Static and Dynamic Big Data Partitioning on Apache Spark

Massimiliano Bertolucci [b] Emanuele Carlini [a] Patrizio Dazzi [a] Alessandro Lulli [a,b]
Laura Ricci [a,b]

[a] *Istituto di Scienze e Tecnologie dell'Informazione, CNR, Pisa, Italy*
[b] *Department of Computer Science, University of Pisa, Pisa, Italy*

Abstract. Many of today's large datasets are organized as a graph. Due to their size it is often infeasible to process these graphs using a single machine. Therefore, many software frameworks and tools have been proposed to process graph on top of distributed infrastructures. This software is often bundled with generic data decomposition strategies that are not optimised for specific algorithms. In this paper we study how a specific data partitioning strategy affects the performances of graph algorithms executing on Apache Spark. To this end, we implemented different graph algorithms and we compared their performances using a naive partitioning solution against more elaborate strategies, both static and dynamic.

Keywords. BigData, Graph algorithms, Data partitioning, Apache Spark

Introduction

In the last years the amount of data produced worldwide has been constantly increasing. For example, in 2012 were created every day 2.5 exabytes (2.5×10^{18}) of data [1], which comes from multiple and heterogeneous sources, ranging from scientific devices to business transactions. Often, data is modelled as a graph in order to organize and extract knowledge from it. Due to its size, it is often infeasible to process this data by exploiting the computational and memory capacity of a single machine. To overcome this limitation, it is common to adopt distributed computing environments. Many approaches of this kind have been designed so far. In this context, approaches based on the MapReduce [2] paradigm are very popular and are able to efficiently exploit large set of computing resources based on commodity hardware. The MapReduce paradigm belongs to the wider family of structured parallel programming approaches, which originated from the proposal of Cole [3] in the late '80s and, across the years, have been implemented by several tools and frameworks [4,5,6,7,8]. However, despite their wide adoption, these frameworks are not the best choices for any purpose. As a consequence, alternative paradigms have been proposed. Some of them have been based on the BSP bridging model [9]. A common trait shared by these frameworks [10] is that they provide the possibility to describe graph processing applications from the point of view of a vertex in the graph. These frameworks are often referred as Think Like a Vertex frameworks (TLAV). Each vertex processes the same function independently, accessing only its local context (usually its neighbourhood), therefore without having a global view on the graph.

As discussed by Gonzalez *et al.* in their GraphX paper [11], the development of efficient algorithms through these frameworks requires to solve several non-functional issues and, among these, the definition of proper graph partitioning strategies is of paramount importance. In fact, the authors show how the assignment of vertices to machines minimising the number of edges crossing partitions leads to good performance with a large class of graphs.

The findings of Gonzalez *et al.* are based on a static partitioning, i.e. the partitioning is computed before starting the execution of the graph algorithm. This solution performs well when the computation on the vertices of the input graph happens to be uniform (e.g. the computation performs at the same rate on different vertices) or according to a communication pattern that can be statically predicted. However, in many graph algorithms the computation is not uniform and neither has a static communication pattern. To evaluate the role of partitioning in these kind of algorithms, in this paper we consider a set of different vertex-centric algorithms, characterised by different communication/computational patterns. The algorithms we studied in this paper belongs to two main families. The first kind of algorithms assumes a static structure of the graph, i.e. it does not change along the computation. A well known algorithm belonging to this class is the PAGERANK [12]. We also consider two further algorithms of this class: TRIANGLE COUNTING [13] and the KCORE decomposition [14]. The other class of algorithms we examined are those in which the topology of the communication graph changes during the computation. Both the algorithms belonging to this class that we studied are aimed at the detection of Connected Components [15,16]. We analysed the behaviour of all these algorithms when adopting different partitioning approaches, both static and dynamic. In conclusion, the main contribution of this paper is the presentation, the analysis and the comparison of a set of vertex-centric problems using different partitioning strategies. We have considered the following algorithms and partitioning approaches:

- *algorithms:* PAGERANK, TRIANGLE COUNTING, KCORE Decompositions, Connected Components
- *partitioner:* Spark Hash, BKW, SWAP, DESC

Each algorithm has been implemented on top of Apache Spark[17] and executed on a cluster composed by 3 machines, each composed by 32 cores and 128 Gbytes of RAM.

