CUTTANA: Scalable Graph Partitioning for Faster Distributed Graph Databases and Analytics

Milad Rezaei Hajidehi
University of British Columbia
Vancouver, Canada
miladrzh@cs.ubc.ca

Sraavan Sridhar
University of British Columbia
Vancouver, Canada
sraavan@student.ubc.ca

Margo Seltzer
University of British Columbia
Vancouver, Canada
mseltzer@cs.ubc.ca

ABSTRACT

Graph partitioning plays a pivotal role in various distributed graph processing applications, including graph analytics, graph neural network training, and distributed graph databases. A "good" graph partitioner reduces workload execution time, worker imbalance, and network overhead. Graphs that require distributed settings are often too large to fit in the main memory of a single machine. This challenge renders traditional in-memory graph partitioners infeasible, leading to the emergence of streaming solutions. Streaming partitioners produce lower-quality partitions, because they work from partial information and must make premature decisions before they have a complete view of a vertex's neighborhood. We introduce CUTTANA, a streaming graph partitioner that partitions massive graphs (Web/Twitter scale) with superior quality compared to existing streaming solutions. CUTTANA uses a novel buffering technique that prevents the premature assignment of vertices to partitions and a scalable coarsening and refinement technique that enables a complete graph view, improving the intermediate assignment made by a streaming partitioner. We implemented a parallel version for CUTTANA that offers nearly the same partitioning latency as existing streaming partitioners.

Our experimental analysis shows that CUTTANA consistently yields better partitioning quality than state-of-the-art streaming vertex partitioners in terms of both edge-cut and communication volume metrics. We also evaluate the workload latencies that result from using CUTTANA and other partitioners in distributed graph analytics and databases. CUTTANA outperforms the other methods in most scenarios (algorithms, datasets). In analytics applications, CUTTANA improves runtime performance by up to 59% compared to various streaming partitioners (i.e., HDRF, Fennel, Ginger, HeiStream). In graph database tasks, CUTTANA results in higher query throughput by up to 23%, without hurting tail latency.
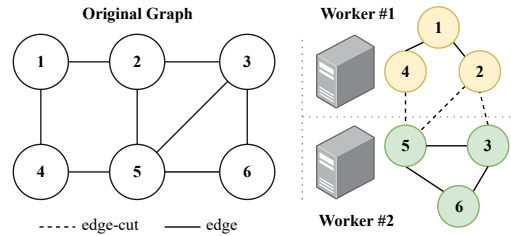
Figure 1: Partitioning a graph for two workers. Assume every vertex must send a message to its neighbors; each edge between machines becomes a network call.

The source code, data, and/or other artifacts have been made available at https://github.com/cuttana.

1 INTRODUCTION

Ubiquity and massive growth of real-world networks sparked the applications of distributed graph processing. A graph is a common data model that can represent complex relationships between real-world entities in myriad domains such as social networks, the World Wide Web, finance, fraud detection, transportation, and biological networks [47]. In practice, many of these graphs are sufficiently large to exceed the memory of a single machine, posing performance challenges for single-node solutions. Using distributed systems with increased memory and parallelism enables high performance for large graph processing. The ubiquity and growth of real-world graphs motivated the development of distributed graph processing solutions for various applications such as graph analytics [7, 8, 11, 18, 19, 22, 31, 56, 60], graph databases [5, 23, 30, 51], and graph neural networks (GNN) [16, 41, 55, 57, 62].

Graph partitioning affects the performance of distributed graph processing. The first step of any distributed graph processing application is to partition the graph into disjoint subgraphs and distribute them to worker machines. Unlike traditional distributed applications such as map-reduce, graph processing workloads exhibit many interactions among partitions [39]. For example, in PageRank, the rank of a vertex is calculated based on the rank of its neighbors in each iteration. To achieve high-quality partitioning, the number of edges that have vertices assigned to different machines (i.e., edge-cuts) should be minimized, since exchanging data along those edges incurs network usage (Figure 1). Another aspect of good partitioning is assigning equal-sized partitions to workers to avoid stragglers. Figure 2 shows that network overhead can be more than 100GB for a 16-worker PageRank computation on the
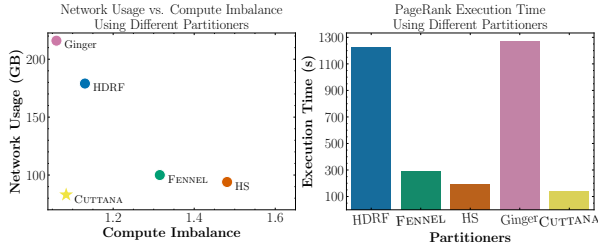
Figure 2: An example of partitioning's effect on network usage, compute imbalance, and PageRank time on the UK07 dataset. CUTTANA improved PageRank execution time by more than 52s (27%).

UK07 dataset and the graph partitioning algorithm has a significant effect on network usage, worker imbalance, and execution time.

Partitioning large graphs is hard and memory-bound. The problem of balanced graph partitioning is $\mathcal{NP}$-hard [17]. However, many domains other than distributed graph processing (e.g., VLSI design [6] and causal inference [3]) demand high-quality partitioning. As a result, many heuristic solutions exist [6, 24, 36, 38, 48, 61]. However, most of these solutions fail when partitioning graphs larger than main memory. For example, METIS [24], long the gold standard for graph partitioning, is unable to partition the Twitter or Web graphs [35, 38], leading to the development of various streaming partitioners for massive graphs [8, 14, 37, 43, 50, 52].

Streaming partitioners are scalable, but the partitionings that they produce are low-quality, having a greater number of cross-partition edges than do in-memory partitioners. Streaming solutions make partitioning decisions by reading vertices or edges one by one and assigning them to partitions based on a scoring function. The score is calculated from minimal summarized information about the vertices/edges already assigned, the current vertex/edge, and the partition sizes. There are two types of partitioners: vertex-partitioners (edge-cut partitioners) [14, 50, 52], which read a stream of vertices and their neighbors and assign each vertex to a partition, and edge partitioners (vertex-cut partitioners) [8, 13, 45], which read a stream of edges and assign each to a partition. In an experimental study, Pacaci and Ozsu [43] reported that edge-cut partitioners yield lower network overhead but greater worker imbalance. Analyses show that streaming partitioners produce more edge-cuts, hence higher network overhead, relative to in-memory partitioners [3, 52]. This is unsurprising given their limited view of the graph.

CUTTANA is a high-quality, scalable partitioner, designed to have the scalability of streaming solutions while providing better partitioning quality. We studied existing streaming vertex partitioners and found three limitations. 1) They prematurely assign vertices when the data needed to calculate an accurate scoring function is not available. 2) They never change vertex assignments, even though, over time, the algorithm gains information about the graph. 3) The significant worker imbalance when using edge-cut partitioners overshadows their network overhead superiority.

We solve the first problem by introducing *score-based dynamic buffering*. We buffer vertices based on the knowledge we have about their neighborhood and avoid premature partitioning when insufficient data is available. However, if done naively, buffering can

result in storing the entire graph in memory, which is obviously not scalable. We solve the second problem by providing a mechanism to move and exchange vertices between partitions to enhance the partitioning quality at the end of the streaming phase. Determining which moves enhance quality requires saving the neighborhood for each vertex and is also impossible (due to memory constraints). Also, the moving phase can be time-consuming due to the large number of possible moves. We introduce a *coarsening* strategy and a theoretically efficient *refinement* algorithm to find the best moves, enabling fast and coarse-grained improvement of partitioning quality. We show that the huge edge imbalance in existing edge-cut partitioners is the cause of worker imbalance in analytics. We solve the third problem by modeling and satisfying an edge-balance condition using an edge-based score function and our refinement algorithm. Finally, to minimize the potential time overhead caused by buffering and refinement, we provide a parallel implementation that yields nearly the same partitioning time for massive graphs compared to streaming solutions, while offering better partitioning quality.

Our contributions are as follows.

- We present a scalable, buffered streaming partitioning model to effectively use main memory to avoid premature vertex assignment. This model can be applied to any existing streaming partitioner to increase its quality.
- We introduce a novel *coarsening* and *refinement* technique that receives the output of a streaming partitioner and improves it to reach a "maximal" quality. This algorithm is theoretically efficient and independent of the graph size.
- We leverage unused cores via a parallel implementation, providing rapid partitioning speed.
- Through experimental analysis, we show CUTTANA's superiority relative to existing edge-cut partitioners. We also demonstrate the edge imbalance of existing partitioners, which is often overlooked in the literature.
- We show the effect of CUTTANA partitioning quality improvement in the execution time of distributed graph analytics. Overall CUTTANA can improve the runtime performance of graph analytics by up to 59% and is the best partitioner in most scenarios.
- We show the effect of CUTTANA partitioning quality improvement in the query throughput of distributed graph databases. CUTTANA can improve the throughput of the JanusGraph distributed graph database by up to 23% over the best existing graph partitioner in the standard LDBC social network benchmark.

