# Graph Sampling with Distributed In-Memory Dataflow Systems

Kevin Gomez [1,2], Matthias Täschner[1,2], M. Ali Rostami[1,2], Christopher Rost[1,2], Erhard Rahm[1,2]

**Abstract:** Given a large graph, graph sampling determines a subgraph with similar characteristics for certain metrics of the original graph. The samples are much smaller thereby accelerating and simplifying the analysis and visualization of large graphs. We focus on the implementation of distributed graph sampling for Big Data frameworks and in-memory dataflow systems such as Apache Spark or Apache Flink and evaluate the scalability of the new implementations. The presented methods will be open source and be integrated into Gradoop, a system for distributed graph analytics.

**Keywords:** Graph Analytics; Distributed Computing; Graph Sampling; Data Integration

## 1 Introduction

Sampling is used to determine a subset of a given dataset that retains certain properties but allows more efficient data analysis. For graph sampling it is necessary to retain not only general characteristics of the original data but also the structural information. Graph sampling is especially important for the efficient processing and analysis of large graphs such as social networks [LF06, Wa11]. Furthermore, sampling is often needed to allow the effective visualization and computation of global graph measures for such graphs.

Our contribution in this paper is to outline the distributed implementation of known graph sampling algorithms for improved scalability to large graphs as well as their evaluation. The sampling approaches are added as operators to the open-source distributed graph analysis platform Gradoop[3] [Ju16, Ju18] and used for interactive graph visualization. Like Gradoop, our distributed sampling algorithms are based on the dataflow execution framework Apache Flink but the implementation would be very similar for Apache Spark. To evaluate horizontal scalability, speedup and absolute runtimes we use the synthetic graph data generator LDBC-SNB [Er15].

This work is structured as follows: We briefly discuss related work in Section 2 and provide background information on graph sampling in Section 3. In Section 4, we explain the distributed implementation of four sampling algorithms with Apache Flink. Section 5 describes the evaluation results before we conclude in Section 6.

[1] University of Leipzig, Database Group & ScaDS.AI Dresden/Leipzig
[2] [gomez, taeschner, rostami, rost, rahm]@informatik.uni-leipzig.de
[3] http://www.gradoop.com

cc by sa doi:10.18420/btw2021-15

## 2 Related Work

Several previous publications address graph sampling algorithms but mostly without considering their distributed implementation. Hu et al. [HL13] surveyed different graph sampling algorithms and their evaluations. However, many of these algorithms cannot be applied to large graphs due to their complexity. Leskovec et al. [LF06] analyze sampling algorithms for large graphs but there is no discussion of distributed or parallel approaches. Wang et al. [Wa11] focuses on sampling algorithms for social networks but again without considering distributed approaches.

The only work about distributed graph sampling we are aware of is a recent paper by Zhang et al. [ZZL18] for implementations based on Apache Spark. In contrast to our work, they do not evaluate the speedup behavior for different cluster sizes and the scalability to different data volumes. Our study also includes a distributed implementation and evaluation of random walk sampling. Zhang et al. [Zh17] have evaluated the influence of different graph sampling algorithms on graph properties such as the distribution of vertex degrees as well as on the visualization of the samples. These aspects do not depend on whether the implementation is distributed and are thus not considered in this short paper.

## 3 Background

We first introduce two basic definition of a graph sample and a graph sample algorithm and then specify some selected sampling algorithms.

### 3.1 Graph Sampling

A directed graph $G = (V, E)$ can be used to represent relationships between entities, for example, interactions between users in a social network. The user can be denoted as a vertex $v \in V$ and a relationship between two users $v$ and $u$ can be denoted as a directed edge $e = (v, u) \in E$.

Since popular social networks such as Facebook and Twitter contain billions of users and trillions of relationships, the resulting graph is too big for both, visualization and analytical tasks. A common approach to reduce the size of the graph is to use graph sampling to scale down the information contained in the original graph.

**Definition 1** (Graph Sample) A graph $S = (V_S, E_S)$ is a sampled graph (or graph sample) that defines a subgraph of graph $G = (V, E)$ iff the following three constraints are met: $V_S \subseteq V$, $E_S \subseteq E$ and $E_S \subseteq \{(u, v)|u, v \in V_S\}$.[4]

[4] In the existing publications, there are different approaches toward the vertices with zero-degrees in the sampled graph. Within this work we choose the approach to remove all zero-degree vertices from the sampled graph.

