AN INTRODUCTION TO GRAPH COMPRESSION TECHNIQUES FOR IN-MEMORY GRAPH COMPUTATION

AMIT CHAVAN
Computer Science Department
University of Maryland, College Park
amitc@cs.umd.edu

ABSTRACT. In this work we attempt to answer the following question: How large a graph can we process using a vertex-centric model of computation in the main memory of a single machine? Specifically, we use a modified Pregel framework to calculate PageRank, identify connected components, and single source shortest path algorithms on two large graphs. While it is not possible to load these graphs into memory with naive representations, we show that by using well known graph compression techniques, we can not only load the graphs but also run vertex centric programs on them, even on machines with fairly limited memory.

We evaluate both adjacency list and adjacency matrix graph compression. For high-degree vertices in an adjacency list, we show a space savings of around 70%, while for all vertices in the graphs we saved around 45%. Using a compressed adjacency matrix representation we saved around 40% for all vertices – high-degree vertices could not be compressed further because of extra data associated with these vertices. After compressing the LiveJournal graph, which has around five million vertices, we were able to run the PageRank algorithm in approximately 10 seconds per superstep, while the shortest path algorithm ran in 8 minutes per superstep.

1. **Introduction.** Graphs are ubiquitous: computer networks, social networks, mobile call networks, the World Wide Web, protein regulation networks, to name a few. In recent years, there has been a sharp surge in the volume of available data across all domains. Analysts, sociologists, computer scientists and others are interested in exploring the nature of relationships, patterns, occurence of communities, etc. to understand certain types of behavior or predict events amongst many other objectives.

Designing scalable systems for analyzing, processing and mining huge real-world graphs has become one of the most timely problems facing systems researchers. Consequently, many existing data-parallel abstractions and new graph-parallel abstractions have been proposed for efficient computing on such graphs. Data parallel abstractions, like MapReduce [7], are argued to be inefficient on graphs (see [14], [5]). To this effect, graph parallel abstractions like Pregel [16] and GraphLab [14], and high-performance memory based computation engines like Spark [18] have been proposed. These systems are able to scale to graphs of billions of edges by distributing the computation. Although distributed computational resources are now available, these systems have to deal with issues of cluster management, fault tolerance, and often unpredictable performance. On the contrary, systems like GraphChi [11], uses persistent storage

Submitted for the fulfillment of requirement of a Scholarly Paper for Master of Science without thesis option. Supervised by Prof. Amol Deshpande(amol@cs.umd.edu).

as a memory extension for processing large graphs. GraphChi exploits properties of sparse graphs to partition them into disk blocks and requires only a small number of sequential disk block transfers to efficiently perform asynchronous computation.

None of the aforementioned systems thoroughly investigate the impact of graph compression strategies to make the computation efficient. Graph compression strategies are geared towards making efficient use of available memory/disk to store graph structure, while permitting a given set of queries that can be answered on the compressed representation. In this work, we investigate the performance trade-offs of using graph compression techniques in a vertex centric framework. The two most common graph representations are adjacency list and adjacency matrix, and many techniques have been proposed to get a compact representation (using smallest number of bits/edge) for both. To the best of our knowledge, this work takes the first steps in using compressed graph representation to process large graphs in the memory of a single machine.

The rest of our report is organized as follows: We outline the salient features of Pregel and GraphLab and highlight the type of problems they are able to solve in Section 2. Section 4 discusses our vertex centric model in detail. Since we want to perform the computation on a single machine, we look at graph compression as a tool to help us represent large-size graphs. We talk about our specific compression techniques in Section 5. We present our preliminary evaluation of these techniques in Section 6 and conclude in Section 7.

2. **Related Work.**

Recently, new parallel abstractions like Pregel [16] and GraphLab [14] [15] have been proposed to address the limitations in applying existing abstractions like MapReduce[7] to Machine Learning algorithms on graph data. The common theme in these abstractions is a *vertex-centric* model of computation. In this model, the framework provides an ability to execute user defined update functions on a vertex which operates on the data associated with small neighborhoods of the vertex. These frameworks are able to scale to graphs of billions of edges by distributing the computation across a cluster of machines. We describe each of them next.

