Balanced Distributed Graph Processing

Anil Yelam, Audrey Randall

Abstract

In big data analytics, balanced execution of a computation is important for maximizing utilization of available resources in the cluster like CPU, Memory and Network. Various works in the past have looked at optimized execution of big data platforms like Apache Hadoop MapReduce, Apache Spark and other sort and shuffle based workloads. In this paper, we go on a similar quest for graph processing frameworks, which are equally commonly used in big data processing and has not been hitherto well studied. We discuss various factors that affect the balanced execution of a graph processing algorithms on a given cluster. Specifically, we focus on the effect of different graph partitioning methods on performance of Pagerank algorithm on a wide range of natural graphs. By both studying the properties of these graphs and results from the pagerank runs on Apache Giraph, we show that even the simplest of the partitioning schemes can have a significant effect on the performance.

1 Introduction

The growing popularity of technologies such as Internet of Things (IoT), mobile devices, smartphones, and social networks has led toward the emergence of ”big data”. Such applications produce not just gigabytes or terabytes of data, but soon petabytes of data that need to be actively processed. A large percentage of this growing dataset exists in the form of linked data, more generally, graphs, of unprecedented sizes. Graphs from today's social networks contain billions of edges while inter-connected data from millions of IoT sensors can generate graphs that are exponentially larger. This requires large-scale graph processing to analyze the data and generate useful statistics. Frequently applied algorithms to this end include shortest paths computations, different flavors of clustering, and different variations of page rank.

Traditional big data analytics frameworks like Hadoop MapReduce[6], Apache Spark[20], etc does not perform well for graph procesing. Previous studies [2], [10] have repeatedly shown that these frameworks are too general and does not make use of properties of graph algorithms that often exhibit poor locality of memory access, very little work per vertex, and a changing degree of parallelism over the course of execution. As a consequence, iterative graph processing systems started to emerge in 2010, starting with Google's Pregel[13] that uses Valiant's Bulk Synchronous Parallel (BSP) processing model for its computation, and promoted a ”think like a vertex” notion for processing large graphs. Following that, there has been an explosion in distributed graph processing frameworks like Apache Giraph[8], GraphLab[12], PowerGraph[9], Stratosphese[1], Blogel[19], etc (to name a few) that offer different programming and computational models.

The importance of balanced execution of running large workloads like Sort on big clusters has been studied before (TritonSort[17]), including studies of big data frameworks like Spark (Osterhout et al.[15] and MapReduce (Themis[16]), however there hasn't been a lot of similar work for graph algortihms. Distributed graph processing platforms, like the ones above, make a lot of design choices like the programming model (vertex or edge centric), distributed coordination (synchronous or asynchronous), partitioning schemes (vertex or edge cuts), specializing for certain kinds of graphs, etc (which are discussed in detail in the next section) that influence the performance of the frameworks.

In this paper, we pick one of these design choices i.e., partitioning schemes that are used to divide input graph among workers, and study how different choices can affect the performance. Specifically, we focus on real-world graphs like social networks and web domains (made available Laboratory for Web Algorithmics[4][3]) and show some properties these (power law) graphs exhibit under two very simple but different paritioning schemes called Hash and Range Partitioning. Later, we run pagerank on these same graphs using a distributed graph processing framework called Apache Giraph[8] using the afforementioned partioning schemes and present our results.

The rest of the paper is structured as follows. In sections 2 and 3, we present background of some design choices made by graph processing platforms and rationale for choices we make for this paper. In sections **??** and 4, we present our partitioning schemes in detail, and theoretically study the effect of these schemes on selected input graphs. In section 4.2, we present results from our pagerank runs of Giraph.

2 Background

Notation A Graph G = (V, E), consists of a set of vertices, V= $\{v_1, v_2, ..., v_n\}$ and a set of edges, E = $\{e_1, e_2, ...,e_m\}$ that indicate pairwise relationships, E = V x V. The edges may be directed or undirected. If $(v_i, v_j) \in$ E, then v_i and v_j are neighbors.

