
# Triparts: Scalable Streaming Graph Partitioning to Enhance Community Structure

Ruchi Bhoot
Indian Institute of Science (IISc)
Bangalore, India
ruchibhoot@iisc.ac.in

Tuhin Khare*
Georgia Institute of Technology
Atlanta, GA, USA
tkhare7@gatech.edu

Manoj Agarwal
GiKA.AI
Bangalore, India
agarwalm@gikagraph.ai

Siddharth Jaiswal*
Indian Institute of Technology (IIT)
Kharagpur, India
siddsjaiswal@kgpian.iitkgp.ac.in

Yogesh Simmhan
Indian Institute of Science (IISc)
Bangalore, India
simmhan@iisc.ac.in

## ABSTRACT

k-way edge based partitioning algorithms for processing large streaming graphs, such as social networks and web crawls, assign each arriving edge to one of the k partitions. This can result in vertices being replicated on multiple partitions. Typically, such partitioning algorithms aim to balance the edge counts across partitions while minimizing the vertex replication. However, such objectives ignore the community structure inherently embedded in the graph, which is an important quality metric for clustering and graph mining applications that subsequently operate on the partitions. To address this gap, we propose a novel optimization goal to maximize the number of local triangles in the partitions as an additional objective. Triangle count is an effective metric to measure the conservation of community structure. Further, we propose TriParts a family of heuristics for online partitioning over an edge stream. They use three complementary state data structures: Bloom Filters, Triangle Map and High degree Map. Each state adds tangible value to meet our objectives. We validate TriParts on six diverse real world graphs with up to 1.6B edges and varying triangle densities. Our best heuristic outperforms the state-of-the-art DBH and HDRF streaming graph partitioners on the triangle-count metric by up to 4-8.3x while maintaining competitive vertex replication factor and edge-balancing. We achieve an ingest rate of 500k edges/sec on a 16 node cluster. We also offer detailed results on the configuration parameters, scalability and overheads of TriParts, and its practical benefits for distributed graph analytics.

**PVLDB Reference Format:**
Ruchi Bhoot, Tuhin Khare, Manoj Agarwal, Siddharth Jaiswal, and Yogesh Simmhan. Triparts: Scalable Streaming Graph Partitioning to Enhance Community Structure. PVLDB, 18(9): 2992 - 3006, 2025.
doi:10.14778/3746405.3746423

**PVLDB Artifact Availability:**
The source code, data, and/or other artifacts have been made available at https://github.com/dream-lab/triparts.

*Based on work done while at the Indian Institute of Science (IISc).


Proceedings of the VLDB Endowment, Vol. 18, No. 9 ISSN 2150-8097.
doi:10.14778/3746405.3746423

## 1 INTRODUCTION

Massive graphs such as web, social, and gene sequencing graphs, often contain billions of vertices and edges, exceeding single-machine memory [31, 41, 60]. To enable distributed memory and parallel computation, these graphs must be partitioned across multiple machines [4, 60]. Efficient partitioning [6, 22, 45, 46] is therefore crucial for distributed processing .

Graph partitioning can be vertex-based or edge-based. A $k$-way *vertex-based* partitioning places each vertex on one of $k$ partitions such that each has a similar number of vertices to achieve load balancing, and the edge-cuts between vertices in different partitions are minimized [17, 44, 67, 77, 83]. A $k$-way *edge-based* partitioning places each edge on one of the partitions to balance the edges per partition, and minimize the vertex replicas across partitions [44]. Edge-based partitioning is more effective for power-law graphs often seen in real-world graphs [31, 58], and the focus of this article.

Optimal graph partitioning is NP-Complete [6, 58] and many heuristics have been proposed [31, 43, 44, 46, 63, 74, 79, 83], in particular for *static graphs* that are available *a priori* [17, 54, 67, 83]. However, graphs that model contemporary applications such as financial transaction networks, Internet of Things sensors and social network feeds are continuously updated with 1k-100k edges/sec [42, 55, 59]. Partitioning such dynamic or *streaming graphs* helps ensure that the incremental graph received at any point in time is available for distributed analysis to serve periodic insights or to identify emerging events in a timely manner [3]. This is in contrast to periodically partitioning the entire accumulated graph every few hours – which may itself take several hours for billion-scale graphs [17, 44] – or waiting for the entire graph to be available, which may be infeasible when graph updates never terminate, before analyzing the graph.

*Streaming graph partitioners* [1, 26, 79] receive the graph updates at a *central machine (leader)*, which decides the partition on which to place the incoming graph element. *Worker machines* receive the updates from the leader and add them to their respective local partition. In *streaming edge-based partitioning* (Fig. 1), edges represented as vertex-pair form the update stream [31, 43, 63, 74, 87]. Each arriving edge is placed in one partition and a vertex whose incident edges are on different partitions is *replicated* on each of these partitions. The objectives are to maintain *edge-balancing*, $\approx \frac{|E|}{k}$ per partition,

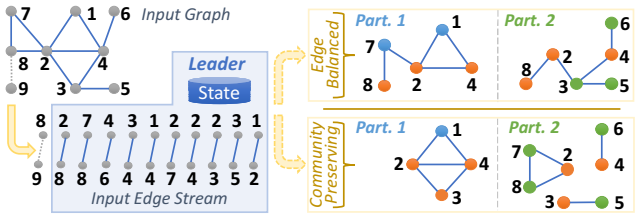



**Figure 1: Edge-based streaming partitioning. Partitioner on bottom preserves communities while top does not. Both are edge-balanced with the same vertex replication factor.**

and to minimize the replicated vertices, with the *vertex replication factor* given by $\rho = \frac{(\sum_1^V r(v))}{|V|}$, where $1 \leq r(v) \leq k$ is the number of replicas per vertex. Balancing the edges per partition helps load-balance the compute and memory load on workers when executing a distributed graph analysis over the partitioned graph [14, 47]. Reducing vertex replication mitigates duplicate elements across partitions and reduces messaging between them during distributed execution [31]. The leader may use *local in-memory state* to make the decision on edge placement; but given the large graph sizes, it is important that this state is bounded in size to $O(|V|)$ and $\ll O(|E|)$.

E.g., in Fig. 1, the leader has processed the edge stream until edge $2 \leftrightarrow 8$ and decided on their placements among $k = 2$ partitions, shown in the top-right, based on a default edge-balancing strategy . Both parts have 5 edges each, making them well-balanced . The orange vertices $(2, 4, 8)$ have a replication factor of $r(v) = 2$, while the others have 1. This gives us $\rho = \frac{11}{8}$.

***Motivation.*** In this article, we go beyond these two standard objectives of edge-balancing and minimizing vertex replication, and argue that a good graph partitioning algorithm should preserve the *local community structure* within each partition [13, 65, 81]. This can help solve a class of graph algorithms such as community evolution [62], PageRank [32, 78], sampling GNN computation graphs [92], etc. using local partition information rather than regularly resort to costly distributed analytics.

Real-world graphs typically exhibit a power-law degree distribution and small-world properties, with a few highly connected hub vertices forming dense communities and sparse inter-community links [57, 72]. Community quality is measured using metrics like transitivity, Local Clustering Coefficient (LCC), modularity, density, conductance, and cohesiveness. Transitivity [30] reflects the likelihood of adjacent nodes being interconnected, indicating tightly knit communities. LCC measures how close a vertex's neighbors are to forming a clique and helps identify small-world networks [84].

This intuition can be extended to graph partitioning, where *preserving community structure* within partitions serves as a quality metric. Prior work emphasizes partition compactness [13, 81] and high intra-partition connectivity, especially for community detection and resilience enhancement [10, 65]. Partitioning into highly connected communities benefits information flow in social networks [37], create sub-graphs resilient to vertex failures in power grids [35], and biological clustering [13, 30].

Therefore, a streaming partitioner should aim to co-locate community vertices and edges within the same partition, while meeting other objectives. Since intra-community interactions are more frequent, this reduces inter-machine communication [32, 62], enables faster local approximations [32, 37].

***Gaps.*** Several edge-based streaming partitioners have been proposed [43, 63, 74] with the objective of balancing edge loads and minimizing vertex replicas. But few preserve the community structure embedded in such graphs while partitioning [65]. Also, these default objectives do not result in community preservation. E.g., in Fig. 1, our proposed community-aware triangle-maximizing partitioner, *TriParts*, results in partitions (bottom-right) that are edge-balanced (5 edges each) and with the same vertex replication ($\rho = \frac{11}{8}$), but is able to preserve the two communities present in the input graph at this point in time, $(1, 2, 3, 4)$ and $(2, 7, 8)$, within the local partitions. State-of-the-art (SOTA) algorithms such as DBH [87] and HDRF [74] cannot preserve communities §6.3.4. Community-aware edge partitioners, CLUGP [52] and 2PS-L [66], do out-of-core partitioning of a materialized stream, taking 3–4 passes over the stream. Even with 1 pass over an online stream, BTH qualitatively out-performs CLUGP's offline approach (§6.2.2). This motivates the need for a fast streaming partitioner that meets this additional objective.

***Contributions.*** In this paper, we propose a novel problem of $k$-way streaming edge-based graph partitioning, with the added objective of preserving the community structure within local partitions, while also balancing edges and minimizing vertex replication. Specifically, we identify that preserving the number of local triangles within each partition helps meet the community-preservation goal, and leverage this along with compact data structures to make fast placement decisions on edge streams, with minimal communication between the leader and workers, and achieving rates of 500$k$ edges/s. *To our knowledge, this is the first edge-based streaming graph partitioner designed to preserve the community structure.*

We make the following specific contributions:

(1) We formalize the problem of edge-based streaming partitioning to enhance the local community structure, discuss the benefits of using triangle preservation to retain community structure, and define a novel objective function for the same (§3).
(2) We develop a scalable multi-threaded distributed partitioning framework *TriParts* that uses compact data structures maintained at the leader to make partitioning decisions (§4).
(3) We propose four heuristics that incrementally leverage these states to partition the incoming edge stream to meet our qualitative objectives (§5).
(4) We compare TriParts with several SOTA baselines for six large real-world graphs and five synthetic graphs, and report partitioning quality, scalability metrics, configuration trade-offs, overheads and practical benefits for distributed analytics (§6).

## 2 RELATED WORK

### 2.1 Static Graph Partitioning

Static graph partitioning focuses on balancing the elements in each partition and minimizing the edge cuts (vertex-based partitioning) or vertex replication (edge-based). Identifying optimal balanced $k$-way partitioning of input graph $G = \langle V, E \rangle$ is NP-Complete. *Andreev and Räcke* [6] give an approximation algorithm with an approximation ratio of $O(\log^{1.5} |V|)$. *Kernighan and Lin* [46] present a

2-way partitioning algorithm which can be extended to $k$ partitions with a time complexity of $O(|V|^2 \cdot \log |V|)$.

There are a number of works on vertex-based partitioning. *METIS* [44] is a popular hierarchical vertex-based algorithm that offers highly balanced partitions with low edge cuts. *SCOTCH* [17] is a recursive multi-level bisection partitioner. Their parallel variants, *ParMETIS* [45] and *PT-Scotch* [17], can be deployed on distributed clusters. *ParHip* [67] uses distributed label propagation to achieve high quality partitioning. We focus on edge-based partitioning, which is effective for power-law graphs [31, 58] and common in streaming scenarios.

*Schlag et al.* [77] proposed a distributed edge-partitioner that uses a Split and Connect Method to partition a graph. Others [41] have offered a distributed memory edge partitioner that uses parallel neighbor expansion to perform the partitioning. However, it is non-trivial to extend static partitioners to a streaming scenario, which we address, as they rely on *a priori* knowledge of the whole graph.

## 2.2 Streaming Graph Partitioning

In a streaming scenario, the partitioning decision is taken at the ingest point of a graph element as it arrives at a leader machine. The partitioning algorithms can either be *stateless*, e.g., using a *hash* function [79], or maintain and use an *incremental state* at the leader based on the previous elements seen [74, 83]. The incoming stream can be in a random order, or follow a traversal order like Breadth/Depth First Search (BFS/DFS), such as observed in web crawls. *Abbas et al.* [2] provide a comprehensive survey of some of the common streaming partitioning algorithms. We address the problem of streaming partitioning by maintaining state at the leader and report competitive results for both random and BFS ordering.