2.1. **Pregel.**

Introduced in 2010, Google's Pregel[16] system is a computational model used for processing large graphs in a scalable manner. Programs in Pregel are executed as a sequence of iterations or rounds. Within each round, a vertex can receive messages sent in the previous round, send messages to other vertices for the next round, and modify its own state or its outgoing edges. In Pregel, the rounds are called supersteps. Pregel operates based on user-defined compute functions which do the tasks of a vertex. After each superstep, the vertices vote to halt, and once all vertices have halted, the execution ends. The halting is determined on some satisfactory computation result. The model has been designed for efficient and scalable execution on clusters of thousands of computers. This model can be used to efficiently compute widely used graph algorithms such as Page Rank, Shortest Paths, Bipartite Matching, and a Semi-Clustering. This open source framework's ease of implementation and powerful computation makes it a great choice for large scale graph algorithms.

2.2. **GraphLab/PowerGraph.**

GraphLab[14][15], compared to Pregel's synchronous model, is an asynchronous (or automatic synchronization) parallel framework geared more for Machine Learning tasks due to higher efficiency for ML tasks. GraphLab is a graph-based data model. It consists of a shared data table which consists of global data,

a scheduler, and update functions. GraphLab operates based on user-defined update functions which are applied on a vertex and transform the data in the scope of the vertex. In other words, compared to Pregel, there are no messages passed from vertices. GraphLab's scheduler determines the order of the update functions. GraphLab has various schedulers available such as a synchronous scheduler, a round-robin scheduler, and dynamic schedulers. With the synchronous scheduler, every vertex is updated simultaneously. With the Round Robin scheduler, every vertex updated sequentially. GraphLab ensures serializability by preventing neighboring program instances from running simultaneously. The GraphLab framework can be used to design and implement parallel versions of ML algorithms such as belief propagation, Gibbs sampling, Co-EM, Lasso and Compressed Sensing.

Pregel and early version of GraphLab were not found to be suitable for natural graphs or very large graphs with billions of vertices and edges. Specifically, natural graphs have large neighborhoods with high degree vertices. Additionally, natural graphs have highly skewed Power Law degrees, where the top 1% are part of a large percentage of the edges. In other words, there are certain super-nodes or popular nodes in the social network setting. To address the challenges of power-law graph computation, the PowerGraph [9] exploits the structure of vertex-programs and explicitly factors computation over edges instead of vertices. The PowerGraph introduces greater parallelism, reduces network communication and storage costs, and provides a new highly effective approach to distributed graph placement. The PowerGraph uses individual vertex-programs, a delta caching procedure which allows computation state to be dynamically maintained, a fast approach to data layout for power-law graphs in distributed environments. The order in which active vertices are executed is up to the PowerGraph execution engine. PowerGraph programs can be executed both synchronously and asynchronously.

2.3. **GraphChi.** GraphChi[11] which is a spin-off of the GraphLab project can run very large graph computations on a *single* machine. It processes the graph from disk, but does so in a manner so as to avoid performing random IO. Their main contribution is the method of processing graph patitions incrementally (in shards) from disk using a technique called as the Parallel Sliding Windows algorithm.

3. **Graph Compression.** Let us consider graphs $G = (V, E)$ where V is the set of vertices and E is the set of edges. Let $n = |V|$ and $e = |E|$. The adjacency matrix representation requires $O(n^2)$ bits and the adjacency list representation requires $e \log n$ bits to represent the graph. We call the *neighbors* of a node $v \in V$ those $u \in V$ such that $(v, u) \in E$.