**Definition 2** (Graph Sample Algorithm) A graph sample algorithm is a function from a graph set $\mathcal{G}$ to a set of sampled graphs $\mathcal{S}$, as $f : \mathcal{G} \rightarrow \mathcal{S}$ in which the set of vertices $V$ and edges $E$ will be reduced until a given threshold $s \in [0, 1]$ is reached. $s$ is called *sample size* and is defined as the ratio of vertices $s_V = |V_s|/|V|$ (or edges $s_E = |E_s|/|E|$) the graph sample contains compared to the original graph.

### 3.2 Basic Graph Sampling Algorithms

Many graph sampling algorithms have already been investigated but we will limit ourselves to four basic approaches in this paper: *random vertex sampling*, *random edge sampling*, *neighborhood sampling*, and *random walk sampling*.

Random vertex sampling [Zh17] is the most straightforward sampling approach that uniformly samples the graph by selecting a subset of vertices and their corresponding edges based on the selected sample size $s$. For the distributed implementation in a shared-nothing approach, the information of the whole graph is not always available in every node. Therefore, we consider an estimation by selecting the vertices using $s$ as a probability. This approach is also applied on the edges in the random edge sampling [Zh17].

We extend the idea of the the simple random vertex approach to improve topological locality using the random neighborhood sampling. Therefore, when a vertex is chosen to be in the resulting sampled graph, all neighbors are also added to the sampled graph. Optionally, only incoming or outgoing edges can be taken into account to select the neighbors of a vertex.

For the random walk sampling [Wa11], one or more vertices are randomly selected as start vertices. For each start vertex, we follow a randomly selected outgoing edge to its neighbor. If a vertex has no outgoing edges or if all edges were followed already, we jump to any other randomly chosen vertex in the graph and continue the walk there. To avoid keeping stuck in dense areas of the graph we added a probability to jump to another random vertex instead of following an outgoing edge. This process continues until a desired number of vertices have been visited, thus the sample size $s$ has been met. All visited vertices and all edges whose source and target vertex was visited will be part of the graph sample result.

## 4 Implementation

The goals of the distributed implementation of graph sampling are to achieve fast execution and good scalability for large graphs with up to billions of vertices and edges. We therefore want to utilize the parallel processing capabilities of shared-nothing clusters and, specifically, distributed dataflow systems such as Apache Spark [Za12] and Apache Flink [Ca15]. In contrast to the older MapReduce approach, these frameworks offer a wider range of transformations and keep data in main memory between the execution of operations. Our implementations are based on Apache Flink but can be easily transferred to Apache Spark.

| Transf. | Type | Signature | Constraints |
|---|---|---|---|
| Filter | unary | $I, O \subseteq A$ | $O \subseteq I$ |
| Map | unary | $I \subseteq A, O \subseteq B$ | $\lvert I\rvert = \lvert O\rvert$ |
| Reduce | unary | $I, O \subseteq A \times B$ | $\lvert I\rvert \geq \lvert O\rvert \wedge \lvert O\rvert \leq \lvert A\rvert$ |
| Join | binary | $O \subseteq I_1 \bowtie I_2$ | $I_1 \subseteq A, I_2 \subseteq B$ |

(I/O : input/output datasets, A/B : domains)

Tab. 1: Selected transformations and their characteristics.

We first give a brief introduction to the programming concepts of the distributed dataflow model. We then outline the implementation of our sampling operators.

### 4.1 Distributed Dataflow Model

The processing of data that exceeds the computing power or storage of a single computer can be handled through the use of distributed dataflow systems. Therein the data is processed simultaneously on shared-nothing commodity cluster nodes. Although details vary for different frameworks, they are designed to implement parallel data-centric workflows, with datasets and primitive transformations as two fundamental programming abstractions. A *dataset* represents a typed collection partitioned over a cluster. A *transformation* is a deterministic operator that transforms the elements of one or two datasets into a new dataset. A typical distributed program consists of chained transformations that form a dataflow. A scheduler breaks each dataflow job into a directed acyclic execution graph, where the nodes are working threads and edges are input and output dependencies between them. Each thread can be executed concurrently on an associated dataset partition in the cluster without sharing memory.