In general, the execution of a graph algorithm in a typical graph processing framework involves *reading input* data (that involves parsing one of the various graph formats like an adjacency list, usually from a distributed file system like HDFS), *pre-processing* the data (depending on the algorithm), *partitioning* the input graph (into sets of vertices and edges that can be assigned to different workers), the actual *computation* which runs the required graph algorithm and *writing output*. The frameworks differ in the implementation of each of these phases in many ways, but more importantly with the partitioning strategies and computation models. Below, we summarize few major (and relevant) aspects where graph processing frameworks differ, based on a study by Heidari et al[11]. For a more detailed taxonomy of a multitude of graph processing systems available, we refer the reader to the original paper.

Architecture The frameworks differ in whether they are shared-memory (single machine) or distributed. Prior to the recent growth in distributed graph processing systems, there have been several works on processing large scale graphs on a single machine. Even recently, there has been work that argues that distributed processing for graph processing incurs too much overhead and not really needed in practice [14]. However, shared memory frameworks are inherently limited in the amount of memory and CPU cores present in that single machine and are not scalable, so we only consider distributed frameworks in this paper. A distributed framework includes several processing units (workers) and each worker has access to only its own private memory. Each partition of the graph is typically assigned to one worker to be processed while the workers interact with each other via messages over network.

Programming Model The programming abstraction for each framework is designed based on a graph topology elements such as vertices and edges. Vertex-centric programming is the most mature distributed graph processing abstraction and several frameworks have been implemented using this concept. A vertex-centric system partitions the graph based on its vertices, and distributes the vertices across different partitions. Edges that connect vertices lying in two different partitions either form remote edges that are shared by both partitions or owned by the partition with the source vertex. Consequently, messages sent along these edges need to be sent over network to the remote worker that hold the neighboring vertex - which is usually taken care of by the framework itself. In edge-centric frameworks, edges are the primary unit of computation and partitioning, and vertices that are attached to edges lying in different partitions are replicated and shared between those partitions. It means that each edge of the graph will be assigned to one partition, but each vertex might exist in more than one partition. In this paper, we only look at vertex-centric frameworks like Giraph, and hence we limit ourselves to vertex partitioning.

Distributed Coordination Graph algorithms are usually iterative, so most graph processing platforms execute algorithms synchronously, meaning that concurrent workers process their share of the work iteratively, over a sequence of globally coordinated and well-defined iterations (generally called Supersteps). For example, in case of PageRank, an iteration consists of each vertex receving messages from its neighbors, computing its rank and sending out new rank to its neighbors. All workers wait until an iteration is finished and move on to the next iteration. This makes programming much more intuitive but it is more prone to stragglers where one worker could delay each iteration affecting the overall performance. Giraph, like most graph processing platforms, uses the synchronous model to allow for intuitive programming.

Other There are other aspects where frameworks differ, like whether the execution is disk-based or memory-based, or how they handle fault tolerance. Disk-based frameworks store data to disk not just while reading input and writing output, but even in between iterations, which hurts performance but does not necessitate the memory that is large enough to hold all the data during the computation. Giraph, like many other frameworks, is memory-based, meaning it never writes intermediate data to disk for better performance. Frameworks alsl provide checkpointing and fault recovery mechanisms to recover from failures, however we did not consider any of those for our purposes.

3 Discussion

Of the features of graph processing frameworks discussed in the last section, input partitioning seems like the one that effects the performance significanlty - so we decided to focus on that in this paper. As we limit ourselves to vertex-centric models, we only focus on vertex partitioning methods. There are two potential ways in which the partitioning affects performance:

- **The distribution of edges across partitions** Given that the computational complexity of most graph algorithms is on the order of number of edges, the share of each worker in the computation depends on number of edges its partition has. In order to avoid stragglers (especially for synchronous coordination frameworks like Giraph), the partitioning scheme should distribute the edges evenly and try to minimize the variation in the number of edges across partitions.
- **The fraction of inter-partition edges** As vertex partitioning puts vertices in different partitions, a lot of edges may end up with their vertices in different partitions - we call these inter-partition edges. As graph algorithms work by sending messages over edges, messages sent over these inter-partition edges need to be sent across network and add to a lot of performance overhead. A good partitioning scheme tries to minimize the number of inter-partition edges. Finding such an optimal partitioning is an NP-complete problem, so we can only go with heuristics.