***Vertex-based Partitioning.*** These systems process a stream of vertices with their adjacency lists. LDG assigns each vertex to the partition with the most neighbors, penalizing larger partitions [79]. FENNEL[83] balances neighbor co-location with non-neighbor separation, offering insights into identifying dense communities, a concept we extend to edge-based partitioning. Adil et al.[18] optimize memory for vertex-based partitioning on trillion-edge graphs by compressing leader structures. CUTTANA [34] reduces replication by buffering the graph to retain global context before partitioning. Though we use edge-based partitioning, we also maintain compact leader state for fast, high-quality decisions.

***Edge-based Partitioning.*** Edge-based partitioning was popularized by *PowerGraph* [31], which proposes a greedy algorithm to place an incoming edge on the partitions where its incident vertices have been placed earlier. It offers a distributed implementation and maintains $O(|V|)$ size state for decision-making at the leader. Our simplest heuristic approximates this behavior using Bloom filters, taking $O(c \cdot k)$ space complexity for a $k$-way partitioning, where $c$ is a large constant representing the Bloom filter bits and $c < |V|$.

*Grid* [43] and *PDS* [2] are stateless hash-based heuristics with an upper bound on vertex replicas but do not account for the graph topology. Our heuristics consider the structure of the incoming graph using state present at the leader, resulting in better quality partitioning. *ADWISE* [63] proposes a window-based algorithm that trades-off quality with latency while *CuSP* [38] partitions graphs maintained in distributed memory of large HPC systems. We focus on partitioning on Cloud and commodity clusters.

*Degree Based Hashing (DBH)* [87] and *High Degree Replicated First (HDRF)* [74] are SOTA edge-based streaming partitioners. DBH maintains the degree of both vertices for each edge using a hash table of size $O(|V|)$, and assigns the edge to a partition based on the lower-degree vertex's ID. HDRF replicates high-degree vertices to reduce replication and balances load using a scoring function, but communicates with all partitions per edge, leading to high overhead. Our proposed heuristics maintain a larger state at the leader than HDRF but smaller than DBH. The leader communicates with the partitions periodically to update its state. Thus, our time and space complexities fall between these two techniques. Also, DBH does not consider the graph structure at all while HDRF is optimized only to reduce the replication factor, ignoring the graph's topology. We empirically compare our heuristics with these contemporaries.

Hybrid techniques have been proposed [16, 25, 91]. *PowerLyra* [16] combines both edges and vertices to get hybrid cuts for partitioning based on the vertex degree. *Leopard* [40] performs vertex duplication with edge cuts and dynamic rebalancing of the partitions when elements are deleted. We consider append-only graph streams, seen in fintech, IoT and even social networks, with a static partition count. We leave dynamic repartitioning to future work.

## 2.3 Partitioning for Platforms and Analytics

Vertex-based platforms like Pregel [61], GraphX [33], Giraph Unchained [36], and TARIS [12] default to hash partitioning on vertex IDs but support custom partitioners. Dist-DGL [92], built for GNNs, uses METIS to localize training neighborhoods and reduce sampling overhead. Edge-based platforms like PowerGraph [32] and Graphbuild [43] prioritize load balancing and minimizing vertex replication. These systems can benefit from our community-preserving edge streaming partitioners, as demonstrated later with GraphX.

Maximizing the internal connectivity within each community can enhance information flow and a clustering-based approach is used to maximize connectivity and resiliency to vertex failures when partitioning a power grid [35]. Highly-connected subgraph clustering [37] partitions a graph while ensuring that the edge connectivity of each part is greater than half the number of vertices in that part. This has been successful in biological networks, including clustering cDNA fingerprints and grouping protein sequences [13]. These are vertex-based static graph partitioners, validated for specific applications. Instead we present an edge-based streaming graph partitioner which creates community-preserved partitions, validated on diverse graphs.

Clustering-based edge partitioners, like 2PS-L [66] and CLUGP [52], have been recently proposed to preserve local community structure, similar in spirit as us. However, these are not streaming partitioners but rather out-of-core ones, streaming the graph from disk and performing multiple passes over them (4 for 2PS-L, 3 for CLUGP) to identify community structure. We perform a single pass to achieve a truly streaming online partitioning. Our empirical comparison with CLUGP also shows us better or comparable on triangle counts but much better in reducing vertex replicas.

Machine learning (ML) has been applied to partitioning, treating it as a classification problem. These tend to be vertex centric and

non-streaming [28, 56, 69], and only consider the usual objectives of load balancing and minimizing cuts and not community preservation. E.g., GAP defines a differentiable formulation of the cut objective as a loss function for to learning of a neural network that predicts the partition for a vertex [69]. GCNSplit [93] does online unsupervised edge classification to a partition using an inductive Graph Convolutional Network (GCN). They use just $< 1MB$ of memory but create more vertex replicas than HDRF. They too do not preserve communities.

# 3 PROBLEM FORMULATION

## 3.1 Preliminaries

Let $G = \langle V, E \rangle$ be a graph, where $V = \{v_1, v_2, \ldots, v_n\}$ is the set of vertices, with $|V| = n$, and $E = \{e_1, e_2, \ldots, e_m\} \subset V \times V$ is the set of undirected edges, with $|E| = m$. This graph arrives incrementally at a *leader* machine as a stream of undirected edges, $S = [e_1, e_2, ..., e_m]$. Each edge in the graph appears exactly once in the input stream and the arrival order is arbitrary, e.g., creation time, traversal order.

The goal of an *edge-based streaming graph partitioner* is for the leader to divide the edge stream $S$ into $k$ subgraphs, $\mathbb{S} = \{S_1, S_2, ..., S_k\}$, each formed from a subset of the stream as the edges arrive. An arriving edge is placed on exactly one of the *Worker* machine $W_i$ that holds the *partition* $S_i$, i.e., $\forall i,\ S_i \subset E,\ \bigcup S_i = S$, and $\forall i, j,\ S_i \cap S_j = \emptyset$. $k$ is static and given *a priori*.

The *first goal* of a high-quality partitioning is *edge balancing*, i.e., $|S_i| \approx \frac{m}{k}$, to allow balanced load on the partitions [14, 47]. Each worker can handle $\approx \frac{m}{k}$ load even as the partition sizes grow, e.g., by scaling up VMs. When different edges incident on the same vertex $v$ are sent to different partitions, the vertex $v$ is *replicated* on each partition. If the number of replicas for a vertex $v$ is $r(v)$, with $1 \leq r(v) \leq k$, the *vertex replication factor* is $\rho = \frac{\sum_{v \in V} r(v)}{|V|}$, with $1 \leq \rho \leq k$. The *second goal* is to *minimize the replication factor*. This reduces data duplication in different partitions and communication across partitions during distributed computation [31].

This is illustrated in Fig. 1 for two different partitioning strategies over a stream of 10 edges placed into two parts.

## 3.2 Preserving Communities

*Local triangles* are 3-cycles of edges that are wholly present within a partition. Let the count of local triangles in partition $i$ be $\tau_i$. These are a subset of the *global triangle count* $\overline{\tau}$ present in the unpartitioned graph, i.e., $\tau \leq \overline{\tau}$. Partitioning the graph can eliminate triangles. E.g., in Figure 1, there are three triangles in the input graph, $\{1, 2, 4\}$, $\{2, 3, 4\}$, and $\{2, 7, 8\}$. However, after the edge-balanced partitioning, only one of these is *preserved*, $\{1, 2, 4\}$ in Partition 1. The *total number of local triangles* in the partitioned graph is given by $\tau = \sum_{i \in k} \tau_i$. The *ratio of triangles preserved* is $\widehat{\tau} = \frac{\tau}{\overline{\tau}}$.

Hypothesis 3.1. *Preserving triangles helps preserve communities.*

*Discussion:* By definition, vertices within a community are more connected within the community than to vertices outside the community. Let the probability of an edge between two vertices in the same community be $p$ and probability of an edge between vertices of different communities be $q$. By definition, $p \gg q$. The expected number of triangles in a community $C$ with $n$ vertices is $(T_{in}^C) = ({}^nC_3) \cdot p^3$. The probability of a triangle existing across communities is either $(T_{out}) = ({}^nC_3) \cdot q^3$, if the three edges are spread across three communities, or $(T_{out}) = ({}^nC_3) \cdot pq^2$, if two edges are in one community and the third in a different one. Since $q \ll p$, $P(T_{in}) \gg P(T_{out})$. So, there is a high chance that all three vertices forming a triangle belong to the same community. So, preserving triangles in a partition can preserve community structure. ∎

Triangle counting is the "basic building block" [90] of many community detection algorithms [29, 71] and is well-studied [49, 76, 90]. The existence of triangles is shown to be important for forming complex networks with an underlying community structure [49], e.g., the presence of cliques or near-cliques [17, 77]. Similarly, a good community is defined as a set of vertices that is dense in terms of triangles [71, 76]. As we show next, metrics to quantify the goodness of clusters created by community detection algorithms, LCC and transitivity, are proportional to triangles in the graph.

Hypothesis 3.2. *If two partitioning methods have similar vertex replication ($\rho$) and edge balancing, the one preserving more triangles ($\tau$) will have a higher Local Clustering Coefficient (LCC).*

*Discussion:* Let $\tau(u)$ be the number of triangles a vertex $u$ is part of in partition $i \leq k$ and $d(u)$ be its degree. The *LCC* of vertex $u$ in partition $i$ is $lcc_i(u) = \frac{2.\tau_i(u)}{d(u)(d(u)-1)}$ and the *average clustering coefficient* of the partition $i$ is $\overline{lcc_i} = \frac{\sum_{u \in V_i} lcc_i(u)}{|V_i|}$. Since both partitioning methods are edge-balanced, each partition will have $\approx \frac{|E|}{k}$ edges, and since both have the same vertex replication factor, the number of vertices in each partition will be similar, $\approx \frac{\rho \cdot |V|}{k}$. Therefore, the average edge degree of each partition will also be same. Hence, if the triangles preserved in one partition, $\tau_i$, increases, then $\overline{lcc}_i$ also increases, assuming a similar degree distribution $d(\cdot)$ for vertices in the partitioned graph under both methods. ∎

At the same time, there is also a trade-off. If $\rho$ decreases, this will cause LCC to decrease since the average edge degree in a partition will increase. We later empirically confirm both of these in §6.3.4.

Hypothesis 3.3. *Preserving triangles increases Transitivity.*

*Discussion:* Transitivity of a graph is $Trans = \frac{3.\tau_i}{|triplets|}$ where $|triplets|$ is a set of three vertices connected by two or more edges. Since a triangle contributes three triplets, the numerator is multiplied by 3. So, transitivity can also be defined as $Trans = \frac{3.\tau}{3.\tau + |o\_triplets|}$, where $|o\_triplets|$ are the number of open triplets having a path but not forming a triangle. Let a graph partitioner $M$ that preserves the number of triangles have transitivity of $Trans_i = \frac{3.\tau_i}{3.\tau_i + |o\_triplets_i|}$ for partition $i$. Let an alternate partitioner $M'$ for that same graph give the same number of vertices and edges in a partition but not preserve as many triangles, with a transitivity of $Trans'_i = \frac{3.\tau'_i}{3.\tau'_i + |o\_triplets'_i|}$, for partition $i$, and with $\tau_i \geq \tau'_i$.

We show that $Trans_i \geq Trans'_i$. Let $T(u, v, w)$ be a triangle preserved in partition $S_i$ of $M$ but not in partition $S'_i$ of $M'$. There are two possible cases for $T$. (1) Two of the edges of $T(u, v, w)$ are present in partition $S'_i$. Here, the denominator in $Trans'_i$ increases by 1 due to $T$ and the numerator remains the same, thus reducing $Trans'_i$ relative to $Trans_i$. (2) Only one of the edges of $T(u, v, w)$ is

part of $S'_i$. Here, the transitivity of the partition is not affected by $T$. Therefore, for every missing triangle in $S'_i$ with respect to $S_i$, the transitivity of $S'_i$ reduces or remains the same. Further, in a fraction, if the numerator and the denominator are increased by the same quantity, the fraction increases. Since $\tau > \tau'$ and $S_i$ and $S'_i$ have same number of edges and vertices, we have $Trans_i \geq Trans'_i$. ■

### 3.3 Optimization Problem

Since preserving triangles is necessary to enhance the community structure, we now fully define our problem with this additional objective.

