Efficiently Loading Big Graph in Distributed System *

Xiaoguang Li
School of Information, Liaoning University
Liaoning Province, P.R.China
xgli@lnu.edu.cn

ABSTRACT

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Theory

Keywords

Big Graph Partition, Graph Sampling, Graph Loading

1. INTRODUCTION

Many real-world complex systems can be represented as graphs and networks from information networks, to communication networks, to biological networks. Given a large massive graph with millions or billions of vertices and edges, loading the graph into a distributed system in a partitioning manner facilitates the parallel processing of graph analysis, such as minimum span tree, shortest path, clique. Partitioning graph also minimizes the total number of cross-partition edges among partitions so as to minimize the data transfer along the edges.

Data loading in distributed system has been extensively studied in the field of database. Many partitioning techniques, such as hash partitioning, range partitioning, round robin partitioning, have been widely applied into massive databases. Partitioning the tuple by such techniques can be processed in a constant time depending on partitioning function and the tuple itself, and therefore whatever the size of data on disk is, the data will be accessed sequentially and read once. It is infeasible, however, to load a graph by an existing algorithm with a linear time-complexity, since graph partitioning is well-known as NP-hard problem.

*A full version of this paper is available as

In comparison with the traditional partitioning techniques, partitioning a vertex always requires not only the vertex itself, but also its neighbors. Moreover, the algorithm of graph partitioning need to repeat scanning the graph and iteratively find an optimal series of partition target, such as minimizing *edge-cuts*. Thus, most of graph partitioning algorithms usually required the graph to fit into main memory, such as Kernighan-Lin (KL) [5, 8], MIN-MAX Greedy algorithm[3, 9], spectral partitioning[12], balanced minimum cut[7]. Many real-world graphs, unfortunately, have grown exceedingly large in recent years and are continuing to grow at a fast rate. For example, the Web graph has over 1 trillion webpages (Google); most social networks (e.g., Facebook, MSN) have millions to billions of users; many citation networks (e.g., DBLP, Citeseer) have millions of publications; other networks such as phone-call networks, email networks, stock-market networks, etc., are also massively large. With the size of graph becoming very large, it is often too expensive to partition large graph in memory. So, the technique called *graph summary* was proposed to shrink the original graph fit into the memory. *Hyper graph* is a kind of summary mentioned in multilevel approaches[10, 14, 6]. *Hyper graph* is built by selecting edges successively and collapsing them into a hyper vertex so as to obtain a graph small enough. *Graph skeleton* constructed by a small random sample from graph's edges[7] was used to compute the minimum cut. But the construction of both such graph summaries is also required to probe the original graph repeatedly, which suffered from potential and massive access to disk.

Besides the graph data to load is stored as the file of vertices and edges by self-defined format on disk, the graph is always of the format of online stream of vertices and edges, flowing into a distributed system continuously. The gigantic size of data stream implies that we generally cannot store the entire stream data set in main memory or even on the disk of partitioner. Certainly, it is impractical to scan through an entire data stream more than once. To the best of our knowledge, there were few works about partitioning graph data stream, except for a streaming graph partitioner proposed by Isabelle.S et.al[13]. They provided a empirical study of a set of natural heuristics for streaming balanced graph partitioning. The quality of partitions highly depends on the locality of stream. Certain systems or applications that need as good a partitioning as possible will still repartition the graph after it has been fully loaded onto the cluster.

Totally, loading big graphs in an manner of graph partitions is becoming a challenge on account of massive disk I/O overhead. In this paper, we propose an I/O efficient

approach to loading a big graph approximately. In this study, our focus is not to design a new algorithm of graph partitioning, but to improve I/O efficiency by trading off the accuracy of partitioning against the efficiency of loading. Firstly loading graph on disk is considered, the proposed approach scans through the graph twice only by two phases. In the first phase, a representative subgraph fitting into main memory is built out of big graph data by the first scanning, and then a set of partitions, called *approximate partitioner*, are retrieved by a gain-based partitioning algorithm in memory. After that, in the second phase, all the remainder of vertices and edges will be partitioned on account of their locality by the approximate partitioner in linear time by the second scanning.

The representative subgraph and its approximate partitioner here play the role as partitioning function as the traditional techniques of database do. Intuitionally, the representative subgraph should be selected for the target of the *edge-cuts*, that is, the edge-cuts by approximate partitioner applied were expected to be as close to the edge-cuts of original graph as possible. However, it is quite difficult to evaluate the edge-cuts in advanced, since the final edge-cuts are not related to the initial partitions only, but also to the partitioning design. We notice that gain-based algorithms partition a vertex by its difference of connections to the partitions, called as *gain of vertex*. Actually, a vertex's partition isn't determined by its absolute value of gain, but by the sign of gain. That is, if a vertex has more connections to the partition i than j, it should be allocated in the partition i. The absolute value of gain just makes the effect of the order of partitioning vertex. So, we consider that if the gain sign of vertex in the representative graph is consistent with the one in the original graph, the vertex will be set to the partition by approximate partitioner as the *true partitioner* of original graph does. Based on this idea, the representative subgraph is built by an unequal sampling on edges to keep the expectation of inconsistence of gain bounded by given threshold. We theoretically give the analysis of the expectation of gain inconsistency, and point out that the unequal sampling should be bias to the vertex of low degree if the expectation of gain inconsistency is held.

The representative subgraph includes most of vertices in the original graph, and holds the structure information(indicated by edges) enough to evaluate the sign of gain. In addition, the approximate partitioner is consistent approximately with the true partitioner with respect to edge-cuts. Hence, partitioning each vertex of remainder may depend on its locality in a constant time complexity at the second phase, and need not to migrate vertices iteratively for partition optimizing. Traditionally, the locality of vertex consists of its neighbors. It works when the graph data are stored by BFS order, where the locality can be retrieved after d edges cached (d is the degree of vertex). For the order of DFS or random, the partitioner can't determine the locality until the end of file or stream. It means that we have to cache all the vertices, and their related edges, which are not contained in representative graph. In fact, there is an widely accepted observation by the study of *Graph Random Walk* in the real world, that is, a random walk started inside a good partition will mostly stay inside the partition [1]. It implied that, beside for the neighbors of vertex, the vertices linking indirectly to the vertex support the determination of partition to some extent. Here, we expand the "neighbors" to the vertices linking to the vertex directly or indirectly in the following window of vertices and edges read sequentially so as to avoid caching all the data. We also put forwards the measure, called *strength of locality support*, to evaluate the vertex's contribution to the locality in terms of *influence* and *attraction*.

For graph stream, each the edge of new arrival is sampled probabilistically to the representative subgraph as mentioned above. The difference from above is that the new vertex of not chosen edge will be partitioned immediately by current approximate partitioner according to its expand locality. The challenge is that the approximate partitioner needs to be adjusted with the representative subgraph varied, and to re-assign the vertices partitioned by previous approximate partitioners.

However it is quite overhead to query the assigning context of vertex, since the vertex has been written to persist disk and the vertices of context maybe reside the different partitions. Querying the context of vertex must lead to the massive I/O operation on disk or network. Our idea is to cache the context of vertex of high likelihood to be re-assigned in the slave site. .....

experiment - conclusion

The rest of the paper is organized as follows. We first present the related works in Section 2. Next, We introduce the background about graph loading in Section 3. Partitioning graph by sampling as the core of graph loading is proposed and discussed in Section 4, including the overall algorithm of loading, the analysis of partitioning and graph sampling design. Section 5 studies loading the real-world graph, including the implementation of graph sampling design - degree bias sampling, and edges assigning. Section 6 focuses on loading the graph stream. We compare the approaches in this paper with other state-of-the-art graph loading algorithms and graph partitioning algorithms in Section 7. Finally, we conclude in Section 8.

2. RELATED WORKS

Graph partitioning as the core of graph loading in distributed system is an NP-hard problem, and therefore all known algorithms for generating partitions merely return approximations to the optimal solution. Numerous algorithms for graph partitioning have been developed in the field of social network analysis, graph theory, et. al. Geometric approaches use the geometric coordinates associated with a graph's vertices to aid in the creation of a partition, but geometric feature also limit its application because geometric coordinate is not the case for all graphs. Spectral methods generally refer to algorithms that assign nodes to communities based on the eigenvectors of matrices, such as the adjacency matrix of the network itself or other related matrices[12].The main disadvantage of spectral algorithms lies in their computational complexity, and spectral clustering is hard to scale up to networks with more than tens of thousands of vertices. Gain-based partitioning algorithms, such as Kernighan-Lin (KL) algorithm[5, 8], MIN-MAX Greedy algorithm[3, 9], are based on the notion of gain metric for quantifying the benefit of allocating a vertex from one subset to the other. The gain of a vertex in KL algorithm and the variations is the reduction in edge-cut if the vertex is moved from its current partition to the other partition. While each iteration in the original KL algorithm had a complexity of $O(|E| \log |E|)$. The partition in the fami-

ly of MIN-MAX Greedy algorithms is immediately obtained by repeatedly adding a vertex to the sets that produces the minimum possible increase of the cut size.

To address the scale problem, *graph summaries* that substantially smaller than the original graph, typically can be used to return approximate partitions. Multilevel graph partitioning algorithms [10, 14, 6] are proposed to coarsen(contract) the input graph successively so as to obtain a graph small enough, partition this small graph and then successively project this partition back up to the original graph, refining the partition at each step along the way. The costs of coarsening and refining are both proportional to the number of edges, and the original graph was scanned many times and the temporary graph of coarsening and refining at each step must be saved.

Sampling graph is another way to obtain a small graph. Sampling on graphs has been used in many different flavors, but few in graph partitioning. Conceptually we can split the graph sampling algorithms into three groups: randomly selecting nodes, randomly selecting edges and the exploration techniques that simulate random walks or virus propagation to find a representative sample of the nodes[11]. Previous work focused on using sampling to condense the graph to allow for better visualization. Works on graph compression focused on transforming the graph to speed up algorithms. Techniques for efficiently storing and retrieving the web-graph were also studied. Internet modeling community studied sampling from undirected graphs and concluded that some graph properties can be preserved by random-node selection with sample sizes down to 30%. Works study separability and stability of various graph properties degree distribution, central betweeness, hop-plot[11]. To our knowledge, most related to our work is the algorithm determining the minimum cut base on *graph skeleton* constructed by a small random sample from graph's edges[7]. The algorithm need scan the original graph many times to probe the sampling ratio and compute the skeleton until a certain minimum cut value on the skeleton is greater than $\Omega((\log n)/\epsilon^2)$.

