2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing

AKIN : A Streaming Graph Partitioning Algorithm for Distributed Graph Storage Systems

Wei Zhang
Department of Computer Science
Texas Tech University
Lubbock, TX, 79409, USA
Email: X-Spirit.zhang@ttu.edu

Yong Chen
Department of Computer Science
Texas Tech University
Lubbock, TX, 79409, USA
Email: yong.chen@ttu.edu

Dong Dai
Department of Computer Science
Texas Tech University
Lubbock, TX, 79409, USA
Email: dong.dai@ttu.edu

***Abstract*—Many graph-related applications face the challenge of managing excessive and ever-growing graph data in a distributed environment. Therefore, it is necessary to consider a graph partitioning algorithm to distribute graph data onto multiple machines as the data comes in. Balancing data distribution and minimizing edge-cut ratio are two basic pursuits of the graph partitioning problem. While achieving balanced partitions for streaming graphs is easy, existing graph partitioning algorithms either fail to work on streaming workloads, or leave edge-cut ratio to be further improved. Our research aims to provide a better solution that fits the need of streaming graph partitioning in a distributed system, which further reduces the edge-cut ratio while maintaining rough balance among all partitions. We exploit the similarity measure on the degree of vertices to gather structural-related vertices in the same partition as much as possible, this reduces the edge-cut ratio even further as compared to the state-of-the-art streaming graph partitioning algorithm - FENNEL. Our evaluation shows that our streaming graph partitioning algorithm is able to achieve better partitioning quality in terms of edge-cut ratio (up to 20% reduction as compared to FENNEL) while maintaining decent balance between all partitions, and such improvement applies to various real-life graphs.**

***Keywords*-Graph Partitioning; Graph Database; Distributed System; Graph Storage**

I. Introduction

Due to the excessive and ever-growing size of graph data, many graph-related applications have to take the advantage of distributed storage systems while facing the challenge of data management in a distributed setting. Distributed graph databases [1], [2], distributed graph-modeled metadata managing system [3], [4] are typical examples of these applications. The size of graphs in these applications are not only large, but also growing over time. Thus, it is necessary to consider a graph partitioning algorithm to distribute the graph data onto multiple machines with the data coming into the distributed system, in other words, the placement of incoming graph data needs to be determined instantaneously.

As one of the most prominent NP-hard problems, graph partitioning has been well studied [5], [6]. Many graph partitioning algorithms were proposed to address the graph partitioning problem. However, not all these algorithms can be applied to dynamic graphs, which are always changing over time. Offline graph partitioning algorithms, such as METIS [7], Chaco [8], and SBV-cut [9], can evaluate the structural features of the entire graph to achieve minimized edge-cut ratio and balanced partition size. However, since they need to load the entirety of a static graph into memory and conduct graph partitioning with multiple iterations of complicated computations, due to the limited memory space and heavy computational workloads, these algorithms are incapable of partitioning dynamic graphs on the fly.

Hence, some online graph partitioning algorithms were proposed recently to address the issue of partitioning such dynamic graphs. Some of them, such as Prefer Big [10] and HoVerCut [11], usually require a buffer for holding a sufficient amount of data, which represent vertices and edges that arrive at the system consecutively in a specified time interval. When the buffer is filled with enough data, these algorithms will make a decision about which partition the buffered data should be sent to. Such a decision making process is done based on the local topological structure of the buffered data. After that, the buffered data will be flushed to the partition, which is recognized as the best choice by the algorithms. However, such partitioning algorithms cannot partition the graph in a timely manner, so they cannot be used in those time-sensitive graph storage systems.

Some other online graph partitioning algorithms work in a purely streaming fashion, in which the partition decision is made immediately as the data of vertices and edges arrives. The most commonly used one is Deterministic Hashing [10]. It only considers the hash value of the vertices and edges and totally ignores the structural features of the incoming data, so it absolutely achieves partition balance but accomplishes nothing in minimizing the cut-size (the number of cross-partition links). Some recent streaming graph partitioning algorithms, such as LDG and Fennel [10], [12], attempt to take structural features into account while maintaining the balance of the partitions. In these algorithms, for each single vertex, the data placement decision is made rapidly based on the number of its edges or its neighbors in each partition. However, these online graph partitioning algorithms have their limitations in minimizing the cut-size due to their simplicity in reflecting the connectivity between a set of closely connected vertices. We believe that such simplicity leaves space for further edge-cut reduction and locality improvement in a distributed graph storage system.

978-1-5386-5815-4/18/$31.00 ©2018 IEEE
DOI 10.1109/CCGRID.2018.00033

To partition graphs on the fly with better data locality on each partition, we propose AKIN - a streaming graph partitioning algorithm. Our algorithm utilizes similarity of vertices to achieve further edge-cut reduction and better data locality. Our major contributions are as follows:

- We propose a new streaming graph partitioning algorithm for distributed graph storage.
- We design and implement our graph partitioning algorithm that utilizes the similarity index of vertices on every incoming edge to further reduce the edge-cut ratio.
- We conduct comprehensive evaluations to validate the proposed graph partitioning algorithm and to verify its effectiveness on various real-life graphs.

The rest of this paper is organized as follows. Section II introduces the background of the graph partitioning problem. After briefly talking about our motivation and designing principles, in Section IV, we introduce the design of proposed AKIN graph partitioning algorithm. Section V provides analysis of critical procedures of our AKIN algorithm. Section VI reports the evaluation results with different criteria on different graph datasets and compares the results against several existing algorithms. Section VII reviews several existing graph partitioning algorithms and compares them with the newly proposed AKIN algorithm. In Section VIII we conclude this study and discuss future work.

II. Background

The formal definition of the k-way partitioning problem can be found in [7] as follows:

> *Given a graph $\mathcal{G} = (V, E)$, where V denotes set of vertices and E denotes set of edges, partition V into k subsets, V_1 ,V_2 ,...,V_k, such that $V_i \cap V_j = \varnothing$ for $i \neq j$, $|V_i| = n/k$, and $\bigcup_i V_i = V$, and the number of edges of E whose incident vertices belong to different subsets is minimized.*

In another words, the objective of the k-way graph partitioning is to split the graph into multiple partitions $(V_1, V_2, ..., V_n)$, so that each partition is equally the same size, and the cut-size (the number of connections spanning over different partitions) can be minimized.

To measure the balance of the partition size, one can quantitively measure the standard deviation of *partition size ratio* of each partition. The partition size ratio can be easily obtained by calculating the quotient of the actual size of a partition and the average size of all partitions. Obviously, a smaller standard deviation of partition size ratio means the size of all partitions is more balanced. Thus, a minimal standard deviation of partition size ratio is another goal the graph partitioning algorithms should achieve.