A fundamental primitive in the study of graphs is adjacency queries: seek the neighboring vertices of a given vertex. The Web graph (Web pages are nodes, hyperlinks are directed edges) is known to be highly compressible, as shown by [4]. In particular, Boldi and Vigna [4] exploit *lexicographic locality* in the Web graph: when web pages are ordered lexicographically by URL, proximal pages have similar neighborhoods. Empirically, two properties of the ordering by URL are observed to hold:

Similarity: pages that are close to each other in the lexicographic ordering tend to have similar sets of neighbors

Locality: many hyperlinks are intra-domain, and therefore likely to point to pages nearby in the lexicographic ordering

These two properties are exploited to compress the Web graph down to an amortized storage of a few bits per link, leading to efficient in-memory data strcutres for Web page adjacency queries. In essence, their compression scheme incorporates three main ideas. First, if the graph has many nodes whose neighborhoods are similar, then the neighborhood of a node can be expressed in terms of other nodes with similar neighborhoods. Second, if the destinations of edges exhibit locality, then small integers can be used to encode them. Third, rather than store the destination of each edge separately, one can use *gap encodings* to store a sequence of edge destinations. We illustrate a simplified version of this scheme in SectionXX.

Chierichetti et al. [6] build upon the scheme mentioned above to compress social networks. Specifically, they make use of link *reciprocity* in social networks. That is, if Alice is Bob's friend, then Bob is very likely to be Alice's friend. However, unlike Web graphs, where lexicographic ordering is both natural and crucial, social networks do not have an obvious natural ordering. One of the heuristics they propose is the *shingle* ordering heuristic: by building a fingerprint (shingle) of a node's neighbors and ordering them according to the shingles, if two nodes have significantly overlapping out-neighbors, i.e., share a lot of common neighbors, then with high probability, they will have the same shingle and hence by close to each other in the final ordering.

In addition to graph compression, [13] consider the application of large scale matrix-vector multiplication for the adjacency matrix representation of the graph. In this scenario, it is desired that the adjacency matrix has clustered edges: smaller number of denser blocks is better than larger number of sparser blocks. This has two benefits. First, dense blocks provide better opportunity for compression. Second, smaller number of denser blocks reduces the number of disk accesses. Thus [13] look for an ordering of nodes such that the adjacency matrix has the mentioned properties. They argue that traditional approaches of using graph partitioning algorithms to find good *cuts* and homogeneous regions so that nodes inside a region form dense communities, thereby leading to better compressions, are not suited well for real world, power law graphs [12].

4. **Graph Computation.** In this section, we describe the computational setting of our framework, outline the Java API and the graph input format we support.

4.1. **Computational Model.** We use the vertex centric model popularized by Pregel. However, we currently support only a subset of the Pregel API. In general, the input to our problem is a directed graph, $G = (V, E)$ where V is the set of vertices and E is the set of edges. We associate a value with each vertex $v \in V$ and possibly with each edge $e \in E$ (we discuss more about this in Section 5). We assume that the vertices are labeled from 0 to $|V|-1$. Given a directed edge $e = (u, v)$ we refer to u as the out-vertex and v as the in-vertex.

A typical computation consists of graph loading and initialization, followed by a sequence of *supersteps* separated by global synchronization points until the algorithm terminates. Within each supertep the vertices execute a user defined function (UDF) in parallel. The UDF has access to the (local) vertex state and a list of messages sent to the vertex by its neighbors. A vertex can modify its state or that of its outgoing edges, receive messages sent to it in the previous superstep and send messages to its neighbors (to be received in the next superstep). Note that we differ from the Pregel model of computation here: Pregel allows the user defined function to mutate the topology of

the graph and send messages to any vertex in the graph. Our current implementation does not support these operations.

As in Pregel, algorithm termination is based on every vertex voting to halt. In superstep 0, every vertex is in the *active* state; all active vertices participate in the computation of any given superstep. A vertex deactivates itself by voting to halt. This means that the vertex has no further work to do unless triggered externally (one of its neighbors sends a message to it), and the our framework will not execute the UDF for this vertex in subsequent supersteps. A vertex can become active/deactive any number of times during the course of the compuration. The computation terminates when all vertices are simultaneously inactive and there are no messages in transit.