Transformations can be distinguished into *unary* and *binary* operators, depending on the number of input datasets. Table 1 shows some common transformations from both types which are relevant for this work. The *filter* transformation evaluates a user-defined predicate function to each element of the input dataset. If the function evaluates to `true`, the element is part of the output. Another simple transformation is *map*. It applies a user-defined map function to each element of the input dataset which returns exactly one element to guarantee a one-to-one relation to the output dataset. A transformation processing a group instead of a single element as input is *reduce* where the input, as well as output, are key-value pairs. All elements inside a group share the same key. The transformation applies a user-defined function to each group of elements and aggregates them into a single output pair. A common binary transformation is *join*. It creates pairs of elements from two input datasets which have equal values on defined keys. A user-defined join function is applied for each pair that produces exactly one output element.

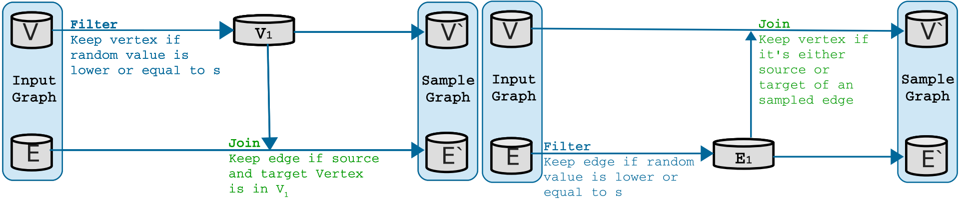


Fig. 1: Dataflow RV Operator.

Fig. 2: Dataflow RE Operator.

## 4.2 Sampling Operators

The operators for graph sampling compute a subgraph by either randomly selecting a subset of vertices or a subset of edges. In addition, neighborhood information or graph traversal can be used. The computation uses a series of transformations on the input graph. Latter is stored in two datasets, one for vertices and one for edges. For each sampling operator a filter is applied to the output graph's vertex dataset to remove all zero-degree vertices following the definition of a graph sample in Section 3.

### 4.2.1 Random Vertex (RV) and Random Edge (RE) Sampling

Both operators RV and RE are very simple to implement. A graph with it's vertex set V and edge set E serves as input for both operators. As well a sample size $s$ has to be defined. The RV operator (Figure 1) starts with applying a filter transformation on the vertex dataset. Within the filter transformation a user-defined-function can be used to generate a random value $r \in [0, 1]$ for each vertex which is compared to the given sample size $s$. If the random value $r$ is lower or equal to $s$, the vertex will be part of a new dataset $V_1$. Otherwise the vertex will be filtered out. In the next step we join the vertices of $V_1$ with the edge dataset $E$. Within the join transformation we only select edges which corresponding source and target vertex is contained within $V_1$. Those selected edges will be part of the resulting edge set $E'$. $V'$ is the resulting vertex set which is equal to $V_1$. The result is a graph sample containing the vertex set $V'$ and the edge set $E'$.

The RE operator (Figure 2) works the other way around, as a filter transformation is applied to the edge dataset $E$ of the input graph. An edge will be kept, again if the generated random value $r \in [0, 1]$ is lower or equal to $s$. After the filter transformation, all remaining edges will be stored in a new dataset $E_1$. In the following we join $E_1$ with the input vertex dataset $V$. Within the join we select all source and target vertices of the edges contained within $E_1$. The selected vertices are stored within the final vertex dataset $V'$. $E'$ contains the final edge dataset which is equal to $E_1$. The result is again a graph sample $G_S = (V', E')$.

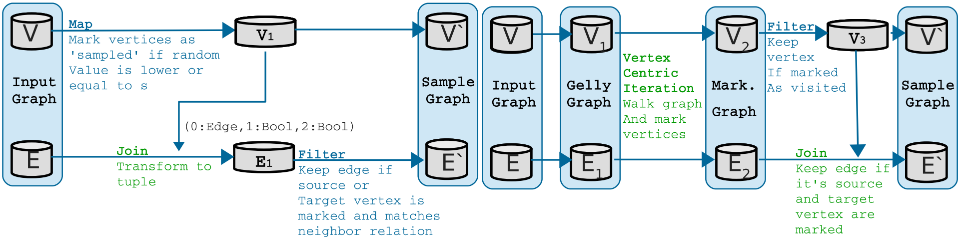


Fig. 3: Dataflow RVN Operator.