To measure the cut-size, given the number of all edges across partitions $\{V_1, V_2, ..., V_n\}$ defined as $C(V_1, V_2, ..., V_n)$, we denote *edge-cut ratio* as follows:

$$\lambda_i = \frac{|C(V_1, V_2, ..., V_n)|}{|E|}$$

Here, $|E|$ represents the total number of edges in the entire graph, and graph partitioning algorithms aim at finding the minimal λ_i.

It has been well known that the k-way partitioning problem is NP-hard [5], [6], hence a huge number of graph partitioning algorithms have been proposed and examined in the past [13]. We roughly categorize them into two paradigms: the *offline* graph partitioning algorithms and their *online* counterparts. Offline graph partitioning algorithms (e.g., METIS [7] and Chaco [8]) requires the the data of an entire graph to be fully loaded into memory before actually performing graph partitioning operations. While the global view of the whole graph is known, these algorithms usually generate considerably superior partitioning quality in terms of both partition balance and cut-size. However, these algorithms need to load the entire graph into memory, and the limitation of memory space in modern computers will become a hindrance to do so for excessively large graphs, especially when the graph is dynamically growing. In addition, the multiple levels of computation during the partitioning process will incur intensive computational overhead, which makes these algorithms impossible to fit for streaming graph partitioning scenarios. Although applying such an algorithm on multiple subgraphs collected periodically can also help in accomplishing the goal of graph partitioning, the partitioning result still fails to reflect the instant changes of the graph. Besides, for a distributed system where vertex accessing is frequent, a distributed hash table needs to be established based on the partitioning result for locating each vertex; this makes the vertex locating procedure inefficient.

To tackle these challenges, online graph partitioning algorithms were proposed recently. The simplest online graph partitioning algorithm is to apply a hashing function on the graph vertices to determine their target partitions. Such an approach normally obtains good balance in partition size, but does not consider the structural nature of the graph at all. Some online graph partitioning schemes, such as Prefer Big [10] and HoVerCut [11], were proposed to improve the partitioning quality by considering the local structural features on the subgraph buffered in a limited portion of memory. Because the local computation takes time and the data buffered in memory needs to be flushed to the actual storage after the computation, the data placement cannot be addressed in a timely manner.

Recently, some other online graph partitioning algorithms, such as LDG [10] and Fennel [12], attempt to achieve the graph partitioning in a streaming fashion as the data continuously arrives. But, they only consider very simple graph features, such as the number of outgoing edges and the number of neighbors. Due to the simplicity of these metrics, these algorithms are able to determine data placement instantaneously. However, such simplicity also becomes an obstacle for these algorithms to explore more structural features of the graph and, hence, makes these algorithms incapable of generating decent partitioning result in terms of edge-cut ratio.

III. Motivation and Requirements

A. An Inspiration From Community Structure

As the saying goes, *"Birds of a feather flock together"*. Graph theory suggests that most real-world graphs tend to show certain *community structure* on some of the vertices [14]. In other words, there are always some groups of highly connected vertices in the graph. These highly connected vertices naturally reveal some clustering features that can be exploited to perform graph partitioning. Also, existing research has revealed that the connectivity among similar vertices tend to be stronger than what it is among those dissimilar vertices [15]. Thus, gathering similar vertices into the same partition can help increase the spatial data locality in distributed graph storage systems and, therefore, reduce the edge-cut ratio.

To determine similarity between vertices, we need a metric that requires a minimum amount of information about the graph, and the calculation of such a metric should be simple so that the computational overhead won't be very high. As a common metric for similarity, we use the Jaccard index [16] to measure the similarity between vertices because it is simple and straightforward.

B. AKIN Algorithm Designing Principles

Based on the discussions in previous sections, we summarize four key requirements for streaming graph partitioning in a distributed system:

1) **Easy to locate a vertex.** In a distributed graph storage, a lookup on vertex requires the address of the vertex, a graph traversal operation start from a vertex, even a simple lookup on an edge requires the address on its source vertex. In general, vertices are frequently accessed, and even more frequently than edges. Therefore, it is a must that every vertex can be located and accessed rapidly.
2) **Balanced partition size.** Since the size of all partitions determines the workload on each of them, the size of all partitions should be approximately equal. As stated above, vertices are more frequently accessed, so the number of vertices in one partition highly decides the workload on that partition. Thus, to balance partition size is to balance the number of vertices in all partitions.
3) **To determine data placement instantly.** For arriving data stream of vertices and edges, the algorithm should address the data placement for continuously arriving vertices and edges in a timely manner, so that the streamed graph can be partitioned on the fly while it is growing.
4) **To minimize edge-cut ratio.** In a distributed graph storage, graph traversal operations are very frequent. However, the more cross-partition edges we have, the more network overhead there would be. Thus, to reduce the excessive network communication between different partitions, the number of cross-partition edges should be minimized and, hence, the data locality should be improved.

IV. AKIN Algorithm Design

In this study, we propose a novel similarity-based streaming graph partitioning algorithm - AKIN that fulfills these requirements summarized above. Our AKIN algorithm works in a streaming fashion. It takes the stream of vertices and the stream of edges in a graph as input, and determines data placement on each vertex and each edge it receives. We generally introduce how our algorithm deals with vertex stream and edge stream, respectively.

A. On Vertex Stream

As stated in our partitioning requirements, the vertices should be easily located after partitioning. Thus, each data placement decision on vertices should be responsible for the simplicity of vertex locating. To achieve this, we apply the deterministic hashing function on each arriving vertex in the vertex stream. Specifically, for each arriving vertex v, our AKIN algorithm will apply hashing function h against the ID of the arriving vertex and determine the initial partition $\mathcal{P}_i$ of the vertex according to the result $i = h(v)$ (as shown in **Algorithm 1**).

Algorithm 1 Determine partition $\mathcal{P}_i$ for arriving vertex v

Input: arriving vertex v, number of all partitions k

```
1: i ← h(v)
2: if i ∈ {0, ..., k − 1} then
3:     if not_exists(v, P_i) then
4:         assign v to P_i
5:     end if
6: end if
```

Here, function $not\_exists(v, \mathcal{P}_i)$ determines whether vertex v (or a reference of it) does not exist in partition $\mathcal{P}_i$. We define the partition determined by the hashing of the vertex to be the **base partition**.

