dataset collection [13]. Here are presented some results comparing graph layouts.

6.1 Generated datasets

To generate large datasets with a structured layout, we designed a large graph generator. Graphs generated this way have grid-like layouts. This graph generator creates a graph from a vertex grid where vertices are suppressed regularly in a hole pattern. For each vertex, five neighbours are drawn randomly in a small radius around the vertex. This method can generate any size of graph. An example of layout generated by this method can be seen in Figure 3, on the left.

6.2 Results on a Hadoop cluster

In Figure 3, results generated by our algorithm on a Hadoop cluster are compared to a layout generated by FM^3 and to the original layout. The generated graph is composed of 8000 vertices and 35000 edges.

Our layout was generated on a cluster of 16 machines each with 24 cores. Each computer in the cluster has 48Gb of RAM. We ran our Spark application on 20 executors. The algorithm ran for 4600 steps in 5 hours. The layout can be seen on the right in Figure 3. The FM^3 layout was obtained using Tulip [2]. It ran on a computer with 8 cores and 16Gb of RAM. The algorithm only took a few seconds. The layout can be seen in the middle in Figure 3.

The original graph layout structure can be recognized in the layout obtained in Spark, as well as in the layout obtained with FM^3. The layout generated in Spark retains the global structure of the original layout, but is more warped than the one obtained with FM^3. This is a result of the center and centroid repulsive force, which are creating local distortions in the layout.

Concerning running times, the graph is too small to take advantage of the power of the MapReduce paradigm. It is not surprising that our method was not faster than FM^3 on a single machine. However, our implementation still needs tuning to perform much faster than it is currently performing.

6.3 Social networks

Some SNAP social networks, like WikiVote, were drawn using our method and compared with FM^3 (results not shown). The results are similar as those presented in previous section. However, the layouts are not very structured and bring little informations.

Conclusion

In this paper, we described a way to implement a graph layout drawing algorithm using the MapReduce paradigm. We showed how our method, while often slower than efficient CPU methods on small graph, ensures horizontal scalability. The layout obtained had most of the qualities of efficient graph drawing algorithms for reasonably large graphs.

Future works

Future works will be focused on tuning our algorithm to better run using Spark. The method will be more finely tuned and further tested in our current cluster but also on a larger cluster of 45 machines. With this finer tuning, we will attempt to draw much larger graphs than the one showed in this paper. We will provide benchmarks comparing our running times with those of algorithms running on a single computer for very large graphs.

Acknowledgements

This work has been carried out as part of the REQUEST project supported by the French Investissement dAvenir Program (Big Data - Cloud Computing topic - PIA O18062-645401).

References

[1] Daniel Archambault, Tamara Munzner, and David Auber. Topolayout: Multilevel graph layout by

topological features. *Visualization and Computer Graphics, IEEE Transactions on*, 13(2):305–317, 2007.

[2] David Auber. Tulip a huge graph visualization framework. In *Graph Drawing Software*, pages 105–126. Springer, 2004.

[3] David Auber and Yves Chriricota. Improved efficiency of spring embedders: Taking advantage of gpu programming. In *Visualization, Imaging, and Image Processing*, volume 2007, pages 169–175, 2007.

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

[5] Arne Frick, Andreas Ludwig, and Heiko Mehldau. A fast adaptive layout algorithm for undirected graphs (extended abstract and system demonstration). In *Graph Drawing*, pages 388–403. Springer, 1995.

[6] Yaniv Frishman and Ayellet Tal. Multi-level graph layout on the gpu. *Visualization and Computer Graphics, IEEE Transactions on*, 13(6):1310–1319, 2007.

[7] Thomas MJ Fruchterman and Edward M Reingold. Graph drawing by force-directed placement. *Software: Practice and experience*, 21(11):1129–1164, 1991.

[8] Pawel Gajer and Stephen G Kobourov. Grip: Graph drawing with intelligent placement. In *Graph Drawing*, pages 222–228. Springer, 2001.

[9] Stefan Hachul and Michael Jünger. Drawing large graphs with a potential-field-based multilevel algorithm. In *Graph Drawing*, pages 285–295. Springer, 2005.

[10] Stefan Hachul and Michael Jünger. An experimental comparison of fast algorithms for drawing general large graphs. In *Graph Drawing*, pages 235–250. Springer, 2006.

[11] David Harel and Yehuda Koren. Graph drawing by high-dimensional embedding. In *Graph Drawing*, pages 207–219. Springer, 2002.

[12] Tomihisa Kamada and Satoru Kawai. An algorithm for drawing general undirected graphs. *Information processing letters*, 31(1):7–15, 1989.

[13] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[14] Donald Miner and Adam Shook. *MapReduce Design Patterns: Building Effective Algorithms and Analytics for Hadoop and Other Systems*. " O'Reilly Media, Inc.", 2012.

[15] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[16] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013.

[17] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.