Isabelle. S et.al[13] proposed a streaming graph partitioning, which reads serial graph data from a disk onto a cluster. However, they just put forward ten heuristics of selecting the index of the partition where a vertex is assigned, and define three stream orderings of vertex arrival. One might need to perform a full graph partitioning once the graph has been fully loaded.

3. PROBLEM BACKGROUND

3.1 Notations

We model a graph to load as an undirected and unlabeled graph, $G = (V, E)$, where V is the set of vertices and E is the set of edges. In real-world, the data of graph are always treated as a sequence of edges and vertices, which stored on the disk of loading server by self-defined format, or streaming into loading server without the storage. Without the lost of generality, the data of graph are defined as the edge sequence $E_{seq} = (e_1, e_2, ..., e_m)$, for $e = (v_i, v_j), e \in E_{seq}$, v_i and v_j are the adjacent vertices joint by e, and we define the corresponding set of vertices of E_{seq} as $V_{seq} = \{v | v \in e, e \in E_{seq}\}$. Let n and m be the size of E_{seq} and V_{seq} respectively. For graph stream, n and m is increased with time. Generally, for most of graph in practice, we have $n << m$.

Graph loader is a program resided in loading server and responsible for reading the graph data from the disk or online stream into a distributed system with k storage nodes. Its goal is to find a close to optimal balanced graph partitioning with as more efficiency as possible, and save each partition into a storage node respectively. Graph partitioning is to divide V of G into k partitions, $P = \{P_1, P_2, ..., P_k\}$ such that $P_i \bigcap P_j = \emptyset$, for any $i \neq j$, and $|P_i| = n/k$, and $\bigcup_i P_i = V$. The number of edges whose incident vertices belong to different partitions is called *edge-cuts*. Optimal balanced graph partitioning is k set of partitions with minimized edge-cuts. We call the algorithm that searching the optimal graph partitioning as *optimal partitioner*.

We assume there is the limitation of the size of main memory in loading server, denoted by M, and $M << m$. Let $Partition|_{V \to P \bigcup \emptyset}$ be the function of mapping the vertex into its partition. If $Partition(u) = i$, we say that u is partitioned into P_i, while if $Partition(u) = \emptyset$, we say that u isn't partitioned yet.

3.2 Order of Vertex and Edge

The order of vertex and edge for a graph G is always following three orders: breadth-first search(BFS), depth-first search(DFS) and random search($Random$). BFS is generated by selecting a starting node from each connected component of the graph uniformly at random and is the result of a breadth-first search that starts at the given node. If there are multiple connected components, the component ordering is done at random. DFS is identical to the BFS ordering except that depth-first search is used. Both BFS and DFS are natural ways of linearizing graphs and are highly simplified models used by most of graph data collectors, such as web crawler. Here, we assume the vertices and edges in sequence obey single ordering. Random search is an ordering that assumes that the vertices arrive in an order given by a random permutation of the vertices. In practice, few graph sequences obey to the order of complete random, but they always are the sequence of the mixture between DFS and BFS.

The order of vertex and edge is important to the computation complexity of partitioning graph, specially, with the format of stream. It challenges the efficiency of graph partitioning not processed in main memory. Finding all the neighbors of a vertex as its locality, or computing the degree of a vertex, for example, is critical operation of graph partitioning, and these operations are easy implementing for BFS order since all the adjacent vertices and edges will be seen immediately in the sequence after the vertex, but for the order of random or DFS, we have to wait until the end of the sequence.

3.3 Gain-based Graph Partitioning

Most of popular graph partitioning algorithms are based on comparison of the external and internal cost of vertex, called as gain-based partitioning algorithm, such as KL and MIN-MAX Greedy. The approach proposed in this study is also based on the concept of vertex gain.

For a graph G, given a set of partitions P_1 and P_2 of V, $P_1 \bigcup P_2 = V$, suppose $u \in P_1$, the *external cost* E_u of u is defined as the number of edges connecting the vertex u and the vertex in P_2, while the *internal cost* I_u of u is the number of edges connecting the vertex u and the vertex in P_1. The D value of u, $D_u = E_u - I_u$, is defined as the

gain obtained by moving u from its current partition. Gain-based algorithms allocate a vertex from one subset to the other if it benefits from the gain of moving. For example, KL algorithm swaps a pair of vertices (u, v) if their total gain is improved iteratively, where u, v belong to different partition. MIN-MAX Greedy algorithm repeatedly add a vertex to the two sets that produces the minimum possible increase δf of the cut size f, where δf is equivalent to D value. For gain-based partitioning, we observed that the migration of vertex was decided by its gain, not by its order of gain,that is, whether the gain of vertex is the best or not, it will be migrated sooner or later if the partitioning benefits from its moving.

k-way graph partitioning usually is the procedure of recursive bisection adopted by the majority of the algorithms due to its simplicity. The procedure has two phases in general. It generates a k-way partition by performing a bisection on the original graph and recursively considering the resulting subgraphs in the first phase, and then repeated the application of the 2-way partitioning procedure to optimal in the second phase. Such algorithms are heuristic to find the best gain of moving at each iteration.

The partitioning process can be described by a full and complete binary tree, where the leafs are the final partitions and the internal nodes are the temporal results. For a binary tree of graph partitioning, we denote $LCA(i, j)$ as the lowest common ascendant of the partition P_i and P_j, that is, the descendants P_i and P_j are split recursively from their LCA.

4. LOADING GRAPH BY REPRESENTATIVE SUBGRAPH

As shown before, the target of this paper is to design a I/O-efficient approach to loading a big graph into k storage nodes with approximately minimum edge-cuts. We consider the loading of graph data on disk by scanning thorough disk twice in this chapter.

4.1 Loading Graph On Disk

Our idea is to design a partitioning function to allocate vertices and edges streamingly as the way of data partitioning processed by the traditional techniques in the field of database. Here, partitioning function is designed as a set of partitions, which is derived from a representative subgraph that fits into the memory and preserves the information of allocating vertex enough to determine vertex's partition with high probability and makes the final edge-cuts approximate to the optimal edge-cuts. The main steps are as follows:

- Firstly, a representative subgraph resided entirely in memory is selected out of the original graph with a sampling design by scanning through graph data.
- Secondly, the representative subgraph is divided into k partitions by an internal-memory gain-based algorithm mentioned in Chapter 3. The k partitions of representative subgraph is called as *approximate partitioner*.
- Finally, the remainder of edges and vertices that don't selected into the representative subgraph are allocated streamingly to the corresponding storage node on account of their locality and approximate partitioner by scanning through the graph data once again.

For such loading strategy, the questions we ask here are:

- What is a "good" representative subgraph out of big graph with the limitation of memory, and how to create it?
- How do we allocate the remainder of vertices and edges besides the representative graph without vertex migration?

Here, the representative subgraph is constructed by the technique of simple random sampling without replacement. As mentioned in related works, state-of-the-art graph sampling techniques can be broadly classified as *sampling by random node selection(RN), sampling by random edge selection(RE) and sampling by exploration(SE)*. Actually, for a very big graph on disk, it is infeasible for *SE* to explore the graph arbitrarily. For *RN*, the sample consists of the induced subgraph over the selected nodes and all edges among the nodes. It means that the vertex's neighbors need to be obtained efficiently, which is suitable for the sequence of BFS order only. Furthermore, It is hard to control the sampled graph size to fit in the memory, since $n << m$. So, we adopt *RE* to select the representative graph.

We firstly give an I/O-efficient loading algorithm for graph on disk, called *Sample-based Graph Loading(SGL)*. *SGL* is a general loading algorithm with no concern of the order of graph sequence and the goodness of representative graph. Both of problems will be leave discussing in the following sections. For *SGL*, the limitation of memory ρ refers to the limitation of edges since $n << m$ in practice, and m and n are unknown in advance. The main steps of *SGL* algorithm are shown as Algorithm 1. The sampling design of *SGL* as shown from Step 2 to Step 9 is a kind of *Reservoir Sampling* [15] with the probability of ρ/m, which is a family of randomized algorithms for randomly choosing a sample of items of given size from the list of either a very large or unknown number.

4.2 Expectation of Error Gain

For Question 1, the "goodness" of representative subgraph intuitively means what the extend of the edge-cuts for approximate partitioner is close to the one for optimal partitioner. However, it is quite difficult for graph partitioning to design a sampling algorithm with the objective of edge-cuts, since the final edge-cuts of partitioning algorithm isn't relevant to the heuristic only, but also the initial partitions. We observed that for gain-based partitioning algorithms, the partition of vertex isn't determined by its absolute value of gain, but the sign of gain. That is, if a vertex has more connections to the partition i than j, it should be allocated in the partition i. The absolute value of gain just makes the effect of the order of vertex allocation. For example, suppose that the vertex v of the partition P_1 has n_1 internal-connections to the partition P_1 and n_2 external-connections to the partition P_2, if $n_1 \geq n_2$, i.e the gain $n_1 - n_2 \geq 0$ then v will be stayed in P_1, otherwise moved into P_2. If the sign of vertex gain for a representative subgraph is consistent with the one for the original graph, we say the partition of vertex for representative subgraph will keep consistence with that for the original graph.

Here, we introduce the notion of *expectation of error gain* to measure the goodness of approximate partitioner instead of the edge-cuts. Since recursive bisection partitioning is

Algorithm 1 Sample-based Graph Loading (SGLd)

Input:
E_{seq}: Edge Sequence $(e_1, e_2, ..., e_m)$;
k: The Number of Storage nodes;
ρ: Limitation of Edges in Memory;
Output:
Graph Partitions $G_P^1, G_P^2, ..., G_P^k$ on k storage-nodes;

```
V_s = ∅, E_s = ∅;
Read the first ρ edges of E_seq into E_s and initialize V_s by E_s;
for i = ρ + 1 to m do
    Read the edge e_i = (u, v) from E_seq;
    j = random(1, i);
    if j < ρ then
        Select an item l from E_s on random and replace E_s[l] by e_i ;
    end if
end for
Build the sample graph G_s by E_s;
Apply a gain-based partitioning algorithm to compute k partitions G_P^s = {G_1^s, ..., G_k^s} for G_s;
for each G_i^s, i = 1, .., k do
    Save G_i^s as G_P^i on the site i;
end for
for each unsampled edge e = (u, v) ∈ E_seq do
    if u or v ∉ V_s then
        Allocate u or v to V_i^s by its locality and save it to the site i;
    end if
    if i ≠ j, where u ∈ V_i^s and v ∈ V_j^s then
        Label e as cut edge, and save e to both E_P^i and E_P^j;
    else
        Save e to E_P^i;
    end if
end for
return G_P;
```

adopted, without the lost of generality, we assume $k = 2$ and it is easy to extend k to any integer of 2^h. Given the graph G, let $P = \{P_A, P_B\}$ be a set of partitions for G, where $P_A \bigcup P_B = V$, and d be the degree of the vertex $v \in V$, $d \geq 0$. Let s be the probability of edge sampling, and $G^s = (V^s, E^s)$ be a subgraph of G selected by an edge-sampling algorithm, where $V^s \subseteq V, E^s \subseteq E$. $P^s = \{P_A^s, P_B^s\}$ is the corresponding partitions for P, where $P_A^s \subseteq P_A, P_B^s \subseteq P_B$. d^s is the degree of $v \in V^s$. We have the expectation of d^s, $E(d^s) = s \cdot d$.