1. Related Works

The problem of graph partitioning has been extensively studied. Recently, due to the increasing size of the datasets, many solutions have been proposed targeting large graphs and suitable to the current TLAV frameworks. Among many, some target distributed implementations [18][19] or adopt the streaming model [20][21]. However, the METIS family of graph partitioning software [22] is still often considered the de facto standard for near-optimal partitioning in TLAV frameworks [10]. An extensive analysis of all the methods is beyond the scope of this work and we concentrate in particular on the effects of a good partitioning strategy in TLAV frameworks. As far of our knowledge, this is the first work evaluating the impact of different graph partitioning strategy in Apache Spark. Other works evaluate the impact of graph partitioning on different TLAV framework. The outcome of such evaluation is not always the same, suggesting that extensive work must be done in order to understand the benefits of a good balanced partitioner. For what concern static partitioner evaluation, Salihoglu et al. [23] propose GPS, a Pregel-like graph processing system, and evaluates static partitioning on it. The framework uses a BSP model of computation and the graph can remain in memory during the computation. Their claim is that, using a balanced k-way partitioning, GPS is able to reduce run time of the PAGERANK algorithm between 2.1x and 2.5x with respect to a random partitioner. Instead, Connected Components and Single Source Shortest Path get less benefits. They perform some tests also using Giraph, another Pregel-like framework, and found that a k-way partitioning gets only marginal improvements. They tested the system also with a dynamic relabelling technique and found marginal improvement in time limited to the PAGERANK algorithm. For what concern Hadoop MapReduce, Ibrahim et al. [24] describe LEEN, a locality-aware and fairness-aware key partitioning to save the network bandwidth dissipation during the shuffle phase of MapReduce. Experimental evaluation shows that this approach is up to 40% better with respect to the default Hash partitioner. However they test their optimization just with a wordcount algorithm and they do not perform evaluation of algorithms exploiting different communication patterns. Different results have been obtained by Shao et al [25]. Using Giraph, they found that the performance over well partitioned graph might be worse than Hash partitioner in some cases. The cause is that the local message processing cost in graph computing systems may surpass the communication cost in several cases. Due to this, they propose a partition aware module in Giraph to balance the load between local and remote communication. One work starting an evaluation of the partition strategy with Apache Spark has been presented recently by Amos et al. [26]. The focus of the work is a system to answer queries using Spark but, in the evaluation, they consider the problem of choosing the proper dimension for the partitions. If the size is too small, the system suffers an overhead for the management of the partitions. If the size is too large, data are sequentially processed. They empirically find a good trade-off for their specific application domain but an analysis of the impact of different partition strategies on other domains and algorithms is not presented.

Spark [17] is a distributed framework providing in memory computation making use of Resilient Distributed Datasets (RDD) [27]. RDDs are distributed data structures where it is possible to apply multiple operators such as Map, Reduce and Join. In Spark, the RDDs are partitioned by default using an Hash partitioner. The framework provides also the possibility to use a custom partitioner for data. However in the original and successive works an evaluation regarding the use of different partitioners is not presented.

2. Data Partitioning

In this paper we consider a set of different vertex-centric algorithms characterised by different communication/computational patterns targeting distributed computations. For each of them we evaluate how, even simple partitioning strategies can enhance their performances. The algorithms tested can be grouped into two main families. The algorithms belonging to the first family keep every vertex of the graph always active during the whole computation, and it communicates with its neighbours at each iteration. The second family of algorithms that we consider in our study selectively deactivate a subset of vertices during the computation. A detailed description of all the algorithms we implemented and exploited in our study is presented in Section 3. To develop all the algorithms considered in our study, we exploited the standard API of Apache Spark. Even if the Spark environment currently offers the GraphX library [11] for graph analysis we developed our own vertex-centric abstraction, in fact, when we started our work, GraphX was not stable yet, especially when used with massively iterative algorithms. To conduct our study, we embodied in Spark the different partitioning strategies we decided to apply. We can describe a partitioner as an entity that takes in input the identifier of a vertex id_V and gives as output the identifier of a partition id_P.

2.1. Spark embedded partitioners

Plain Spark already provides two different partitioners *hash* and *range*. The first one is based on a hashing function that takes in input id_V and returns the id_P of the partition depending on the behavior of the hashing function adopted. The standard function used by Spark is based on the module function. Basically, it aims at distributing all the identifier of the graph in a uniform way. Conversely, the *range* function aims at distributing the vertices of the graph on the basis of user provided ranges, e.g., all the nodes whose identifier is in the range $100 - 200$ are assigned to partition 2. In our study we exploited the first one of these two strategies, which we used as a baseline in our comparisons.