***Problem Statement.*** The objectives of a community-preserving edge-based partitioner are to partition the stream of input edges $S$ into $k$ partitions such that we:

(1) Balance the edges across partitions within a *load-balancing factor* $\epsilon$: $\forall i,\ |S_i| \in [(1-\varepsilon)\frac{m}{k},\ (1+\varepsilon)\frac{m}{k}]$ ;
(2) Minimize the vertex replication factor, $\rho$; and
(3) Maximize the triangle count ratio, $\widehat{\tau}$.

E.g., in Figure 1, the triangle-preserving partitioner (bottom-right) achieves a triangle count ratio of $\frac{2}{3}$, preserving two of the three global triangles, compared to just $\frac{1}{3}$ for the edge-balanced partitioner, while balancing edges and having the same $\rho$.

We now formally analyze our objective function. Say, we have an input graph $G\langle V, E\rangle$, generated uniformly randomly, with the probability of generating an edge being $q$. Let us assume that the largest clique in the graph is of size $\bar{n}$.

Theorem 3.1. *In an edge-based partitioner, when vertices are replicated ($\rho > 1$), some triangles will be missed ($\widehat{\tau} < 1$).*

*Proof:* When an edge that is part of a clique results in a vertex replication, it can break one or more triangles in the clique. Say, $K \in G\langle V, E\rangle$ is a clique of size $\bar{n}$ and $\bar{V}$ is its set of vertices. $K$ has $\frac{\bar{n}\cdot(\bar{n}-1)}{2}$ edges. The number of unique triangles in the clique is ${}^{\bar{n}}C_3$, and each edge is part of $(\bar{n}-2)$ triangles in the clique. So, a missing edge in the clique can result in missing up to $(\bar{n}-2)$ triangles.

Say, the replication of a vertex in $K$ results in a missing triangle with probability $\sigma$ over the entire stream. So, for a replication factor of $\rho$, $\rho - 1$ extra copies of a vertex exist, resulting into loss of up to $\sigma \cdot (\rho-1) \cdot (\bar{n}-2)$ triangles from the clique. But, if an edge $e(v_i, v_j) \notin K$, involving $v_i \in K$ and $v_j \notin K$ is assigned to partition $S_{k \neq l}$, this will not result in a missing triangle. ■

This highlights that for any replication factor, $\rho > 1$, a fraction of triangles are bound to be missed in $k$-way partitioning of the graph, irrespective of the partitioning strategy. In the rest of the discussion, we ignore the factor $\sigma$ as it is not central to our argument.

Theorem 3.2. *A higher vertex replication factor will cause a higher number of triangles to be missed.*

*Proof:* Since the clique has $\bar{n}$ nodes and an edge is adjacent to 2 vertices, the loss in total triangles due to vertex replication is bounded by $((\rho-1)\cdot(\bar{n}-2)\cdot\frac{\bar{n}}{2})$ triangles, i.e., $O((\rho-1)\cdot\bar{n}^2)$ triangles are lost. Say, $\mathcal{N}(\bar{n})$ is the number of cliques of size $\bar{n}$ in the graph. Since, the probability of an edge being present in the graph is $q$ and a clique must contain all the edges, all $O(\bar{n}^2)$ edges must be present. So, the probability of a clique of size $\bar{n}$ is $q^{\bar{n}^2}$. Hence, the number of cliques of size $\bar{n}$ is $\mathcal{N}(\bar{n}) \approx O(|V|^{\bar{n}} \cdot q^{\bar{n}^2})$. Therefore, a total of $O((\rho-1)\cdot\bar{n}^2\cdot\mathcal{N}(\bar{n}))$ triangles will be missed for all the cliques of size $\bar{n}$ in the graph due to a $k$-way partitioning with replication factor $\rho$. The expected number of total triangles missed in the graph, $T_{miss}$, for cliques of all sizes $s$ due to replication is:

$$T_{miss} = O(\Sigma_{s=3}^{\bar{n}}((\rho-1)s^2\mathcal{N}(s)) \quad (1)$$

$$\approx O(\Sigma_{s=3}^{\bar{n}}((\rho-1)s^2|V|^s q^{s^2})) \quad (2)$$

$$\approx O\Big(\frac{(\rho-1)|V|q}{q^4(1-q^2)^3}\Big) \quad (3)$$

We omit the intermediate steps from Eqn. 2 to Eqn. 3 for brevity.

The total number of triangles expected in the graph are:

$$T_{tot} = \Sigma_{s=3}^{\bar{n}}({}^sC_3\mathcal{N}(s)) \approx O(\Sigma_{s=3}^{\bar{n}}(s^3\mathcal{N}(s))$$

$$\approx \Sigma_{s=3}^{\bar{n}}({}^sC_3|V|^s q^{s^2}) \approx O\Big(\frac{|V|q}{q^6(1-q^2)^4}\Big)$$

Hence, the fraction of missing triangles is:

$$\frac{T_{miss}}{T_{tot}} = O((\rho-1)q^2(1-q^2))$$

$$\approx O((\rho-1)q^2), \quad \text{for } q \ll 1$$

Hence, the fraction of missing triangles is higher for a high replication factor in edge-balanced partitioner. Further, for a given replication factor, the fraction of missing triangles is more if the graph is dense, which is also intuitive. ■

This theorem proves that we can improve the quality of graph partitioning, i.e., the retained communities due to number of triangles preserved, by improving the replication factor. So, **our objectives of minimizing the replication factor while simultaneously increasing the triangle count are not contradictory**, and hence there exists a solution. This is confirmed in our experiments, where for our algorithm, the replication factor is smaller in some cases where the triangle count ratio is more, and vice versa.

The second insight revealed by the theorem is that it is non-trivial to design such an algorithm. Although the two objectives support each other, existing algorithms that **optimize just the vertex replication factor do not *automatically* result in increased triangle counts**. As shown by our experiments, the SOTA algorithm HDRF consistently performs worse than our algorithm by 3× to 5× in terms of triangles preserved for diverse graphs, even though it matches or marginally does better on replication factor. While we can improve the community structure by improving the replication factor, there is still a compelling need to design a non-trivial partitioning algorithm to specifically enhance triangle preservation. We propose a novel algorithm to address this gap.

## 4 SYSTEM ARCHITECTURE

TriParts is a distributed partitioner with one *leader machine* and $k$ *workers* for a $k$-way partitioning (Fig. 2). Each worker holds one partition. While by default each worker is on a separate machine, we can configure multiple workers per machine, e.g., allowing workers to be migrated to new machines as their partition sizes grow.

The leader receives an input stream of edges from an external source which is accessed through a FIFO *Reader Queue* by the leader. The leader's partitioning logic decides the partition to which each edge should be assigned to, and then sends that edge to the worker hosting that partition. The leader exploits concurrency using a

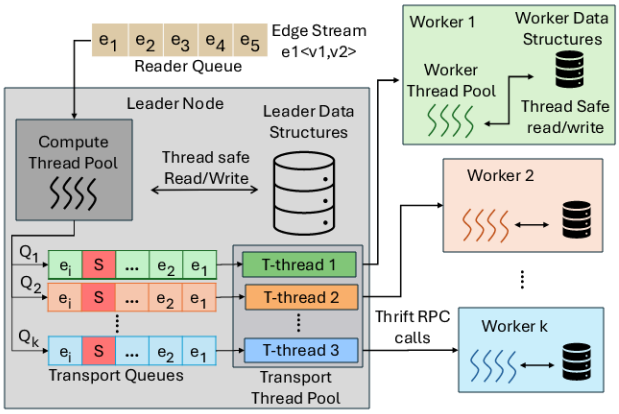



**Figure 2: TriParts streaming partitioner architecture**

*compute thread pool*, whose threads can access the read queue and in-parallel decide on the placement of edges with high throughput. This can have some impact on the quality of partitioning but this amortized across the long stream of edges (millions–billions) except for adversarial scenarios. It also has $k$ *transport queues*, one per worker, to send edges assigned to them asynchronously. The leader maintains three *state* data structures – a *Bloom filter* per partition, a *Triangle map* and a *High Degree map*, discussed in § 4.2.

Each worker also uses a *thread pool* to handle the stream of edges assigned to it by the leader. It adds the received edge to the local partition maintained in-memory as an adjacency list. It also maintains a *local* triangle map data structure. The number of threads in the leader compute and worker thread pools are configurable.

## 4.1 Workflow

The reader queue is unbounded, populated from the incoming edge stream (or from an external file, in our experiments). It is a thread-safe Java `ConcurrentQueue` and uses atomic operations, allowing all the compute threads to concurrently remove the edges from the queue. Each thread independently executes the partitioning heuristic (introduced in § 5) using the local state information to assign the edge to one of $k$ partitions. Once decided, it may also update the thread-safe state data depending on the heuristic.

The edge is then inserted into the respective transport queue for the partition and the compute thread moves on to read the next edge from reader queue. Multiple compute threads may be inserting into the same transport queue, again requiring a concurrent queue. Meanwhile, the transport thread removes an edge from its queue for its partition and sends it to the worker using an asynchronous RPC call. We use Apache Thrift's *fire and forget* mode that maintains an implicit receive queue at the worker.

A thread in the worker's thread pool is invoked with an incoming edge, and acquires a lock to update the *local adjacency list* for the partition. It also calculates and updates the *local triangle map* for each edge, which maintains the number of local triangles that this edge is participating in. Concurrent threads at the worker do not affect the quality or correctness.

The leader periodically *synchronizes* and updates its state information from the workers, at a configurable interval of $\sigma_e$ edges. After reading and processing $\sigma_e$ edges from the reader queue, one of the compute thread serves as a *coordinator* for the sync. It pauses all compute threads after they finish handling their current edge, and inserts a *sync message* into the transport queue for all partitions ($S$ in red box in Fig. 2). When a worker thread receives a sync message, it too pauses all its local threads after they process their current edge, after which it calculates the *high-degree map* for the local partition; unlike the triangle map, this is done on-demand since the definition of a "high-degree" vertex changes over time and decided by the leader. Both the pre-computed triangle map and the high degree map are sent to the coordinator thread at the leader. Once all workers respond, the coordinator updates the leader's triangle and high degree maps, and resumes all its compute threads.

These threads, concurrent data structures and locks are carefully designed, selected and scoped to be fine-grained to minimize overheads and maximize concurrency that can benefit from many-core CPUs, as evaluated in § 6.3. These architectural choices help exploit data parallelism across edges at the leader, across workers for different partitions, and across edges within a worker. We also leverage pipelining of the partitioning logic, state updates and edge transfers. These limit the execution time per edge to $O(10\ \mu s)$ and achieves high scalability.

## 4.2 States Maintained at the Leader

Streaming graph partitioners need to operate with low latency, processing thousands of edges per second. So, the network communication between the leader and the workers should be minimal during decision-making. The leader makes decisions using the local states it maintains. At the same time, the leader has limited memory to hold the state and cannot retain the complete graph seen so far. Even if possible, the cost for traversals over the full graph is high. *Thus, the choice of the states maintained at the leader and how they are used affect the quality, speed and scalability of partitioning large graphs.* We next discuss these states and associated data structures at the leader used by our partitioning heuristics (Fig. 3).

*4.2.1 Bloom Filters (BF).* One of the quality-metrics of partitioning is the vertex replication factor. When a new edge arrives, its two incident vertices may already be present in some partitions. Knowing such partitions helps the heuristic prioritize them when deciding where to place the new edge, and avoid additional replicas.

Others use a hashmap to track every vertex to the partitions with their replicas; this takes $O(k \cdot |V|)$ space [87]. Instead, we use *Bloom filters*, which are compact but approximate data structures [26]. We maintain one filter per partition and insert vertices into them when assigning an edge to a partition. We later check each Bloom filter for the presence of each vertex of an incoming edge.

Bloom filters have fast *constant-time* insertion and lookup speeds. Their lookups have *zero false negatives* but can have a *small false positive rate*. However, for a given false positive rate and count of entries in the filter, its space required is smaller than the number of items stored. If $n$ vertices are inserted in a Bloom filter and $f$ is the false positive rate, the size of the filter is $\frac{-n \cdot ln(f)}{ln(2)^2}$ *bits* [26].