Definition 1. A *boundary vertex* of partition is the vertex that has at least one link to other partitions. Let $\mathcal{B}$ be the set of boundary vertices for a graph partitions.

Definition 2. Given the graph G, and its partitions P, the *gain* of vertex $v \in V$ with respect to P is defined as the connection difference of v between P_A and P_B, and denoted as δ. Obviously, we have $-d \leq \delta \leq d$. For G^s and P^s, the *gain* of v with respect to P^s is defined as the connection difference of v between P_A^s and P_B^s, and denoted as δ^s.

Definition 3. For the graph G and the graph sample G^s, and the corresponding partitions P and P^s, given the vertex v and its degree d, if $\delta^s \cdot \delta > 0$, where $\delta^s \neq 0, \delta \neq 0$, or $\delta = 0 \wedge \delta^s = 0$, we say δ^s is *consistent* with δ, denoted as $\delta^s \equiv \delta$, otherwise δ^s is *inconsistent* with δ, denoted as $\delta^s \not\equiv \delta$. With $\delta^s \not\equiv \delta$, we introduce the variable $\tilde{\delta}$ as the error gain of v, and $\tilde{\delta} = |\delta|$.

THEOREM 1. *For the graph G and its sample G^s with sample ratio s, given the vertex $v \in V$ and its degree d, the expectation of error gain of v, denoted as $E(\tilde{\delta})$ satisfies that*

$$E(\tilde{\delta}) \leq \frac{1}{s}(1 - \exp(-\frac{sd}{2})) \tag{1}$$

PROOF. For $v \in V$, let $P(\delta^s \not\equiv \delta, \delta)$ be the probability that δ^s is inconsistent with given δ, and we have

$$P(\delta^s \not\equiv \delta, \delta) = P(\delta)P(\delta^s \not\equiv \delta|\delta) \tag{2}$$

Since $-d \leq \delta \leq d$, and we assume δ obeys uniform distribution with the interval $[-d, d]$, then

$$P(\delta) = \frac{1}{2d} \tag{3}$$

Let $X_1, X_2, ..., X_{d^s}$ be independent random variables taking on values -1 or 1, which means a connection to P_A or P_B is sampled, then we have $\delta^s = \sum_{i=1}^{d^s} X_i$. Here let $\epsilon = E\delta^s$. According to Definition 3, for $\delta^s \not\equiv \delta$, we have $\delta^s < E\delta^s - \epsilon$ if $\delta > 0$, or $\delta^s > \epsilon - E\delta^s$ if $\delta < 0$. Obviously, the expectation of δ^s is $s \cdot \delta$. According to the *Hoeffding Inequality*, we have

$$P(\delta^s \not\equiv \delta|\delta) \leq \exp(-\frac{2\epsilon^2}{4d^s}) = \exp(-\frac{s \cdot \delta^2}{2d}) \tag{4}$$

With Eq.(3) and Eq. (4), we have

$$P(\delta^s \not\equiv \delta, \delta) = \frac{1}{2d} \cdot \exp(-\frac{s \cdot \delta^2}{2d}) \tag{5}$$

With Eq. (5), we have

$$\begin{aligned} E(\tilde{\delta}) &\leq \sum_{\delta=-d}^{d} |\delta| \cdot P(\delta^s \not\equiv \delta, \delta) \\ &= 2\sum_{\delta=0}^{d} \delta \cdot P(\delta^s \not\equiv \delta, \delta) \\ &= \sum_{\delta=0}^{d} \frac{\delta}{d} \cdot \exp(-\frac{s \cdot \delta^2}{2d}) \end{aligned} \tag{6}$$

Let $x = \frac{\delta}{d}$ and with Eq. (6), then

$$\begin{aligned} E(\tilde{\delta}) &\leq \sum_{\delta=0}^{d} d \cdot x \cdot \exp(-\frac{sdx^2}{2}) \cdot \frac{1}{d} \\ &= \int_0^1 d \cdot x \cdot exp(-\frac{sdx^2}{2}) d(x) \\ &= \int_0^1 -\frac{1}{s} \exp(-\frac{sdx^2}{2}) d(-\frac{sdx^2}{2}) \\ &= -\frac{1}{s}[\exp(-\frac{sdx^2}{2})]_0^1 \\ &= \frac{1}{s}(1 - \exp(-\frac{sd}{2})) \end{aligned}$$

□

LEMMA 1. *The upper-bound of $E(\tilde{\delta})$, denoted as $\hat{E}(\tilde{\delta})$, is the decreasing function of sampling probability s, and the increasing function of the degree d.*

PROOF. The partial derivative of $\hat{E}(\tilde{\delta})$ with respect to s is

$$\frac{\partial \hat{E}(\tilde{\delta})}{\partial s} = \frac{1}{s^2}((\frac{sd}{2}+1)\exp(-\frac{sd}{2})-1)$$

According to the inequality $e^{-x} < \frac{1}{1+x}$, where $x > -1$, we have $\frac{\partial \hat{E}(\tilde{\delta})}{\partial s} < 0$. The partial derivative of $\hat{E}(\tilde{\delta})$ with respect to d is

$$\frac{\partial \hat{E}(\tilde{\delta})}{\partial d} = \frac{1}{2}\exp(-\frac{sd}{2}) > 0$$

□

LEMMA 2. *The change of $\hat{E}(\tilde{\delta})$ is more significant to the sampling probability s than to the degree d.*

PROOF. According to Lemma 1, we have

$$\begin{aligned}
&|\frac{\partial \hat{E}(\tilde{\delta})}{\partial s}| - |\frac{\partial \hat{E}(\tilde{\delta})}{\partial d}| \\
&= \frac{1}{s^2}(1-(\frac{sd}{2}+1)\exp(-\frac{sd}{2})) - \frac{1}{2}\exp(-\frac{sd}{2}) \\
&= \frac{1}{s^2\exp(sd/2)}(\exp(sd/2) - (1+sd/2) - \frac{s^2}{2})
\end{aligned}$$

Since $s^2 \geq 0$ and $\exp(sd/2) > 0$, the maximum of s is 1, and the minimum of d is 1, we have

$$|\frac{\partial \hat{E}(\tilde{\delta})}{\partial s}| - |\frac{\partial \hat{E}(\tilde{\delta})}{\partial d}| > 0$$

□

THEOREM 2. *$E(\tilde{\delta_G})$ is defined as the total expectation of error gain for G, that is $E(\tilde{\delta_G}) = \sum_{v\in\mathcal{B}} E(\tilde{\delta_v})$, where $E(\tilde{\delta_v})$ is the expectation of the error gain of v. The following inequality is satisfied*

$$E(\tilde{\delta_G}) \leq \sum_{1\leq d\leq d_{max}} \frac{n_d}{s}(1-(1-\frac{1}{k})^d)(1-\exp(-\frac{s\cdot d}{2})) \quad (7)$$

where n_d is the number of the vertex of degree d, and d_{max} is the maximum degree

PROOF. Obviously, the total expectation of error gain for G is determined by boundary vertex that has at least one neighbor on other partitions. Under k partitions, an edge's vertices have the probability $1-\frac{1}{k}$ to be on different partitions, and then we have the probability that a vertex of degree d has at least one link to another partition is $1-(1-\frac{1}{k})^d$. Due to the linearity of expectation and with Theorem 1, Eq. (7) is derived. □

4.3 Sampling Representative Subgraph

Naturally, the sampling design is preferred to set the sampling probability with $E(\tilde{\delta_G})$ bounded by a given threshold. For the inequality (7), however, it is quite difficult to estimate the uniform s for all degrees. So, the sampling probability for each vertex is set to hold $E(\tilde{\delta}) \leq \alpha$ with the uniform threshold α here. That is, with the degree d, if the sampling probability s satisfies

$$1-\alpha\cdot s \leq \exp(-\frac{s\cdot d}{2}) \quad (8)$$

according to Theorem 1, $E(\tilde{\delta}) \leq \alpha$ will be hold, and then $E_{\tilde{\delta_G}}$ is bounded by $\sum_d n_d(1-(1-\frac{1}{k})^d)\alpha$. Moreover, n_d can be estimated by $n\cdot P(d)$ with the probability distribution of degree $P(d)$ provided, such as the power-law-like distributions observed in many real graphs, and α can be set as $\frac{\beta}{n\sum_d P(d)(1-(1-\frac{1}{k})^d)}$ with the upper-bound β of $E_{\tilde{\delta_G}}$. In such case, the size of sample edges is $n\sum_d \frac{P(d)\cdot d}{s_d}$, where s_d is the sampling ratio for the degree d.

For the order of BFS, the degree of vertex can be calculated by caching locally a part of edge sequence, and the edge sampling probability of vertex can be determined with Equation (8). However, for the order of random or DFS, the degree of vertex cann't be determined until all of its adjacent edges have been confirmed to reach at the end of edge sequence. In such case extra scan of the edge sequence is required to count the degree for all the vertices, whereas it is time-consuming for big graph and infeasible for edge stream. Moreover, selecting β in advance is quite difficult since the cut value is unknown. To this problem, we relax the constrain of $E(\tilde{\delta}) \leq \alpha$ and resort to the target that making $E(\tilde{\delta})$ as small as possible. So an unequal sampling algorithm, called *Degree Bias Sampling (DBS)*, is proposed here. Note that $E(\tilde{\delta})$ in Theorem 1 can be transformed as

$$E(\tilde{\delta}) \leq \frac{d}{E(d^s)}(1-\exp(-\frac{E(d^s)}{2})) \quad (9)$$

Similar to Lemma 1 and Lemma 2, $\hat{E}(\tilde{\delta})$ is decreasing function of the expectation of degree d^s. The change of $\hat{E}(\tilde{\delta})$ is more significant to $E(d^s)$ than d. It implies that if the $\hat{E}(\tilde{\delta})$ is kept at the same level, the high sampling probability is expected for the vertex of low degree, otherwise the low sampling probability is preferred for the vertex of high degree. Based on this idea, *DBS* replaces the edge in the reservoir according to the current degree of its end vertices with the bias to low degree. Obviously, *DBS* also need the degree, but it does depend not on the absolute and final degree, but the relative and current degree.