B. On Edge Stream

For each edge in edge stream, our AKIN algorithm performs the following steps to determine where the edge should be assigned to and whether vertex migration should happen:

First, AKIN will evaluate the default partition where the edge should be stored. Without further evaluating the similarity of the two vertices on the edge, it is ideal to assign the edge alongside its source vertex, so that the data locality between the source vertex and its out-going edge is ensured. Such data locality may facilitate the graph traversal operations.

Secondly, AKIN will evaluate the similarity between two vertices. To achieve this, AKIN will maintain a **fixed-length neighbor list** for each vertex on its base partition, as shown in Figure 1. Such a neighbor list is sorted by the degree of those neighboring vertices. Each item in this neighbor list consists of the ID of the neighbor vertex and the ID of its current hosting partition, along with the degree of this vertex. The neighbor vertices with smallest degree will be eventually replaced by the ones with higher degrees.

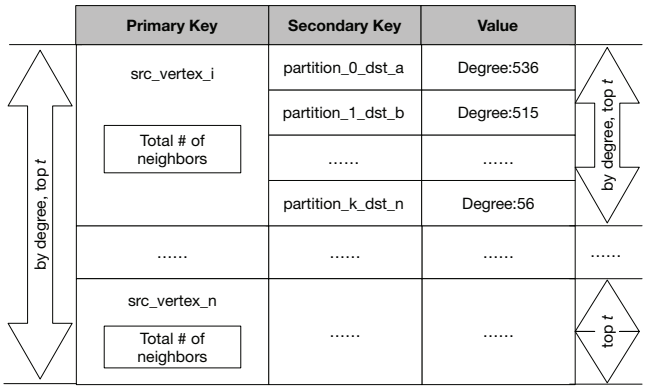


Figure. 1: Neighbor List Sorted by Degree

Given such a sorted, fixed-length neighbor list, our AKIN algorithm is able to apply the similarity evaluation on the two vertices of any arriving edge. To be specific, each time when an edge $e(u, v)$ in the edge stream is received, our AKIN algorithm will retrieve two neighbor lists of vertices (u and v) from their base partition, and then apply similarity-based heuristic $H(u, v)$ on each partition to find the partition $\mathcal{P}_t$ where the maximum score of $H(u, v)$ is obtained. Once $\mathcal{P}_t$ is found, AKIN will assign the edge $e(u, v)$ and both vertices (u and v) to this partition $\mathcal{P}_t$, only if both u and v are available in all partitions (as shown in **Algorithm 2**). Otherwise, the arriving edge will be assigned to the partition where its source vertex resides. After vertex migration happens, we define the new partition hosting the vertex to be the **hosting partition**.

Afterwards, AKIN will update the neighbor list of both vertices u and v on their base partitions. Also, a reference to the migrated vertex will be created on the base partition for locating purposes. We denote the reference of a vertex v by $\mathcal{K}_v$. The reference simply plays the role of pointer to the vertex by recording the vertex ID and its hosting partition ID.

Algorithm 2 Determine vertex migration for arriving edge $e(u, v)$

Input: arriving edge $e(u, v)$, number of all partitions k
1: $i \leftarrow h(u)$
2: $j \leftarrow h(v)$
3: **if** ($\mathcal{K}_u \in \mathcal{P}_i$ **or** $u \in \mathcal{P}_i$) **and** ($\mathcal{K}_v \in \mathcal{P}_j$ **or** $v \in \mathcal{P}_j$) **then**
4: $max\_score \leftarrow 0$
5: $t \leftarrow nil$
6: **for all** p such that $0 \leq p < k$ **do**
7: $x \leftarrow H_p(u, v)$
8: **if** $x > max\_score$ **and** $isFull(\mathcal{P}_x) =$ **false then**
9: $max\_score \leftarrow x$
10: $t \leftarrow p$
11: **end if**
12: **end for**
13: **if** $t \neq nil$ **then**
14: migrate u and v to partition $\mathcal{P}_t$.
15: $\mathcal{K}_u \leftarrow Tuple(u, \mathcal{P}_t)$
16: $\mathcal{K}_v \leftarrow Tuple(v, \mathcal{P}_t)$
17: maintain reference key $\mathcal{K}_u$ at partition P_i.
18: maintain reference key $\mathcal{K}_v$ at partition P_j.
19: **end if**
20: **end if**

Here, $isFull$ function takes a partition and determines whether the partition size constraint is reached. A coefficient of such constraint ν is provided so that our algorithm is able to loosen the size constraint and gather more similar vertices in one partition.

AKIN algorithm always seeks to join similar vertices together rather than tearing them apart. We consider vertex migration on every edge insertion operation to be sufficient, namely, the deletion of an edge is out of our concern.

C. Heuristic Function

For any partition $\mathcal{P}_i$, our similarity-based heuristic $H(u, v)$ is defined as follows:

$$H_i(u, v) = \alpha \times sim_i(u, v) - \beta \times pnl(i) \tag{1}$$

Here, $sim_i(u, v)$ calculates the similarity score of vertices u and v on partition $\mathcal{P}_i$ and the $pnl(i)$ calculates a score based on the size of partition $\mathcal{P}_i$, serving as the penalty against the similarity score.

The purpose of having a penalty score is to guarantee that, while the similarity of two vertices is considers, the balance of partition size can also be taken into account.

With α and β serving as coefficient of similarity score and penalty score, for each arriving edge $e(u, v)$, the objective of our AKIN algorithm is to find a certain partition $\mathcal{P}_i$ among k partitions, when the value of the heuristic function $H(u, v)$ reaches to its maximum, namely,

$$\underset{i \in \{0,\ldots,k-1\}}{\arg\max} \{H(u, v)\} \tag{2}$$

D. Similarity Score of a Partition $\mathcal{P}$

In the AKIN algorithm, we use the Jaccard index [16] as the similarity measurement, since its simplicity leads to efficient similarity evaluation while its mathematical meaning makes it intuitive to be applied on the evaluation of vertex similarity. The definition of the Jaccard index is as follows, given two sample sets, A and B:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|} \tag{3}$$

Based on this equation, for any edge $e(u, v)$, we define A is the set of outgoing neighbors of u, and B is the set of outgoing neighbors of v. Then, we can extend this similarity index to our graph partitioning scenarios, where we have multiple partitions. The similarity score for both u and v on a particular partition $\mathcal{P}$ can be defined as:

$$sim_{\mathcal{P}}(u, v) = \frac{|A_{\mathcal{P}} \cap B_{\mathcal{P}}|}{|A \cup B|} = \frac{|A_{\mathcal{P}} \cap B_{\mathcal{P}}|}{|A| + |B| - |A_{\mathcal{P}} \cap B_{\mathcal{P}}|} \tag{4}$$

where $A_{\mathcal{P}}$ and $B_{\mathcal{P}}$ are the sets of outgoing neighbors on a particular partition $\mathcal{P}$ for vertex u and v, respectively. With the support of sorted neighbor list, the Jaccard index on the neighboring sets of two vertices can be efficiently calculated to achieve streaming graph partitioning. We will discuss the complexity of similarity calculation in Section V.