2 BACKGROUND

Formal definition of vertex partitioning problem. Given a graph $G = \langle V, E \rangle$, the $\mathcal{K}$-way vertex-balanced graph partitioning problem is to assign vertices to the disjoint sets $\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_{\mathcal{K}}$ such that $\bigcup_{i=1}^{\mathcal{K}} \mathcal{V}_i = V$ and the $\mathcal{V}_i$ satisfy the balance condition:

$$|\mathcal{V}_i| \leq (1 + \epsilon) \cdot \frac{|V|}{\mathcal{K}} \quad (1 \leq i \leq \mathcal{K}) \tag{1}$$

The $\epsilon \geq 0$ is the balance slack parameter that constrains how imbalanced the partitions can be. The balance condition can also be defined based on the number of edges in a partition. With $\mathcal{N}(v)$

representing the set of neighbors for vertex v, we define the edge-balance condition for vertex partitioning as:

$$\sum_{v \in \mathcal{V}_i} |\mathcal{N}(v)| \leq (1 + \epsilon) \cdot \frac{2 \cdot |E|}{\mathcal{K}} \quad (1 \leq i \leq \mathcal{K}) \quad (2)$$

Optimization objectives. The quality metrics for graph partitioning are based on minimizing the interdependency of partitions. A common metric is *edge-cut*, the number of edges whose endpoints are in different partitions. Given $\mathcal{P} : V \rightarrow \mathbb{N}_{\leq \mathcal{K}}$, the function that returns the partition ID to which a vertex is assigned, the normalized number of edge-cuts is:

$$\lambda_{EC} = \frac{|\{\langle x, y \rangle \in E | \mathcal{P}(x) \neq \mathcal{P}(y)\}|}{|E|} \quad (3)$$

Minimizing edge-cuts is equivalent to minimizing network cost, since whenever a vertex requires data from a neighboring vertex in a different partition, the two corresponding workers must transmit the data over the network.

A common optimization in bulk synchronous systems, mostly in analytic workloads, is *sender-side aggregation* [4, 39, 43]. In these systems, the workload is iterative (e.g., PageRank iteration) and at the end of each iteration, if multiple vertices in the same worker are connected to the same vertex in a different worker, the neighboring vertex sends the data once. This causes all of the edges between the neighboring vertex and the vertices in the first worker to need only a single network message. In Figure 1, vertices 2 and 5 can benefit from this optimization. *Communication-volume* is the metric that models the network cost of such systems. Given $\mathcal{D} : V \rightarrow \mathbb{N}_{\leq \mathcal{K}}$, a function that returns the number of partitions in which a given vertex has neighbors, excluding its own partition ($\mathcal{P}(v)$), the normalized communication volume is:

$$\lambda_{CV} = \frac{\sum_{u \in V} \mathcal{D}(u)}{\mathcal{K}|V|} \quad (4)$$

General streaming model for edge-cut graph partitioning. At each iteration t ($1 \leq t \leq |V|$), where we read the tth vertex in the stream, a streaming edge-cut partitioner reads the vertex u_t and its neighbors $\mathcal{N}(u_t)$ and assigns u_t to one of the partitions. The assignment is based on evaluating a *score* function for each partition based on u_t, $\mathcal{N}(u_t)$, and the state of each partition in the tth iteration ($\mathcal{V}_i^t$). A general model for the assignment of u_t is:

$$\underset{1 \leq i \leq \mathcal{K}}{\text{argmax}} \left[\mathbf{h}(|\mathcal{V}_i^t \cap \mathcal{N}(u_t)|) - \mathbf{g}(|\mathcal{V}_i^t|) \right] \quad (5)$$

where $\mathbf{h}$ biases assigning the vertex to the partition that contains the greatest number of neighbors, thus minimizing the number of edge cuts, and $\mathbf{g}$ is the penalty term for the current size of the partition to satisfy balance constraints, thus encouraging equal partition growth. This heuristic is at the core of many edge-cut partitioners, and variants of Equation 5 can be found in them [3, 12, 21, 42, 50, 52].

3 CUTTANA ALGORITHM

Scope. CUTTANA is a vertex partitioner that operates on a static snapshot of a graph and is designed to improve workload latency and combined workload/partitioning latency for jobs on distributed vertex-centric systems. We designed CUTTANA so that it can be executed on commodity machines commonly used for distributed

processing in the cloud (concerning their memory constraints). The main focus of CUTTANA is on massive graphs (e.g., billion-scale graphs) for which in-memory partitioners (e.g., METIS) fail.

Overview. CUTTANA is a two-phase partitioner. The first phase is a streaming partitioner with delayed placement that creates an initial partitioning of the graph. The second phase is the refinement of the initial partitioning. We move vertices among the partitions to increase the partitioning quality (e.g., reducing edge-cuts or communication volume) while maintaining the balance condition.

The delayed placement in the first phase is incorporated into a streaming algorithm through a buffered streaming model. This model enables any classic streaming partitioner to delay the assignment of a vertex whenever necessary; we discuss this in Section 3.1. In Section 3.2, we explain the challenges of refinement and how our solution addresses them. Finally, in Section 3.3, we explain how we reduce the time overhead introduced by buffering and refinement.

3.1 Phase 1: Prioritized Buffered Streaming

Premature assignments: a problem in streaming partitioners. The primary intuition behind streaming partitioners is to assign each vertex to the partition containing the greatest number of neighboring vertices. The corresponding term for this greedy assignment in Equation 5 is $|\mathcal{V}_i^t \cap \mathcal{N}(u_t)|$. However, a partitioner frequently encounters a vertex for which many, or even all, of its neighbors are not yet assigned. We call such assignments *premature*. Mathematically, premature assignments happen when partitioning u_t and the number of assigned neighbors, $\sum_{1 \leq i \leq \mathcal{K}} |\mathcal{N}(u_t) \cap \mathcal{V}_i|$, is small or zero. Without adequate information about the assignment of neighboring vertices, the assignment of u_t can cause random/low-quality assignment and may increase the number of edge-cuts.

Challenges of avoiding premature assignments. A simple fix for premature assignment is delaying the assignment of these vertices and prioritizing the assignment of vertices with more already-assigned neighbors. As we partition more vertices, more information becomes available for vertices that were subjected to premature assignment. However, delayed partitioning requires storing all delayed vertices as well as their neighbors in a buffer. We have to store both vertices and their neighbors, because we need this information later to compute partitioning scores; in the streaming phase after reading a vertex and its neighbors, they are no longer accessible unless explicitly stored. Storing vertices and their neighbors could require $O(E)$ buffer space if done naively. To avoid consuming all of main memory, we should limit the buffer size to some constant, based on the system's available memory. High-degree vertices will consume more space (for their neighbors) than low-degree vertices. The key challenge is determining which vertices to buffer, when to evict, and how to manage the buffer efficiently.

Key Finding: buffering low-degree vertices is sufficient. Most large networks exhibit a power-law degree distribution [2, 27, 45], in which the majority of vertices are low-degree, and the majority of edges have at least one low-degree side [38]. Figure 3 (left) illustrates that most vertices are low-degree and (right) that most edges have at least one low-degree endpoint, in two large network datasets. Thus, high-quality placement of low-degree vertices leads to high-quality partitioning (i.e., fewer edge-cuts), because most edges have at least one low-degree endpoint. These low-degree
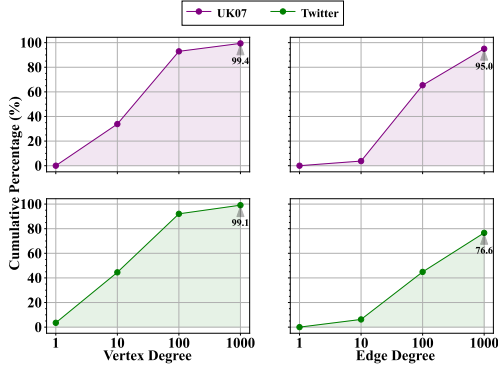
Figure 3: In large power-law graphs, the majority of vertices have low degrees (≤ 1000) (left charts), and the majority of edges have at least one low-degree endpoint (right charts), even though the maximum degree in these networks exceeds a million.

vertices are exactly the ones most likely to be prematurely assigned, because the probability of having zero or a low number of assigned neighbors is inversely proportional to the number of neighbors (Theorem 3.1). Recall that making good buffer eviction and placement decisions requires buffering vertices and their neighbors, so high-degree vertices consume more buffer space than low-degree vertices. Intuitively, buffering can be viewed as a way to reorder the input stream. Inevitably, the first vertices we place are, by definition, premature assignments. However, they guide later placement decisions; high-degree vertices provide more information to guide these decisions; that is, they attract neighboring vertices. Therefore, we choose to buffer low-degree vertices, because they are most likely to be prematurely assigned and consume little buffer space.