The algorithm of *DBS* is shown in Algorithm 2. The element of V_s has two items: *vex* and *d*, representing the information of vertex and its degree respectively. $V_s[v]$ is the element of the vertex v. The element in E_s has four items: the item *edge* that includes the vertices u and v of edge, the item *weight* that is the sampling weight determined by the inverse of the minimum degree of u and v and the item with high relative weight has high probability of sampling, the item *rand* that is a random real in $[0,1]$, and the item *key* that is the key for weighted sampling and calculated by $rand^{1/weight}$. The edge is selected into the reservoir according to the relative *weight* by the algorithm of weighted random sampling in *DBS*, similar to *A-Res* proposed by Pavlos S. Efraimidis etc [4]. The differences between *DBS* and *A-Res* are that:

(1) for *A-Res* the weight of the item of population must be given in advanced, while for *DBS* the degree observed by now is used to determine the weight. The assumption under *DBS* is that the degree of vertex is relative high in the observations if it is of the high relative degree in the population. The assumption can be proved by the following: Suppose that the degrees of u_1 and u_2 are d_1 and d_2, $d_1 \geq d_2$, the total sampling degree of u_1 and u_2 is l and let k be the sampling degree of u_1, then the probability distribution of k is the distribution of *Bernouli* , that is, if l is big enough, $p(k) = C_l^k p^k (1-p)^{l-k} = \frac{\lambda^k}{k!}e^{-\lambda}$ where $p = \frac{d_1}{d_1+d_2}, \lambda = lp$.

Since $d_1 \geq d_2$, $p \geq 1/2$, then we have $p(k \geq l/2) > 1/2$.

(2) In *A-Res* the items except for the first ρ items will be checked one by one, while such strategy will cause the phenomenon *"first arrival, first dead"* for *DBS*, that is, the degrees of the vertices of the first arrival edge will be increased firstly and then the edges attached on these vertices are of high replaced probability even though the degrees are relative low in the final. Ideally, the random order is expected to avoid such phenomenon, but for DFS order or mixture order, the locality of arrival edges influences seriously the quality of weighted sampling. So, *DBS* introduces an extra buffer of size η, and updates the degrees after each η items read, which reduces the locality in some sense. The parameter of η is set by experience.

Algorithm 2 Degree Bias Sampling

Input:
Edge Sequence E_{seq};
Output:
Sample Graph G_s;
1: $V_s = \emptyset, E_s = \emptyset$;
2: **for** each $e = (u, v)$ in the first ρ edges of E_{seq} **do**
3: $E_s = E_s \bigcup (e, 0, 0, 0)$; //(edge, weight, random, key)
4: $V_s = V_s \bigcup (u, V_s[u].d + 1) \bigcup (v, V_s[v].d + 1)$;
5: **end for**
6: **for** each item $it \in E_s$ **do**
7: $it.weight = \frac{1}{min(V_s[item.edge.u].d, V_s[item.edge.v].d)}$;
8: $it.rand = random(0, 1)$;
9: $it.key = it.rand^{1/it.weight}$
10: **end for**
11: **repeat**
12: Read the next η edges into E' from E_{seq} at the begin of the position $\rho + 1 + i * \eta$ for $i = 0, 1, ...$
13: **for** each $e = (u, v)$ in the set of E' **do**
14: $V_s = V_s \bigcup (u, V_s[u].d + 1) \bigcup (v, V_s[v].d + 1)$;
15: **end for**
16: For each item of E_s, re-caculate its key ;
17: $T = \arg\min_{it \in E_s} \{it.key\}$;
18: **for** each edge $e = (u, v) \in E'$ **do**
19: $w = \frac{1}{min(V_s[u].d, V_s[v].d)}$;
20: $r = random(0, 1)$;
21: $key = r^{1/w}$;
22: **if** key > T.key **then**
23: Replace T by the item (e, w, r, key);
24: $T = \arg\min_{it \in E_s} \{it.key\}$;
25: **end if**
26: **end for**
27: **until** the end of E_{seq}
28: Build the sample graph G_s by E_s
29: **return** G_s;

4.4 Allocating Unsampled Vertex

In the second pass of scanning the graph, *SGL* allocates the unsampled edges and vertices according to the approximate partitioner of G_s. For the unsampled edge $e = (u, v)$, as shown from Step 15 to Step 24 of Algorithm 1, if both u and v exist in G_s, e is allocated by the partitions of u and v, whereas if there exits a vertex of edge that doesn't belong to G_s, *SGL* has to wait for allocating it until its locality is retrieved. Here, the locality of vertex is defined as following:

Definition 4. For graph $G = (V, E)$, the *locality* of vertex $u \in V$ is defined as the set of its adjacent vertices, that is $LOC(u) = \{v | v \in V, (u, v) \in E\}$.

An unsampled vertex is allocated until more than $s \cdot d$ percent of the adjacent vertices in its locality have been read and partitioned, where s is decided by Equation 8 with the bound of given error α, otherwise the vertices of locality should be cached. For a vertex of degree d, the likelihood that none of its d edges is sampled is $(1 - s)^d$ and the expectation of the number of unsampled vertex of degree d is $n_d(1 - s)^d$, where n_d is the number of vertices of degree d. The expectation of required space is $dn_d(1 - s)^d$. With the sum of expectation with respect to the degree $1 \sim d_{max}$, We have the average space complexity of locality cache is $O(\sum_{d=1}^{d_{max}} dn_d(1 - s)^d)$. Since $s < 1$, few of vertices are unsampled and the cache will be small. Moreover, the approximate partitioner needn't to be re-adjusted with unsampled vertex allocated, because it is considered to be qualified to allocating the vertex with the bound of error gain.

5. LOADING STREAM GRAPH

For loading the stream graph , it is infeasible visiting the stream graph twice, since the edges flow continuously into the loading system and usually unable to be stored in the disk of loading node. To this problem, *SGL* is modified here to fit into streaming graph loading.

5.1 Streaming Loading Algorithm

The algorithm of loading stream graph is shown in Algorithm 3. The element of vertex in Algorithm 3 has three items: the item *sampled* represents whether the vertex is selected in V_s or not; the item *partition* indicates the partition of the vertex; the item *AC* is the *Allocation Context* of vertex determining the partition of unsampled vertex. The algorithm mainly includes two phases as following.

Phase I: For each new arrival edge e^+, *DBS* is applied to select e^+ into the reservoir E_s. If e^+ substituted for e^- in E_s, and then it will be checked whether the representative graph should be re-partitioned to update the approximate partitioner or not, as shown from Step 8 to Step 12. if it required, G_s will be repartitioned after η edges read, as shown from Step 27 to 29. The details of re-partitioning will be discussed in Chapter 5.3. Otherwise, e^+ will be allocated and saved immediately into the corresponding storage node in terms of current approximate partitioner and its *AC* as shown from Step 14 to Step 25. Here, *AC* is the expand locality for determining the partition of unsampled vertex of e^+, which will be discussed further in Chapter 5.2.

Phase II: At the end of stream, the partition of unsampled vertex maybe different from the one of its allocation, so the following should be checked in parallel on each storage node.

i. If there are vertices in the storage node that were migrated into different partitions on updating the approximate partitioner, they must be removed from the current node, as shown by Step 35;

ii. For unsampled vertex, if there is the vertex in its *AC* whose partition is different from the one in the final approximate partitioner, it must be re-allocated, as shown from Step 37 to Step 40;

iii. After (i) and (ii), the edge with the partition of its unsampled vertex changed should be re-allocated, as shown from Step 43 to Step 50.

5.2 Allocation Context

Ideally, allocating the unsampled vertex is based on its locality defined by Definition 4, which is feasible for *SGLd* regardless of the search order. However, retrieving the locality under DFS or mixture order has to wait for the end of stream for *SGLs*. To this problem, the allocation context is proposed here based on the widely accepted observation in the field of *Random Walk*, which is a random walk started inside a good cluster will mostly stay inside the cluster [1]. Here, by a good cluster is meant a cluster with relatively fewer number of cross-edges in comparison with the one inside. The search of DFS order is generally considered as a kind of *Self-avoiding Random Walk(SARW)*. The difference between DFS and SARW is that DFS allows the search to trace back when all of neighbors are visited, whereas SARW will stop searching in such case. It means that the vertices in DFS traversal besides the adjacent vertices can make the contribution to the determination of partition in some senses. So, the locality of vertex is expanded to the connected vertices to the unsampled vertex in the traversal, which makes the vertex allocated as early as possible.

Definition 5. For a unsampled vertex u_0, where $u_0 \in e_i, e_i \in E_{seq}$, the *Allocation Context* of u_0, denoted as $AC(u_0)$, is defined as the subgraph (V_{u_0}, E_{u_0}), where V_{u_0} is the set of vertices with the connection to u_0 in the following traversal from u_0. Formally, $V_{u_0} = \{u | u \text{ is connected to } u_0, u \in e_j, e_j \in E_{seq}, j = i+1, ...\}$. E_{u_0} is the set of edges attached on V_{u_0}. $|V_{u_0}| = l$, l is the size of AC.

According to the observation, if u_0 belongs to the partition i, the other vertices in its AC would stay in the partition i with high probability. Here two factors of the vertex u in AC are considered to contributing to the allocation of u_0:

(1) *Influence* to u_0 measured by the distance from u to u_0 is the factor that if u is of less distance to u_0, it would stay in the same partition to u_0 with higher probability. For $u \in V_{u_0}$, the influence of u to u_0 is

$$influence(u, u_0) = \frac{1}{sp(u, u_0)} \tag{10}$$

where $sp(u, u_0)$ is the length of shortest path between u and u_0 in the graph $AC(u_0)$.

(2) *Attraction* to a partition measured by the proportion of its links to the partition is the factor that the more links to the partition is, the higher probability of traversing in the same partition is. For $u \in V_{u_0}$, the attraction of u to the partition i is

$$attraction(u, i) = \frac{d_u^i}{d_u} \tag{11}$$

where d_u^i is the number of links of the vertex u to the partition i, d_u is the degree of u.