E. The Calculation of Penalty Score

The penalty score of a partition $\mathcal{P}_i$ aims at avoiding biased evaluation towards the similarity score when calculating the heuristic score H. Since it is related to partition size, depending on the definition of partition size, it can be calculated in many ways. For example, if we only consider the generic graph model, the size of a partition $\mathcal{P}_i$ could be represented by the number of vertices in the partition, denoted as $|V_{\mathcal{P}_i}|$. With the size constraint of partition $\mathcal{P}_i$ defined as $max(|V_{\mathcal{P}_i}|)$, the penalty score can be calculated as:

$$pnl(i) = \frac{|V_{\mathcal{P}_i}|}{max(|V_{\mathcal{P}_i}|)} \tag{5}$$

V. Complexity Analysis in Distributed System

In this section, we conduct theoretical analysis about the complexity of AKIN algorithm. Since determining vertex placement simply relies on the hashing function and requires $\mathcal{O}(1)$ time, the most critical part of our algorithm is to determine the edge placement. We will analyze the edge placement from three aspects: 1. Cost of Neighbor List, 2. Complexity of Local Similarity Evaluation, 3. Data Migration Cost.

A. Cost of Neighbor List

Each time when evaluating the similarity index of two vertices on all partitions, AKIN can simply retrieve the neighbor lists of the two vertices from their base partition. Since the size of each neighbor list is globally fixed (which can be denoted by n), the communication cost for such an operation is also fixed. To be specific, suppose the length of vertex ID takes 64 bits, which is identical to 8 Bytes, and the length of partition ID as well as the length of vertex degree takes 32 bits, namely 4 Bytes, each record in the neighbor list will cost 16 Bytes or so. For a neighbor list of 500 records, the entire list will take up 8KB, which will not impose a heavy burden on the storage system and can also be easily transmitted over the network without taking up too much bandwidth. In addition, one of our evaluation results in Section VI shows that our AKIN algorithm does not require the length of the neighbor list to be very long. With the length of 100 records for every neighbor list, our algorithm already outperforms the state-of-the-art streaming graph partitioning algorithm - FENNEL with a much better edge-cut ratio. Thus, in our AKIN algorithm, introducing such a neighbor list for each vertex will neither exhaust storage resource nor network bandwidth. Furthermore, with some optimizations, the network communication overhead for transmitting neighbor lists still remains little. For example, if the comparison of two neighbor lists happens at the server side, there is no need to transmit both neighbor lists. After transmitting one neighbor list from a partition to another, the in-situ comparison of two lists can be conducted and consequent edge placement can be performed by the server where such a comparison is made. Overall, in modern distributed systems where large graphs can be stored and queried, it would not be a big deal for AKIN to introduce such a neighbor list.

B. Complexity of Local Similarity Evaluation

To evaluate the similarity between two vertices, our algorithm needs to compare between two neighbor lists of both vertices. Although the complexity of comparing the two neighbor lists would be proportional to the size of the lists, given the fact that the size of all neighbor lists are globally fixed, such operations can actually finish in constant time. Also, with some optimizations, the computation time can be further reduced. For example, with proper data organization of the neighbor list and the support of parallelism in modern computers, the number of common neighbors on different partitions can be evaluated in parallel. One of our evaluation results in Section VI shows that the computational overhead for local similarity evaluation is very small, even without the support of any parallelism mechanism.

C. Data Migration Cost

Since the AKIN algorithm only considers edge placement rather than edge migration, there is no need to consider edge migration cost. The only data migration happens to the vertices on an edge. Each time for an arriving edge, the possibility of migrating the two vertices is $\frac{k-1}{k}$ for k partitions. However, since each data migration only involves two vertices, the migration cost when determining each placement for each arriving edge remains constant. Although multiple data migrations may happen to many pairs of vertices over a relatively long period of time, considering the goal of minimizing edge-cut ratio, such data migration cost is inevitable and worthwhile. More importantly, as a graph partitioning algorithm which aims to partition an ever-growing graph on the fly, the data migration cost of only two vertices already enables AKIN to achieve so.

VI. Experimental Evaluation

A. Experimental Setup

We carried out all evaluations on c6320 computing nodes from CloudLab [17]. For the c6320 cluster, each computing node has two Intel E5-2683 v3 14-core CPUs at 2.00 GHz, 256GB ECC Memory, two 1 TB 7.2K RPM 3G SATA HDDs, and dual-port Intel 10Gbe NIC (X520) and Qlogic QLE 7340 40 Gb/s Infiniband HCA (PCIe v3.0, 8 lanes).

In the following evaluations, we applied multiple partitioning algorithms on various graph datasets to generate multiple partitions (eight) for comparison. Since our focus in this paper is to evaluate the capability of our AKIN algorithm in terms of keeping the partition size balanced, reducing the edge-cut ratio and controlling the overhead of making partition decisions, we measure their corresponding metrics for quantitative comparison. The evaluated metrics include: 1) the most overweighed partition size ratio and standard deviation of partition size ratio, which show how balanced the resulted partitions can be; 2) the edge-cut ratio, which reflects the data locality from the flip-side; 3) the maximum, average, 90-percentile running time of each partition decision making, which indicates the overheads that our algorithm may bring to the real system.

B. Evaluation Datasets

We select 14 different graphs' data sets from the Stanford Large Network Dataset Collection [18] to conduct the evaluations. The datasets were selected to cover the wide range of use cases: from co-purchasing network to social network, from citation network to the autonomous system network, from p2p network to World Wide Web, from Internet interactions to citation network.

Although the graphs in the SNAP graph data set are not excessively large, it is still valid to consider a distributed system where a large number of clients access any of these graphs. A single server definitely will not be capable of responding such a huge query workload, so it is still necessary to distribute any of these graphs onto multiple machines in a distributed storage system. Also, for streaming graph partitioning algorithms, any of these graphs can be considered as a subgraph of the ever-growing graph collected from the data stream during a limited period of time. A streaming graph partitioning algorithm should work well at any given time period, thus it is not necessary to consider excessively huge graphs. As long as the streaming graph partitioning algorithms achieve two major pursuits, i.e., the balanced partition size and the minimized edge-cut ratio, at any given time for any type of graph, we consider the algorithm to be effective.