The partitioning heuristics can be as simple as hash-based ones that hash vertices into buckers without regard to their connectivity or complex ones that try to reduce the edge cuts across partitions [18] that comes at added computational overhead of partitioning itself. Our intial goal was to implement a mix of simple and complex partitioning heuristics and evaluate the tradeoff between performance benefits of good partitioning versus the overhead of partitioning itself. However, due to limited time and the complexity of implementing these heuristics on Apache Giraph, we restricted ourselves to evaluating two simple yet very different partitioning schemes that does not incur any computational overhead:

- **Round-robin Partitioning** In this method of partitioning, we assign each vertex to partition on a round-robin basis (we use modulo operator on vertex id to acheive this). The idea is that for a graph with vertices indexed according to certain traversal algorithm, each partition would get vertices from all over the graph in a uniform way.
- **Range Partitioning** In this one, we divide the vertex index space into equal ranges (number

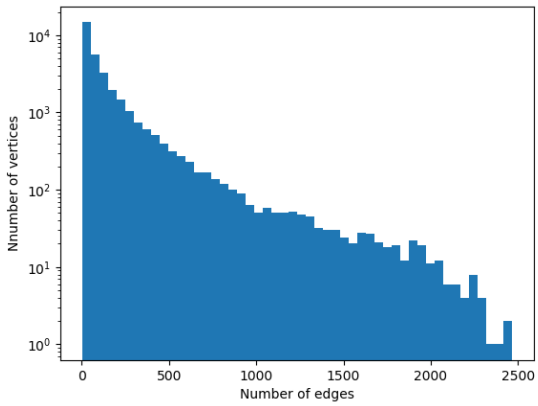

Figure 1: Degree distribution for a representative graph in our dataset, demonstrating that the graphs we chose are power-law graphs.

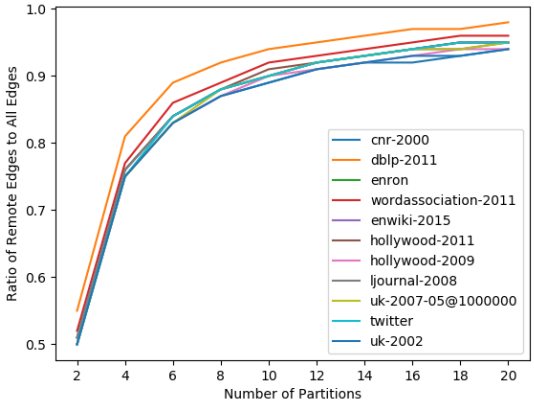

Figure 2: Ratio of remote edges (requiring messages to be sent across the network) to all edges for various numbers of partitions, where partitioning is done by taking the modulus of the vertex ID and number of workers.

of ranges being number of partitions required) and vertices from each range is assigned to each partition. This partitioning roughly divides the graph spatially and so partitions will exhibit better locality.

Since these partitioning methods are straightforward, we first perform a theoretical analysis of our input graphs to see the distribution of edges in and across partitions, which is presented in the next section. Later, we run PageRank algorithm on these graphs using Giraph and analyze the running times to see if our observations hold in practice.

4 Results

4.1 Input Graph Characteristics

We selected eleven graphs to analyze. Although we originally chose several others, we were limited by the hardware we had the funds to rent to analyzing relatively small graphs. A tradeoff exists between the size of a graph, the number of workers assigned to handle its vertices, and the amount of memory on a server. First, there is a minimum amount of memory that a Giraph worker requires in order to run. This puts an upper bound on the number of workers that can be run on any machine. Second, there appears to be a minimum number of workers that can be assigned to a graph of a given size in Giraph: when too few are assigned, the job fails. Presumably, this is because each worker can only feasibly handle a certain number of vertices, although it is unclear why Giraph chooses to terminate jobs rather than allowing them to eventually complete. This factor puts a lower bound on the number of workers that can be assigned to a graph, which is dependent on the size of the graph. Given these constraints, we restricted ourselves to graphs with approximately 200 million edges or fewer.

Our graphs came from the Laboratory for Web Algorithmics (LAW) [4, 3]. We considered using artificial graphs generated by Facebook's Darwini project [7], but eventually concluded that they were too large. The LAW graphs come from a variety of places. We selected a diverse subset of them, as shown in Table 1.

We also paid attention to the distribution of the degrees of the vertices in the graphs that we chose. Since most natural graphs are power-law distributed, our graphs primarily follow that pattern. Figure 1 shows a representative example of the degree distribution of a graph in our dataset.