Fig. 4: Dataflow RW Operator.

### 4.2.2 Random Vertex Neighborhood (RVN) Sampling

As for RV and RE, a graph and a sample size $s$ serves as input for the RVN operator (Figure 3). This approach is similar to the RV operator but also adds the direct neighbors of a vertex to the graph sample. The selection of the neighbors can be restricted according to the direction of the connecting edge (incoming, outgoing or both). In the implementation, we start with using a map transformation and randomly selecting vertices of the input vertex dataset $V$ and mark them as sampled with a boolean flag, iff a generated random value $r \in [0, 1]$ is lower or equal than the given sample size $s$. In a second step, the vertex dataset $V_1$ is joined with the input edge dataset $E$, transforming each edge into a tuple containing the edge itself and the boolean flags for its source and target vertex. In an additional filter transformation on the edge tuples, we retain all connecting edges of the vertices of $V_1$ and apply the given neighborhood relation to create the final edge dataset $E'$. This relation will be either a neighbor on an incoming edge of a sampled vertex, a neighbor on an outgoing edge, or both. Note that the vertex set $V'$, which is equal to $V_1$, of the resulting graph can contain vertices with a degree of zero. Since we choose the approach to remove all zero-degree vertices from the resulting graph sample, an additional filter transformation has to be applied.

### 4.2.3 Random Walk (RW) Sampling

This operator uses a random walk algorithm to walk over vertices and edges of the input graph. Each visited vertex and edges connecting those vertices will then be returned as the sampled graph. Figure 4 shows the dataflow of an input graph to a sampled graph of this operator. The input for this operator is a graph, a sample size $s$, an integer $w$ and a jump probability $j$. At the beginning we transform the input graph to a specific Gelly format.

We are using Gelly[5], the Google Pregel [Ma10] implementation of Apache Flink, to implement a random walk algorithm. Pregel utilizes the bulk-synchronous-parallel [Va90] paradigm to create the vertex-centric-programming model. An iteration in a vertex-centric

[5] For more technical details see: https://bit.ly/39UuEWS

| SF | $\|V\|$ | $\|E\|$ | Disk usage | $s$ |
|---|---|---|---|---|
| 1 | 3.3 M | 17.9 M | 2.8 GB | 0.03 |
| 10 | 30,4 M | 180.4 M | 23.9 GB | 0.003 |
| 100 | 282.6 M | 1.77 B | 236.0 GB | 0.0003 |

Tab. 2: LDBC social network datasets.

program is called *superstep*, in which each vertex can compute a new state and is able to prepare messages for other vertices. At the end of each superstep each worker of the cluster can exchange the prepared massages during a synchronization barrier. In our operator we consider a message from one vertex to one of its neighbors a *walk*. A message to any other vertex is considered as *jump*.

At the beginning of the random walk algorithm $w$ start vertices are randomly selected and marked as visited. The marked vertices will be referred to as *walker*. In the first superstep each walker either randomly picks one of its outgoing and not yet visited edges, walks to the neighbor and marks the edge as traversed. Or, with the probability of $j \in [0, 1]$ or if there aren't any outgoing edges left, jumps to any other randomly selected vertex in the graph. Either the neighbors or the randomly selected vertices will become the new walker and the computation starts again. Note, this algorithm is typically executed using only one walker. Since this would create a bottleneck in the distributed execution we extended the algorithm with the multi-walker approach as just explained.

For each completed superstep the already visited vertices are counted. If this number exceeds the desired number of sampled vertices, the iteration is terminated and the algorithm converges. Having the desired number of vertices marked as visited, the graph is transformed back and is now containing the marked vertex set $V_2$ and the marked edge set $E_2$. Using a filter transformation on $V_2$ we create the resulting vertex set $V'$. A vertex will be kept if it is marked as visited. By joining $V'$ with $E_2$ we only select edges which source and target vertex occur in the final vertex dataset $V'$ and create the final edge dataset $E'$. The result of the operator is the graph sample $G_S = (V', E')$.

## 5 Evaluation

One key feature of distributed shared-nothing systems is their ability to respond to growing data sizes or problem complexity by adding additional machines. Therefore, we evaluate the scalability of our implementations with respect to increasing data volume and computing resources.