TABLE I: Basic Information of Graph Datasets

Graph	Type	Num_V	Num_E
amazon0601	Purchase	403394	3387388
as-skitter	Network	1696415	11095298
ca-AstroPh	Social	18772	198110
cit-HepPh	Citation	34546	421578
cit-Patents	Citation	3,774,768	16,518,948
email-Enron	Social	36692	183831
higgs-mention_network	Social	116408	150818
higgs-retweet_network	Social	256491	328132
higgs-social_network	Social	456626	14855842
loc-brightkite_edges	Geo	58228	214078
p2p-Gnutella31	Network	62586	147892
soc-Slashdot0902	Social	82168	948464
web-Google	Web	875713	5105039
wiki-Talk	Social	2394385	5021410

C. Experiment Parameters

TABLE II: Experiment Parameters

Experiment Name	Parameters	ν
METIS	Default Parameters (p=k-way, c=shem, obj=cut)	
Hash	None	
FENNEL Cost 1	$c(x) = \alpha x^{\gamma}, \gamma = \frac{3}{2}, \alpha = \sqrt{k}\frac{m}{n^{3/2}}$	1.1
FENNEL Cost 2	$c(x) = \frac{1}{2}x^2$	1.1
AKIN 100	$\beta = 1$ (with Penalty), $n = 100$	1.1
AKIN 200	$\beta = 0$ (without Penalty), $n = 200$	1.1
AKIN 300	$\beta = 0$ (without Penalty), $n = 300$	2
AKIN 400	$\beta = 0$ (without Penalty), $n = 400$	3

We evaluate our AKIN algorithm against the state-of-the-art streaming graph partitioning algorithm - FENNEL. To observe how close we can get to the extreme case of edge-cut ratio and the extreme case of partition size balance, we also conduct the experiment on one classic offline graph partitioning algorithm - METIS (which is our extreme case of edge-cut ratio), and one most commonly used streaming data partitioning algorithm - the deterministic hashing (which is our extreme case of partition size balance).

We conduct these algorithms with different parameter settings. As shown in Table II, for deterministic hashing, there are no specific parameters that need to be set up due to its plainness and simplicity.

For METIS, we just use the default parameters since they are already the best case as described in their paper [7].

For FENNEL, we choose two specific families of the cost function mentioned in the original paper [12]. For the first family of cost function $c(x) = \alpha x^{\gamma}$, we choose the same value for α and γ as what was used in their experiments. For the second family of cost function, we choose $c(x) = \frac{1}{2}x^2$ as discussed in the original paper too.

For AKIN, we also setup multiple parameters to better evaluate its performance characteristics. Specifically, we introduce four different parameter sets. In each parameter set, we control three key parameters: 1) whether to consider penalty or not (β); 2) the size of neighbor list (n); and the coefficient of partition size constraint (ν). We use the length of neighbor list to denote the name of each parameter set. As shown in Table II, we only consider penalty under the minimal number of adjacency list (i.e., AKIN 100). For the other three cases, we consider removing the penalty by setting α to be 0, and we set up the partition size constraint factor ν as 1.1, 2 and 3 respectively.

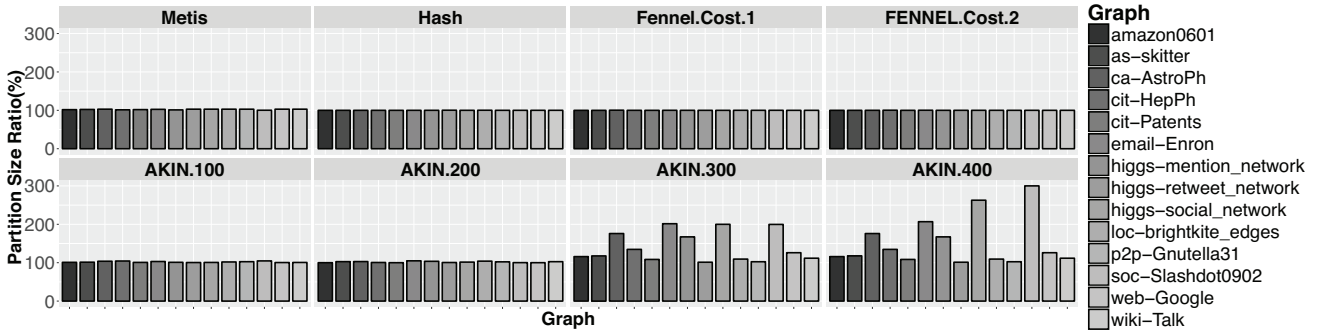
D. Experimental Results

1) Partition Size Ratio: One major pursuit of the graph partitioning algorithm is to balance the partition size. After partitioning a graph, the size of all partitions should be roughly equal, so that when queries against the graph arrive, the workload of each partition can be well balanced. We define the partition size ratio of partition $\mathcal{P}$ to be

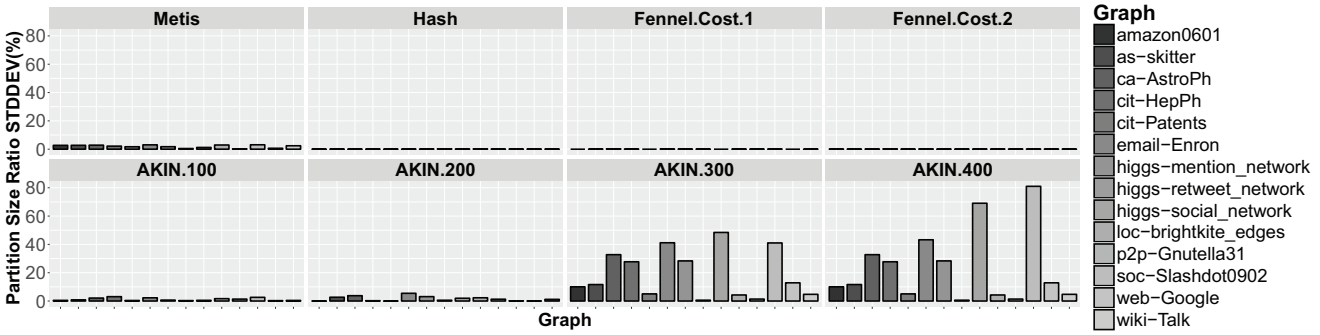
$$\rho_{\mathcal{P}} = \frac{\text{the actual number of vertices in partition } \mathcal{P}}{\text{the average number of vertices in each partition}} \quad (6)$$

As it shows in Figure 2a, the most balanced partitioning results are generated by Hash and FENNEL. METIS also conducts a decent job in maintaining the balance of all partitions. The balance of the AKIN algorithm is largely determined by the selected parameters. For example, AKIN still generates balanced partitions when the coefficient of partition size constraint ν is 1.1. But, when ν increases, like in the case of $\nu = 2$(AKIN.300) and $\nu = 3$(AKIN.400), the balance between partitions is obviously violated.