4.2. **Java API.** Writing a program in our framework involves subclassing the **`PVertex`** class and implementing the **`PVertexFactory`** interface (see Figure 1). Each vertex has an associated value of type **`double`**. The user overrides the abstract **`compute()`** method which will be executed at each active vertex in every superstep. Additional methods are available to access meta-data about the vertex and the graph. The **`PVertexFactory`** interface is needed so that our framework can create new vertices of the user defined type. The user specifies the initialization parameters of a new vertex in the **`create()`** method.

```
class PVertex {
        abstract void compute(final Iterable<Double> messages);

        int getVertexId();
        int getSuperStep();

        double getValue();
        void setValue(final double newValue);

        Iterable<Integer> getOutNeighborIds();
        void sendMessage(final int outNeighborId, final double messageValue);
        void sendMessageToAllOutNeighbors(final double messageValue);

        void voteToHalt();
}

interface PVertexFactory<V extends PVertex> {
        V create(int id);
}
```

FIGURE 1. Vertex API

Vertices communicate with their neighbors by sending messages. A common pattern is for a vertex v to iterate over its outgoing edges and sending a message to the destination vertex of each edge. In our implementation, the destination for a message sent by v must be a neighbor of v.

4.3. **Input Formats.** The input file for the graph is an edge list consisting of **`(outVertexId, inVertexId)`** pairs or **`(outVertexId, inVertexId, edgeValue)`** triples, one pair/triple per line. In the former case, we use an adjacency list representation while in the latter case we use the adjacency matrix representation.

5. **System Design and Implementation.** In this section we describe the two strategies we use to represent the input graph compactly.

5.1. **Adjacency List.** In the *raw* adjacency list representation, we maintain two arrays at each vertex: One array stores all in-neighbors of the vertex while the other array stores all out-neighbors. Since these lists contain 4-byte integers, the space required to store the edges for a high-degree vertex is considerable. However, if we assume that the neighbor vertex ids dont vary considerably in magnitude, we can use delta encoding to reduce the memory footprint of our edge lists (see [2]).

Our delta encoding scheme works by first sorting the adjacency lists in ascending order (see [4],[6] for more refined versions of this scheme). Then, we store the full id of the first neighboring vertex and the remaining vertices are represented as the difference of their ID with the preceding vertex's ID. An example of this scheme in action is below:

Original	17	19	24	24	33	119
Encoded	17	2	5	0	9	86

TABLE 1. Example of delta encoding

From Table 5.1 it is apparent that delta encoding is less successful when the difference between vertex ids in a sorted list is large. We were not sure if we could use delta encoding to compress social network datasets because of this issue, but in practice (as shown in the evaluation section) it worked effectively.

Amazon's version of delta encoding stores the entire vertex id in a 4-byte integer if its difference cannot be stored in 1 byte. Their claim is that the time spent decoding differences is not worth the minimal space gains if those differences are large. However, their decoding process has to differentiate between vertices that do not need to be decoded versus ones that do. We decided to go for the more simple scheme, encoding every difference regardless of its magnitude, in the hope that our graph was significantly compressible. Since we did not assume that the differences could be encoded in a single byte, we used variable-length encoding ([1]), to store the differences using the least number of bytes.

In the above discussion we made the assumption that we have the entire edge lists in memory before compression. For graphs that are very large (millions of vertices), this assumption might not hold. Next, we describe how to incrementally compress the edge lists such that the entire list is never stored directly.