THEOREM 3.1. *In a streaming partitioner, the degree of a vertex is inversely proportional to the probability of the vertex being partitioned without knowledge of its neighbors.*

PROOF. When placing the tth vertex, v_t, with degree $d = |\mathcal{N}(v_t)|$, it will be partitioned with no knowledge if all of its d neighbors come after it in the stream. There are $|V| - t$ such positions, so there are $\binom{|V|-t}{d}$ possible orderings. All of the possible ways to place these d vertices in the stream is $\binom{|V|-1}{d}$. Thus, the probability that v_t is partitioned with no knowledge is the ratio of the number of orderings in which all the neighbors come after v_t to the total number of possible orderings of v_t's neighbors:

$$P = \frac{\binom{|V|-t}{d}}{\binom{|V|-1}{d}} = \frac{(|V|-1-d)!(|V|-t)!}{(|V|-1)!(|V|-t-d)!} = \prod_{i=1}^{t-1} \frac{|V|-d-i}{|V|-i}$$

As d decreases, the numerator increases, yielding higher probability for low-degree vertices. Since $|V| >> d$ for low-degree vertices, the fraction is close to 1, and P can be high, even for large t. □

Prioritized buffered streaming model. We take advantage of our key finding by buffering vertices that have a degree lower than a threshold, D_{max}. We select D_{max} such that the majority of edges have at least one endpoint that has the opportunity to be buffered.

Algorithm 1: CUTTANA's First Phase with User-Defined Buffer Score and Partitioning Score Functions

Data: Graph File: F, Degree Threshold: D_{max},
Vertex Count: $|V|$, Queue Size: max_qsize

```
// The buffer is a priority queue
// Storing vertices in decreasing
// order of buffer score
```
1   $Q \leftarrow PriorityQueue()$
2   **for** $i \leftarrow 1$ **to** $|V|$ **do**
```
      // Reading a vertex and neighbors
```
3     $v, \mathcal{N}(v) \leftarrow readLine(F)$
4     $v\_score \leftarrow bufferScore(v, \mathcal{N}(v))$
5   **if** $|\mathcal{N}(v)| \geq D_{max}$ **then**
6       $partitionVertex(v, \mathcal{N}(v))$
7   **else**
8       $Q.push(\{v\_score, v, \mathcal{N}(v)\})$
9   **if** $Q.size() == max\_qsize$ **then**
10      $t\_score, t, \mathcal{N}(t) \leftarrow Q.pop()$
11      $partitionVertex(t, \mathcal{N}(t))$

12   **while** $Q.size() > 0$ **do**
13     $t\_score, t, \mathcal{N}(t) \leftarrow Q.pop()$
14     $partitionVertex(t, \mathcal{N}(t))$

15   **function** $partitionVertex(v, \mathcal{N}(v))$
```
      // Finding best partition among the
      // K partitions using
      // partitioning score function.
```
16     $\mathcal{V}_{best} = findBestPartition(v, \mathcal{N}(v))$
17     $\mathcal{V}_{best} = \mathcal{V}_{best} \cup v$
18     $updateBufferScores(\mathcal{N}(v))$

This property prematurely assigns high-degree vertices to attract their neighbors into the same partition.

Algorithm 1 presents the pseudocode for CUTTANA's prioritized buffered streaming model. The buffer, denoted by Q, is a priority queue sorted in descending order of buffer score and has a capacity of max_qsize vertices. The buffer score is a user-defined function designed to prevent premature assignments. Our buffer score function for v_t is:

$$\frac{|\mathcal{N}(v_t)|}{D_{max}} + \theta \frac{\sum_{1 \leq i \leq \mathcal{K}} |\mathcal{N}(v_t) \cap \mathcal{V}_i|}{|\mathcal{N}(v_t)|} \tag{6}$$

Once the buffer fills, we prioritize partitioning the vertex with the highest *buffer score*. The rationale behind this buffer score is to assign higher buffer scores (leading to earlier eviction/placement) to vertices with more assigned neighbors, while simultaneously favoring assigning high-degree vertices and delaying the assignment of low-degree vertices (based on our key-finding).

θ is a hyperparameter whose value indicates how much to favor the number of assigned neighbors over the degree. By giving more weight to the fraction of assigned neighbors, more vertices will have a chance to be buffered. However, this means the vertices will

spend less time in the buffer and will be evicted with less information about their neighborhood. When a vertex is evicted from the buffer, it needs to be assigned to a partition. This assignment can be done using the same *partitioning score* function used in existing partitioners (Equation 5). In our implementation, we use the Fennel [52] partitioning score function, with a minor adjustment to achieve more edge-balanced partitions. To select the best partition for vertex v_t at time t, we use:

$$\underset{1 \leq i \leq \mathcal{K}}{\mathrm{argmax}}(|\mathcal{V}_i^t \cap \mathcal{N}(v_t)| - \delta(|\mathcal{V}_i^t| + \mu \sum_{x \in \mathcal{V}_i^t} |\mathcal{N}(x)|)), \qquad (7)$$

where δ is the exact penalty function used by Fennel. However, unlike Fennel, which considers only existing vertices in the partition for the penalty ($|\mathcal{V}_i^t|$), we adopt PowerLyra's hybrid-cut model [8], which incorporates the number of edges in the partition ($\sum_{x \in \mathcal{V}_i^t} |\mathcal{N}(x)|$) into the penalty function. Given that the number of edges exceeds the number of vertices, μ is the ratio of vertices to edges, normalizing their sum to ensure balanced growth of both vertices and edges within partitions during streaming.

After a vertex is assigned to a partition, we increase the scores of its buffered neighbors, since they have a newly assigned neighbor. This score update helps us prioritize partitioning vertices with more assigned neighbors, rather than partitioning vertices based on the random order in which they appear in the stream. We also perform a check: if all the neighbors of a vertex are assigned, we evict that vertex, a step omitted in Algorithm 1 for simplicity.

Space and time complexity. In the implementation, the buffer maintains a pair of the score and the vertex ID of each buffered vertex. We store neighbors of buffered vertices in a separate unordered map to avoid inserting/deleting the neighbor list to/from buffer after each single score update. Once a vertex is evicted from the buffer, we remove the map entry. In phase 1, the worst-case space overhead is $O(V)$ for storing a partition ID for each vertex and $O(D_{max} \times max\_qsize)$ for at most max_qsize buffered vertices, each of which can have at most D_{max} neighbors. To measure the initial buffer score for a vertex, we iterate through all its neighbors and check if they are assigned, which is $O(V+E)$ in total. For measuring the partitioning score, we added an optimization to Fennel. In partitioning each vertex, we first store the initial scores (balance penalties) for all $\mathcal{K}$ partitions in a heap, based on their current size. We iterate the neighbors, and after each assigned neighbor, we increase the score of the corresponding partition in the heap. At the end, we select the partition with the greatest score in the heap. With this optimization, the cost of measuring the partitioning score becomes $O(|V| + |E| \log(K))$ rather than the $O(K|V| + |E|)$ in the original implementation [1, 52]. This optimization is particularly important for sub-partitioning, which we describe in Section 3.2. Finally, after assigning each vertex, we update the score of its buffered neighbors. Worst case, we update the score for all of its neighbors. In total, for all vertices, this step is $O(|V| + |E| \log(max\_qsize))$. So, the overall phase 1 time complexity is $O(|V| + |E| \log(max\_qsize) + |E| \log(K))$.

3.2 Phase 2: Quality Refinement

Definition 3.2 (Trade & Maximality). We call a pair of vertex and partition index, $\langle v, b \rangle$ ($v \in V$ and $1 \leq b \leq \mathcal{K}$), a *trade* if, after moving v from its current partition to V_b, the total partitioning

quality increases and the balance condition is maintained. If there exists no trade for a partitioning, we call the partitioning *maximal*.

The quality of the streaming output is not maximal. After partitioning a graph using a streaming partitioner, it is possible to apply trades to improve the partitioning quality, because, in practice, the balance is relaxed ($\epsilon > 0$), and the streaming partitioner, even with buffering, places many vertices based only on partial information. However, when applying trades, we have a more complete view of the graph. We now present our scalable refinement algorithm to enhance the partitioning produced in Phase 1 using these trades.