Name	Vertices	Edges	Vertex means...	Edge means...
cnr-2000	325557	3216152	Website	Hyperlink
dblp-2011	986324	6707236	Scientist	Paper collaboration
enron	69244	276143	Person	Recipient of email
wordassociation-2011	10617	72172	Word	Interpreted association
hollywood-2011	2180759	228985632	Actor	Appearance in movie
hollywood-2009	1139905	113891327	Actor	Appearance in movie
ljournal-2008	5363260	79023142	User	Friend
uk-2007-05@1000000	1000000	41247159	Website	Hyperlink
twitter	41652230	1468365182	User	Follower
uk-2002	18520486	298113762	Website	Hyperlink

Table 1: Details of graph datasets

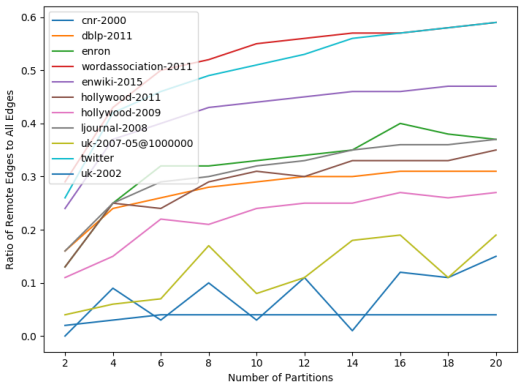

Figure 3: Ratio of remote edges to all edges for various numbers of partitions, where range partitioning is used.

We ran experiments to look at two metrics that we believe are predictors of performance: the number of edges given to each worker that require network requests, and the range of the number of edges given to each worker. We chose the first metric because we predict that if a worker is assigned a high proportion of edges that require it to make network requests, the time taken to run each step of computation will be much higher. Recall that in PageRank, every computational step requires a message to be sent along every edge. If that message begins and ends at vertices that are stored on the same machine, the overhead is much lower than it would be if the source and destination vertices are stored on two separate machines, thus necessitating a network request. Therefore, we measure the ratio between the number of local edges and the total edges assigned to a worker for various numbers of partitions and two partitioning schemes. Figures 2 and 3 show the results.

Figure 2 shows the partitioning scheme where vertices are assigned to workers by taking the modulo of their vertex ID with the number of workers in the cluster. Figure 3 shows the results of ranged partitioning. We hypothesized that both partitioning schemes would yield a graph similar to Figure 2, because if vertices are truly randomly assigned, then there is a greater chance of a vertex being assigned to a remote worker than a local one as the number of workers increases. However, range partitioning is not random - it takes locality into account. The graphs we analyze are taken from real-world data. Since they have to be created by some sort of crawler or traversal algorithm, they exhibit a certain amount of locality. Range partitioning therefore does a significantly better job than modulo partitioning at reducing the fraction of remote edges to local edges, as shown in Figure 3.

The second predictor of performance that we measured was the variation of the number of edges assigned to each worker. We assigned an equal number of vertices to each worker with all of our partitioning schemes, but this did not guarantee that the number of edges was the same. To our surprise, large graphs tend to approach edge equanimity even with the sim-

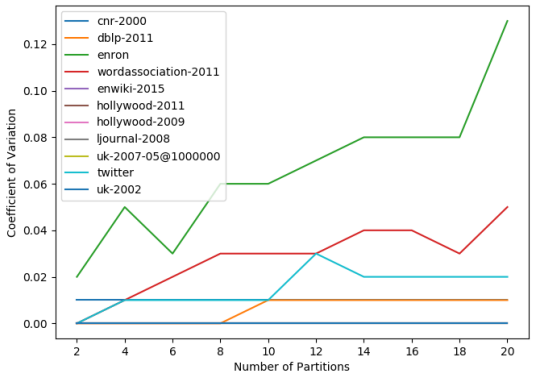

Figure 4: Range of the maximum number of edges handled by each worker to the minimum handled by any worker, expressed by the coefficient of variation, for various numbers of partitions, with modulo partitioning.