However, tolerating imbalance between different partitions helps in reducing the edge-cut ratio even further. For example, as shown in Figure 3, when $\nu = 1.1$(AKIN.200), the edge-cut ratio of *soc-Slashdot0902* is above 50%(54.09% actually), whereas in the case of $\nu = 2$(AKIN.300) and $\nu = 3$(AKIN.400), the edge-cut ratio of *soc-Slashdot0902* is around 53.19% and 45.98% respectively.



(a) Partition Size Ratio of the Most Overweighed Partition



(b) Standard Deviation of the Partition Size Ratio

Figure. 2: Partition Balance Evaluation

In general, all partitioning algorithms in our evaluation are capable of maintaining balanced partition size. Specifically, AKIN neither falls behind the state-of-the-art streaming graph partitioning algorithm, nor fails to maintain balanced partition size as deterministic hashing does, unless the partition size constraint is set to be violated for better edge-cut ratio.

2) Edge-cut Ratio: Less edge-cut ratio means better data locality and less communication overhead between different graph storage servers. We compare the edge-cut ratio of our selected graph partitioning algorithms. Please note that we include METIS in this comparison to see how close our AKIN algorithm can approach to the extreme case of edge-cut ratio - METIS, as compared to the-state-of-the-art streaming graph partitioning algorithm - FENNEL. Through the discussion in previous sections, it is naturally difficult for an online streaming graph partitioning algorithm to achieve superior edge-cut ratio while offline graph partitioning algorithms like METIS can.

We show the results in Figure 3. It can be easily seen that METIS has the minimum edge-cut ratio while the deterministic hashing algorithm generates the partitioning result with the highest edge-cut ratio. As we discussed earlier, the reason is simply because offline algorithms have the information of the entire graph and can find the best partitioning solution on a global basis, while the deterministic hashing algorithm does not consider any graph structural features at all. For the FENNEL algorithm with its cost function 1, we can see its edge-cut ratio of most graphs is slightly reduced as compared to deterministic hashing. In some graphs such as *higgs-activity_time*, *higgs-mention_network* and *higgs-retweet_network*, the edge-cut ratio is reduced even more significantly. However, for the FENNEL algorithm with its cost function 2, the edge-cut ratio of all the graphs is not reduced at all as compared to the result of deterministic hashing. Figure 3 also shows the results of AKIN. We can observe that the AKIN algorithm is able to reduce more edge-cut ratio compared to Fennel with both cost functions. Even with the penalty and maximum 100 elements in the adjacency list, the edge-cut ratio of AKIN still witnesses a good reduction for most graphs. In terms of edge-cut ratio, AKIN did better in getting close to the quality of an offline graph partitioning algorithm, as compared to FENNEL.

After removing the penalty and increasing the maximum length of adjacency list to 200, the edge-cut ratio of every graph is substantially reduced. After increasing the coefficient of partition size constraint ν from 1.1 to 3, and increasing the maximum length of adjacency list too 400, the edge-cut ratio of all different graphs are reduced even more significantly, and approaches even closer to the edge-cut ratio of the offline graph partitioning algorithm - METIS.

3) Decision Making Time per Edge Chunk: Since our AKIN algorithm mainly works on the edge stream, for each arriving edge, we need to calculate the intersect of the neighbors of its two vertices to get the similarity. While transmitting neighbor lists, which is less than 8KB and is not a big deal for modern distributes systems, it is also critical to understand how much time will be taken for compare the neighbor lists

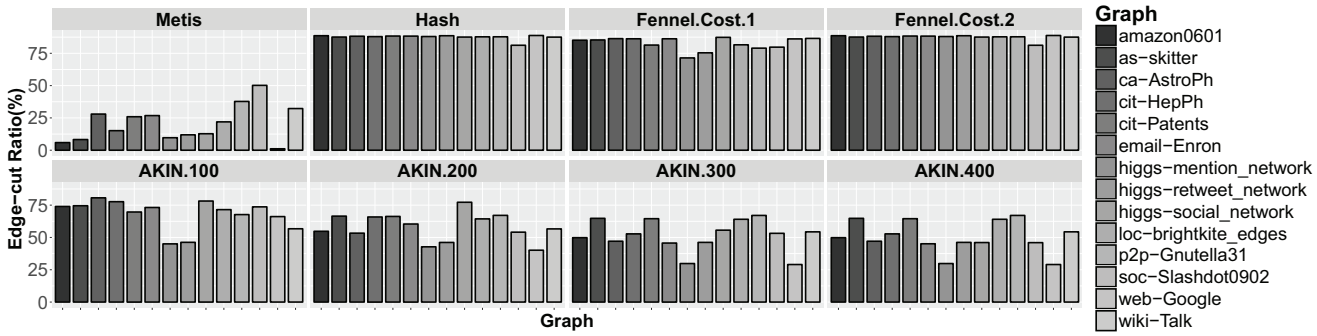


Figure. 3: Edge-cut Ratio Comparisons

of two vertices. So, we evaluated the time for such in-memory comparison. Specifically, we show the maximum and average decision making time, plus the 90-percentile decision making time, which is the decision making time below 90 percent of observations throughout all the partitions. Namely, if the 90-percentile decision making time is acceptable, we should expect that 90% of the partition decision making takes place within an acceptable amount of time.

As shown in Figure 4, throughout the entire 14 graph datasets and all their edges, the maximum decision making time is slightly over 600ms for a particular graph dataset - *cit-Patent*. Except for the Hash algorithm, FENNEL also has the similar maximum running time over 500ms on that graph. In this worst case, AKIN.100 only spends less than 10% more time compared to FENNEL(FENNEL.Cost1), and AKIN.400 spends around 20% more time than the best case of FENNEL(FENNEL.Cost2). This suggests a comparable small computational overhead towards the state-of-the-art streaming partitioning algorithm. Looking deeper into the 90-percentile decision making time, it is clear that, for the slowest case of dataset *cit-Patent*, 90% percent of the decisions were made really fast (within 1ms). For other datasets, 90% percent of the comparisons were made only within 0.5ms, regardless of the length of the neighbor list. On average, the comparison between neighbor lists of every two vertices on an edge can be finished within 1ms, which ensures that our AKIN streaming graph partitioning algorithm can partition the graph on the fly.