2.2. Balanced k-way partitioners

The first class of data decomposition strategies we embodied in Spark is based on the balanced k-way partitioning. It consists in a static partitioning strategy that decomposes the graph in k parts each one composed by the same amount of nodes, approximately. This approach is interesting because, by accepting a certain degree of approximation, it admits a polynomial time verification of its results (NP-complete). In fact, graph partition problems usually fall under the category of NP-hard problems. Solutions to these problems are generally derived using heuristics and approximation algorithms. However, balanced graph partition problem can be shown to be NP-complete to approximate within any finite factor. To conduct our study, we decided to use the suite of algorithms for partitioning provided by Metis. It provides solutions for partitioning a graph into a configurable amount of parts using either the multilevel recursive bisection or the multilevel k-way partitioning paradigms. As we mentioned, we used the latter approach, indeed, the multilevel k-way partitioning algorithm provides additional capabilities (e.g., minimize the resulting subdomain connectivity graph, enforce contiguous partitions, minimize alternative objectives, etc.). To exploit the features provided by Metis in Spark we decided to organise the computations in two steps instead of embedding the Metis library directly in the Spark framework. Basically, we give the graph in input to the multilevel balanced k-way partitioning provided by Metis, then we exploit the result to re-label the vertices of the graph accordingly to the partitions returned by Metis so that the Spark hash partitioner will be able to assign the vertices belonging to the same partition to the same machine.

2.3. Dynamic partitioners

The other kind of partitioning strategies that we have studied in our work falls in the class of dynamic partitioners. These approaches do not rely on a pre-computed decomposition of data, instead, they aim at adjusting the distribution of vertices to the machines that are more suited to compute them, depending on the behavior of the actual computation. In particular, we are interested in adjusting the distribution of the computation workload when the amount of active nodes decreases. To this end, in our study we investigated two different approaches of dynamic partitioning. The first one is based on a reduction of the partitions that is merely proportional to the number of active nodes. This strategy can be applied to any algorithm because it does not require any additional information about the nodes except the number of nodes that are active at a certain stage of the computation. We called this strategy "DESC". The other strategy we studied can be applied only to the algorithms that during their computation label the nodes of the graph in a way such that the nodes that communicate more one each others share the same label. In this case the dynamic partitioning strategy "clusters" the nodes sharing the same label in the same partition, when possible. The name of this strategy is "SWAP".

3. Case studies

As we mentioned above, we evaluated the impact of different partitioning strategies with several algorithms. This section briefly presents such algorithms.

3.1. PageRank

PageRank [12] is one of the most popular algorithms to determine the relevance of a node in a graph. It is measured by means of an iterative algorithm characterised by a neighbour-to-neighbour communication pattern. Basically, every node is characterised by a value, representing its relevance. For each iterative step of the computation, every node redistributes its value among its neighbours. At the end of the iterative step each node sums up the "contributions" received, the resulting value represents its updated relevance.

3.2. Triangle Counting

Triangle Counting is a well known technique for computing the clustering coefficient of a graph. More in detail, a *triangle* exists when a vertex has two adjacent vertices that are also adjacent to each other. Several sequential and parallel solutions have been proposed for this problem. The algorithm we adopted in this paper is a vertex-centric version of the solution proposed by Suri and Vassilvitskii [13]. In the first step of the algorithm each vertex, in parallel, detects all the triples it belongs to that have two neighbours having a higher node degree. In the second step each vertex, in parallel, gathers all the messages received and, for each of them, checks the existence of a third edge closing the triple. The algorithm is characterized by a neighbour-to-neighbour communication pattern. The neighbourhood of each node is fixed, i.e. does not change during the computation steps. In the first step all the vertices are performing computation, whereas in the second step only the nodes which have received a message actually compute.