Challenges of finding trades and *sub-partitioning*. Finding and applying trades requires keeping track of vertex neighborhoods. While this is possible for small graphs, it is not scalable to large graphs. To solve this problem, we coarsen the graph into a summarized version with a substantially reduced number of vertices and edges. Each coarsened vertex consists of a subset of the original vertices from the same partition. The coarsened vertices are connected with edges that are weighted according to the number of edges between their members (vertices in the original graph). We call this process *sub-partitioning* and the coarsened vertices *sub-partitions*.

Definition 3.3 (Sub-Partitioning). Assuming $\mathcal{K}' \in \mathbb{N}$, equally-sized disjoint sets $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_{\mathcal{K}'}$ are a sub-partitioning of $\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_{\mathcal{K}}$, if $\bigcup_{1 \leq i \leq \mathcal{K}'} \mathcal{S}_i = V$, and for all $\mathcal{S}_i$ there exists only one $\mathcal{V}_j$ such that $\mathcal{S}_i \subset \mathcal{V}_j$ and $\mathcal{K}'$ is the total number of sub-partitions.

Definition 3.4 (Sub-Partition Graph). A sub-partition graph consists of sub-partitions $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_{\mathcal{K}'}$ as its vertices and the edge between $\mathcal{S}_i, \mathcal{S}_j$ is a weighted edge denoted by:

$$\mathcal{W}(\mathcal{S}_i, \mathcal{S}_j) = \left| \{ \langle u, v \rangle \in E | u \in \mathcal{S}_i \wedge v \in \mathcal{S}_j \} \right|$$

.

Refinement as trades on the sub-partition graph. The sub-partitions can be moved between partitions via trades. Moving a sub-partition involves relocating all of its members to another partition. The goal is to reduce the edge cuts, realized as a reduction of the sum of weights of the edges between sub-partitions from different partitions. We present a scalable algorithm designed to find and apply all trades in the sub-partition graph to improve the overall quality.

Coarsening and assigning sub-partitions is another partitioning problem. The vertices comprising a sub-partition always remain together after each trade. Our goal is to maximize the number of internal edges within a sub-partition, thereby reducing the total edges between sub-partitions. Additionally, we want to control the size of the sub-partitions and avoid skewed sizes, as such imbalances complicate maintaining the balance condition during trades. This problem mirrors the original graph partitioning problem, and we approach it similarly. We assume a constant number of subpartitions ($\frac{\mathcal{K}'}{\mathcal{K}}$) in each partition. During Phase 1, when a vertex is placed in a partition, it is also assigned to a sub-partition within the selected partition. Cuttana's Phase 1 can be implemented using any partitioning algorithm. By uniformly assigning sub-partitions to partitions, we ensure that, by the end of streaming, we have
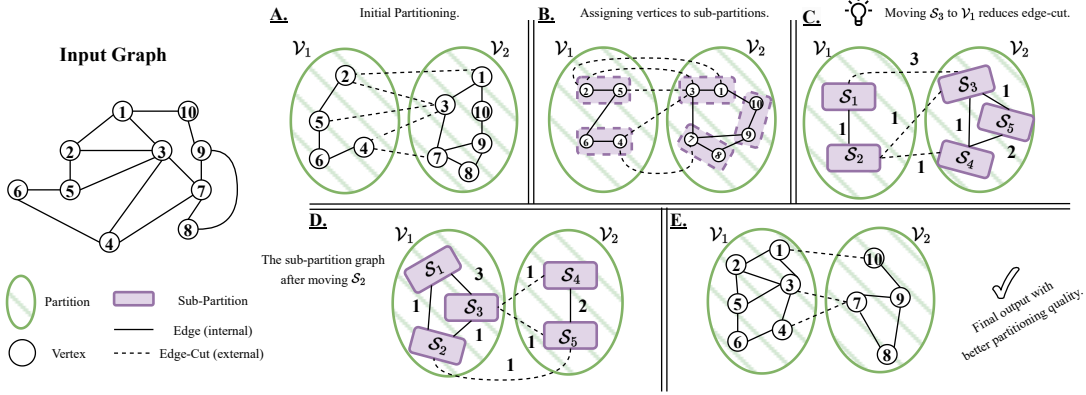
Figure 4: Partitioning of a graph and applying sub-partitioning and refinement with $\epsilon = 0.2$ balance condition and $\mathcal{K}' = 5$.

an initial state for refinement that is as good as the result of $\mathcal{K}$-way partitioning of the streaming algorithm used in the first phase. Hence, any streaming partitioner guarantees better results after refinement. Additionally, we solve a smaller partitioning problem by having $O(\frac{\mathcal{K}'}{\mathcal{K}})$ choices for each vertex instead of $O(\mathcal{K}')$. We use the scoring function described in Equation 7 to assign vertices to sub-partitions but with different hyperparameters.

Refinement Algorithm. Although we coarsen the graph, the scalability and efficiency of the refinement algorithm determine how large we can make $\mathcal{K}'$. Finer-grained sub-partitions (larger $\mathcal{K}'$) produce better and more precise refinements. Our refinement algorithm is a greedy iterative algorithm that, in each step, applies the trade that produces the greatest quality improvement. The algorithm stops when no further trade is possible, and the partitioning is maximal. In each iteration, we consider all pairs of partitions and find the best subpartition trade among them. To implement this algorithm efficiently, we define and use data structures that we can calculate once in Phase 1 and update efficiently during Phase 2.

Let $\mathcal{P}'(\mathcal{S}_i)$ represent the index of the partition containing $\mathcal{S}_i$. Let *ECP* (edge cut per partition) be a data structure holding the number of edge cuts produced by placing a particular sub-partition in a partition. Hence, $ECP_{\mathcal{S}_i, \mathcal{V}_{dest}}$ is the sum of all the edge weights between $\mathcal{S}_i$ and the sub-partitions that are not in partition $\mathcal{V}_{dest}$:

$$ECP_{\mathcal{S}_i, \mathcal{V}_{dest}} = \sum_{1 \leq j \leq \mathcal{K}'} \mathcal{W}(\mathcal{S}_i, \mathcal{S}_j) \ [\mathcal{P}'(\mathcal{S}_j) \neq dest] \quad (8)$$

Next, define $DEC_{\mathcal{S}_i, \mathcal{V}_{src}, \mathcal{V}_{dest}}$ as the decrease in edge-cut produced by moving $\mathcal{S}_i$ from $\mathcal{V}_{src}$ to $\mathcal{V}_{dest}$, where $src = \mathcal{P}'(\mathcal{S}_i)$ and all other subpartition assignments are unchanged. When this value is negative, moving $\mathcal{S}_i$ to $\mathcal{V}_{dest}$ increases the edge-cut and worsens quality. The value of *DEC* can be computed as:

$$DEC_{\mathcal{S}_i, \mathcal{V}_{src}, \mathcal{V}_{dest}} = ECP_{\mathcal{S}_i, \mathcal{V}_{src}} - ECP_{\mathcal{S}_i, \mathcal{V}_{dest}}. \quad (9)$$

We store all *DEC* values in the move-score structure (*MS*). Each $MS_{\mathcal{V}_{src}, \mathcal{V}_{dest}}$ stores all $DEC_{\mathcal{S}_i, \mathcal{V}_{src}, \mathcal{V}_{dest}}$. To find the best trade, we iterate through all possible partition pairs $(\mathcal{V}_i, \mathcal{V}_j)$ and query $MS_{\mathcal{V}_i, \mathcal{V}_j}$ to determine the best trade (largest *DEC*) assuming the source partition is $\mathcal{V}_i$ and the destination is $\mathcal{V}_j$. Thus, we iterate over a total of $O(\mathcal{K}^2)$ move-score sets. To maintain the balance

condition, we keep track of the size of each partition. If, at any move, the destination partition reaches capacity, we exclude this move from the set of possible moves.

LEMMA 3.5. *The size of $MS_{\mathcal{V}_i, \mathcal{V}_j}$ and the number of sub-partitions in a partition at any point of refinement is $O(\frac{\mathcal{K}'}{\mathcal{K}})$.*

PROOF. By the definition of trade, we always maintain the balance condition, and since sub-partitions are equal-sized, a partition can have at most $(1 + \epsilon) \frac{\mathcal{K}'}{\mathcal{K}}$ subpartitions. The number of sub-partitions in a partition is $O(\frac{\mathcal{K}'}{\mathcal{K}})$, because ϵ is a small constant. Also, the size of $MS_{\mathcal{V}_i, \mathcal{V}_j}$ is bounded by the number of sub-partitions currently in $\mathcal{V}_i$. □

The size of each move-score set is $O(\frac{\mathcal{K}'}{\mathcal{K}})$ (Lemma 3.5). We implement each move-score set as a Segment Tree [9], which means we can find the maximum value of a set in $O(1)$ and update it in $O(log(\frac{\mathcal{K}'}{\mathcal{K}}))$. Updating is implemented by deleting the *DEC* value and inserting a new value.