VII. Related Work

The existing graph partitioning schemes mainly falls into two categories - edge-cut and vertex-cut regarding to whether the partitioning scheme is applied on the edge or vertex. By edge-cut graph partitioning schemes, the vertices are assigned into different partitioning, while edges running between different partitions, intuitively, the edges are cut by the boundary of different partitions. Such an example can be seen in most of the existing works, such as Chaco [19], METIS [7], ParMETIS [20], Hashing and LDG [21]. By vertex-cut schemes, the critical vertices are replicated onto multiple partitions, so that there are much less links running between edges, and intuitively the vertices are cut into pieces, some of the existing practice and applications include SBV-cut [9], PowerGraph [22], GraphX [23], etc. Recently, people even attempt to apply hybrid-cut graph partitioning scheme to the distributed graph computing system, which is to apply edge-cut on low-degree vertices and to apply vertex-cut on high-degree vertices [24]. Here, the degree of a vertex measures the number of incident edges of the vertex.

However, in addition to reducing the cut-size(defined as the number of inter-partition links), we might also shed light on community structure of the graph. Community detection [25] is the technique that can help people to find some special groups of vertices with a high concentration of edges inside each group and a low concentration of edges between these groups. One big difference between graph partitioning and community detection is that the latter doesn't need to know how many communities are in the graph in advance, whereas the former needs to know how many partitions need to be generated. In other words, the process of graph partitioning can be viewed as a typical type of supervised machine learning [26], namely, classification, while community detection can be viewed as a typical case of unsupervised learning [27], namely, clustering. In comparison with the above algorithms that consider minimizing the cut-size, our approach focuses more on increasing the spatial data locality that naturally lies in the community structure of the graph. In this sense, our approach is basically semi-supervised graph clustering using the *Jaccard Index* as a similarity measure.

Although most of the graph partitioning algorithms or community detection algorithms are able to divide the graph into smaller subgraphs, some of them need to load the data of the entire graph into the memory, so that the algorithm itself is able to generate a superior partitioning result with minimized cut-size. Obviously, in the real practice when we consider the system where an increasingly large amount of data is generated from time to time, such as GraphMeta(where metadata graph is growing as the number of files increases) and TAO(where large-scale social graph data is growing everyday with the growth of the number of active users), loading the entire graph into memory is impossible due to the excessive and ever-growing size of data, the limited capacity of memory and storage. In addition, most offline graph partitioning schemes come with multiple iterations of complex computation, and such intensive computational cost makes them incapable of

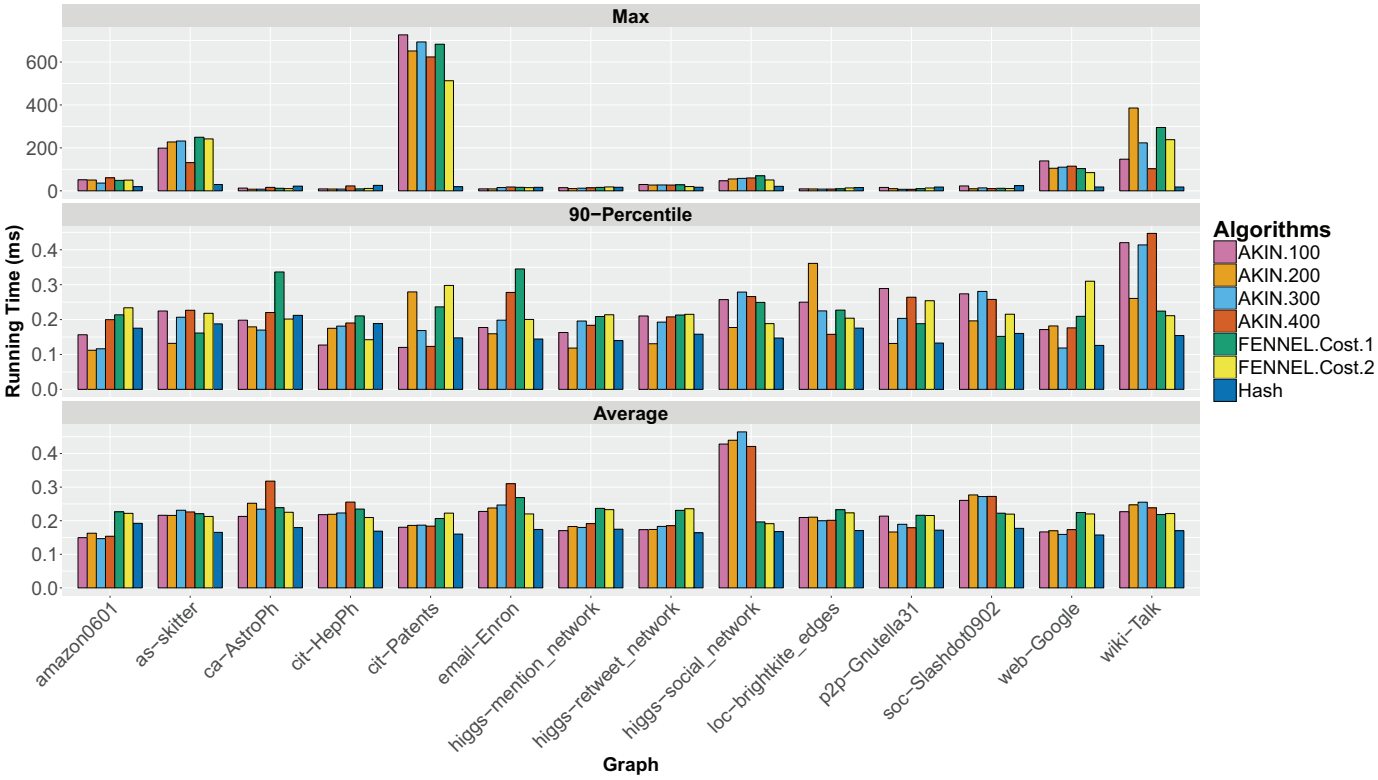


Figure. 4: Decision Making Time Comparisons (per Edge Chunk)

partitioning dynamic graph data on the fly. Since the data comes into the system incrementally, and the size of the data is growing all the time, in this scenario, streaming graph partitioning comes into play so that the data is able to be placed into a certain partition as long as it arrives to the system. As discussed in the last paragraph, our graph partitioning approach is basically doing semi-supervised graph clustering with *Jaccard Index* similarity measure, Although such an objective can be found in [28], our approach works in a streaming fashion.