The unsampled vertex u_0 is allocated by the following equation.

$$Partition(u_0) = \arg\max_{i \in [1...k]} \{w_i\} \tag{12}$$

$$w_i = \sum_{partition(u_j)=i \bigwedge u_j \in V_{u_0}} influence(u_j, u_0) attraction(u_j, i) \tag{13}$$

5.3 Adjusting Approximate Partitioner

When the edge $e^+ = (u^+, v^+)$ in E_{seq} is selected to substitute for the edge $e^- = (u^-, v^-)$ in E_s, the representative graph G_s should be re-partitioned in case of the gain of associated vertex inverse. The procedure of repartitioning need-n't start at the root of the binary tree of graph partitioning but at the the internal node $LCA(i, j)$.

For an non-boundary vertex of G_s with adding or removing incident edge, its sign of gain will not change and then it is unnecessary re-partitioning G_s. Whereas for an boundary vertex, its sign of gain will be inverse and G_s should be re-partitioned in the following cases.

i. If $\exists u \in \{u^-, v^-\} \wedge j \in [1...k]$, $Partition(u) = i$, $u \in \mathcal{B} \wedge \{u^-, v^-\} \setminus u \not\subseteq \mathcal{B} \wedge \delta_u^{ij} = 0$, where δ_u^{ij} is the gain of u between the partition of i and j before e^- removed, then the partition i and j should be adjusted.

ii. If $\{u^+, v^+\} \subseteq \mathcal{B}$, $Partition(u^+) = i$, $Partition(v^+) = j$, $i \neq j$, $\delta_{u^+}^{ij} = 0$ or $\delta_{v^+}^{ji} = 0$ before e^+ added, then the partition i and j should be adjusted.

As shown by the condition (i) and (ii) above, the likelihood of adjusting partition is related to the number of boundary vertex of zero gain. The probability that a vertex of degree d becomes the boundary vertex with one link at least to the different partitions is $1 - (1 - \frac{1}{k})^d$, and the probability that the boundary vertex is of zero gain is $\frac{1}{d}$. Thus the probability that G_P^s is adjusted when a edge on sample incident to the boundary vertex of degree d is $\frac{s}{d}(1 - (1 - \frac{1}{k})^d)p(d)$, where $p(d)$ is the probability that the vertex is of the degree d. A constant $c_1 < 1$, for simplicity, is introduced to substitute for $(1 - (1 - \frac{1}{k})^d)$. Moreover, we assume the graph follows a *power law* as most of big graph do in the real world, and then $p(d) = \frac{d^{-\beta}}{\zeta(\beta)}$, where $\zeta(\beta) = \sum_{1 \leq d \leq \infty} d^{-\beta}$, $\beta > 0$, which $\zeta(\beta)$ is *Riemann Zeta Function* for the purpose of normalized constant. Finally, the probability that G_P^s is adjusted with a edge sampled is $\sum_{1 \leq d \leq \infty} sc_1 \frac{d^{-\beta}}{d\zeta(\beta)} = sc_1 \frac{\zeta(\beta+1)}{\zeta(\beta)}$. In total, the expectation of the number of adjusting is $(1 - (1 - sc_1 \frac{\zeta(\beta+1)}{\zeta(\beta)})^\eta)m/\eta$.

Obviously, G_s should be re-partitioned in the case that the new vertex is added with the edge sample, while it maybe not in the case that the sample edge attaches the exists vertices in G_s, except that the condition (i) and (ii) satisfied. As a result, G_s will be re-partitioned frequently at the early phase of *SGLs* until most of vertices of the original graph are observed, because G_s in *SGLs* that allocates unsampled edges comes from the partial graph of the current stream under *BFS* or *DFS* search order, not the global graph as *SGL* does. Hence, the tip recommended here to reduce the times of re-partitioning at the early phase is that the size of E_s in *SGLs* in practice should be larger than the one in *SGL*. it is the better that the analysis on the size of E_s theoretically and the suggestion of size are given.

Moreover, if an edge wasn't selected by *DBS* in *SGLs*, it should be allocated immediately since the edge stream was never saved persistently at the loader. As the result, the partition of the vertex saved as unsampled in the storage node at early allocation should be re-examined at the end of stream

graph, because the vertex itself or the vertex in its AC maybe migrated in the following re-partitioning. The time complexity of the re-allocation in storage node is related to the number of unsampled vertex, that is, $O(\sum_{d=1}^{d_{max}} dn_d(1-s)^d)$.

6. EXPERIMENTAL EVALUATION

In this section we present our experimental findings. Specifically, Section 6.1 describes the experimental setup. Sections 6.2 presents our findings for synthetic and real-world graphs respectively.

6.1 Experimental Setting

The real-world graphs and synthetic graphs used in our experiments are shown in Table 2. Multiple edges, self loops, signs and weights were removed. All real-world graphs are publicly available on the Web[1]. In our synthetic experiments, The Barabasi-Albert model[2] is used to generate the synthetic graph of power-law degree distribution. Numerical simulations and analytic results indicate that this network evolves into a scale-invariant state with the probability that a node has d edges following a power law with an exponent 3. All algorithms have been implemented in C++, and all experiments were performed on a single machine with Intel i7-3770k cpu at 3.5GHz, and 32GB of main memory. To simulate the limitation of main memory, we implement the virtual memory system to swap the item of adjacent table. The multi-thread program is implemented to simulate the distributed loading with *SGLs*. The *SGLd* and *SGLs* adopted the KL algorithm to partition the sample graph in all experiments.

As our competitors we use state-of-the-art heuristics. Specifically, in our evaluation we consider the following heuristics from [13] and [16]

- **Balanced Parition(B)**: place v to the partition P_i with minimal size.
- **Hash Partitioning (Hash)**: place v to a partition chosen uniformly at random.
- **Deterministic Greedy (DG)**: place v to P_i that maximizes $|N(v) \bigcap P_i|$.
- **Linear Weighted Deterministic Greedy (LDG)**: place v to P_i that maximizes $|N(v) \bigcap P_i| \times (1 - \frac{|P_i|}{\frac{n}{k}})$.
- **Exponentially Weighted Deterministic Greedy (EDG)**: place v to P_i that maximizes $|N(v) \bigcap P_i| \times (1 - exp(|P_i| - \frac{n}{k}))$.
- **Triangles (Tri)**: place v to P_i that maximizes $t_{P_i(v)}$.
- **Linear Weighted Triangles (LTri)**: place v to P_i that maximizes $t_{P_i(v)} \times (1 - \frac{|P_i|}{\frac{n}{k}})$.
- **Exponentially Weighted Triangles (ETri)**: place v to P_i that maximizes $t_{P_i(v)} \times (1 - exp(|P_i| - \frac{n}{k}))$.
- **Non-Neighbors (NN)**: place v to P_i that minimizes $|P_i \\ N(v)|$
- **FENNEL(FNL)**: place v to P_i such that $\delta g(v, P_i) \geq \delta g(v, P_j)$.

[1]http://snap.stanford.edu/snap/

v is the newly arrived vertex. $N(v)$ denotes the set of neighbors of vertex v. For FENNEL, the setting of the parameters we use throughout our experiments is $\gamma = 1.5$, $\alpha = \sqrt{k}\frac{m}{n^{1.5}}$, and $\nu = 1.1$ as mentioned in [16].

We evaluate our algorithms by measuring two quantities from the resulting partitions. In particular, for a partition we use the measures of the fraction of edges cut ς and the normalized maximum load τ [16], defined as

$\varsigma = \frac{\#edges\ cut}{m}$, and

$\tau = \frac{\#vertices\ of\ maximum\ partition}{\frac{n}{k}}$.

6.2 Performance Evaluation

The experimental results reported in this section are organized as follows: we first report on experiments that show the ability of *SGL* to deliver the better overall performance in terms of execution time and edge cut with the smaller restriction of memory and close to optimal load balance when executed on real-world graphs. We then study how *SGL* performance is affected by changes in the input stream ordering, changes in the size of memory, changes in the number of partitions, and changes in the length of AC when executed on synthetic graph.

6.2.1 Performance Comparison on Real Graph

We measured the performance of *SGLd* and *SGLs* against other streaming partitioning algorithms on the set of real-world graphs. These experiments were run by partitioning the input graphs on a set of target partitions of $k = 4$ with the memory limitation in the range of $30\%, 50\%$ and 80%. The running times are reported in Table 2, Table 3 and Table 4. the fractions of edges cut are reported in Table 5 and Table 6. The normalized maximum loads are reported in Table 7 and Table 8.

It can be observed that **Balanced Partition** and **Hash Partition** are the fastest on average than existing streaming algorithms, and insensitive to the memory size. This excellent performance is achieved by their succinct partition measure, but the measures also bring the worst the fractions of edges cut. The values of ς almost are three times lager than the ones of **SGLd** and **SGLs**. The others ς of existing partitioning algorithms are better than **Balanced Partition** and **Hash Partition**, especially, the ς of **FNL** is close to **SGLd** and **SGLs**. As also reported in [16], **FNL** shown a better performance than the other existing ones. However, they are more sensitive to the memory size than **SGLd** and **SGLs** because of massive disk I/O. As shown from Table 2 to Table 4, **FNL** suffers the runtime increment of 240.2% and 148.5% from the range of memory size proportion 80% to 50% and 50% to30% respectively. On contrary, there is no memory swap for **SGLd** and **SGLs**, so they are faster with the smaller memory. The time increment of **SGLd** are 12.4% and 10.5% from 30% to 50%, and 50% to 80% respectively, **SGLs** are 7.3% and 6.1%. **SGLd** and **SGLs** outperform existing algorithms under the smaller memory limitation. For the memory limitation of 30%, the runtime of **SGLd** is 26.5% of **FNL**, **SGLs** is 52% of **FNL**. For the memory limitation of 50%, the runtime of **SGLd** is 74.1% of **FNL**, while **SGLs** is 138% of **FNL**. For larger memory, the probability of swapping is small for all the existing algorithm including**FNL**, while the larger approximate graph in memory is required for **SGL**, so existing algorithms outperform **SGL** in runtime. In the comparison with **SGLd** and **SGLs**, although **SGLd** read the whole graph twice, **S-**

GLd only compute the approximate partitioner once, while **SGLs** have to adjust the approximate partitioner for each reading. So, **SGLd** perform better than **SGLs**.