3.3. KCORE *Decomposition*

In many cases the study of the structure of a large graph can be eased by partitioning it into smaller sub-graph, which are easier to handle. To this aim, the concept of the KCORE decomposition of a graph results very useful. A KCORE of a graph G is a maximal connected subgraph of G in which all vertices have degree at least k. K-coreness is exploited to identify cohesive group of nodes in social networks, i.e. subset of nodes among which there are strong, frequent ties or to identify the most suitable nodes to spread a piece of information in epidemic protocols. Our algorithm is based on the solution proposed by Montresor *et al.* [14], which computes the K-coreness of each node using an iterative epidemic algorithm. Each node maintains an estimation of the coreness of its neighbours, which is initialized to infinity, whereas the local coreness of each node is initialized to its degree. At each step, each node sends the current value of its local coreness to its neighbours. When a node is updated about the coreness of one of its neighbours, if it is lower than its current local estimation, it records the new estimated value and checks if its local coreness has been changed by this update. In case, it sends its new coreness to its neighbours. The algorithm terminates when messages are no longer sent. As in PAGERANK, the communication pattern characterizing the algorithm is a fixed neighbour-to-neighbour pattern. However, differently from PAGERANK, nodes not participate to the message exchange at each iteration because only nodes that updates their coreness send messages in that iteration.

3.4. HASH-TO-MIN

Rastogi *et al.* [15] surveyed several algorithms for computing the connected components of a graph. All of them are characterized by associating a unique identifier to each node of the graph and by identifying each connected component through the minimum identifier of the nodes belonging to that component. In HASH-TO-MIN, each node initializes its own cluster to a set including itself and its neighbours. During the computation each node selects the node v_{min} with the minimum identifier in its cluster and sends its cluster to v_{min} and v_{min} to all other nodes of the cluster. In the receive phase, each node updates its cluster by merging all the received clusters. If the resulting cluster is unchanged from the previous iteration, the node do not exchange messages in the following steps, but it may receive a message which re-activates its computation. At the end of the computation, each node of a connected component is tagged with the minimum identifier, whereas this latter node has gathered all the identifiers of its connected component. In the HASH-TO-MIN algorithm, the communication pattern is not fixed, since each node may choose a different set of recipients at each iteration (the set of recipients is defined by the nodes in its cluster). Furthermore, as in KCORE, a node may be de-activated at some iteration.

3.5. CRACKER

CRACKER [16] is an algorithm optimizing HASH-TO-MIN. Each node checks its neighbourhood and autonomously decides if its participation to the computation of the connected components can be stopped. In any case, the node maintains the connectivity with its neighbours. Like in HASH-TO-MIN, the communication pattern is not fixed, but, with respect to HASH-TO-MIN when a node decides to stop its participation to the computation of the connected components, it may not be reactivated.

Table 1. Completion time with synthetic graphs (best times are in bold)

	size	KCORE		PAGERANK		TRI COUNT		HASHTOMIN		CRACKER	
		HASH	BKW	HASH	BKW	HASH	BKW	HASH	BKW	HASH	BKW
NEWMAN	1M	44	**42**	111	**100**	43	**42**	160	**118**	65	**63**
	2M	60	**58**	162	**137**	65	65	331	**188**	95	**82**
	4M	90	**87**	274	**217**	110	**109**	633	**347**	153	**128**
POWERLAW	1M	**109**	110	149	**135**	**83**	88	125	**90**	66	**58**
	2M	182	**180**	235	**207**	**156**	167	**147**	152	85	**81**
	4M	327	**275**	431	**358**	**340**	358	276	**273**	138	**134**
WATTZ	1M	111	**107**	107	**82**	41	**40**	137	**97**	73	**58**
	2M	171	**167**	150	**123**	62	**61**	172	**149**	101	**80**
	4M	286	**275**	253	**196**	103	**101**	326	**270**	160	**130**

4. Experimental evaluation

The experimental evaluation has been conducted on 96 cores of a cluster of 3 machines, each composed by 32 cores and 128 GB of RAM. The evaluation has considered both the static partitioning, in which the partition is statically created before the computation, and the dynamic partitioning in which the partitioning is performed during the computation. Experiments measured the completion time without considering the time needed to compute the partitioning, i.e. the partitioning of the nodes is supposed to be given at the start of the computation.

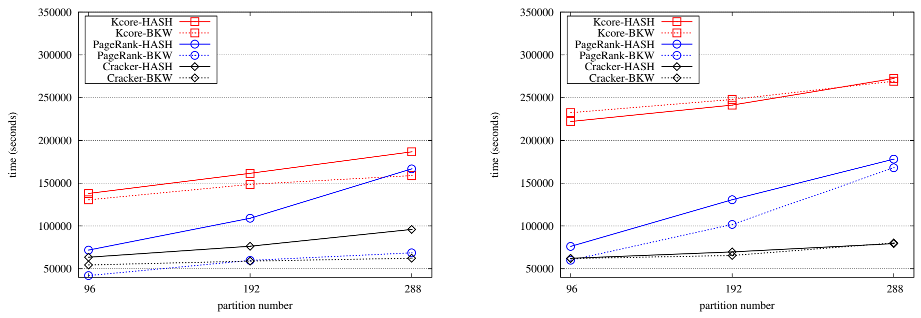
4.1. Static Partitioning, Synthetic Graphs