In spite of a higher proportion of inter-partition links, the streaming graph partitioning approach requires neither intensive computational overhead nor huge memory space. In [21], 10 different heuristics were proposed, but to get rid of a mapping table between vertices and partitions, the best way is still the hash-based function. While some graph processing frameworks seek to apply the hash-based streaming graph partitioning scheme with some replication on vertices or edges [22], [23], [29], others are trying to apply more complicated graph partitioning techniques on the fly [12]. However, for each partition decision made on a single vertex or edge, since only very limited local information about the vertex or edge is available, most of the streaming graph partitioning algorithms only consider the speed of placing each vertex or edge while failing to consider the community structure of the graph. Thus, in terms of the partitioning result, these online graph partitioning schemes usually features a decent time efficiency while having much lower cut-size minimization for the partitioning result. Our approach strives for better data locality of each partition, so we take the local community structure into account while maintaining a fast speed for making each data placement decision in a streaming fashion. At the same time our partitioning result is compatible with the partitioning result of deterministic hashing mentioned by [21], which makes it possible to accelerate the data retrieval for one single vertex or one single edge.

VIII. Conclusion and Future Work

In this study, we propose a streaming graph partitioning algorithm - AKIN, which utilizes the similarity between vertices to increase the data locality. Compared to the state-of-the-art streaming graph partitioning algorithm (FENNEL), AKIN is able to keep partition size balanced while achieving further reduced edge-cut ratio. In addition, the partitioning overhead of AKIN is also small, making it suitable for partitioning a graph in a streaming fashion for a distributed graph storage system. However, AKIN does not aim at addressing the problem of power-law degree distribution, since the way it considers structural features of graphs has nothing to do with the degree of vertices. In the future, we would like to seek optimizations to address such challenge and would like to explore other structural features that a streaming graph partitioning algorithm can apply to achieve better partitioning quality.

ACKNOWLEDGMENT

This material is based upon work supported by the National Science Foundation under grant CCF-1409946 and CNS-1338078.

REFERENCES

[1] "Titan," http://titan.thinkaurelius.com.

[2] "Flockdb," http://engineering.twitter.com/2010/05/introducing-flockdb.html.

[3] D. Dai, R. B. Ross, P. Carns, D. Kimpe, and Y. Chen, "Using property graphs for rich metadata management in hpc systems," in *Parallel Data Storage Workshop (PDSW), 2014 9th*. IEEE, 2014, pp. 7–12.

[4] D. Dai, Y. Chen, P. Carns, J. Jenkins, W. Zhang, and R. B. Ross, "A Graph-based Engine for Managing Large-Scale HPC Rich Metadata," in *the IEEE International Conference on Cluster Computing, (Cluster'16)*, 2016.

[5] M. R. Garey, D. S. Johnson, and L. Stockmeyer, "Some simplified np-complete problems," in *Proceedings of the sixth annual ACM symposium on Theory of computing*. ACM, 1974, pp. 47–63.

[6] T. N. Bui and C. Jones, "Finding good approximate vertex and edge partitions is np-hard," *Information Processing Letters*, vol. 42, no. 3, pp. 153–159, 1992.

[7] G. Karypis and V. Kumar, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1999.

[8] B. Hendrickson and R. Leland, "The chaco users guide: Version 2.0," Technical Report SAND95-2344, Sandia National Laboratories, Tech. Rep., 1995.

[9] M. Kim and K. S. Candan, "Sbv-cut: Vertex-cut based graph partitioning using structural balance vertices," *Data & Knowledge Engineering*, vol. 72, pp. 285–303, 2012.

[10] G. Kliot, "Streaming Graph Partitioning for Large Distributed Graphs Categories and Subject Descriptors," *Acm Kdd*, pp. 1222–1230, 2012. [Online]. Available: http://kdd2012.sigkdd.org/

[11] H. P. Sajjad, A. H. Payberah, F. Rahimian, V. Vlassov, and S. Haridi, "Boosting vertex-cut partitioning for streaming graphs," in *Big Data (BigData Congress), 2016 IEEE International Congress on*. IEEE, 2016, pp. 1–8.

[12] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, "Fennel: Streaming graph partitioning for massive scale graphs," in *Proceedings of the 7th ACM international conference on Web search and data mining*. ACM, 2014, pp. 333–342.

[13] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz, "Recent advances in graph partitioning," 2013.

[14] M. E. Newman, "The structure and function of complex networks," *SIAM review*, vol. 45, no. 2, pp. 167–256, 2003.

[15] J. Scott, *Social network analysis*. Sage, 2012.

[16] L. Hamers, Y. Hemeryck, G. Herweyers, M. Janssen, H. Keters, R. Rousseau, and A. Vanhoutte, "Similarity measures in scientometric research: the jaccard index versus salton's cosine formula," *Information Processing & Management*, vol. 25, no. 3, pp. 315–318, 1989.

[17] "CloudLab," https://www.cloudlab.us/.

[18] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[19] B. Hendrickson and R. W. Leland, "A multi-level algorithm for partitioning graphs." *SC*, vol. 95, p. 28, 1995.

[20] G. Karypis and V. Kumar, "Parallel multilevel series k-way partitioning scheme for irregular graphs," *Siam Review*, vol. 41, no. 2, pp. 278–300, 1999.

[21] I. Stanton and G. Kliot, "Streaming graph partitioning for large distributed graphs," in *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2012, pp. 1222–1230.

[22] J. Gonzalez, Y. Low, and H. Gu, "Powergraph: Distributed graph-parallel computation on natural graphs," *OSDI'12 Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pp. 17–30, 2012. [Online]. Available: https://www.usenix.org/system/files/conference/osdi12/osdi12-final-167.pdf

[23] R. S. Xin, J. E. Gonzalez, M. J. Franklin, I. Stoica, and E. AMPLab, "GraphX: A Resilient Distributed Graph System on Spark," *First International Workshop on Graph Data Management Experiences and Systems*, p. 2, 2013.

[24] R. I. o. P. Chen, D. Systems), J. Shi, Y. Chen, H. Guan, H. Chen, and B. Zang, "PowerLyra : Differentiated Graph Computation and Partitioning on Skewed Graphs," 2013.

[25] S. Fortunato, "Community detection in graphs," *Physics reports*, vol. 486, no. 3, pp. 75–174, 2010.

[26] S. Sarkar and P. Soundararajan, "Supervised learning of large perceptual organization: Graph spectral partitioning and learning automata," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 5, pp. 504–525, 2000.

[27] F. D. Malliaros and M. Vazirgiannis, "Clustering and community detection in directed networks: A survey," *Physics Reports*, vol. 533, no. 4, pp. 95–142, 2013.

[28] S. E. Schaeffer, "Graph clustering," *Computer science review*, vol. 1, no. 1, pp. 27–64, 2007.

[29] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.