Updating Variables After a Trade. The main challenge in the refinement is efficiently updating *MS*. Moving S_x from $\mathcal{V}_{src}$ to $\mathcal{V}_{dest}$ involves updating the *ECP* values and changing the *DEC* values stored in *MS*. We need to update *ECP* only for neighboring vertices of S_x. For any neighbor $S_i \in \mathcal{N}(S_x)$, we perform:

$$\begin{aligned} ECP_{\mathcal{S}_i, \mathcal{V}_{src}} &= ECP_{\mathcal{S}_i, \mathcal{V}_{src}} + \mathcal{W}(\mathcal{S}_i, \mathcal{S}_x) \\ ECP_{\mathcal{S}_i, \mathcal{V}_{dest}} &= ECP_{\mathcal{S}_i, \mathcal{V}_{dest}} - \mathcal{W}(\mathcal{S}_i, \mathcal{S}_x) \end{aligned} \quad (10)$$

In the worst case, S_x can be neighbors to all other sub-partitions, making this step $O(\mathcal{K}')$. Updating $DEC_{\mathcal{S}_i, \mathcal{P}'(\mathcal{S}_i), \mathcal{V}_j}$ naively can result in $O(\mathcal{K}'\mathcal{K})$ updates. Changing each entry in the move-score set is $O(log(\frac{\mathcal{K}'}{\mathcal{K}}))$, so the naive approach is $O(\mathcal{K}'\mathcal{K} log(\frac{\mathcal{K}'}{\mathcal{K}}))$ in total, because the moved sub-partition can have $O(\mathcal{K}')$ neighbors, and those neighbors can go from their partitions to $O(\mathcal{K})$ other partitions. However, it can be done in $O(\mathcal{K}' log(\frac{\mathcal{K}'}{\mathcal{K}}))$ by exploiting the following theorem.

THEOREM 3.6. *After applying each trade, we require updating only $O(\mathcal{K}')$ entries in the move-score sets.*

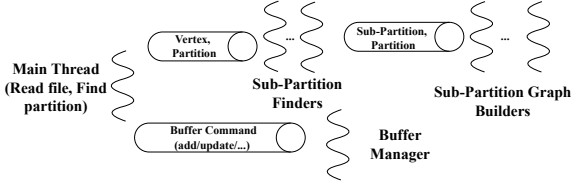PROOF. After moving S_x, we categorize its neighbours $S_i \in \mathcal{N}(S_x)$ into two cases:

19

Figure 5: Parallel model of CUTTANA execution.

(1) $\mathcal{P}'(\mathcal{S}_i) \in \{src, dest\}$: In this case, for all of the partitions $\mathcal{V}_j$, we have to update $DEC_{\mathcal{S}_i, \mathcal{P}'(\mathcal{S}_i), \mathcal{V}_j}$. However, because we have $O(\frac{\mathcal{K}'}{\mathcal{K}})$ sub-partitions in both $\mathcal{V}_{src}, \mathcal{V}_{dest}$ (Lemma 3.5), updating $DEC_{\mathcal{S}_i, \mathcal{P}'(\mathcal{S}_i), \mathcal{V}_j}$, even for all of the partitions $\mathcal{V}_j$, is $O(\mathcal{K}.\frac{\mathcal{K}'}{\mathcal{K}})$ or $O(\mathcal{K}')$.

(2) $\mathcal{P}'(\mathcal{S}_i) \notin \{src, dest\}$: In this case, if the neighbor is in neither the source nor destination partitions, we need to only update $DEC_{\mathcal{S}_i, \mathcal{P}'(\mathcal{S}_i), \mathcal{V}_j}$ where $\mathcal{V}_j \in \{\mathcal{V}_{src}, \mathcal{V}_{dest}\}$, because the number of edges from $\mathcal{S}_i$ to the sub-partitions in other partitions are unchanged. Since the number of neighbors is bounded by the number of sub-partitions, we also have to perform $O(\mathcal{K}')$ updates, but this time only for two target partitions.

□

Finally, we have to update DEC variables and move-score sets for $\mathcal{S}_x$. For all partitions $\mathcal{V}_j$, we have to remove all of $DEC_{\mathcal{S}_x, \mathcal{V}_{src}, \mathcal{V}_j}$ from $MS_{\mathcal{V}_{src}, \mathcal{V}_j}$ and add all $DEC_{\mathcal{S}_x, \mathcal{V}_{dest}, \mathcal{V}_j}$ to $MS_{\mathcal{V}_{dest}, \mathcal{V}_j}$, since $\mathcal{S}_x$ changed its partition. In summary, we find the best move for each iteration in $O(\mathcal{K}^2)$ and update all variables in $O(\mathcal{K}' \log(\frac{\mathcal{K}'}{\mathcal{K}}))$. The number of refinement steps is finite, as we decrease edge cut in each step, and edge-cut is finite. The algorithm stops at the maximal partitioning when there is no move left. However, due to the coarse granularity of sub-partitions and weighted edges, in practice, the improvements are also coarse-grained. It is possible to stop the refinement process early when the best move improves edge-cut by less than a threshold ($Thresh$). This early stop provides a time/quality trade-off, and the threshold produces a worst-case bound for the number of steps of $\frac{|E|}{Thresh}$, since the upper bound for edge-cuts is $|E|$, and each step improves at least $Thresh$ edge-cuts.

3.3 Parallel Partitioning and Implementation

Parallelization. Existing streaming partitioners use only a single thread. This underutilizes resources in modern multicore computers. We leverage the unused cores to parallelize CUTTANA, thereby reducing the overhead introduced by buffering and refinement. Our approach involves dividing the computational load in such a way that different threads do not write to shared variables, thus avoiding the need for locking. Thread communication is facilitated using hardware-optimized lock-free queues [32, 54]. At a high level, Phase 1 runs in parallel, partitioning, sub-partitioning, and building the data structures for refinement; Phase 2 simply applies trades.

The primary thread reads the file, selects vertices for buffering, and partitions them after eviction from the buffer. Once a vertex is assigned to a partition, other threads are notified to determine

the sub-partition of that vertex and update variables for refinement (Figure 5). Additionally, we have a buffer manager thread that pushes/pops the buffer based on main-thread commands and applies changes to the buffer score whenever a new vertex has been partitioned. Threads of the same kind (i.e., subpartition finders) shard the shared variables based on a key (e.g., vertex ids, sub-partition id).

4 EXPERIMENTAL ANALYSIS

Our experimental analysis answers the following research questions.

- **RQ1:** How does CUTTANA partition quality compare to that of existing approaches?
- **RQ2:** Given that existing vertex partitioners impose a vertex-balance constraint, how much edge imbalance do they produce? How does the partition quality change if they adopt an edge-balance constraint?
- **RQ3:** How much do buffering and refinement affect partition quality?
- **RQ4:** How does CUTTANA partitioning affect the performance of Distributed Graph Analytics?
- **RQ5:** How does CUTTANA partitioning affect the performance of a Distributed Graph Database?

Datasets. Table 1 shows the characteristics of datasets used in our study. We selected graphs of different sizes and domains to represent various use cases. The web graphs are hyperlink networks where vertices are webpages and the edges are links. In social networks, the vertices are users and the edges are follow/friend relationships. All of the datasets were obtained from the Konect network repository [29] except for the LDBC social network benchmark, which we obtained using the LDBC generator [10], the US-Roads dataset [53], and the RMAT synthetic dataset which we generated using ParMAT [26]. We used real-world natural graphs, including both big and small graphs, to analyze the quality of partitioning for different algorithms. We used large graphs, both real-world and synthetic, for distributed graph analytics, and finally, we used the LDBC benchmark for our graph database evaluation.

Baselines. We compare CUTTANA to three other streaming partitioners: FENNEL [52], LDG [50], and HEISTREAM [14]. FENNEL and LDG are score-based streaming vertex partitioners. FENNEL is the best baseline to show the benefits of buffering and refinement, since CUTTANA is implemented on top of FENNEL and uses the same scoring function. HEISTREAM is recent work that has the same motivation as CUTTANA, i.e., bridging the gap between streaming and in-memory solutions in quality and scalability. HEISTREAM reads and assigns vertices in batches and claims to beat FENNEL [14]. We used the implementation of FENNEL and LDG provided by Pacaci and Ozsu [43], since the official code is not available, and we obtained HEISTREAM from the authors. In graph analytic benchmarks, where it's possible to use edge-partitioners [43], we also compare CUTTANA to HDRF [45] and GINGER [8].