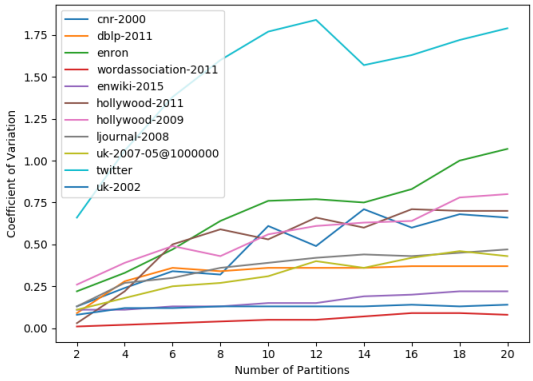

Figure 5: Range of the maximum number of edges handled by each worker to the minimum handled by any worker, expressed by the coefficient of variation, for various numbers of partitions, with range partitioning.

ple partitioning algorithms we use, which make no attempt to assign edges equally. We quantified this relationship in terms of the coefficient of variation of the number of edges assigned to each worker, for a varied number of workers, as shown in Figures 4 and 5.

Figure 5 shows that range partitioning led to a significantly higher difference between the number of edges assigned to each worker. This is likely due to locality again - range partitioning tends to capture any clustering present in the graph, whereas modulo partitioning evenly distributes clusters and breaks them up. One graph in particular stood out from the others: twitter, which draws edges between followers on the Twitter social network. Twitter is the largest graph we analyze, but large graphs do not necessarily appear to correspond to high coefficients of variation when it comes to edge distribution. It is possible that the twitter graph has more severe clustering than the other graphs, but why it should be so much higher than (for example) another social network graph like ljournal-2008, we do not know. Figure **??** shows a lower range of number of edges assigned to each worker, perhaps because modulo partitioning is closer to completely random than range partitioning. Here, many graphs have such even edge partitioning that their coefficient of variation is zero for all numbers of partitions.

Our takeaway from these experiments is that there is a tradeoff to be made when choosing a partitioning algorithm. If a particular job is concerned with having workers that lag behind the fastest workers in the cluster, it is probably important to choose to distribute edges as well as vertices evenly among the workers, and modulo partitioning should be used. If a job is likely to be network bound, range partitioning may be more effective, since it leverages locality to reduce the number of remote edges each worker must handle.

4.2 Apache Giraph

Apache Giraph[8] is a popular open-source implementation of Pregel[13]. Giraph runs on Hadoop MapReduce and uses Map-only jobs to schedule and coordinate the vertex-centric workers and uses HDFS for storing and accessing graph datasets. It is developed in Java and has a large community of developers and users such as Facebook[5]. Giraph has a faster input loading time compared to Pregel because of using byte array for graph storage. On the other hand, this method is not efficient for graph mutations, which lead to decentralized edges when re-

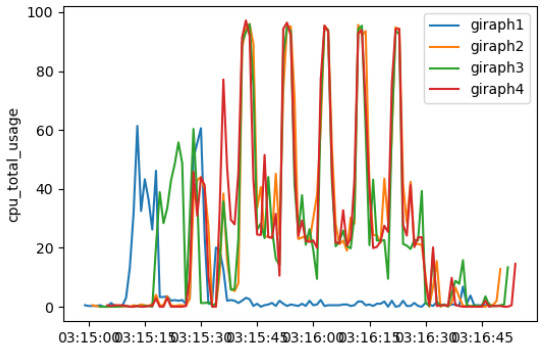

Figure 6: CPU usage on all the nodes from a sample PageRank run over time. The five spikes indicate five iterations of the algorithm.

moving an edge. Giraph inherits the benefits and deficiencies of the Pregel vertex-centric programming model. We picked Giraph for the supposed ease of use of this framework and the community support.

We ran Giraph on a 4-node hadoop cluster. Each of these nodes are t2.xlarge AWS machines each with 4 vCPUs, 16 GB Memory, 32 GB SSD and upto 1 Gbps Network. Giraph workers run as Map jobs, each in their YARN containers. YARN allows setting limits on the number of cores and memory to each container, so we limit each giraph worker to 1 vcore and 2 GB memory - so we could run a maximum of around 20 giraph workers given our cluster capacity. We chose PageRank as our graph algorithm since it is global and involves computation at all the vertices and edges in every iteration. We ran five iterations of PageRank (called SuperSteps) on each graph, with the two partitioners that we implemented in Giraph using *HashPartitionerFactory* and *SimpleLongRangePartitionerFactory* classes. The CPU usage on all the machines from a sample run of Giraph using round-robin partitioning on Facebook's Darwini graphs is shown in Figure 6.