Conceptually, we treat the Bloom filters for the partitions collectively as a "map" data structure that, given a vertex, returns the set of workers on which its replicas *may be* present. E.g., in Fig. 1, the Bloom Filter for Part. 1 of the community preserving partitioner

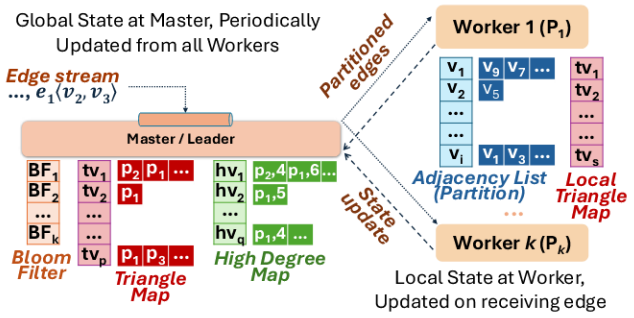



**Figure 3: Data structures maintained on leader and workers**

will have entries for vertices $v1, v2, v3, v4$ while for Part. 2 will have $v2, v3, v5, v5, v6, v7, v8$. The decisions taken by our heuristics will not violate the correctness of partitioning even with false positives.

*4.2.2 Triangle Map (T).* The novel partitioning quality metric we introduce is the number of local triangles in the partitions.

HYPOTHESIS 4.1. *Sending an edge $e(v_i, v_j)$ to the partition where $v_i$ or $v_j$ already forms triangles, increases the preserved triangle count.*

*Discussion:* Since a triangle is the smallest clique possible and is present in all larger cliques, if a vertex $v_i$ is part of one or more local triangles in a partition $S_x$, it indicates a closely-knit neighborhood around $v_i$ increasing the chance of community presence in its locality. Further, if vertex $v_i$ is part of a local community, it will have more neighbors within the community than outside due to strong locality properties of dense communities. So, there is high chance that $v_j$ also belongs to the same community. Hence, sending the edge $e(v_i, v_j)$ to partition $S_x$ can enhance its community structure and increase the chance of forming new local triangles. ■

We implement this intuition using a *Triangle Map* data structure at the leader that maps vertices (key) to the list of partitions on which they form triangles (value). E.g., in Fig. 1, $T(v_1) = \{P_1\}$ while $T(v_2) = \{P_1, P_2\}$ as it forms a triangle in both these partitions. The leader by itself cannot determine if an input edge indeed forms a triangle when placed on a partition. Instead, it periodically *syncs* with the workers to get this. As discussed, each worker updates its local triangle map for an incoming edge $e\langle v_i, v_j\rangle$ (Fig. 3). The worker thread performs a set intersection of the neighbors of $v_i$ and $v_j$ to check if new triangles are formed. This takes $O(\min(\deg\{v_i\},\deg\{v_j\}))$. A single edge can form multiple local triangles, e.g., adding $\langle v_2, v_4\rangle$ to $P_1$ forms 2 new local triangles. When the leader sends a *triangle map sync request*, e.g., after every $\sigma_e = 10\%$ edges are seen by default, each worker returns vertices that were newly added to its triangle map and the leader updates its map. As we show in § 6.3, real-world power-law graphs are sparse and only a small fraction of vertices participate in triangles. So the triangle map grows sub-linearly with input stream.

*4.2.3 High Degree Map (H).* Power-law graphs have a small fraction of vertices with very high degrees.

HYPOTHESIS 4.2. *Sending an edge $e(v_i, v_j)$, where $v_i$ is a high-degree vertex, to the partition containing $v_i$, increases the total preserved triangle count ($\tau$).*

*Discussion:* Most SOTA partitioners including HDRF and DBH [74], replicate high-degree vertices to achieve better load balancing. On the contrary, we claim that *not replicating high-degree vertices, increases* $\tau$, while still achieving edge balancing. In knowledge graphs, communities are formed based on common interests. Here, high-degree vertices like $v_i$ represent influential vertices, which seed the community. Its neighbors share a common interest (common neighbor $v_i$) and are likely to belong to the same community and hence be connected, forming a triangle. So, we avoid replicating high-degree vertices and instead consolidate edges incident on them to one partition to conserve triangles and maximize $\tau$. ■

We maintain a *High-degree Map* data structure on the leader that maps from a high-degree vertex to the partition(s) on which it has a high-degree, and their actual degree. A vertex $v$ is *high-degree* in a partition if $d(v) > \eta \times \delta$, where $d(v)$ is the local degree of $v$ on that partition, $\delta = \frac{|E|}{|V|}$ is the current *average degree* for all vertices seen at the leader, and $\eta > 1$ is a constant; we default to $\eta = 2$ in our experiments. A vertex that was once a high degree may later not meet this threshold if the average degree of the graph increases.

While the leader can independently compute $\delta$ from the input stream, it solicits the high-degree vertices from each workers during its sync by sending it the current $\delta$. Each worker computes afresh and returns the HD vertices that meet the threshold, along with their actual degree. In our runs, these form only $\approx 4\%$ of the vertices of the graph. Besides the default sync interval of $\sigma_e$, the leader sends an additional *HD sync request* when $\delta$ changes by 1.0 at the leader.

## 4.3 Load Balancing

Adding more edges to a partition increases the chance of finding more local triangles and reduces the vertex replication. But this can skew the load on a partition and violate the edge-balancing goal. We avoid this by defining a *dynamic load threshold* for each Worker that incrementally grows as edges arrive. Our heuristics use this to avoid overloading partitions and to keep them balanced.

We exploit the $(1 \pm \varepsilon)$ imbalance factor allowed on the final partition capacity of $\frac{m}{k}$ to trade-off the partition quality and load balancing. The leader has a simple count of the edges sent to each worker. It also maintains the current *load threshold* $\lambda_l, \lambda_h \in (0, (1+\varepsilon)\frac{m}{k}]$ for each worker, with $\lambda_h \geq \lambda_l$. Here, $\lambda_l$ is the current *pessimistic* (or lower) limit, and $\lambda_h$ the *optimistic* (or higher) limit for the worker.

A worker's load count must stay below the pessimistic limit $\lambda_l$ while assigning a new edge to a worker *which does not contain any of its incident vertices*. This avoids skewing the partition load if the qualitative benefit of placing the edge on a partition is low. We allow the higher limit $\lambda_h$ for a partition when *either or both the vertices of an input edge are already present* on it. Thus, we use the relaxed threshold if it improves the chances of finding local triangles by selecting the best partition despite a skew. Both $\lambda_l$ and $\lambda_h$ are increased by a step size of $\xi_l$ and $\xi_h$ by the leader for all partitions when all have exhausted their thresholds.

# 5 PARTITIONING HEURISTICS

We propose four partitioning heuristics that incrementally use these state data structures. The leader uses the three state data structures, Bloom Filter "Map" ($BF$), Triangle Map ($T$) and High-degree Map ($H$) to make the partitioning decision When a new

**Algorithm 1** Primary decision points for BTH heuristic

```
1: procedure ComputePartition(v_i, v_j)
2:     if T(v_i) ∩ T(v_j) ≠ φ then
3:         return GetPartFromTIntersect(v_i, v_j)
4:     if T(v_i) ≠ φ && T(v_j) ≠ φ && T(v_i) ∩ T(v_j) = φ then
5:         return GetPartFromTUnion(v_i, v_j)
6:     if T(v_i) ≠ φ && T(v_j) = φ then          ▷ & vice versa
7:         return GetPartTIndiv(v_i, v_j)
8:     if H(v_i) ∩ H(v_j) ≠ φ && T(v_i) = φ && T(v_j) = φ then
9:         return GetPartFromHIntersect(v_i, v_j)
10:    if H(v_i) ≠ φ && H(v_j) ≠ φ && H(v_i) ∩ H(v_j) = φ then
11:        return GetPartHUnion(v_i, v_j)
12:    if H(v_i) ≠ φ && H(v_j) = φ then          ▷ & vice versa
13:        return GetPartHIndiv(v_i, v_j)
14:    ⟹ T(v_i) = φ && T(v_j) = φ && H(v_i) = φ && H(v_j) = φ
15:    return GetPartFromBF(v_i, v_j)
```

edge $e(v_1, v_2)$ arrives, each of its two incident vertices may have an entry in the three maps, $BF(v_1), BF(v_2), T(v_1), T(v_2), H(v_1)$ and $H(v_2)$. When $BF(v_1) \cap BF(v_2) \neq \phi$, the partitions that these two vertices are already present on may overlap and we can avoid creating new replicas by collocating them. If $T(v_1) \cap T(v_2) \neq \phi$ and/or $H(v_1) \cap H(v_2) \neq \phi$, the vertices are also part of a triangle and/or have a high-degree in the overlapping partitions. This can improve their triangle conservation. We also test for intersection across map types. This forms our intuitive approach.

The states used and the order for testing these partition-overlaps forms the basis for each heuristic. We prioritize: *(1) enhancing the community structure, (2) avoiding replication of vertices and (3) balancing of edges across partitions.* We prefer partition overlaps in T-Map, as local triangles have already formed, over H-Map, which indicates future potential to form triangles.

***Bloom Filter, Triangle Map and High-degree Map (BTH).***

This is the core heuristic that uses all three maps to select a partition for an edge. Alg. 1. gives the *six primary decision points* taken by this heuristic, which narrows the choices based on partitions common to both vertices of an edge in the T-Map, H-Map or BF, in that order. For an edge $\langle v_1, v_2 \rangle$, we first test if partitions in their T-Map intersect (Alg. 1, line 2) and return the least loaded among them — if it has capacity (Alg. 2). Else, if either vertex forms triangles on partitions but they do not overlap (Alg. 1, line 4), we use a variant of Alg. 2 (not shown) to check if the triangle partitions of a vertex intersect with the high-degree partitions for the other. If multiple triangle partitions match (Alg. 2), we prioritize those that also have a high-degree vertex, giving preference to the partition with a higher sum-of-degrees for the vertices. We break ties by marking the first partition a vertex forms a triangle on as its "home" partition and using it by default to improve co-location.

In the absence of any triangle partitions, we select from overlapping high-degree partitions for the vertices (Alg. 1, line 6). Failing this, we chose from common partitions in their BF to avoid replication (Alg. 1, line 11). If even the BF partitions do not intersect, we pick the least loaded of the partitions having either of the vertices, which may cause the other vertex to replicate. If all tests fail, we send the edge to the least loaded partition. In all cases, if a partition's high and low thresholds are reached, as applicable, the partition is removed from consideration.

*Time complexity analysis:* For a vertex $v_i$, an entry in T-map or H-map contains a subset of partition IDs. So, $T(v_i)$ and $H(v_i)$ can contain at most $k$ values. Partitioning heuristics calculate union and intersections on these subsets to choose one partition. Therefore, the time complexity per edge is $O(k)$.

**Algorithm 2** Secondary decisions *(△ vertices are in same part.)*

```
1: procedure GetPartFromTIntersect(v_i, v_j)
2:     T_ij = T(v_i) ∩ T(v_j)
3:     H_ij = H(v_i) ∩ H(v_j)
4:     if T_ij ∩ H_ij ≠ φ then
5:         return GetMaxDegSumPartition(T_ij ∩ H_ij)
6:     if T_ij ∩ (H(v_i) ∪ H(v_j)) ≠ φ then
7:         return GetMaxDegPartition(T_ij ∩ (H(v_i) ∪ H(v_j)))
8:     if H_ij ∩ (T(v_i) ∪ T(v_j)) ≠ φ then
9:         return GetMaxDegSumPartition(H_ij ∩ (T(v_i) ∪ T(v_j)))
10:    if T_ij ≠ φ then
11:        return GetLeastLoadedPartition(T_ij)
12:    if H_ij ≠ φ then          ▷ only if there is no space available above
13:        return GetMaxDegSumPartition(H_ij)
14:    if BF(v_i) ∩ BF(v_j) ≠ φ then
15:        return GetLeastLoadedPartition(BF(v_i) ∩ BF(v_j))
16:    if BF(v_i) ∪ BF(v_j) ≠ φ then
17:        return GetLeastLoadedPartition(BF(v_i) ∪ BF(v_j))
18:    return GetLeastLoadedPartition(*)
```

***Bloom Filter with Triangle Map (BT).*** In this variant, we include the T-Map to the leader's state but not the H-Map. This behaves as if the H-Map is empty, allowing us to scope the benefits of T-Map alone.