Experimental Setup and Reproducibility. We conducted all our experiments, including partitioning, analytics, and graph database benchmarks, on a private cluster of 16 machines, each equipped with an 8-core Intel® Xeon® Silver 4309Y Processor and 64 GB of RAM. Unless otherwise specified, we ran CUTTANA with
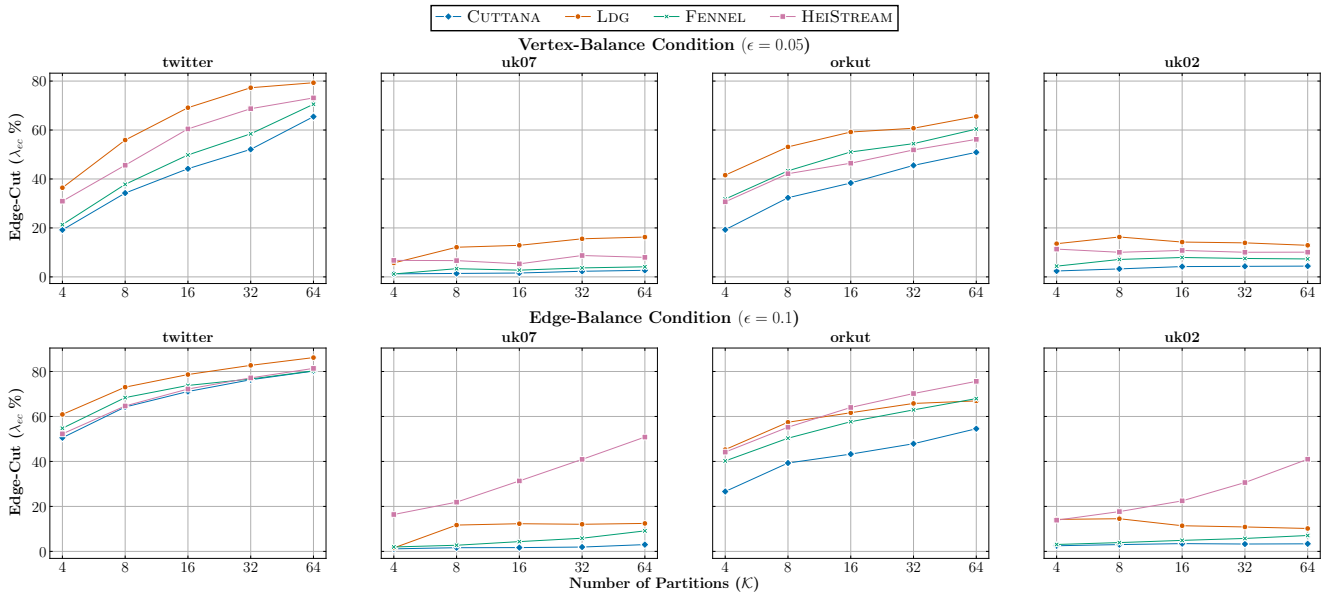
Figure 6: The partitioning quality of CUTTANA excels consistently across varying numbers of partitions.

Table 1: Graph datasets used in the evaluation.

Name (Domain)	# Edges # Vertices	Degree Distribution (Avg./Max. Degree)
US-Roads (Road)	28M 23M	Low Degree (2.4,18)
Orkut (Social)	117M 3M	Power Law (76,33K)
UK domains - 2002 (Web)	261M 18M	Power Law (28,195K)
LDBC-SNB-SF1000 (Social)	490M 3M	Heavy Tailed (248,3K)
RMAT Large (Synthetic)	1B 10M	Power Law (200,236K)
Twitter (Social)	1.4B 41M	Power Law (70,3M)
UK domains - 2007 (Web)	3.3B 105M	Power Law (62,975K)

D_{max} = 1000, *max_qsize* = 10^6 vertices (consuming at most 4 GB of DRAM) and determined the number of sub-partitions such that $\frac{\mathcal{K}'}{\mathcal{K}}$ = 4096 for all datasets except Twitter on which we set D_{max} = 100 and determined the number of sub-partitions such that $\frac{\mathcal{K}'}{\mathcal{K}}$ = 256. The code for CUTTANA and the framework for the application study (analytics, databases) are publicly available. We conducted the application study based on the benchmarking framework provided by Pacaci and Ozsu [43], making minor modifications to update deprecated packages, add support for HEISTREAM

and CUTTANA, and add some additional features. The partitioning process is deterministic as we fixed the random seed used for tie-breaking among partitions with the same score. We also disabled the buffer-manager thread. The use of a buffer-management thread introduces scheduling randomness, which we disable by offloading the task to the main thread to ensure reproducibility. We used the baselines with default hyperparameters.

4.1 Quality Metrics Analysis

Improving Edge-cut and Communication Volume. We address **RQ1** by partitioning datasets under both edge/vertex-balance constraints using all baseline algorithms. We measure the communication volume and edge-cut as indicators of network overhead in distributed applications with/without message aggregation, respectively. Table 2 shows that CUTTANA produces better quality partitions, relative to Fennel, in nearly all scenarios. The benefit is most pronounced for the largest graphs (twitter and uk07) in edge-balance mode, suggesting that it is possible to effectively partition massive graphs that cannot be partitioned by in-memory partitioners. Since the reported metrics are normalized, their relative difference $\left(\frac{|\lambda_1-\lambda_2|}{\max(\lambda_1,\lambda_2)}\right)$ is an underestimate of the reduction in network overhead.

CUTTANA consistently improves partitioning quality from 6% to 59%. This improvement reflects network overhead, which is the dominant overhead in distributed graph processing, so we anticipate a more significant improvement in end-to-end application latency as well. In large graphs such as Twitter and UK07, CUTTANA produces better partitioning quality than HEISTREAM by up to 19% and 93%, respectively. However, in the US-Roads datasets, HEISTREAM produces better partition quality than CUTTANA. HEISTREAM's authors told us that the algorithm is sensitive to ordering and performs best when each batch consists of vertices

Table 2: Analysis of Partitioning Quality on eight partitions ($\mathcal{K} = 8$). The boldfaced numbers shaded blue indicate the best result for each graph and balance condition. The Improv. column shows the improvement of CUTTANA over FENNEL.

| Quality Metric | Dataset | Edge-Balance Condition (*EB*) ($\epsilon = 0.10$) | | | | Vertex-Balance Condition (*VB*) ($\epsilon = 0.05$) | | | | Improv. | |
		CUTTANA	FENNEL	HEISTREAM	LDG	CUTTANA	FENNEL	HEISTREAM	LDG	*EB*	*VB*
edge-cut λ_{EC} (%)	*usroad*	27.93	31.22	**16.84**	30.06	22.5	31.15	**10.48**	30.05	11%	28%
	orkut	**39.3**	50.33	55.22	57.43	**32.33**	43.31	42.15	53.11	22%	26%
	uk02	**3.03**	3.91	17.7	14.53	**3.26**	7.12	10.05	16.3	23%	55%
	twitter	**64.21**	68.39	64.67	73.04	**34.09**	37.80	45.62	55.9	6%	10%
	uk07	**1.64**	2.73	21.9	11.71	**1.4**	3.35	6.65	12.11	40%	59%
communication volume λ_{CV} (%)	*usroad*	7.93	9.06	**4.41**	8.68	6.09	9.04	**2.97**	8.68	13%	33%
	orkut	**44.82**	63.83	65.48	63.42	**44.09**	55.95	48.43	61.01	30%	22%
	uk02	**4.25**	5.45	6.74	6.74	4.68	5.63	**3.78**	6.78	22%	17%
	twitter	**40.91**	43.72	50.23	46.77	**41.3**	47.04	44.04	47.39	6%	13%
	uk07	**4.5**	7.21	8.98	6.12	**3.88**	5.29	3.99	6.01	38%	27%

from the same neighborhood with many edges among them. The size and original ordering of US-Roads are ideal for HEISTREAM. On the other hand, CUTTANA's buffering is robust to input order; the only case in which CUTTANA does not provide the lowest edge-cut (communication volume) is when the original input order happens to be ideal for HEISTREAM. Figure 6 shows partition quality as a function of the number of partitions. While FENNEL and HEISTREAM outperform each other depending on the dataset and balance condition, CUTTANA outperforms both.