**Setup.** The evaluations were executed on a shared-nothing cluster with 16 workers connected via 1 GBit Ethernet. Each worker consists of an Intel Xeon E5-2430 6 x 2.5 Ghz CPU,

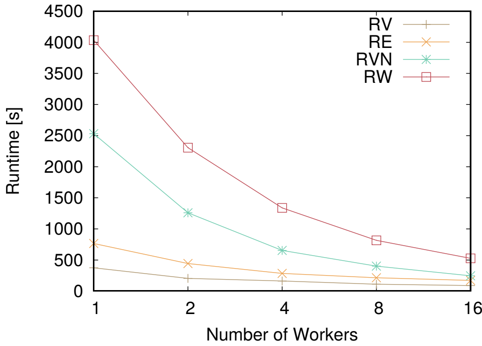


Fig. 5: Increase worker count.

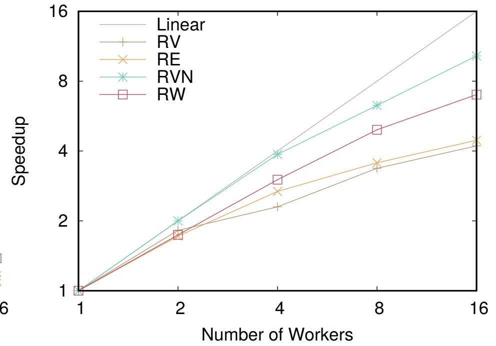


Fig. 6: Speedup over workers.

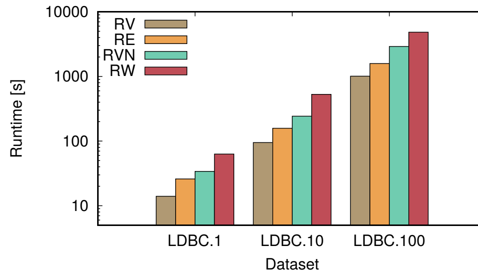


Fig. 7: Increase data volume.

48 GB RAM, two 4 TB SATA disks and runs openSUSE 13.2. We use Hadoop 2.6.0 and Flink 1.9.2. We run Flink with 6 threads and 40 GB memory per worker.

To evaluate the scalability of our implementations we use the LDBC-SNB data set generator [Er15]. It creates heterogeneous social network graphs with a fixed schema. The synthetic graphs mimic structural characteristics of real-world graphs, e.g., node degree distribution based on power-laws and skewed property value distributions. Table 2 shows the three datasets used throughout the benchmark. In addition to the scaling factor (SF) used, the cardinality of vertex and edge sets, the dataset size on hard disk as well the used sample size $s$ are specified. Each dataset is stored in the Hadoop distributed file system (HDFS). The execution times mentioned later include loading the graph from HDFS (hash-partitioned), computing the graph sample and writing the sampled graph back to HDFS. We run three executions per setup and report the average runtimes.

## 5.1 Scalability

In many real-world use cases data analysts are limited in graph size for visual or analytical tasks. Therefore, we run each sampling algorithm with the intention to create a sampled

graph with round about 100k vertices. The used sample size $s$ for each graph is contained in Table 2. As configurations, we use `Direction.BOTH` for the RVN algorithm and $w = 3000$ walker and a jump probability $j = 0.1$ for the RW algorithm.

We first evaluate the absolute runtime and scalability of our implementations. Figure 5 shows the runtimes of the four algorithms for up to 16 workers using the `LDBC.10` dataset. One can see, that all algorithms benefit from more resources. However *RVN* and *RW* gain the most. For *RVN*, the runtime is reduced from 42 minutes on a single worker to 4 minutes on 16 workers. In comparison the *RW* implementation needs 67 minutes on a single worker and 9 minutes using the whole cluster. The more simpler algorithms *RV* and *RE* are already executed relatively fast on a single machine. Their initial runtime of 405 seconds and 768 seconds on a single worker only got reduced to 105 seconds and 198 seconds.

In the second experiment we evaluate absolute runtimes of our algorithms on increasing data sizes. Figure 7 shows the runtimes of each implementation using 16 workers on different datasets. The results show that the runtimes of each algorithm increases almost linearly with growing data volume. For example, the execution of the *RVN* algorithm required about 314 seconds on `LDBC.10` and 2907 seconds on `LDBC.100`. The *RW* algorithm shows equal results with 549 seconds on `LDBC.10` and 5153 seconds on `LDBC.100`. The more simple algorithms *RV* and *RE* show very good scalability results as well.