The technique we use, called *batch encoding*, compresses an edge list whenever its size reaches a threshold. This threshold is determined after considering both the graph properties (number of vertices, number of edges) and system properties (available memory). Instead of loading the entire edge lists for all vertices before compression beings, we store a compressed byte array of edges as well as a number of uncompressed vertices bounded by the batch size. When we've finished reading in the graph and

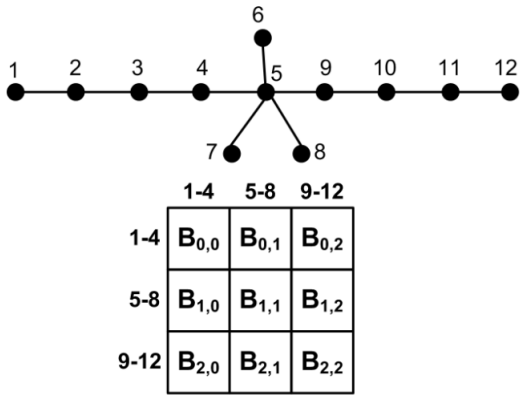

FIGURE 2. Block encoding of adjacency matrices

loading the edges, we might still have edges in the uncompressed vertices list. Thus, we also have to call the compression algorithm on every vertex once the graph has been read. Batch delta encoding makes it possible to loading a large graph like the live-journal data set into memory, which is infeasible when using the naive adjacency list representation.

5.2. **Adjacency Matrix.** To represent the adjacency matrix of the graph compactly, we use a scheme similar to the one proposed in [10] (see Figure 2). The idea here is to view the adjacency matrix as sub-matrices of size $b \times b$ and write each sub-matrix to its own file. Inside the file, we use $2 \log b$ bits to encode the source and destination vertex of each edge, 8 bytes for the edge value, 8 bytes for the message value (we are assuming message values to be of type `double`) and 1 byte boolean to indicate the state of the message value. We note that while $\log b$ bits are enough to encode a vertex id inside the block, we have only looked at byte level encoding. Once we encode all edges inside a sub-matrix, we compress the file using GZip compression.

6. Evaluation.

6.1. **Test Setup.** We evaluated our implementation on a quad-core 2.4 GHz Intel Core i7 processor, 6GB of memory given to the Java process (8GB system memory) and running Windows 8 (64-bit). We ran three algorithms: Page Rank (PR), finding weakly connected components (CC) and single source shortest paths (SSSP). Details about the input graphs for the three algorithms is shown in Table 6.1.

Graph name	Vertices	Edges
pokec	1632803	30622564
live-journal	4847571	68993773

TABLE 2. Experiment graphs

	pokec	live-journal
Graph creation time (ms)	21771	139055
Compression Time (ms)	2822	48941
Total links (2x)	61245128	137987546
Total number of bytes for all links	141766607	289780956
Space savings	**0.42**	**0.47**

TABLE 3. Adjacency list compression

Total Edges (in and out)	Byte array size	Space savings
20518	21680	0.73
16978	18144	0.73
15177	16139	0.73
8733	9757	0.72
7559	7834	0.74
6770	8541	0.68
4667	6314	0.66
4204	4935	0.70
2442	3569	0.63
2029	2984	0.63

TABLE 4. Savings for top 10 high degree vertices in Pokec dataset

6.2. **Compression Gains.** On an average (taken over 5 runs), the PR algorithm took 386 seconds on the pokec graph and 883 seconds on the live-journal graph. Runtime for SSSP algorithm is discussed in the next sub-section.

In the following discussion, we calculate the *compression ratio* as the ratio of the size of the data structure after compression to the un-compressed size. We define the *space savings* to be (1 − compression ratio).

6.2.1. *Adjacency list compression.* Table 6.2.1 gives a summary of the delta encoding scheme applied on the two datasets.

Table 6.2.1 and 6.2.1 outline the space savings for the top 10 high degree vertices in the Pokec and Live Journal dataset respectively.

6.2.2. *Adjacency matrix compression.* We used the adjacency matrix representation in order to run the SSSP algorithm. In the raw representation, we use 25 bytes per edge – 4 bytes each for the vertex ids, 8 bytes each for the edge weight and message value and 1 byte boolean to indicate the state of the message with respect to the current super step. Since we have to store additional data per edge, we process the graph in *partitions* with each partition having size at most 100000. Table 6.2.2 presents the summary of our results.