For the fractions of edges cut, all the existing algorithms are insensitive to the memory size. **FNL** achieves 0.165 on average and perform the best in the existing algorithms. As proved by Lemma 1, **SGLd** and **SGLs** are more sensitive to the memory size, that is, the upper-bound of $E(\tilde{\delta})$ will be decreased with the sample ratio increased, and the experimental results also proves it. As shown in Table 6, **SGLd** perform better than **FNL** for all memory configurations and takes the improvement of 2.4%, 6.7% and 9.1% respectively. **SGLs** outperform **FNL** only for 80% limitation and takes 9% improvement, while suffers the reduction of 4.8% for 30% limitation and 1.2% for 50% limitation. **SGLs** shows the worse fractions of edges cut than **SGLd** in all cases and **FNL** in the memory limitation of 30% and 50%. This is because the approximate partitioner in **SGLd** is built on the whole graph, which represent the characteristic of graph more accurately, however, the partitioner of **SGLs** modified continuously until the end of stream is worse representative to graph. For the normalized maximum loads, as depicted by Table 7 and Table 8, both **SGLd** and in all configurations outperform the competitors except for **Balanced Partition**. In comparison to **FNL**, **SGLd** achieves the average improvements of 5.6%, 5.4% and 5.4% for the memory setting of 30%, 50% and 80%. **SGLs** outperform **FNL** with the higher memory size and have 2.1% improvement at 80% memory limitation. But it suffers a little reduction of 0.9% for 30% memory and 0.2% for 50% memory. Although the edges cut and maximum load of **SGLs** are a little worse than **FNL** for the small memory, it shows much better efficiency of about 50% improvement.

6.2.2 *Performance Evaluation on Synthetic Graph*

We firstly present the experimental evaluation of runtime, edges cut and maximum load in comparison with **SGL** and **FNL** for each setting of memory limitation and edges orders. Here, we also provide the experiments of **Random Uniform Sampling (RUS)** beside **DBS**, which marked by '-eq' in the figure. The runtime comparisons are depicted in Figure 1. **SGLd** with the sampling of RUS and DBS perform well over Random, DFS and BFS orders, especially outperforms **FNL** greatly under the small memory. More accurately, the runtime of RUS is a little better than DBS, but very closely. **SGLs** performs worse than **SGLd**, but much better than **FNL** with the small memory limitation. Both **SGLd** and **SGLs** have little sensitivity to the order, but **FNL** is quite sensitive. As shown by Figure 1(c), **FNL** in BFS order performs the worst in comparison to the others, and shows much overhead of runtime. The comparisons of edges cut are depicted in Figure 2. **SGL** performs much better than **FNL** for Random and DFS and closely for BFS. **SGLd** is close to **SGLs** in all cases. **RUS** is worse than **DBS**, especially for Random order. The comparisons of maximum loads are shown by Figure 3. **SGLd** including **RUS** and **DBS** has very good load balancing properties. **SGLs** achieve the comparable performance only in BFS order. For the other orders, **SGLs** has the similar behaviors as **FNL**.

Secondly, we present the experimental evaluation with respective to the number of partitions k and input orders. Figure 4 shows the runtime when k ranges from 2 to 16 for each input order. We fix the memory limitation $\rho = 0.3$ and $|AC| = 50$. As we see, the factor of k has less effects on the runtime of **FNL** than **SGL**, since **SGL** applies optimal graph partitioning in memory to build the approximate partitioner. But the factor of input order has much more effects on **FNL** than **SGL**, which also shown in the experiments of Figure 1. As depicted by Figure 4(c), the runtime of **FNL** increases greatly for BFS order in comparison to the other orders, while the runtime of **SGL** changes slightly over three orders. Figure 5 gives the experimental result of edges cut, where the experimental setting is the same as the one of Figure 4.

Table 1: Datasets in the experiments

DataSet	Vertices	Edges	Mean	Var	Description
com-dblp.all.cmty	17578	12073	1.52	4	DBLP
com-youtube-all-cmty	17776	13505	12.1	1391	Youtube
as-733	7587	23240	1.43	1.437	733 daily instances
higgs-reply-network	38918	29612	6.13	686	Twitter
facebook	4035	85003	43.01	2362	Facebook
ca-hepph	12008	118445	1.53	64	Arxiv High Energy Physics
higgs-mention-network	116406	128024	19.74	2175	Twitter
email-enron	36114	181385	21.11	934	Email from Enron
ca-astroph	18772	198035	2.35	200	Arxiv Astro Physics
soc-sign-slashdot081106	77347	461004	10.08	1206	Slashdot Zoo
soc-sign-slashdot090216	81866	490047	12.03	1372	Slashdot Zoo
soc-sign-slashdot090221	82137	492351	12.07	1380	Slashdot Zoo
syn-v4k-e85k	4000	85099	39.76	2213	Synthetic

Table 2: Runtime with $\rho = m \times 30\%,\ k = 4$(unit:sec)

Dataset	Hash	B	DG	LDG	EDG	Tri	LTri	ETri	NN	FNL	SGLd	SGLs
com-youtube-all-cmty	4.512	3.923	506	552	533	512	534	518	520	518	397	425
com-dblp.all.cmty	4.418	4.598	518	502	453	439	439	440	435	435	409	441
as-733	8.82	8.739	125	125	125	139	137	138	125	123	112	169
facebook	27	27	209	207	204	582	603	600	222	227	74	142
higgs-reply-network	30	31	2324	2290	2550	2252	2438	2370	2272	2361	1140	1230
ca-hepph	62	63	544	542	542	1543	1310	1290	501	491	251	489
ca-astroph	70	70	1143	1168	1209	1558	1472	1528	1140	1145	558	1113
higgs-mention-network	233	232	16507	16715	16716	16704	16811	16343	16541	16581	4520	8047
email-enron	421	417	3179	3104	3110	3934	3884	3988	3031	3067	832	1647
soc-sign-slashdot081106	1420	1426	9023	10348	10395	13252	13377	13255	10841	10520	2465	5634
soc-sign-slashdot090216	1522	1616	12053	12364	13016	13471	13811	13719	12319	12636	2420	5678
soc-sign-slashdot090221	1347	1308	11069	11187	11093	12379	12573	11977	10923	10912	2484	5704
avg.	429.1	433.9	4766.7	4925.3	4995.5	5563.8	5615.8	5513.8	4905.8	4918	1305.2	2559.1

Table 3: Runtime with $\rho = m \times 50\%,\ k = 4$(unit:sec)

Dataset	Hash	B	DG	LDG	EDG	Tri	LTri	ETri	NN	FNL	SGLd	SGLs
com-youtube-all-cmty	1.9	1.7	185	180	184	185	184	187	180	183	226	479
com-dblp.all.cmty	2.096	1.783	165	166	164	171	173	170	163	163	443	451
as-733	2.994	2.686	47	47	47	53	52	51	46	47	103	198
facebook	8.148	8.207	103	106	102	257	243	243	103	102	77	154
higgs-reply-network	9.01	8.463	878	877	925	933	933	903	881	897	956	1358
ca-hepph	17	17	141	141	140	502	482	472	140	141	269	549
ca-astroph	75	74	318	327	326	478	453	473	333	390	852	1769
higgs-mention-network	78	78	7232	7257	7244	7280	7421	7488	7267	7447	5307	8721
email-enron	112	115	1183	1180	1173	1408	1406	1402	1127	1135	1018	1769
soc-sign-slashdot081106	393	390	3920	4166	4180	5142	5085	5107	4116	4122	2676	5798
soc-sign-slashdot090216	443	449	4930	4318	4315	4760	4684	4680	4428	4584	2678	5799
soc-sign-slashdot090221	404	406	4491	4662	4572	4796	4773	4746	4592	4537	2999	5891
avg.	128.8	129.3	1966.1	1952.3	1947.7	2163.8	2157.4	2160.2	1948	1979	1467	2744.7

Table 4: Runtime with $\rho = m \times 80\%,\ k = 4$(unit:sec)

Dataset	Hash	B	DG	LDG	EDG	Tri	LTri	EDTri	NN	FNL	SGLd	SGLs
com-youtube-all-cmty	1.975	1.698	103	104	104	109	108	106	100	112	214	497
com-dblp.all.cmty	2.166	1.854	99	99	98	100	99	99	95	94	226	501
as-733	2.993	2.667	19	20	19	26	25	24	19	20	94	207
facebook	8.187	8.542	55	58	55	215	201	198	53	53	110	197
higgs-reply-network	8.323	8.341	480	478	476	486	488	482	476	475	915	1465
ca-hepph	17	17	56	57	57	393	405	401	58	61	325	784
ca-astroph	70	71	121	122	122	259	242	245	124	123	971	1889
higgs-mention-network	77	77	3240	3240	3221	3248	3249	3251	3281	3212	5741	9245
email-enron	112	112	62	55	55	311	307	307	56	56	1755	1997
soc-sign-slashdot081106	384	370	819	812	823	1730	1683	1744	835	818	3001	6021
soc-sign-slashdot090216	407	409	985	985	986	1425	1422	1425	981	983	3025	6045
soc-sign-slashdot090221	410	411	974	972	988	1415	1413	1427	958	974	3078	6105
avg.	125.1	124.2	584.4	583.5	583.7	809.8	803.5	809.1	586.3	581.8	1621.3	2912.75

Table 5: ς with $k = 4$

Dataset	Hash	B	DG	LDG	EDG	Tri	LTri	ETri	NN	FNL
com-youtube-all-cmty	0.006	0.007	0.001	0.002	0.002	0.005	0.005	0.005	0.002	0.001
com-dblp.all.cmty	0.749	0.886	0.048	0.213	0.219	0.663	0.666	0.666	0.186	0.054
as-733	0.665	0.66	0.029	0.375	0.385	0.224	0.386	0.389	0.408	0.377
facebook	0.751	0.754	0.023	0.131	0.266	0.051	0.148	0.185	0.305	0.071
higgs-reply-network	0.742	0.85	0.048	0.209	0.231	0.651	0.654	0.654	0.211	0.057
ca-hepph	0.75	0.763	0.003	0.168	0.169	0.107	0.188	0.201	0.399	0.125
ca-astroph	0.969	0.975	0.003	0.353	0.373	0.065	0.344	0.345	0.415	0.218
higgs-mention-network	0.717	0.751	0.098	0.241	0.287	0.638	0.636	0.636	0.275	0.112
email-enron	0.75	0.768	0.001	0.268	0.269	0.084	0.258	0.26	0.469	0.177
soc-sign-slashdot081106	0.751	0.754	0.008	0.359	0.348	0.26	0.347	0.33	0.551	0.248
soc-sign-slashdot090216	0.751	0.754	0.007	0.406	0.342	0.667	0.665	0.665	0.555	0.271
soc-sign-slashdot090221	0.752	0.754	0.006	0.406	0.344	0.667	0.664	0.664	0.557	0.271
avg.	0.696	0.723	0.023	0.261	0.270	0.340	0.413	0.417	0.361	0.165