### 5.2 Speedup

In our last experiment we evaluate relative speedup of our implementations on increasing cluster size. Our evaluation (Figure 6) show a good speedup for all our implementations executed on the `LDBC.10` dataset. The algorithms *RVN* and *RW* show the best speedup results of 11.2 and 8.5 for 16 workers. We assume that the usage of multiple join transformations within the *RVN* operator and the utilization of the iterative Gelly implementation used in the *RW* operator limits the speedup performance of both algorithms. However, for *RV* and *RE* we report the lowest speedup results of around 4.0. This behaviour can be explained with the already low runtimes we discovered on a single worker during the scalability evaluation in Figure 5. Hence, both operators won't benefit much of increasing cluster size, since increasing communication costs have an negative influence in both runtimes and speedup capabilities.

## 6 Conclusion

We outlined distributed implementations for four graph sampling approaches using Apache Flink. Our first experimental results are promising as they showed good speedup for using multiple workers and near-perfect scalability for increasing dataset sizes. In our ongoing work we will provide distributed implementations for further sampling algorithms and optimization techniques such as custom partitioning.

## 7 Acknowledgements

This work is partially funded by the German Federal Ministry of Education and Research under grant BMBF 01IS18026B in project ScaDS.AI Dresden/Leipzig.

## Bibliography

[Ca15] Carbone, Paris; Katsifodimos, Asterios; Ewen, Stephan; Markl, Volker; Haridi, Seif; Tzoumas, Kostas: Apache Flink: Stream and Batch Processing in a Single Engine. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 36(4), 2015.

[Er15] Erling, Orri et al.: The LDBC social network benchmark: Interactive workload. In: Proc. SIGMOD. 2015.

[HL13] Hu, Pili; Lau, Wing Cheong: A Survey and Taxonomy of Graph Sampling. CoRR, abs/1308.5865, 2013.

[Ju16] Junghanns, Martin; Petermann, André; Teichmann, Niklas; Gómez, Kevin; Rahm, Erhard: Analyzing Extended Property Graphs with Apache Flink. In: Proc. ACM SIGMOD Workshop on Network Data Analytics (NDA). 2016.

[Ju18] Junghanns, Martin; Kiessling, Max; Teichmann, Niklas; Gómez, Kevin; Petermann, André; Rahm, Erhard: Declarative and distributed graph analytics with GRADOOP. PVLDB, 11:2006–2009, 2018.

[LF06] Leskovec, Jure; Faloutsos, Christos: Sampling from Large Graphs. In: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. KDD '06, ACM, New York, NY, USA, pp. 631–636, 2006.

[Ma10] Malewicz, Grzegorz; Austern, Matthew H.; Bik, Aart J.C; Dehnert, James C.; Horn, Ilan; Leiser, Naty; Czajkowski, Grzegorz: Pregel: A System for Large-scale Graph Processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data. SIGMOD '10, ACM, New York, NY, USA, pp. 135–146, 2010.

[Va90] Valiant, Leslie G.: A Bridging Model for Parallel Computation. Commun. ACM, 33(8):103–111, August 1990.

[Wa11] Wang, T.; Chen, Y.; Zhang, Z.; Xu, T.; Jin, L.; Hui, P.; Deng, B.; Li, X.: Understanding Graph Sampling Algorithms for Social Network Analysis. In: 2011 31st International Conference on Distributed Computing Systems Workshops. pp. 123–128, June 2011.

[Za12] Zaharia, Matei; Chowdhury, Mosharaf; Das, Tathagata; Dave, Ankur; Ma, Justin; McCauley, Murphy; Franklin, Michael J; Shenker, Scott; Stoica, Ion: 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. USENIX Association, 2012.

[Zh17] Zhang, Fangyan; Zhang, Song; Chung Wong, Pak; Medal, Hugh; Bian, Linkan; Swan II, J Edward; Jankun-Kelly, TJ: A Visual Evaluation Study of Graph Sampling Techniques. Electronic Imaging, 2017(1):110–117, 2017.

[ZZL18] Zhang, Fangyan; Zhang, Song; Lightsey, Christopher: Implementation and Evaluation of Distributed Graph Sampling Methods with Spark. Electronic Imaging, 2018(1):379–1–379–9, 2018.