These experiments measured the completion time of KCORE, PAGERANK, CRACKER, TRIANGLE COUNTING and HASH-TO-MIN with three different dataset sizes (number of nodes) in the set $\{1000000, 2000000, 4000000\}$. The graphs type considered are the Watts-Strogartz (WATTZ), Newman-Watts-Strogartz (NEWMAN) and Power-law (POWERLAW) and have been generated by means of the Snap library [28]. The results are shown in Table 1. We can observe that a BKW partitioning of the graph yields better completion times in almost all cases, with the exception being the TRIANGLE COUNTING algorithm, when applied to the powerlaw graph. The TRIANGLE COUNTING is not an iterative algorithm like the others and consists of just two steps of computation. As a consequence it performs communication only one time, hence the advantages of a BKW partitioner are not evident.

4.2. Static Partitioning, Real Graphs

This experiment measured the completion time of three algorithms: CRACKER, PAGERANK and KCORE. We considered two real graphs: the Youtube social network and the road of Texas[1]. All the values are computed considering the average of six independent runs. From the results shown in Figure 1, we can conclude what follows. First, using HASH the completion time is almost always greater than when using BKW for all the algorithms considered. Interestingly, the difference is smaller for the Youtube social network. In the road network dataset by nature a lot of nodes have degree equals to 1 or 2 and also the maximum vertex degree is in the order of tens. Due to this a BKW partitioner is able to cut a smaller amount of edges with respect to a social network graph. This results in the majority of the computation not requiring communication between different par-

[1]both graphs have been taken from `snap.standford.edu`



(a) RoadTX

(b) Youtube

Figure 1. Execution time with various static partitioning strategies

titions. Second, the biggest improvement from HASH to BKW is obtained with PAGERANK, due to its communication pattern. In PAGERANK all the nodes communicate with their neighbours at every iteration. Therefore, a partitioning of the nodes according to the topology of the graph reduces the inter-partition communications, which in turns reduces the completion time. The third aspect worth noticing is the fact that increasing the number of partitions worsens the performances. In principle, an higher number of partitions is beneficial as it would allow the scheduler for a better allocation of the load on the workers. However, the increased costs in term of inter-partition communications is dominant, and it leads to a longer completion time.

4.3. Dynamic Partitioning

In order to evaluate the performances of the dynamic partitioning, we set up an experiment in which we compare the performance of CRACKER when adopting the HASH strategy against the performance it achieves with DESC and SWAP dynamic partitioners. The experiment has been conducted using the graph representing the road of California[2]. This graph consists of around 2 million nodes and 2.7 million edges, with a large connected components that almost includes all the graph. The results are presented in Figure 2. The SWAP strategy yields slightly better results than HASH, as it successfully reallocates active nodes among the workers. However, we believe this behaviour is only beneficial when executing algorithms that do not reactivate previously deactivated nodes and in which the rate of deactivation is relatively steady. Both characteristics match with the behaviour of CRACKER. By comparisons, the DESC has basically the same performances that HASH, as it takes a longer time to reallocate, as in CRACKER the label of the nodes change continuously.

5. Conclusion

[2]taken from `snap.standford.edu`

In this paper we conducted an analysis focusing on the adoption of partitioning strategies supporting BigData graph computations. The analysis was conducted by measuring the impact of both static and dynamic partitioning approaches on several different algorithms, working on data structured as a graph. Our work especially focused on the impact of different partitioning strategies when applied to BSP-like computational frameworks. In particular our investigation focused on Apache Spark, one of the most widely used distributed framework targeting BigData computations. From the result obtained, we observed that a carefully chosen partitioning strategy can lead to an improvement in the computational performances, both with static and dynamic partitioning. As a future work, we plan to implement more elaborate strategy dynamic partitioning, and to experiment with them in larger graphs.