Total Edges (in and out)	Byte array size	Space savings
22889	25568	0.72
15554	17249	0.72
15218	19313	0.68
14029	15526	0.72
13439	17815	0.67
13183	15830	0.70
12422	16442	0.67
11861	13243	0.72
9787	11009	0.72
9312	10957	0.70

TABLE 5. Savings for top 10 high degree vertices in Live Journal dataset

	pokec	live-journal
Number of partitions	17	49
Per edge data after compression (bytes)	15.1	14.74
Time for one superstep (ms)	199956	472312
Min time for one partition (ms)	869	622
Max time for one partition (ms)	24468	49827

TABLE 6. Adjacency matrix compression

7. **Conclusion and Future Work.** In this work, we showed how using techniques from graph compression literature, we can load and process graphs that are too large to fit in main memory of a single machine. In addition, we can also run graph analysis algorithms, using the vertex centric model of computation, over the compressed graphs. Our results show that even simple compression techniques can dramatically improve the capability of a single machine to store and process large graphs. Graph compression techniques can also be used in existing graph parallel systems to make effective use of available resources.

Some of the directions for future work include identifying appropriate orderings of the vertices at runtime for effective adjacency list compression. We can also look at more sophisticated techniques of graph structure compression (like [8], [3], [17]). In addition to compressing the graph structure, we can also identify opportunities to compress graph data, i.e., vertex and edge attributes and messages passed around during computation.

REFERENCES.

[1] *Variable length quantity*, 2013 (accessed April, 2013). URL http://en.wikipedia.org/wiki/Variable-length_quantity.

[2] *Netflix graph*, 2013 (accessed April, 2013). URL http://techblog.netflix.com/2013/01/netflixgraph-metadata-library_18.html.

[3] Daniel K Blandford, Guy E Blelloch, and Ian A Kash. Compact representations of separable graphs. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 679–688. Society for Industrial and Applied Mathematics, 2003.

[4] Paolo Boldi and Sebastiano Vigna. The webgraph framework i: Compression techniques. In *In Proc. of the Thirteenth International World Wide Web Conference*, pages 595–601. ACM Press, 2003.

[5] Rishan Chen, Xuetian Weng, Bingsheng He, and Mao Yang. Large graph processing in the cloud. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 1123–1126, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0032-2. doi: 10.1145/1807167.1807297. URL http://doi.acm.org/10.1145/1807167.1807297.

[6] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. On compressing social networks. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, pages 219–228, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-495-9. doi: 10.1145/1557019.1557049. URL http://doi.acm.org/10.1145/1557019.1557049.

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

[8] Tomas Feder and Rajeev Motwani. Clique partitions, graph compression and speeding-up algorithms. In *Journal of Computer and System Sciences*, pages 123–133. ACM press, 1991.

[9] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, Hollywood, CA, 2012. USENIX. ISBN 978-1-931971-96-6. URL https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez.

[10] U Kang. Mining tera-scale graphs: Theory, engineering and discoveries. *Carnegie Mellon University*, 2012.

[11] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 31–46, Hollywood, CA, 2012. USENIX. ISBN 978-1-931971-96-6. URL https://www.usenix.org/conference/osdi12/technical-sessions/presentation/kyrola.

[12] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Statistical properties of community structure in large social and information networks. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, pages 695–704, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-085-2. doi: 10.1145/1367497.1367591. URL http://doi.acm.org/10.1145/1367497.1367591.

[13] Yongsub Lim, U. Kang, and C. Faloutsos. Slashburn: Graph compression and mining beyond caveman communities. *Knowledge and Data Engineering, IEEE Transactions on*, 26(12):3077–3089, Dec 2014. ISSN 1041-4347. doi: 10.1109/TKDE.2014.2320716.

[14] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, Catalina Island, California, July 2010.

[15] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *PVLDB*, 2012.

[16] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 international conference on Management of data*, pages 135–146. ACM, 2010.

[17] Saket Navlakha, Rajeev Rastogi, and Nisheeth Shrivastava. Graph summarization with bounded error. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 419–432. ACM, 2008.

[18] 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*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1863103.1863113.