A significant amount of the time spent on this project was spent setting up our cluster and installing Giraph and all of its dependencies. Even for a well-supported and widely used piece of software like Giraph, this was a time consuming challenge. To begin with, version mismatches of Java, HDFS, Ubuntu, and Giraph itself caused a number of difficult-to-debug errors. For example, installing the most recent version of Java caused a mysterious build error that took hours to track down a workaround for (the workaround being to downgrade Java). Additionally, once Giraph had been successfully built, we spent eight hours tuning various settings before the example job provided in the documentation would run without error. This was due to a number of factors. First, configuration settings are scattered across many different files, making it challenging to identify the source of a problem and difficult to predict how the settings would interact with each other. Second, few facilities existed for fine-tuning resource allocation. Many jobs failed with "out of memory" exceptions because it was difficult to tune the amount of memory they would be allocated. Third, error messages are not gathered in a central location - the existence of numerous log files made tracking down errors time consuming. Finally, Giraph seems to be a highly sensitive and delicate system. Unfortunately, as a result it is fragile: the slightest misallocation or imbalance of a resource causes jobs to fail with little explanation.

Given limited total memory capacity of our cluster (4*16 GB), we could only run medium sized graphs (with around 100 million edges) as Giraph does everything in memory. We ran PageRank on each of the graphs we discussed in previous section with both round-robin and range partitioners while varying the number of giraph workers (each with 2 GB memory) from 2 to 20. The execution times (excluding any preprocessing time) for these graphs using different partitioners are shown in figures 7 and 8. The missing data points for few input graphs at 2, 4 and 6 workers is due to the fact that these graphs are too big and cannot be run with fewer workers.

In both cases, we expected the computational time to go down as the number of workers increased, which only happened with the range partitioner. Computation times for range partitioner are less compared to round-robin partitioner for every graph, which seems to indicate that network overhead affects the performance more than straggler problems caused by the imbalance in partitioning. In

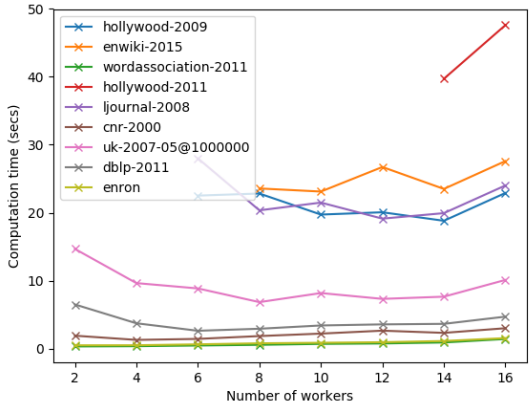

Figure 7: PageRank computation time for various graphs on Giraph with RoundRobin vertex partitioner as number of workers (partitions) are increased.

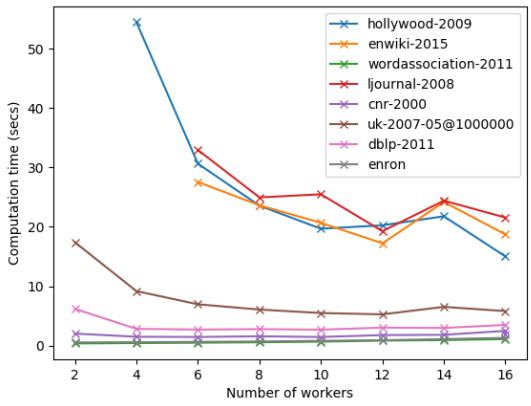

Figure 8: PageRank computation time for various graphs on Giraph with Range vertex partitioner as number of workers (partitions) are increased.

general, we didn't see the results we were expecting. Our conclusion was that Giraph has a lot of setup and preprocessing overheads that adds a lot of noise that significantly affects our runs since we use medium sized graphs that take only few seconds to run. We believe that using much larger graphs (like Twitter or Facebook) would give us the results that could let us clearly draw conclusions, but we could not run them due to resource limitations. We leave that to future work.

5 Conclusion

We present a small study on the effects of partitioning algorithms on performance predictors (such as the distribution of edges on workers) and performance itself on Apache Giraph. We conclude that a tradeoff exists between reducing the number of messages that must be sent over the network and equalizing the number of edges that are distributed to each node: to optimize the former, range partitioning should be used, and to optimize the latter, modulo partitioning is best.