Table 6: SGLd and SGLs - ς with $k = 4$, $\rho = 0.3, 0.5, 0.8$

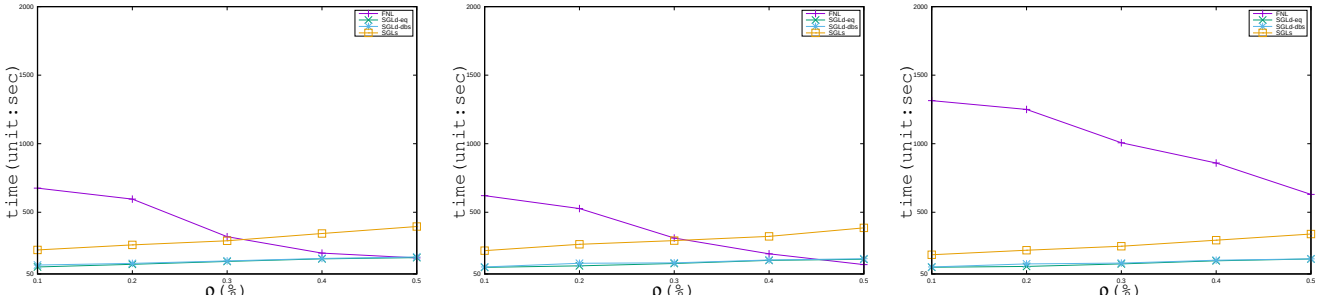
Dataset	SGLd-0.3	SGLd-0.5	SGLd-0.8	SGLs-0.3	SGLs-0.5	SGLs-0.8
com-youtube-all-cmty	0.003	0.001	0.001	0.003	0.002	0.001
com-dblp.all.cmty	0.051	0.048	0.04	0.05	0.048	0.039
as-733	0.35	0.341	0.321	0.389	0.347	0.312
facebook	0.069	0.06	0.047	0.149	0.108	0.068
higgs-reply-network	0.061	0.055	0.043	0.064	0.057	0.051
ca-hepph	0.142	0.123	0.112	0.134	0.124	0.12
ca-astroph	0.2	0.194	0.154	0.21	0.197	0.181
higgs-mention-network	0.112	0.11	0.101	0.121	0.19	0.136
email-enron	0.167	0.16	0.154	0.174	0.165	0.16
soc-sign-slashdot081106	0.241	0.238	0.21	0.244	0.24	0.215
soc-sign-slashdot090216	0.267	0.26	0.252	0.269	0.261	0.258
soc-sign-slashdot090221	0.264	0.259	0.254	0.268	0.263	0.254
avg	0.161	0.154	0.141	0.173	0.167	0.15

Table 7: τ with $k = 4$

Dataset	Hash	B	DG	LDG	EDG	Tri	LTri	ETri	NN	FNL
com-youtube-all-cmty	1.027	1.001	1.827	1.376	1.491	1.344	1.346	1.344	1.333	1.001
com-dblp.all.cmty	1.022	1.001	1.445	1.34	1.373	1.345	1.345	1.345	1.315	1.001
as-733	1.025	1	3.96	1.652	1.679	2.533	1.837	1.848	2.635	1.101
facebook	1.025	1.001	2.997	1.346	1.361	2.832	1.385	1.392	2.792	1.081
higgs-reply-network	1.012	1.001	1.562	1.34	1.412	1.344	1.344	1.344	1.444	1.001
ca-hepph	1.029	1.001	3.746	1.428	1.445	2.435	1.619	1.62	2.395	1.007
ca-astroph	1.028	1.001	3.827	1.371	1.407	3.112	1.506	1.544	3.141	1.101
higgs-mention-network	1.006	1.001	1.992	1.338	1.492	1.343	1.343	1.343	1.313	1.001
email-enron	1.014	1.001	3.771	1.617	1.614	2.711	1.807	1.809	2.754	1.101
soc-sign-slashdot081106	1.007	1.001	3.982	1.616	1.639	1.973	1.785	1.781	1.985	1.101
soc-sign-slashdot090216	1.008	1.001	3.986	1.12	1.634	1.347	1.356	1.355	1.357	1.1
soc-sign-slashdot090221	1.008	1	3.986	1.127	1.63	1.347	1.351	1.35	1.357	1.101
avg.	1.018	1.001	3.091	1.39	1.515	1.973	1.502	1.507	1.986	1.059

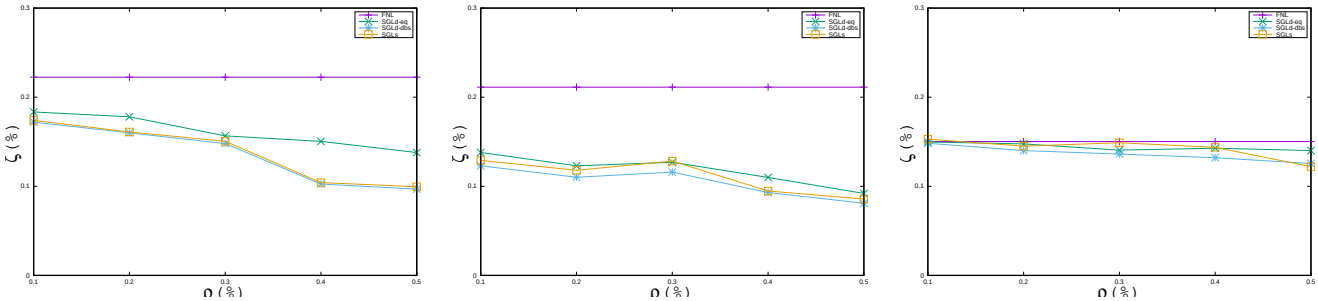
Table 8: SGLd and SGLs - τ with $k = 4$, $\rho = 0.3, 0.5, 0.8$

Dataset	SGLd-0.3	SGLd-0.5	SGLd-0.8	SGLs-0.3	SGLs-0.5	SGLs-0.8
com-youtube-all-cmty	1	1.001	1	1.092	1.072	1.033
com-dblp.all.cmty	1.001	1.001	1.001	1.088	1.064	1.023
as-733	1	1	1.001	1.091	1.078	1.071
facebook	1	1.001	1.001	1.054	1.05	1.04
higgs-reply-network	1.001	1	1	1.026	1.02	1.024
ca-hepph	1.001	1.001	1.001	1.021	1.018	1.017
ca-astroph	1	1	1.001	1.033	1.031	1.019
higgs-mention-network	1	1	1	1.066	1.064	1.018
email-enron	1	1.001	1	1.09	1.087	1.015
soc-sign-slashdot081106	1	1.001	1.001	1.08	1.081	1.065
soc-sign-slashdot090216	1.001	1.001	1	1.09	1.089	1.06
soc-sign-slashdot090221	1	1.001	1.001	1.094	1.091	1.06
avg	1	1.001	1.001	1.069	1.062	1.037