The Case for Edge-Balance using Vertex Partitioners. Pacaci and Ozsu [43] uncovered two key properties of state-of-the-art vertex partitioners. They demonstrated that both FENNEL and LDG exhibit lower network overhead than edge partitioners, but they suffer from significant worker imbalance, rendering them less compelling. In some scenarios, random partitioning produced better application performance due to its superior load balancing. We determined that the root cause lies in using a vertex-oriented balance constraint. Balancing the number of vertices in a partition does not necessarily balance the number of edges. However, edge-balance is crucial from a computational load-balancing perspective, because almost all graph algorithms iterate over edges. In other words, the number of edges in a partition determines the workload on each participant in a distributed computation, and edge imbalance leads to stragglers. Edge balance is more critical than vertex balance since the number of edges dominates the number of vertices. Moreover, the balanced assignment of edges is crucial in memory-constrained scenarios. When the number of edges in each partition varies, we must either over-provision memory (which is expensive) or suffer the consequences that some workers will be computing on out-of-memory data, thus exacerbating the delay that stragglers impose.

We demonstrate this issue in Figure 7. We use all the baselines and modify CUTTANA to use a vertex-balance constraint instead of its preferred edge-balance constraint. We set $\epsilon = 0.05$ and show the ratio of the maximum number of edges in any partition to the average number of edges across all partitions. Although the

vertices are balanced among the partitions, the edges are hugely imbalanced. This suggests that, regardless of the partitioning scheme and dataset, using the vertex-balance condition, which is prevalent in the literature[1, 3, 14, 50, 52], yields partitions with too many edges, leading to stragglers when computing in parallel. For example, on Twitter, using all of the partitioners in vertex balance mode causes one worker machine to have at least *4x* more load than other machines. Overweight partitions also risk producing more network overhead. CUTTANA offers the user both vertex and edge balance options. HEISTREAM was originally implemented for vertex-balance, but the authors added the edge-balance feature upon our request. We added edge-balance support to FENNEL and LDG using the same approach as that used in CUTTANA.

The answer to **RQ2** can also be found in Figures 6 and 7 and Table 2, which show that 1) satisfying edge balance makes edge cut worse, and 2) vertex-balance produces significant edge imbalance (as discussed above). However, CUTTANA produces the best partition quality when satisfying either balance constraint. In the rest of our evaluation, we use CUTTANA's edge-balance mode and the original baseline implementations. Experimentally, especially for communication volume and on graphs other than Twitter, the additional overhead of quality difference introduced by satisfying edge-balance was amortized in execution time by having more even computation and network overhead distribution.

Ablation Study & Partitioning Latency. We analyzed the isolated contributions of the two main components of CUTTANA, as shown in Table 3, to answer **RQ3**. Generally, buffering was the main contributor to quality improvement. The relative improvement of refinement was higher when there was no buffering and the initial partitioning had lower quality. We also demonstrate the benefit of prioritizing low-degree vertices by removing degree constraints and biases in the first phase (CUTTANA without Refine & Degree Bias in Table 3). The results for this mode were slightly better than FENNEL, as we consider assigned neighbors, but compared to prioritizing low-degree vertices, the results were considerably worse since the motivations mentioned in Section 3.1 no longer hold.
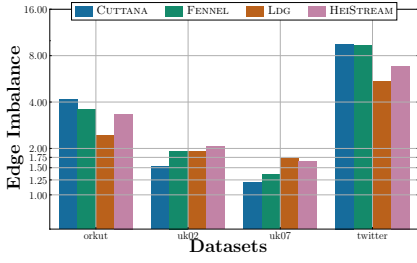
Figure 7: Baselines partitioners and Cuttana when using a vertex-balance condition (which is not Cuttana's default) can lead to uncontrolled edge-imbalance and uneven load distribution.

Table 3: Contribution of different components of Cuttana to the final partitioning quality ($\mathcal{K} = 16$). The numbers represent the normalized edge-cut (λ_{EC}), and the percentages indicate the improvement over Fennel.

Algorithm	Orkut	Twitter	UK07	UK02
Cuttana	38.3 (25%)	44.1 (11%)	1.5 (52%)	2.7 (66%)
Cuttana w/o Refine	40.7 (20%)	47 (6%)	1.7 (45%)	4.9 (38%)
Cuttana w/o Refine & Degree Bias	49.4 (3%)	48.1 (3%)	2.9 (6%)	6.6 (16%)
Cuttana w/o Buffer	45.9 (10%)	48.2 (3%)	2 (35%)	6.2 (22%)
Cuttana w/o Buffer & Refinement (Fennel)	51	49.8	3.1	7.9

Figure 8: Memory overhead (log-scale) and time efficiency of Cuttana compared to baselines and single-thread implementation (Cuttana-S)

Figure 9: Impact of Buffer Size on Time and Quality for uk07

Figure 10: Effect of D_{max} and θ on Partitioning Quality

Figure 8 compares Cuttana's memory consumption and partitioning time to the baselines. The memory overhead is high relative to Fennel and Ldg; however, this is not a cause for concern as the overhead is independent of graph size and consumes only a small fraction of the main memory available on today's commodity computers. Cuttana has a small additional time overhead compared to Fennel and is nearly twice as fast as HeiStream for large graphs. In Table 4, we demonstrate that we more than compensate for Cuttana's time overhead, relative to Fennel, by running analytic tasks much more quickly.

Figure 9 highlights the tradeoffs that a user can make when configuring Cuttana. Both partitioning time and memory consumption are governed by the selection of the buffer size, $|Q|$, and the number of subpartitions, K'. Cuttana performs more work than Fennel due to buffering, updating buffer scores, selecting sub-partitions, and updating the data structures we use to optimize refinement. Using lock-free queues, we assign these tasks to a background thread. This reduces the overhead of enqueuing requests, draining the buffer at the end of streaming, and refinement.

Large buffer sizes and numbers of sub-partitions can cause the background thread to fall behind the main thread. If the queues
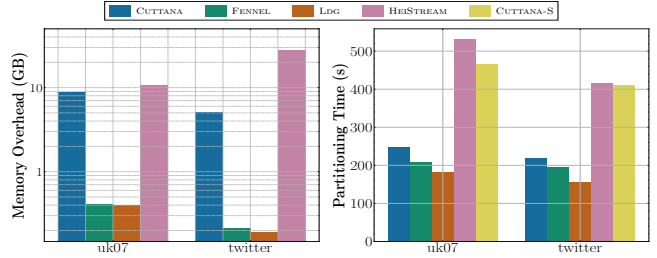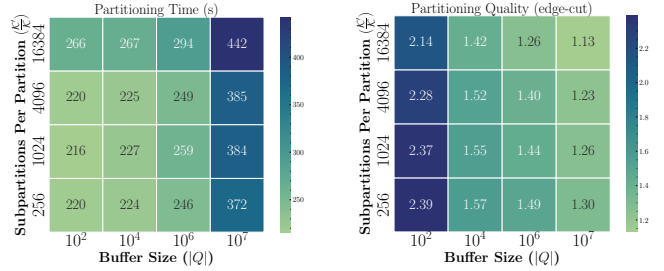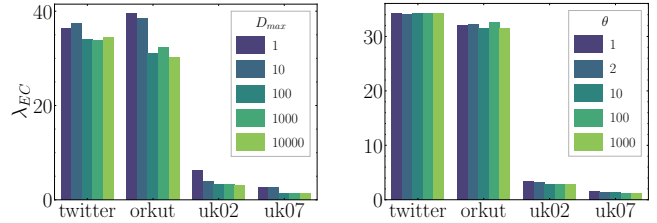
fill, then the main thread blocks. Cuttana maintains competitive performance up to 4096 sub-partitions and 10^6 vertices in the buffer. We use these as the default values. While selecting larger parameters can yield higher quality partitions, it should be undertaken cognizant of the impact on partitioning time. Figure 10 shows the effects of the buffering threshold, D_{max}, and the score function scale parameter, θ. It is important to set D_{max} such that the majority of edges have at least one endpoint that can be buffered. For a real-world power-law graph, this value is approximately 100 and is an attribute of the graph's degree distribution, independent of its size. Figure 10 shows values of D_{max} greater than 100 produce only minor changes, since the fraction of vertices that can be buffered does not change considerably (see Figure 3). However, increasing D_{max} increases the memory/time overhead of buffering, or for fixed memory, it reduces the number of vertices that can be buffered. Therefore, we suggest setting D_{max} between 100 and 1000. θ is a hyperparameter for our algorithm, and the optimal value varies per dataset. However, we used a fixed value in all other experiments to ensure that the default setting can still outperform baselines.

Table 4: The latency, in seconds, of PageRank (PR), Connected Components (CC), and Single Source Shortest Path (SSSP) workloads using different partitioners. The boldfaced numbers shaded blue indicate the best result for each workload.