***Bloom Filter with High Degree Map (BH).*** In this heuristic, we include the H-Map to the leader's state but not the T-Map. This behaves as if the Triangle Map is empty, and helps evaluate the benefit of maintaining the H-Map alone.

***Only Bloom Filter (B).*** This variant only uses BF and offers a simple baseline. It only avoids vertex replication by collocating edges on partitions that may already have replicas of both vertices, i.e., picks from $BF(v_1) \cap BF(v_2)$. The least loaded partition from these is chosen. This does not aim to enhance community structure but offers fast decisions using a compact state that mimics partitioners like PowerGraph.

## 6 EXPERIMENTS

In this section, we report detailed experimental results and analysis of our TriParts streaming edge-based partitioning heuristics and SOTA baselines. Our empirical claims are:

- *Superior Quality*. For diverse graphs and edge stream orderings, our heuristics out-perform SOTA on local triangle count, are comparable or better on vertex replicas, and achieve edge-balancing.
- *High Scalability*. Varying the compute threads on the leader improves the partitioning throughput linearly with has minimal impact on the partitioning quality. We have modest memory overheads on leader, low locking overheads on workers and low sync overheads between leader and workers.

### 6.1 Setup

***System Setup.*** The experiments are run on a commodity cluster (*sirius*) with compute nodes having an Intel Xeon Gold 6226R CPU with 16 cores@2.9$GHz$, 128$GB$ RAM and 10$Gbps$ Ethernet. Leader and workers each run on a compute node with CentOS v7, Java v8 and Apache Thrift 0.13.0.

**Table 1: Graph datasets used for experimental evaluation**

| Graph | $V$ | $E$ | $deg_{avg}$ | $T$ | $\frac{T}{V}$ | $\frac{T}{E}$ | NW Type |
|---|---|---|---|---|---|---|---|
| **USRN** [75] | 23.9$M$ | 28.9$M$ | 2.4 | 438.8$K$ | 0.02 | 0.02 | Road/Planar |
| **Orkut** [89] | 3.1$M$ | 117.2$M$ | 76.28 | 627.6$M$ | 202.4 | 5.35 | Social |
| **DBPedia** [7] | 18.3$M$ | 126.9$M$ | 13.89 | 328.7$M$ | 17.96 | 2.6 | Knowledge |
| **Brain** [5] | 693$K$ | 133.7$M$ | 385.86 | 17.1$B$ | 24675 | 128 | Brain |
| **MAG** [24] | 67.4$M$ | 1.03$B$ | 17.75 | 1.61$B$ | 13.79 | 1.55 | Biblio. |
| **Twitter** [15] | 52.6$M$ | 1.61$B$ | 30.70 | 55.4$B$ | 499.1 | 34.3 | Social |

***Datasets.*** We evaluate the partitioners on a diverse set of large *real-world graphs*, summarized in Table 1. They are carefully chosen to span knowledge graphs (DBPedia, MAG), social networks (Twitter, Orkut), road network (USRN) and neuro-science (Brain) domains, and have varying sizes, degrees, triangles counts, and planar or power-law degree distributions. Later, we complement these with synthetic graphs with diverse distributions. The graph stream is generated from a *random* permutation of the edges, as is common [2, 74, 79], and the same ordering used across partitioners. We also generate a *BFS ordering* [74, 79] where a vertex is selected randomly, and the edge order generated as a BFS traversal from it.

***Baselines.*** Streaming edge partitioners can be classified as *heuristic driven* [32, 64, 74], *hashing-based* [32, 87], and *clustering-based* [52, 66]. We compare TriParts with one SOTA baseline from each.

- **HDRF** [73, 74]. *Highest Degree Replicated First (HDRF)* partitioner is tailored for power law graphs. It uses a heuristic which targets workloads with highly skewed graphs. The key idea is that since power-law graphs have a few high degree vertices and many low degree vertices, it is beneficial to prioritize replicating high degree vertices to radically reduce the number of vertex cuts.
- **DBH** [73, 87]. *Degree-based Hashing (DBH)* also prioritizes replicating vertices that have the highest degree. However, unlike HDRF, DBH employs hashing for partitioning. For an input edge $e$, DBH computes the partial degree of its endpoint vertices $v_1$ and $v_2$, $\delta_1$ and $\delta_2$. After that, $e$ is assigned to the partition ID computed as the hash of the vertex with lower degree.
- **CLUGP** [52]. This is a clustering-based, 3-pass edge partitioning. In pass 1, the input graph is processed to generate intermediate vertex clusters. In pass 2, these clusters are refined using a game-theoretic approach to map them to partitions. In pass 3, the vertex partitions are converted into edge partitions. We run the single-machine Java version that is available [51].

***Configurations.*** The leader is configured by default to use 32 threads while all worker nodes use 16 threads. The number of partitions $k$ created is same as the number of workers. The sync interval, $\sigma_e$, is configured by default to sync after reading every 10% of edges or when the global average degree of the graph changes by 1.0 for BH and BTH. Since BT does not have high degree maps it only syncs after reading every 10% of edges. We run the experiments for $k \in \{8,12,16\}$ partitions. We use $\lambda_l = (1 \pm 0.05)\frac{m}{k}, \lambda_h = (1 \pm 0.1)\frac{m}{k}$ and $\xi_l = 0.05\frac{m}{k}, \xi_h = 0.1\frac{m}{k}$. We run the SOTA baselines with similar leader and worker configurations.

All partitioners achieve good edge load balancing across workers within $\epsilon = 0.001$. We omit reporting these in the analysis.

## 6.2 Analysis of Partitioning Quality

Fig. 4 shows the performance of all four TriParts heuristics B, BH, BT and BTH and the two SOTA baselines DBH and HDRF when run on all 6 graphs on 3 different numbers of partitions. We limit BFS stream order to only BTH, DBH and HDRF. Figs. 4a and 4c show the *normalized percentage of triangles preserved locally* relative to BTH (bars on left Y axis) and the *vertex replication factor* ($\rho$, markers on right Y axis), when edges are ingested in random and BFS order. The bar-labels for BTH are the *% of triangles it preserves* ($\widehat{\tau}$%). Figs. 4b and 4d show the *end-to-end time (E2E)* to process the entire edge-stream (bars on left Y axis, log scale) and the *average throughput* (markers on right Y axis, millions) for the random and BFS order.

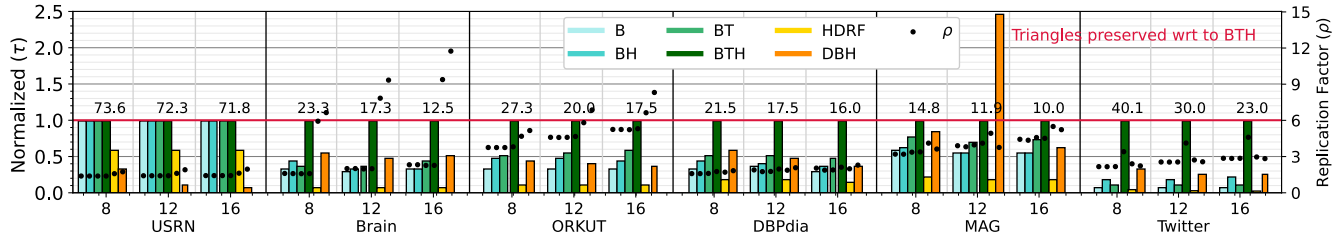
*6.2.1 Comparison within TriParts heuristics.* Figs. 4a and 4c show that both BH and BT increase the triangle count by 10–50% and 20–70%, respectively, when compared to B by using the high-degree map and triangle map. This supports our Hypotheses 4.1 and 4.2, and shows the importance of using these two topological entities in decision-making. For the USRN graph there is little difference across the heuristics due to the planar nature of the network.

Also, BT is marginally better than BH on vertex replication as it uses knowledge of vertices already forming triangles while BH is anticipating future triangles around HD vertices. Hence, while states are important, we prefer T-map over H-map for partitioning decisions in BTH (§5). Using both the maps together further boosts the triangle preserved by BTH by 200–250% compared to B.

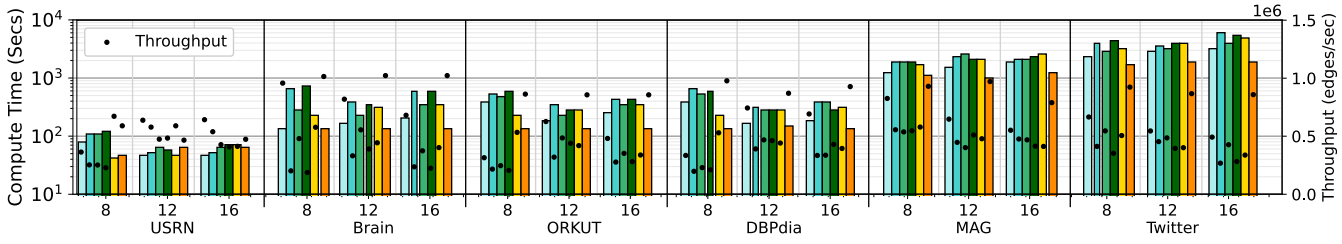
All the four heuristics uses the Bloom filter to ensure that we do not replicate vertices unless explicitly triggered by the load balancing constraint, or if triangles can be preserved. Hence, they all show similar vertex replication factors in Figs. 4a and 4c. Also, B does not have triangle preservation as an objective and only avoids vertex replication. BH, BT and BTH focus on triangle preservation but still achieve similar vertex replication as B. This supports our claim that the two objectives are not contradictory (Theorem 3.2). At the same time, as we see for Brain and DBPedia with BFS ordering, triangle preservation does not automatically follow from reducing vertex replication and these graphs show an inverse trend between the two metrics, as we propositioned in §3.3.

Since B has no sync operations and only uses Bloom filters, it is the fastest, as seen in Figs. 4b and 4d, achieving rates of over 1$M$ edges/s (Brain, 8 parts, BFS) and 650$k$ edge/s (MAG, 8 parts, random). In BT, the workers update their local triangle map as they receive edges in a pipelined manner, and transfer these triangle maps to the leader at 10% syncs. This causes BT to be 15–40% slower than B. Both BH and BTH uses high degree map, that is computed on-demand at the workers when the sync is triggered. The number of sync calls are also higher for BH and BTH since these are additionally performed when the high-degree threshold is breached. Hence, they are 30–80% slower than B.

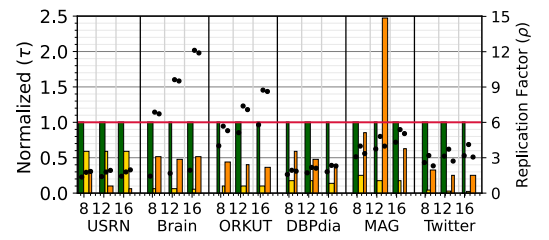
Overall, BTH preserves more triangles than the other heuristics while maintaining the same replication factor, and offers a competitive throughput, e.g., achieving 500$k$ edges/s (ORKUT, 12 parts, random) and 350$k$ edges/s (DBPedia, 16 parts, random). Hence, it is our recommended heuristic for TriParts and discussed further. As expected, as the number of partitions increase the percentage of conserved triangles decreases. This is accompanied by a consistent increase in the replication factor as the partitions increase. There is limited impact of the stream order, BFS or random, on the quality or performance. We focus future results on random order.