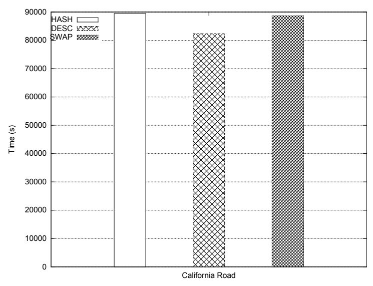


Figure 2.: Comparison of the dynamic partitioners when executing CRACKER

References

[1] Andrew McAfee, Erik Brynjolfsson, Thomas H Davenport, DJ Patil, and Dominic Barton. Big data. *The management revolution. Harvard Bus Rev*, 90(10):61–67, 2012.

[2] Dean Jeffrey and Ghemawat Sanjay. Mapreduce: Simplified data processing on large clusters. *Communication ACM*, 1, 2008.

[3] Murray Cole. *Algorithmic skeletons*. Springer, 1999.

[4] Marco Aldinucci, Marco Danelutto, and Patrizio Dazzi. Muskel: an expandable skeleton environment. *Scalable Computing: Practice and Experience*, 8(4), 2007.

[5] Marco Danelutto and Patrizio Dazzi. A java/jini framework supporting stream parallel computations. 2005.

[6] M. Leyton and J.M. Piquer. Skandium: Multi-core programming with algorithmic skeletons. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 289–296, Feb 2010.

[7] Herbert Kuchen. *A skeleton library*. Springer, 2002.

[8] Marco Danelutto, Marcelo Pasin, Marco Vanneschi, Patrizio Dazzi, Domenico Laforenza, and Luigi Presti. Pal: exploiting java annotations for parallelism. In *Achievements in European Research on Grid Systems*, pages 83–96. Springer US, 2008.

[9] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[10] Robert Ryan McCune, Tim Weninger, and Greg Madey. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing.

[11] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, pages 599–613, 2014.

[12] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: bringing order to the web. 1999.

[13] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th International Conference on World Wide Web, WWW 2011, Hyderabad, India, March 28 - April 1, 2011*, pages 607–614, 2011.

[14] Alberto Montresor, Franscesco De Pellegrini, and Daniele Miorandi. Distributed k-core decomposition. *IEEE Trans. Parallel Distrib. Syst.*, 24(2):288–300, 2013.

[15] Vibhor Rastogi, Ashwin Machanavajjhala, Laukik Chitnis, and Akash Das Sarma. Finding connected components in map-reduce in logarithmic rounds. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 50–61. IEEE, 2013.

[16] Alessandro Lulli, Laura Ricci, Emanuele Carlini, Patrizio Dazzi, and Claudio Lucchese. Cracker: Crumbling large graphs into connected components. In *20th IEEE Symposium on Computers and Communication (ISCC) (ISCC2015)*, Larnaca, Cyprus, July 2015.

[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*, volume 10, page 10, 2010.

[18] Fatemeh Rahimian, Amir H Payberah, Sarunas Girdzijauskas, Mark Jelasity, and Seif Haridi. Ja-be-ja: A distributed algorithm for balanced graph partitioning. In *Self-Adaptive and Self-Organizing Systems (SASO), 2013 IEEE 7th International Conference on*, pages 51–60. IEEE, 2013.

[19] Emanuele Carlini, Patrizio Dazzi, Andrea Esposito, Alessandro Lulli, and Laura Ricci. Balanced graph partitioning with apache spark. In *Euro-Par 2014: Parallel Processing Workshops*, pages 129–140. Springer, 2014.

[20] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM international conference on Web search and data mining*, pages 333–342. ACM, 2014.

[21] Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1222–1230. ACM, 2012.

[22] George Karypis and Vipin Kumar. Multilevel graph partitioning schemes. In *ICPP (3)*, pages 113–122, 1995.

[23] Semih Salihoglu and Jennifer Widom. Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, page 22. ACM, 2013.

[24] Shadi Ibrahim, Hai Jin, Lu Lu, Song Wu, Bingsheng He, and Li Qi. Leen: Locality/fairness-aware key partitioning for mapreduce in the cloud. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 17–24. IEEE, 2010.

[25] Yingxia Shao, Junjie Yao, Bin Cui, and Lin Ma. Page: A partition aware graph computation engine. In *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management*, pages 823–828. ACM, 2013.

[26] Brandon Amos and David Tompkins. Performance study of spindle, a web analytics query engine implemented in spark. In *Cloud Computing Technology and Science (CloudCom), 2014 IEEE 6th International Conference on*, pages 505–510. IEEE, 2014.

[27] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion 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.

[28] Jure Leskovec and Rok Sosič. SNAP: A general purpose network analysis and graph mining library in C++. `http://snap.stanford.edu/snap`, June 2014.