References

[1] Alexandrov, A., Bergmann, R., Ewen, S., Freytag, J.-C., Hueske, F., Heise, A., Kao, O., Leich, M., Leser, U., Markl, V., Naumann, F., Peters, M., Rheinländer, A., Sax, M. J., Schelter, S., Höger, M., Tzoumas, K., and Warneke, D. The stratosphere platform for big data analytics. *The VLDB Journal 23*, 6 (Dec. 2014), 939–964.

[2] Ammar, K., and Özsu, M. T. Experimental analysis of distributed graph systems. *Proc. VLDB Endow. 11*, 10 (June 2018), 1151–1164.

[3] Boldi, P., Rosa, M., Santini, M., and Vigna, S. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World Wide Web* (2011), S. Srinivasan, K. Ramamritham, A. Kumar, M. P. Ravindra, E. Bertino, and R. Kumar, Eds., ACM Press, pp. 587–596.

[4] Boldi, P., and Vigna, S. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)* (Manhattan, USA, 2004), ACM Press, pp. 595–601.

[5] Ching, A., Edunov, S., Kabiljo, M., Logothetis, D., and Muthukrishnan, S. One trillion edges: Graph

processing at facebook-scale. *Proc. VLDB Endow. 8*, 12 (Aug. 2015), 1804–1815.

[6] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In *Symposium on Operating System Design and Implementation (OSDI)* (San Francisco, CA, USA, December 2004).

[7] EDUNOV, S., LOGOTHETIS, D., WANG, C., CHING, A., AND KABILJO, M. Darwini: Generating realistic large-scale social graphs. *arXiv:1610.00664 [cs]* (Oct. 2016). arXiv: 1610.00664.

[8] GIRAPH. http://giraph.apache.org/. accessed: 2019-05-03.

[9] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 17–30.

[10] GUO, Y., BICZAK, M., VARBANESCU, A. L., IOSUP, A., MARTELLA, C., AND WILLKE, T. L. How well do graph-processing platforms perform? an empirical performance evaluation and analysis. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium* (Washington, DC, USA, 2014), IPDPS '14, IEEE Computer Society, pp. 395–404.

[11] HEIDARI, S., SIMMHAN, Y., CALHEIROS, R. N., AND BUYYA, R. Scalable graph processing frameworks: A taxonomy and open challenges. *ACM Comput. Surv. 51*, 3 (June 2018), 60:1–60:53.

[12] LOW, Y., BICKSON, D., GONZALEZ, J., GUESTRIN, C., KYROLA, A., AND HELLERSTEIN, J. M. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow. 5*, 8 (Apr. 2012), 716–727.

[13] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2010), SIGMOD '10, ACM, pp. 135–146.

[14] MCSHERRY, F., ISARD, M., AND MURRAY, D. G. Scalability! but at what cost? In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems* (Berkeley, CA, USA, 2015), HOTOS'15, USENIX Association, pp. 14–14.

[15] OUSTERHOUT, K., RASTI, R., RATNASAMY, S., SHENKER, S., AND CHUN, B.-G. Making sense of performance in data analytics frameworks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2015), NSDI'15, USENIX Association, pp. 293–307.

[16] RASMUSSEN, A., LAM, V. T., CONLEY, M., PORTER, G., KAPOOR, R., AND VAHDAT, A. Themis: An i/o-efficient mapreduce. In *Proceedings of the Third ACM Symposium on Cloud Computing* (New York, NY, USA, 2012), SoCC '12, ACM, pp. 13:1–13:14.

[17] RASMUSSEN, A., PORTER, G., CONLEY, M., MADHYASTHA, H. V., MYSORE, R. N., PUCHER, A., AND VAHDAT, A. Tritonsort: A balanced and energy-efficient large-scale sorting system. *ACM Trans. Comput. Syst. 31*, 1 (Feb. 2013), 3:1–3:28.

[18] SALIHOGLU, S., AND WIDOM, J. Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management* (New York, NY, USA, 2013), SSDBM, ACM, pp. 22:1–22:12.

[19] YAN, D., CHENG, J., LU, Y., AND NG, W. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proc. VLDB Endow. 7*, 14 (Oct. 2014), 1981–1992.

[20] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. 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* (Berkeley, CA, USA, 2012), NSDI'12, USENIX Association, pp. 2–2.