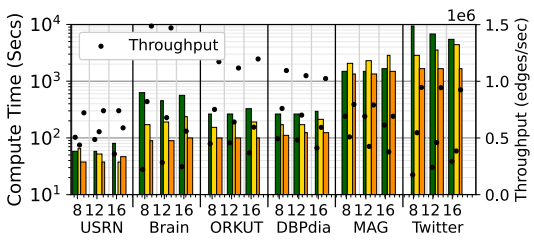

(a) Normalized Local Triangle Count and Vertex Replication Factor in *Random order*



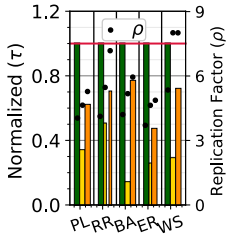

(b) Total compute time and throughput in *Random order*




(c) Normalized Triangle Count and Vertex Rep. Factor in *BFS order*




(d) Total compute time and throughput in *BFS order*




(e) Synthetic Graphs

**Figure 4: Performance of partitioning techniques on 6 graphs for different parts ($k$) with two different stream orders.**

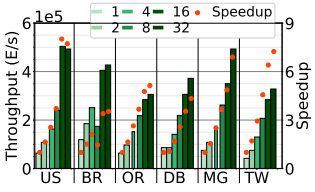
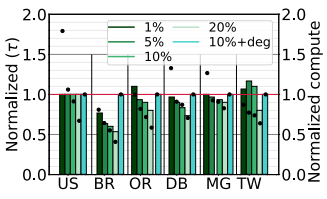
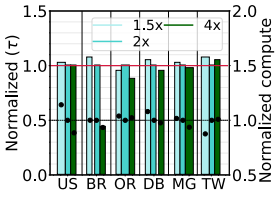



**Figure 5: Scalability with # of threads in leader.**

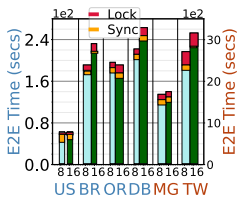



(a) Varying sync frequency

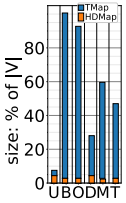



(b) Varying HD threshold

**Figure 6: Varying Sync and HD thresholds.**




(a) Lock and sync overheads.




(b) Memory

**Figure 7: TriParts Overheads**

*6.2.2 Comparison with SOTA Baselines.* BTH consistently outperforms *DBH and HDRF* baselines on *triangles preserved*, with 5–40× improvement over HDRF and 3–9× over DBH, for the 6 real-world graphs (Figs. 4a and 4c). These are consistent for both random and BFS ordering. BTH also has *similar or better replication factor* when compared to these baselines. In BTH, a vertex of an edge is replicated only if the other vertex helps preserve triangles, thus reducing the overall vertex replication factor and/or improving triangle counts. For USRN, DBPdia and Twitter using BFS, these baselines and BTH have similar values of $\rho$. But BTH shows 20–80% reduction in vertex replication for Brain, ORKUT and MAG.

The time taken by BTH is similar to or slightly higher when compared to HDRF (Figs. 4b and 4d). However, DBH is much faster than BTH and takes 2–4× less time. However, the throughput of $> 1M$ edges/s achieved by DBH is unlikely to be of real-world benefit and the throughput of $300k$–$550k$ edges/s achieved by BTH will suffice for most practical applications. Our simplest heuristic B preserves more triangles with a lower replication factor, compared to HDRF, and at a lower compute time.

We further compare these partitioners for *diverse synthetic graphs* generated using `networkx` [53], with different $\{|V|, |E|\}$: Power-law cluster graph (PL $\{2M, 40M\}$) with triangle probability 0.7 [39]; Barabasi Albert graph (BA $\{10M, 300M\}$) [9]; Random Regular graph (RR $\{20M, 400M\}$) [48]; Erdos Reyni graph (ER $\{20M, 150M\}$) [8, 50]; and Watts Strogatz graph (WS $\{10M, 500M\}$) [85].

Fig. 4e shows the *normalized local triangle count* relative to BTH (bars on left Y axis) and the *vertex replication factor* ($\rho$, markers on right Y axis) for $k = 8$ partitions. For all five graphs, BTH preserves 2.9–6.8× *more triangles* than HDRF and 1.5–2× than DBH. BTH also reduces replication factor by 15–50% and 30–75% compared to HDRF and DBH, respectively. These satisfy Theorem 3.2. These are comparable to or better than the results for real-world graphs, confirming the generalizable efficacy of BTH. Despite having a lower replication factor than DBH, HDRF has a lower triangle count. This shows that just reducing replication factor alone will not automatically increase the triangle counts.

Lastly, we compare BTH against the recent *CLUGP* [52] community preserving partitioner (plots omitted for brevity). The triangle count of BTH is higher at $109.8M$ for Orkut/16p compared to $92.7M$ for CLUGP, and with a much lower replication factor of 5.3 against 8.4. For Brain/16p, both preserve a similar number of triangles ($2.12B$ vs. $2.19B$) but with an even lower replication factor for BTH (2.3 vs. 8.3). This confirms that community preservation (CLUGP) and triangle conservation (TriParts) are correlated, but we additionally achieve lower vertex replications.

## 6.3 Performance of TriParts

*6.3.1 Scalability.* A key design of TriParts is its ability to scale well with the number of threads and cores, leveraging the potentially 100+ cores available in modern servers. Fig. 5 shows the *throughput* (bar, left Y axis) for our heuristics when the *leader's compute threads* are increased from 1 to 32, using $k = 8$ partitions. We also report the *speedup* relative to 1 thread as a marker on the right y-axis. The system shows good *strong scaling* when increasing the leader threads, with speedup of 7–10× seen for 32 threads, and a throughput growth from $\approx 100k$ to $750k$ edges/s, except for Brain which has a very high density and many more triangles, causing bottlenecks. This growth indicates further head-room as we run on a leader machine with more cores than the 16 (32 hyperthreads) we have, allowing more compute threads. Our careful data-parallel pipelined design with minimal thread contention allows the compute threads to execute the edge partitioning logic independently.

When we examine the effect of the *number of partitions* on the performance in Figs. 4b and 4d, the total compute time for our heuristics does not increase much from 8 to 16 parts, indicating good scaling. As $k$ increases, the effort involved at the leader and the workers also increase, but so does the number of workers. By overlapping the edge transfers to workers with the partitioning logic at the leader through pipelining, we avoid a linear growth in the time taken by the leader with the number of partitions.

*6.3.2 Sensitivity to Configuration Parameters.* We analyze the impact of changing two configurations parameters of TriParts. Fig. 6a shows the performance of BTH for $k = 8$ partitions of the six real world graphs as we vary the *sync frequencies* to 1%, 5%, 10% and 20% of total edges in the graph. This complements the *default* hybrid sync at every 10% of the stream or when the average edge degree seen changes by 1.0. Fig. 6b shows the performance of BTH when the *threshold for a high degree vertex* is changed from 2× *default* to 1.5× and 4× of the graph's average vertex degree. Both figures show the *normalized triangle count* (bars on left Y axis) and *normalized compute time* (markers on right Y axis) relative to their defaults.

Including HD sync besides 10% syncs usually increases the triangles preserved by 10–60% (Fig. 6a), but at the cost of runtime overheads. The runtime is lower with 10%-only sync by 9–13%, relative to default, for graphs with lower degree (USRN, DBPdia, MAG), and 27–45% lower for the others with higher average degree. As expected, increasing the sync frequency from every 10% to every 1% of edge arrivals causes the compute time to increases by 20–95% since there are 10× more syncs. In return, the 1% sync preserves 10–30% more triangles. Increasing HD threshold from 1.5× to 2× to 4× decreases the preserved triangles by 1.5–8% and 3–10% as the advantage of Hypothesis 4.2 diminishes. The runtime also reduces when the HD threshold increases, dropping by 2–12% and 6–12%.

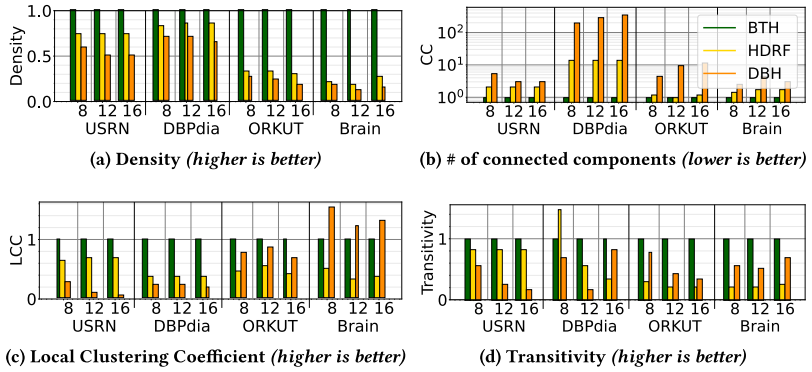
*6.3.3 Overheads.* We evaluate the *locking overheads* on the multi-threaded worker to access the adjacency list for adding a new edge and computing the new triangles count. Fig. 7a shows the average E2E time and within this, the sum of the locking overheads for *all threads* of a worker for the six real-world graphs for $k = 8$ partitions, with 8 and 16 threads per worker. This *lock overhead* (red stack) is an overestimate since only one thread is blocked in this duration while the others are still productive. Only 5–8% (8 threads) and 8–10% (16 threads) of E2E time is spent on locking. As expected, more threads introduces more contention. The actual overheads are likely to be even lesser as this is a weak upper bound.

Fig. 7a also shows the cumulative time to update data structures from the workers during *sync operations* (yellow bar) for these same scenarios. Here, we are syncing only after reading every 10% of edges. The sync overheads forms only 4–6.5% of the E2E time with 8 worker threads and 1.1–4% with 16 threads, except for USRN where it is 27 and 15%, respectively. This is a small cost for the benefits from these states in achieving the partitioner objectives.

Lastly, Fig. 7b shows the peak size of the triangle map and high degree map as a % of total vertices, when running BTH with $k = 8$. The peak High Degree map size is 2.3–4.2% of all vertices since most are power law graphs with few highly connected vertices. The peak Triangle map size is 4–98% of vertex-count. As expected, dense graphs like Brain maintain a larger T-map compared to sparse ones like USRN. Despite this, the peak memory usage on the leader remains modest and comparable to the workers' memory usage. E.g., even for the largest graph Twitter, the leader's peak memory usage is just 15% of $128GB$ for $k = 8$, growing to 17% for $k = 16$. This is comparable to the workers' peak memory of 21% and 16%.

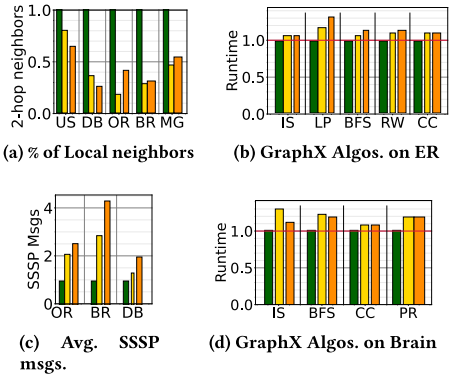
*6.3.4 Community preservation.* We measure common quality used for community structure preservation on the partitions created by BTH, HDRF and DBH. Fig. 8 shows the density, LCC, transitivity and number of connected components (CC), averaged across partitions and normalized with respect to BTH; higher is better for the first three while lower is better for CC.

As seen before, BTH preserves 3–40× more triangles than HDRF and DBH. This translates to its ability to better preserve the community structure using these standard metrics. DBH and HDRF have




(a) Density *(higher is better)*
(b) # of connected components *(lower is better)*
(c) Local Clustering Coefficient *(higher is better)*
(d) Transitivity *(higher is better)*

**Figure 8: Community metrics (8 parts)**




(a) % of Local neighbors
(b) GraphX Algos. on ER
(c) Avg. SSSP msgs.
(d) GraphX Algos. on Brain

**Figure 9: Performance benefits 8 parts)**

3–350× and 1.2–15× more disconnected components on average compared to BTH, in each partition. Hence, BTH is able to maintain connectivity between vertices in each partition, strengthening the community structure significantly. As a result of these disconnected components, DBH and HDRF also have 27.5–85% and 15–80% lower Density. As expected from Hypothesis 3.3, preserving triangles also helps BTH increase Transitivity by 18–80% and 20–82% compared to HDRF and DBH. For USRN and DBPdia graphs, BTH, HDRF and DBH have similar vertex replication factor ($\rho$). According to Hypothesis 3.2, since BTH has 2–10× more triangles it also has a better local clustering coefficient (LCC) by a factor of 3.5–10×. For ORKUT and Brain, BTH also has $2-6\times$ lower vertex replication, which decreases the LCC but increases the average degree. Hence, the LCC of BTH is still better for ORKUT, although by a factor of only 1.2–2×. However, effect of low vertex replication surpasses the effect of higher triangle preservation in Brain, where the LCC reduces to 0.6–0.75× compared to DBH.