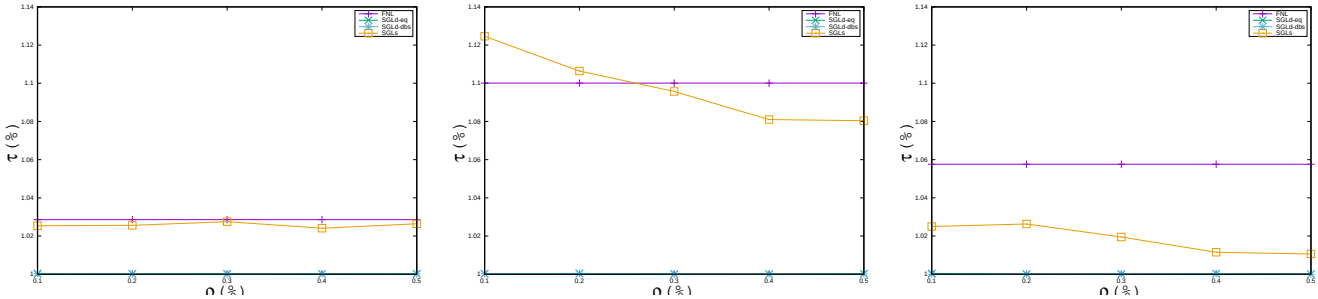
(a) RANDOM (b) DFS (c) BFS

Figure 1: Runtime VS. ρ on syn-v4k-e85k Dataset, $k = 4$



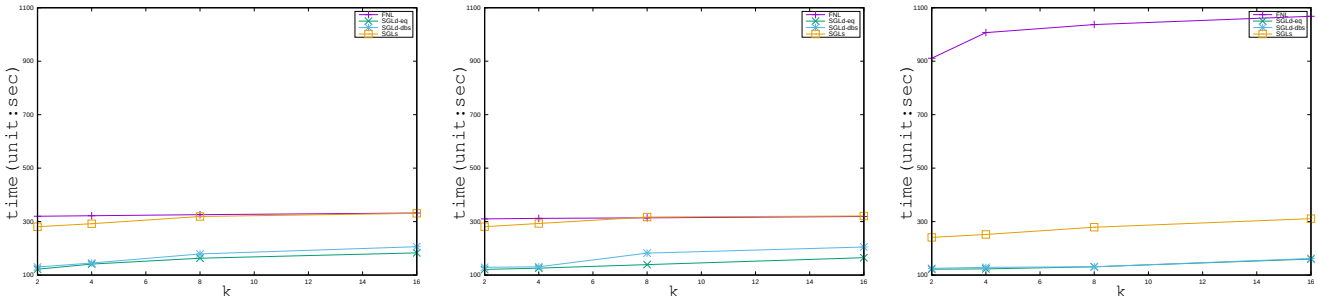
(a) RANDOM (b) DFS (c) BFS

Figure 2: ς VS. ρ on syn-v4k-e85k Dataset, $k = 4$



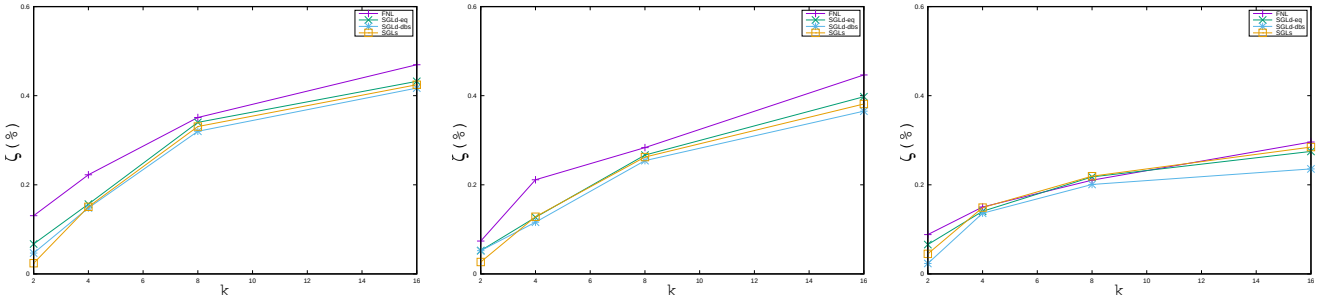
(a) RANDOM (b) DFS (c) BFS

Figure 3: τ VS. ρ on syn-v4k-e85k Dataset, $k = 4$



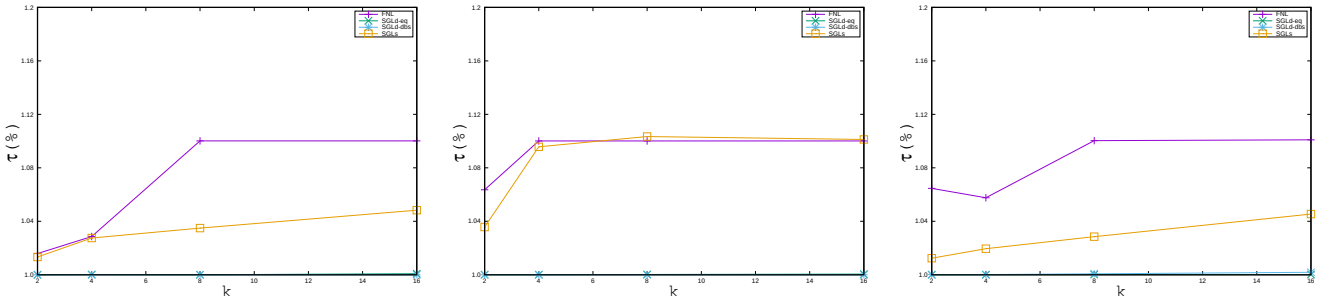
(a) RANDOM (b) DFS (c) BFS

Figure 4: Time VS. k on syn-v4k-e85k Dataset, $\rho = 0.3$



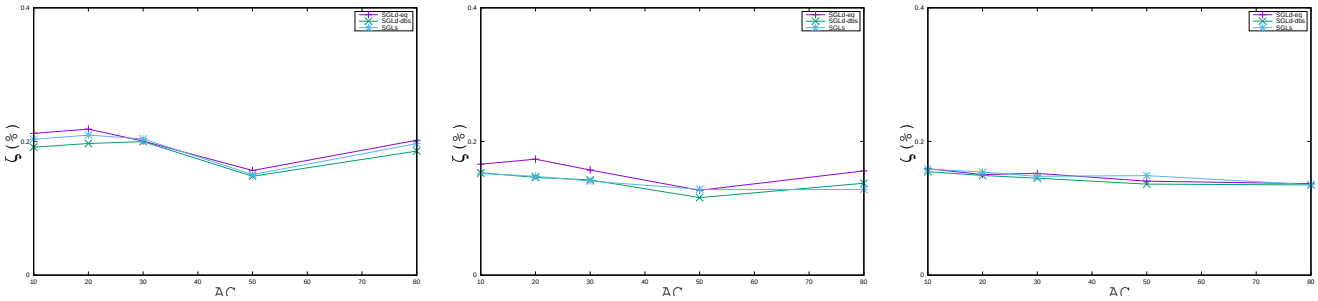
(a) RANDOM (b) DFS (c) BFS

Figure 5: ς VS. k on syn-v4k-e85k Dataset, $\rho = 0.3$



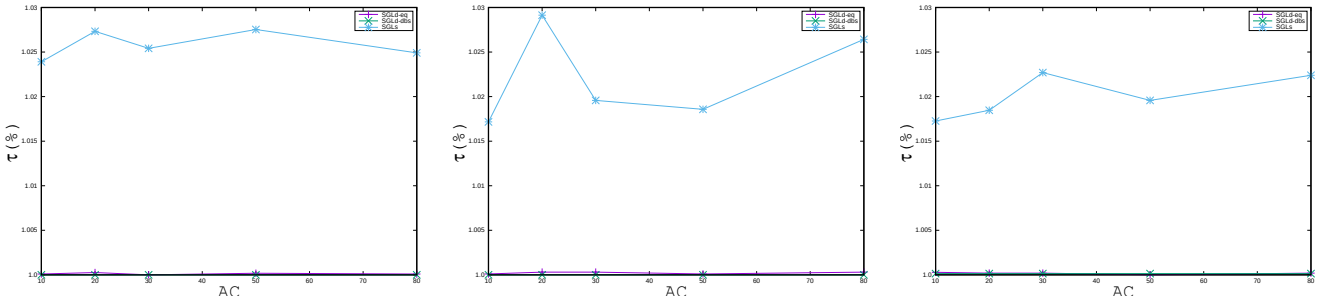
(a) RANDOM (b) DFS (c) BFS

Figure 6: τ VS. k on syn-v4k-e85k Dataset, $\rho = 0.3$



(a) RANDOM (b) DFS (c) BFS

Figure 7: ς VS. AC on syn-v4k-e85k Dataset, $\rho = 0.3$, $k = 4$



(a) RANDOM (b) DFS (c) BFS

Figure 8: τ VS. AC on syn-v4k-e85k Dataset, $\rho = 0.3$, $k = 4$

7. CONCLUSIONS

//We have demonstrated that simple, one-pass streaming graph partitioning heuristics can dramatically improve the edge-cut in distributed graphs.

8. ACKNOWLEDGMENTS

9. REFERENCES

[1] C. C. Aggarwal, editor. *Social Network Data Analytics*. Springer, 2011.

[2] A.-L. Barabasi and R. Albert. Emergence of scaling in random networks. *Science*, 286:509–512, 1999.

[3] R. Battiti and A. A. Bertossi. Greedy, prohibition, and reactive heuristics for graph partitioning. *IEEE TRANSACTIONS ON COMPUTERS*, 48:361–385, 1999.

[4] P. S. Efraimidis and P. G. Skpirakis. Weighted random sampling with a reservoir. *Information Processing Letters*, 97:181–185, 2006.

[5] C. Fiduccia and R. Mattheyses. A linear time heuristics for improving network partitions. In *19th ACM/IEEE Design Automation Conf.*, pages 175–181, 1982.

[6] I.S.Dhillon, Y.Guan, and B.Kulis. Weighted graph cuts without eigen-vectors: A multilevel approach. *IEEE Trans. Pattern Anal. Mach. Intell.*, 29(11):1944–1957, 2007.

[7] D. R. Karger. Random sampling in graph optimization problems. Technical Report 95-1541, Stanford University, January 1995.

[8] B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49:0–0, 1970.

[9] M. Laguna, T.A.Feo, and H.C.Elrod. A greedy randomized adaptive search procedure for the two-partition problem. *Operations Research*, 42:677–687, 1994.

[10] K. Lang and S. Rao. A flow-based method for improving the expansion or conductance of graph cuts. *Lecture notes in computer science*, 0:325–337, 2004.

[11] J. Leskovec and C. Faloutsos. Sampling from large graphs. In *KDD*, pages 631–636, 2006.

[12] U. V. Luxburg. A tutorial on spectral clustering. *Statistics and Computing*, 17(4):395–416, 2007.

[13] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. Technical Report MSR-TR-2011-121, Microsoft Research Institution, November 2011.

[14] S. Teng. Coarsening, sampling, and smoothing: Elements of the multilevel method. *Algorithms for Parallel Processing*, 105:247–276, 1999.

[15] Y. Till, editor. *Sampling Algorithms*. Springer, 2006 edition, 2006.

[16] C. E. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. Technical Report MSR-TR-2012-113, Microsoft Research, November 2012.

Algorithm 3 Sample-based Stream Graph Loading (SGLs)

Input:

E_{seq}: Edge Sequence;
k: The Number of Storage nodes;
ρ: Limitation of Edges in Memory;

Output:

Graph Partitions $G_P^1, G_P^2, ..., G_P^k$ on k storage-nodes;

Read the first ρ edges of E_{seq} as an initial graph sample G_s;
Apply gain-based partitioning algorithm on G_s to obtain k approximate partitioner $G_P^s = \{G_1^s, ..., G_k^s\}$;
$V_P^i = \emptyset$, $E_P^i = \emptyset$, $i = 1...k$;
repeat
  Apply *DBS* to sample the next η edges E' and let E^+ be the set of selected edge and E^- be the set of substituted edges;
  $isRepartition = false$;
  for each edge $e^+ = (u, v) \in E'$ **do**
    if $e^+ \in E^+$ **then**
      Let $e^- \in E^-$ be the edge substitued by e^+;
      $E_s = E_s \bigcup \{e^+\} / \{e^-\}$; $V_s = V_s \bigcup \{u, v\}$;
      $u.sampled = true$; $v.sampled = true$;
      Check if G_s should be re-partitioned; if yes, set $isRepartition = true$;
    else
      for each $u' \in \{u, v\}$ **do**
        if $\exists i \in [1...k], u' \in V_s^i$ **then**
          $u'.partition = i$;
        else
          Set $u'.partition = j$ by Eq. 12 and $u'.AC$;
          $u'.sampled = false$;
          $V_s^j = V_s^j \bigcup \{u'\}$; $V_P^j = V_P^j \bigcup \{u'\}$;
        end if
      end for
      $E_P^{u.partition} = E_P^{u.partition} \bigcup \{e^+\}$;
      $E_P^{v.partition} = E_P^{v.partition} \bigcup \{e^+\}$;
    end if
  end for
  if $isRepartition == true$ **then**
    Re-partition G_s and update the approximate partitioner G_P^s;
  end if
until the end of E_{seq}
//parallel implementation on each storage node
for each storage-site $i \in [1...k]$ **do**
  for each vertex $v \in V_P^i$ **do**
    if $v \notin V_s^i$ **then**
      $V_P^i = V_P^i / v$;
    else
      if $!v.sampled$ and $\exists v' \in v.AC, v'' \in V_s$, $v' == v'' \wedge v'.partition \neq v''.partition$ **then**
        Re-allocate $v.partition = j$ by Equation 12;
        $V_P^i = V_P^i / v$; $V_P^j = V_P^j \bigcup \{v\}$
      end if
    end if
  end for
  for each edge $e = (u, v) \in E_P^i$ **do**
    if $\exists u' \in \{u, v\}, u' \notin V_P^i$ **then**
      $E_P^i = E_P^i / \{e\}$;
      $E_P^{u.partition} = E_P^{u.partition} \bigcup \{e\}$;
      $E_P^{v.partition} = E_P^{v.partition} \bigcup \{e\}$;
    end if
  end for
  $V_P^i = V_P^i \bigcup V_S^i$; $E_P^i = E_P^i \bigcup E_S^i$;
end for