*6.3.5 Workload benefits.* The benefits of BTH also translates to better performance when executing distributed graph algorithms over the streaming graph partitions. We report two metrics: one is the fraction of 2-hop neighbors that are local within a partition, which is helpful when sampling labeled vertices during distributed GNN training and inferencing [92] (Fig. 9a); and the other is the average number of messages sent between partitions when performing a Single Source Shortest Path (SSSP) from 3–5 random source vertices [19](Fig. 9c). These are indicators of the communication overheads when executing distributed graph analytics.

BTH has 66–300% and 25–400% more 1 hop and 2 hop neighbors present locally compared to DBH and HDRF, thus reducing the sampling overheads when performing distributed GNN inferencing over the incremental graph. For iterative distributed bulk synchronous algorithms that have a flavor SSSP, HDRF and DBH have 2–4× more messages sent across machines compared to BTH. This indicates the benefits from lower communication costs for BTH.

We further evaluate the *practical real-world benefits* of our streaming partitioner for several common graph algorithms using Apache Spark's GraphX [33] platform. Edge-centric ones include 2-hop BFS common for GNN sampling [82], Influence Spread (IS) [68, 70], Random Walk (RW) [21, 86] and Label Propagation (LP) [27, 88], with the former three run from 20 random source vertices, while vertex-centric ones are PageRank (PR) [11, 80] and Triangle Count (TC) [20, 23]. These execute on $k = 8$ parts of Erdos-Reyni (ER) and Brain, partitioned using BTH, HDRF and DBH. From Fig. 9b and 9d, we see that for all these algorithms, HDRF is slower by 10–30% for Brain and 6–16% for ER, while DBH is slower by 10–20% for Brain and 7–30% for ER. Here, BTH benefits both from a better replication factor and higher triangle counts, compared to these baselines, lading to faster application runtimes.

## 7 CONCLUSIONS

In this paper, we propose a novel objective of preserving local triangles within the $k$-way edge-based streaming graph partitioning problem to help collocate dense communities within each partition. This complements the usual objectives of edge-balancing and minimizing vertex replicas, which by themselves do not preserve communities as we formally and empirically show. We design a TriParts as a set of partitioning heuristics to meet these objectives. Our innovative T-map, H-map and Bloom filter states at the leader help make partitioning decisions and offer measurable benefits. Our rigorous experiments over diverse real-world and synthetic graphs show that our BTH heuristic out-performs contemporary HDRF, DBH and CLUGP baselines. BTH is consistently better at conserving triangles while offering comparable or better vertex replication. Our optimizations make TriParts scalable and performant, processing $\approx 500k$ edges/sec using 16 workers.

Future work includes reducing high-degree sync frequency and dynamically adjusting syncs to avoid redundancy. We plan to explore edge duplication heuristics for better community localization and use ML to train community-aware edge classifiers. Rebalancing the partitions as size grow, combined with out-of-core storage, can lower memory use. Resilience can be improved via secondary leaders with Bloom filter updates and failover syncs, along with durable, append-only edge logs per worker. Though our triangle preservation method handles diverse graphs well, it may weaken when communities rely on paths longer than 3. Here, BTH reduces to BH, suggesting the need to explore alternative community metrics.

# REFERENCES


[1] Zainab Abbas, Vasiliki Kalavri, Paris Carbone, and Vladimir Vlassov. 2018. Streaming Graph Partitioning: An Experimental study. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1590–1603. https://doi.org/10.14778/3236187.3236208

[2] Zainab Abbas, Vasiliki Kalavri, Paris Carbone, and Vladimir Vlassov. 2018. Streaming graph partitioning: an experimental study. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1590–1603.

[3] Manoj K. Agarwal, Krithi Ramamritham, and Manish Bhide. 2012. Real time discovery of dense clusters in highly dynamic graphs: identifying real world events in highly dynamic environments. *Proc. VLDB Endow.* 5, 10 (2012), 980–991. https://doi.org/10.14778/2336664.2336671

[4] Yaroslav Akhremtsev, Peter Sanders, and Christian Schulz. 2020. High-Quality Shared-Memory Graph Partitioning. *IEEE Transactions on Parallel and Distributed Systems* 31, 11 (2020), 2710–2722. https://doi.org/10.1109/TPDS.2020.3001645

[5] Katrin Amunts, Claude Lepage, Louis Borgeat, Hartmut Mohlberg, Timo Dickscheid, Marc-Étienne Rousseau, Sebastian Bludau, Pierre-Louis Bazin, Lindsay B. Lewis, Ana-Maria Oros-Peusquens, Nadim J. Shah, Thomas Lippert, Karl Zilles, and Alan C. Evans. 2013. BigBrain: An Ultrahigh-Resolution 3D Human Brain Model. *Science* 340, 6139 (2013), 1472–1475.

[6] Konstantin Andreev and Harald Racke. 2006. Balanced graph partitioning. *Theory of Computing Systems* 39, 6 (2006), 929–939.

[7] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. 2007. DBpedia: A Nucleus for a Web of Open Data. *Lecture Notes in Computer Science* 4825 (2007), 722–735.

[8] Arindam Banerjee and D. Yogeshwaran. 2021. *Edge ideals of Erdös-Rényi random graphs: Linear resolution, unmixedness and regularity.* Technical Report 2007.08869. https://arxiv.org/abs/2007.08869

[9] Albert-László Barabási and Réka Albert. 1999. Emergence of Scaling in Random Networks. *Science* 286, 5439 (1999), 509–512. https://doi.org/10.1126/science.286.5439.509

[10] Punam Bedi and Chhavi Sharma. 2016. Community detection in social networks. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 6 (2016). https://doi.org/10.1002/widm.1178

[11] Joost Berkhout. 2016. Google's PageRank algorithm for ranking nodes in general networks. In *13th International Workshop on Discrete Event Systems (WODES)*. 153–158. https://doi.org/10.1109/WODES.2016.7497841

[12] Ruchi Bhoot, Suved Sanjay Ghanmode, and Yogesh Simmhan. 2024. TARIS: Scalable Incremental Processing of Time-Respecting Algorithms on Streaming Graphs. *IEEE Transactions on Parallel & Distributed Systems* 35, 12 (2024), 2527–2544. https://doi.org/10.1109/TPDS.2024.3471574

[13] Ivan A. Bliznets and Nikolai Karpov. 2017. *Parameterized Algorithms for Partitioning Graphs into Highly Connected Clusters.* Technical Report 6. 6:1–6:14 pages.

[14] Florian Bourse, Marc Lelarge, and Milan Vojnovic. 2014. Balanced graph edge partition. In *20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1456–1465. https://doi.org/10.1145/2623330.2623660

[15] Meeyoung Cha, Hamed Haddadi, Fabricio Benevenuto, and Krishna Gummadi. 2010. Measuring User Influence in Twitter: The Million Follower Fallacy. In *International AAAI Conference on Web and Social Media*. 10–17. https://doi.org/10.1609/icwsm.v4i1.14033

[16] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. *ACM Trans. Parallel Comput.* 5, 3 (2019), 39 pages. https://doi.org/10.1145/3298989

[17] Cédric Chevalier and François Pellegrini. 2008. PT-Scotch: A tool for efficient parallel graph ordering. *Parallel computing* 34, 6-8 (2008), 318–331.

[18] Adil Chhabra, Florian Kurpicz, Christian Schulz, Dominik Schweisgut, and Daniel Seemaier. 2024. *Partitioning Trillion Edge Graphs on Edge Devices.* Technical Report. https://arxiv.org/abs/2410.07732

[19] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One trillion edges: graph processing at Facebook-scale. *Proc. VLDB Endow.* 8, 12 (2015), 1804–1815. https://doi.org/10.14778/2824032.2824077

[20] Shumo Chu and James Cheng. 2012. Triangle listing in massive networks. *ACM Trans. Knowl. Discov. Data* 6, 4 (2012), 32 pages. https://doi.org/10.1145/2382577.2382581

[21] Atish Das Sarma, Danupon Nanongkai, Gopal Pandurangan, and Prasad Tetali. 2013. Distributed Random Walks. *J. ACM* 60, 1 (2013), 31 pages. https://doi.org/10.1145/2432622.2432624

[22] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2021. Theoretically Efficient Parallel Graph Algorithms Can Be Fast and Scalable. *ACM Trans. Parallel Comput.* 8, 1 (2021), 70 pages. https://doi.org/10.1145/3434393

[23] David Ediger and David A. Bader. 2013. Investigating Graph Algorithms in the BSP Model on the Cray XMT . In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*. 1638–1645. https://doi.org/10.1109/IPDPSW.2013.107

[24] Suhendry Effendy and Roland H.C. Yap. 2016. Investigations on Rating Computer Sciences Conferences: An Experiment with the Microsoft Academic Graph Dataset. In *25th International Conference Companion on World Wide Web*. 425–430. https://doi.org/10.1145/2872518.2890525

[25] Wenfei Fan, Ruochun Jin, Muyang Liu, Ping Lu, Xiaojian Luo, Ruiqi Xu, Qiang Yin, Wenyuan Yu, and Jingren Zhou. 2020. Application Driven Graph Partitioning. In *2020 ACM SIGMOD International Conference on Management of Data*. 1765–1779. https://doi.org/10.1145/3318464.3389745

[26] Wenfei Fan, Muyang Liu, Chao Tian, Ruiqi Xu, and Jingren Zhou. 2020. Incrementalization of graph partitioning algorithms. *Proc. VLDB Endow.* 13, 8 (2020), 1261–1274. https://doi.org/10.14778/3389133.3389142

[27] Sara E. Garza and Satu Elisa Schaeffer. 2019. Community detection with the Label Propagation Algorithm: A survey. *Physica A: Statistical Mechanics and its Applications* 534 (2019), 122058. https://doi.org/10.1016/j.physa.2019.122058

[28] Alice Gatti, Zhixiong Hu, Tess Smidt, Esmond G. Ng, and Pieter Ghysels. 2022. Graph partitioning and sparse matrix ordering using reinforcement learning and graph neural networks. *J. Mach. Learn. Res.* 23, 1 (2022), 28 pages.

[29] Michelle Girvan and Mark EJ Newman. 2002. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences* 99, 12 (2002), 7821–7826.

[30] M. Girvan and M. E. J. Newman. 2002. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences* 99, 12 (2002), 7821–7826. https://doi.org/10.1073/pnas.122653799

[31] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 17–30. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez

[32] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: distributed graph-parallel computation on natural graphs. In *10th USENIX Conference on Operating Systems Design and Implementation*. 17–30.

[33] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: graph processing in a distributed dataflow framework. In *USENIX Conference on Operating Systems Design and Implementation*. 599–613.

[34] Milad Rezaei Hajidehi, Sraavan Sridhar, and Margo Seltzer. 2024. *CUTTANA: Scalable Graph Partitioning for Faster Distributed Graph Databases and Analytics.* Technical Report. https://arxiv.org/abs/2312.08356

[35] Ibrahim Abou Hamad, Per Arne Rikvold, and Svetlana V. Poroseva. 2011. Floridian high-voltage power-grid network partitioning and cluster optimization using simulated annealing. *Physics Procedia* 15 (2011), 2–6. https://doi.org/10.1016/j.phpro.2011.05.051

[36] Minyang Han and Khuzaima Daudjee. 2015. Giraph unchained: barrierless asynchronous parallel execution in pregel-like graph processing systems. *Proc. VLDB Endow.* 8, 9 (2015), 950–961. https://doi.org/10.14778/2777598.2777604

[37] Erez Hartuv and Ron Shamir. 2000. A clustering algorithm based on graph connectivity. *Inform. Process. Lett.* 76, 4 (2000), 175–181. https://doi.org/10.1016/S0020-0190(00)00142-3

[38] Loc Hoang, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. 2019. Cusp: A customizable streaming edge partitioner for distributed graph analytics. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 439–450.

[39] Petter Holme and Beom Jun Kim. 2001. Growing scale-free networks with tunable clustering. *Physical review. E, Statistical, nonlinear, and soft matter physics* 65 2 Pt 2 (2001), 026107. https://api.semanticscholar.org/CorpusID:4643442

[40] Jiewen Huang and Daniel J. Abadi. 2016. Leopard: lightweight edge-oriented partitioning and replication for dynamic graphs. *Proc. VLDB Endow.* 9, 7 (2016), 540–551. https://doi.org/10.14778/2904483.2904486

[41] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. 2016. Time-evolving graph processing at scale. In *Fourth international workshop on graph data management experiences and systems*. 1–6.

[42] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. 2016. Time-evolving graph processing at scale. In *Fourth International Workshop on Graph Data Management Experiences and Systems*. 6 pages. https://doi.org/10.1145/2960414.2960419

[43] Nilesh Jain, Guangdeng Liao, and Theodore L Willke. 2013. Graphbuilder: scalable graph etl framework. In *First international workshop on graph data management experiences and systems*. 1–6.

[44] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing* 20, 1 (1998), 359–392.

[45] George Karypis, Kirk Schloegel, and Vipin Kumar. 1997. Parmetis parallel graph partitioning and sparse matrix ordering library, In https://api.semanticscholar.org/CorpusID:9818727. *University of Minnesota* 1, 1, 315–320.

[46] Brian W Kernighan and Shen Lin. 1970. An efficient heuristic procedure for partitioning graphs. *The Bell system technical journal* 49, 2 (1970), 291–307.

[47] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. 2013. Mizan: a system for dynamic load balancing in large-scale graph processing. In *8th ACM European conference on computer systems*. 169–182.

[48] J. H. Kim and V. H. Vu. 2006. Generating Random Regular Graphs. *Combinatorica* 26, 6 (2006), 683–708. https://doi.org/10.1007/s00493-006-0037-7

[49] Christine Klymko, David Gleich, and Tamara G Kolda. 2014. *Using triangles to improve community detection in directed networks.* Technical Report. arxiv pages. https://arxiv.org/abs/1404.5874

[50] Donald E. Knuth. 1997. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*.

[51] Deyu Kong. 2022. Clustering-based Partitioning for Large Web Graphs. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 593–606. https://github.com/USTC-DataDarknessLab/GraphPartitioning/tree/main/CLUGP

[52] Deyu Kong, Xike Xie, and Zhuoxu Zhang. 2022. Clustering-based Partitioning for Large Web Graphs. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 593–606. https://doi.org/10.1109/ICDE53745.2022.00049

[53] Los Alamos National Laboratory, United States. Department of Energy. Office of Scientific, and Technical Information. 2008. *Exploring Network Structure, Dynamics, and Function Using Networkx*. https://books.google.co.in/books?id=yOiJAQAACAAJ

[54] Dominique LaSalle and George Karypis. 2013. Multi-threaded graph partitioning. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. 225–236.

[55] Danh Le-Phuoc, Hoan Nguyen Mau Quoc, Hung Ngo Quoc, Tuan Tran Nhat, and Manfred Hauswirth. 2016. The graph of things: A step towards the live knowledge graph of connected things. *Journal of Web Semantics* 37 (2016), 25–35.

[56] Hyeonbyeong Lee, Jeonghyun Baek, Sangho Song, Yuna Kim, Hyunjung Hwang, Jongtae Lim, Dojin Choi, Kyoungsoo Bok, and Jaesoo Yoo. 2025. Efficient Large Graph Partitioning Scheme Using Incremental Processing in GPU. *IEEE Access* 13 (2025), 43889–43903. https://doi.org/10.1109/ACCESS.2025.3547976

[57] Lun Li, David Alderson, John Doyle, and Walter Willinger. 2005. Towards a Theory of Scale-Free Graphs: Definition, Properties, and Implications. *Internet Mathematics* 2 (2005). https://doi.org/10.1080/15427951.2005.10129111

[58] Lingda Li, Robel Geda, Ari B. Hayes, Yanhao Chen, Pranav Chaudhari, Eddy Z. Zhang, and Mario Szegedy. 2017. A Simple Yet Effective Balanced Edge Partition Model for Parallel Computing. *Proc. ACM Meas. Anal. Comput. Syst.* 1, 1 (2017), 21 pages. https://doi.org/10.1145/3084451

[59] Yang Liu, Xiang Ao, Zidi Qin, Jianfeng Chi, Jinghua Feng, Hao Yang, and Qing He. 2021. Pick and Choose: A GNN-based Imbalanced Learning Approach for Fraud Detection. In *Web Conference*. 3168–3177. https://doi.org/10.1145/3442381.3449989

[60] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-Scale Graph Processing. In *ACM SIGMOD International Conference on Management of Data*. 135–146. https://doi.org/10.1145/1807167.1807184

[61] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *ACM SIGMOD International Conference on Management of Data*. 135–146. https://doi.org/10.1145/1807167.1807184

[62] Anna Mastikhina, Oleg Senkevich, Dmitry Sirotkin, Danila Demin, and Stanislav Moiseev. 2024. *An improvement of degree-based hashing (DBH) graph partition method, using a novel metric.* Technical Report. 1–15 pages. https://api.semanticscholar.org/CorpusID:269043059

[63] Christian Mayer, Ruben Mayer, Muhammad Adnan Tariq, Heiko Geppert, Larissa Laich, Lukas Rieger, and Kurt Rothermel. 2018. Adwise: Adaptive window-based streaming edge partitioning for high-speed graph processing. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 685–695.

[64] Christian Mayer, Ruben Mayer, Muhammad Adnan Tariq, Heiko Geppert, Larissa Laich, Lukas Rieger, and Kurt Rothermel. 2018. Adwise: Adaptive window-based streaming edge partitioning for high-speed graph processing. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. 685–695.

[65] Ruben Mayer, Kamil Orujzade, and Hans-Arno Jacobsen. 2022. Out-of-Core Edge Partitioning at Linear Run-Time . In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 2629–2642. https://doi.org/10.1109/ICDE53745.2022.00242

[66] Ruben Mayer, Kamil Orujzade, and Hans-Arno Jacobsen. 2022. Out-of-Core Edge Partitioning at Linear Run-Time . In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 2629–2642. https://doi.org/10.1109/ICDE53745.2022.00242

[67] Henning Meyerhenke, Peter Sanders, and Christian Schulz. 2017. Parallel graph partitioning for complex networks. *IEEE Transactions on Parallel and Distributed Systems* 28, 9 (2017), 2625–2638.

[68] Seth A. Myers, Chenguang Zhu, and Jure Leskovec. 2012. Information diffusion and external influence in networks. In *18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 33–41. https://doi.org/10.1145/2339530.2339540

[69] Azade Nazi, Will Hang, Anna Goldie, Sujith Ravi, and Azalia Mirhoseini. 2019. *GAP: Generalizable Approximate Graph Partitioning Framework*. Technical Report. https://api.semanticscholar.org/CorpusID:67855909

[70] David F. Nettleton. 2013. Data mining of social networks represented as graphs. *Computer Science Review* 7 (2013), 1–34. https://doi.org/10.1016/j.cosrev.2012.12.001

[71] Mark EJ Newman. 2006. Modularity and community structure in networks. *Proceedings of the National Academy of Sciences* 103, 23 (2006), 8577–8582.

[72] M. E. J. Newman. 2004. Fast algorithm for detecting community structure in networks. *Phys. Rev. E* 69 (2004), 066133. https://doi.org/10.1103/PhysRevE.69.066133

[73] Fabio Petroni. 2015. VGP : Vertex-cut balanced Graph Partitioning. In *24th ACM international on conference on information and knowledge management*. 243–252. https://github.com/fabiopetroni/VGP

[74] Fabio Petroni, Leonardo Querzoni, Khuzaima Daudjee, Shahin Kamali, and Giorgio Iacoboni. 2015. Hdrf: Stream-based partitioning for power-law graphs. In *24th ACM international on conference on information and knowledge management*. 243–252.

[75] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. all. http://networkrepository.com/road-road-usa.php

[76] Matthew Saltz, Arnau Prat-Pérez, and David Dominguez-Sal. 2015. Distributed community detection with the wcc metric. In *24th International Conference on World Wide Web*. 1095–1100.

[77] Sebastian Schlag, Christian Schulz, Daniel Seemaier, and Darren Strash. 2019. Scalable edge partitioning. In *Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 211–225.

[78] Yogesh Simmhan, Alok Gautam Kumbhare, Charith Wickramaarachchi, Soonil Nagarkar, Santosh Ravi, Cauligi S. Raghavendra, and Viktor K. Prasanna. 2014. GoFFish: A Sub-graph Centric Framework for Large-Scale Graph Analytics. In *Euro-Par 2014 Parallel Processing - 20th International Conference*, Vol. 8632. 451–462. https://doi.org/10.1007/978-3-319-09873-9_38

[79] Isabelle Stanton and Gabriel Kliot. 2012. Streaming graph partitioning for large distributed graphs. In *18th ACM SIGKDD international conference on Knowledge discovery and data mining*. 1222–1230.

[80] Ana-Andreea Stoica, Nelly Litvak, and Augustin Chaintreau. 2024. Fairness Rising from the Ranks: HITS and PageRank on Homophilic Networks. In *ACM Web Conference*. 2594–2602. https://doi.org/10.1145/3589334.3645609

[81] Rahul Swamy, Douglas M. King, and Sheldon H. Jacobson. 2024. *Highly Connected Graph Partitioning: Exact Formulation and Solution Methods*. Technical Report. https://arxiv.org/abs/2406.08329

[82] Manuel Then, Moritz Kaufmann, Fernando Chirigati, Tuan-Anh Hoang-Vu, Kien Pham, Alfons Kemper, Thomas Neumann, and Huy T. Vo. 2014. The more the merrier: efficient multi-source graph traversal. *Proc. VLDB Endow.* 8, 4 (2014), 449–460. https://doi.org/10.14778/2735496.2735507

[83] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. 2014. Fennel: Streaming graph partitioning for massive scale graphs. In *7th ACM international conference on Web search and data mining*. 333–342.

[84] Duncan J. Watts and Steven H. Strogatz. 1998. Collective dynamics of 'small-world' networks. *Nature* 393 (1998), 440–442. https://api.semanticscholar.org/CorpusID:3034643

[85] Duncan J. Watts and Steven H. Strogatz. 1998. Collective dynamics of 'small-world' networks. *Nature* 393 (1998), 440–442. https://api.semanticscholar.org/CorpusID:3034643

[86] Feng Xia, Jiaying Liu, Hansong Nie, Yonghao Fu, Liangtian Wan, and Xiangjie Kong. 2020. Random Walks: A Review of Algorithms and Applications. *IEEE Transactions on Emerging Topics in Computational Intelligence* 4, 2 (2020), 95–107. https://doi.org/10.1109/tetci.2019.2952908

[87] Cong Xie, Ling Yan, Wu-Jun Li, and Zhihua Zhang. 2014. Distributed power-law graph computing: theoretical and empirical analysis. In *27th International Conference on Neural Information Processing Systems - Volume 1*. 1673–1681.

[88] Rong Yan, Wei Yuan, Xiangdong Su, and Ziyi Zhang. 2023. FLPA: A fast label propagation algorithm for detecting overlapping community structure. *Expert Systems with Applications* 234 (2023), 120971. https://doi.org/10.1016/j.eswa.2023.120971

[89] Jaewon Yang and Jure Leskovec. 2012. *Defining and Evaluating Network Communities based on Ground-truth*. Technical Report. https://snap.stanford.edu/data/com-Orkut.html

[90] Abdurrahman Yaşar, Sivasankaran Rajamanickam, Jonathan W Berry, and Ümit V Çatalyürek. 2021. A Block-Based Triangle Counting Algorithm on Heterogeneous Environments. *IEEE Transactions on Parallel and Distributed Systems* 33, 2 (2021), 444–458.

[91] Wei Zhang, Yong Chen, and Dong Dai. 2018. AKIN: A Streaming Graph Partitioning Algorithm for Distributed Graph Storage Systems. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 183–192. https://doi.org/10.1109/CCGRID.2018.00033

[92] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. 2021. *DistDGL: Distributed Graph*

*Neural Network Training for Billion-Scale Graphs*. Technical Report. https://arxiv.org/abs/2010.05337

[93] Michał Zwolak, Zainab Abbas, Sonia Horchidan, Paris Carbone, and Vasiliki Kalavri. 2022. GCNSplit: bounding the state of streaming graph partitioning. In *Fifth International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. 12 pages. https://doi.org/10.1145/3533702.3534920