Dataset/ Algorithm		Partitioning Scheme						Performance over the best Vertex Partitioner	Performance over the best Partitioner
		Cuttana	Fennel	Ldg	HeiStream	Hdrf	Ginger		
twitter	PR	**168**	813	811	488	413	492	66%	59%
	CC	**33**	76	80	81	70	108	57%	53%
	SSSP	**42**	176	202	117	81	77	64%	45%
uk07	PR	**141**	293	336	193	1227	1269	27%	27%
	CC	**63**	86	96	84	419	548	25%	25%
	SSSP	**49**	61	63	54	181	147	9%	9%
rMat-XL	PR	**144**	514	576	376	205	430	61%	29%
	CC	**61**	93	112	86	74	82	29%	25%
	SSSP	**53**	89	97	65	59	81	18%	10%

4.2 Application Study

We conduct case studies focusing on distributed graph analytics and graph databases to investigate how enhancements in quality metrics impact the performance metrics of these applications (e.g., throughput and execution time). We develop our application study framework on top of the benchmarking framework provided by Pacaci and Ozsu [43]. We report performance metrics, but due to space constraints, we refer the reader to the original work for more information about the specifications [43].

Distributed Graph Analytics. Table 4 shows the results of running three different algorithms on a Powerlyra cluster with 16 machines [8] to assess the performance of various partitioning schemes on graph analytics to answer **RQ4**. We ran PageRank for 30 iterations, connected components until we found all connected components, and single-source shortest path from a random vertex. We report the average latency of three runs. In practice, an algorithm can be executed multiple times (e.g., finding the shortest path from multiple sources in graphs with millions of vertices), which further increases the total latency improvement by using Cuttana. We show results for only the three largest graphs since small graphs can be processed more efficiently on a single machine than on multiple machines, because the network overhead introduced by adding a machine is not amortized by the parallelization achieved [40].

In social network graphs, vertex partitioners other than Cuttana, suffer from significant load imbalance, overshadowing any advantages in network usage. This aligns with findings reported by Pacaci and Ozsu [43]. We illustrate the reason for this imbalance in Figure 7. We used the edge-balance version of Cuttana and the original implementation of other baselines. In web graphs, the worker imbalance was less pronounced, and vertex partitioners with better network overhead outperformed edge partitioners due to different message-passing protocols and low replication of low-degree vertices [43]. More performance metrics are in Figure 2. Among all algorithms, we observed the greatest performance improvement in PageRank. In this algorithm, most vertices are active in all iterations, stressing the system's network. Cuttana,

Table 5: Partitioning Metrics and Performance (throughput in the unit of query per second) of one-hop and two-hop neighborhood retrieval on LDBC social network benchmark.

Metric	Cuttana	Fennel	HeiStream	Ldg
Edge-cut	37.49	47.72	53.26	74.22
Edge-imbalance	1.09	1.13	1.8	1.89
Vertex-imbalance	1.03	1.00	1.05	1.05
one-hop (q/s)	**2776**	2595	2381	1998
two-hop (q/s)	**232**	189	164	131

which both balances edges like edge partitioners do and exploits the lower network overhead of vertex partitioners, provides a "best-of-both-worlds" choice. Cuttana achieves our goal of producing sufficiently high-quality partitioning for large graphs that it improves runtime performance in analytical workloads.

Distributed Graph Database. We conducted the LDBC social network benchmark [10] on a JanusGraph cluster of 4 machines with 24 concurrent client threads, using Cassandra as the backend storage engine. Our goal is to observe how improvements in edge-cut and edge imbalance can translate into improved throughput in a distributed graph database, addressing **RQ5**. The queries and graph were generated by the LDBC generator. The improvement in graph databases is smaller than in analytics, because existing partitioners produce less edge imbalance on the LDBC-generated graph than we observed on other graphs, e.g., Twitter. Additionally, LDBC one-hop and two-hop queries limit the number of returned neighbors, which we believe puts the system under less stress, and many queries can be answered locally, although there exist neighbors in other partitions (machines). We observed a 23% improvement in the throughput of two-hop queries and a 7% improvement for one-hop queries, without a major difference in tail latency.

5 RELATED WORK

Distributed graph analytics has gained significant attention [7, 8, 11, 18, 19, 22, 31, 56, 60] since the introduction of Pregel [33]. Many partitioning strategies have been proposed to reduce network overhead and address load imbalance [3, 8, 28, 37, 45, 46, 50, 52, 58], since partitioning plays a crucial role in application latency [43]. Most of the recent advances in graph partitioning are in edge partitioning, which is unsurprising, since edge partitioners produce better edge balance than do vertex partitioners, and edge balance leads to even load distribution. Cuttana takes a different approach and imposes an edge-balance condition while partitioning by vertex. Hdrf [45] and Ginger [8] are two popular partitioners that reduce vertex replication and exhibit the best performance on large graphs [43]. Clugp [28] and Hpcd [46] are more recent edge-partitioners. Clugp provides a fast restreaming partitioning solution, while Hpcd transforms the problem into a combinatorial design problem.

However, some systems require vertex-partitioning [14, 43], and the message-passing protocol of the system changes when the graph is vertex-partitioned [43]. Stanton et al. analyzed multiple scoring functions for streaming vertex partitioners and proposed Ldg [50]. Later, Fennel [52] introduced a new scoring function with the same greedy, score-based model that outperformed Ldg and remained state-of-the-art for an extended period. HeiStream [14] and Spnl [58] are recent streaming partitioners whose evaluations showed it to be better than Fennel. We found that Cuttana outperforms both Fennel and HeiStream, especially on large graphs, which is the most compelling use case for streaming partitioners. Unfortunately, the code for Spnl was not available. However, in the common datasets UK07 and UK02, our model exhibited better partitioning quality than that reported by Spnl.

Finally, there are in-memory partitioners for both edge- and vertex-partitioning [24, 44, 48, 61]. Metis [24] is considered the gold standard for vertex-partitioning and Ne [61] for edge-partitioning. In-memory partitioners inspired our coarsening and refinement strategy, but we adopted a different approach to facilitate scalability. In-memory partitioners offer better quality than streaming partitioners in medium-sized graphs; however, they fail to partition billion-scale graphs [35, 38, 59]. Kl [25] and Fm [15] are partitioning methods based on vertex exchange. Cuttana differs from both of these approaches. First, it includes a coarsening phase to efficiently reduce graph size. Our coarsening approach also differs from that of Metis, which relies on multiple maximal matching iterations, rendering it unscalable for large graphs. We reframe coarsening as another streaming partitioning problem (sub-partitioning). Second, at the core of refinement, Kl swaps vertices while we move subpartitions; our approach provides asymptotically better performance. While the greedy heuristics and moves in Fm are similar to Cuttana's, its bucket listing technique does not apply to our case as it only works for small unweighted graphs. Fm requires $\mathcal{K}^2$ buckets for $\mathcal{K}$-way partitioning, each sized by the quality gain, making it unscalable for our weighted graph with gains up to $O(\frac{V^2}{\mathcal{K}})$. Finally, we initiate refinement on a graph partitioned by a streaming partitioner, which converges more quickly. This facilitates parallel coarsening and data structure preprocessing during streaming, thereby enhancing overall efficiency. Another approach to improving the partitioning quality of streaming partitioners is restreaming

[3, 42], where the graph is read multiple times to iteratively improve partitioning quality. The restreaming technique is orthogonal to our work, and Cuttana can be used in a restreaming system for faster convergence. There are also distributed graph partitioners that improve runtime and memory constraints over single-node solutions [20, 34, 36, 49].

In graph databases, updates can degrade partition quality over time. Cuttana can be combined with incremental graph partitioning techniques, such as those of Leopard [21] and Fan et al. [12] to work in the dynamic graph setting. Another possibility is a periodic coarse-grained repartitioning of the entire graph or fine-grained recalculation of the scoring function to determine when to move vertices. Repartitioning can be performed in the background, and its overhead can be negligible in long-running applications.

6 LIMITATIONS, CONCLUSION, AND FUTURE WORK

Limitations. Using Cuttana for dynamic graphs requires repartitioning or incorporating existing incremental approaches [12, 21], which we have not yet undertaken. Using Cuttana for small and mid-scale graphs, such as Orkut, may not be a good choice in single-run analytics, as the performance gain in analytical job runtime will not amortize the partitioning latency. Because our additional overhead compared to streaming solutions is independent of graph size, Cuttana's sweet spot is large graphs where the additional overhead is small relative to streaming time. Cuttana is designed to be a partitioner for distributed vertex-centric applications on massive graphs; in-memory solutions can provide better partitioning for small-to-medium scale graphs with higher quality.

Future Work & Conclusion. We introduced a novel streaming partitioner that incorporates prioritized buffering to improve the quality of classic streaming graph partitioners. We then conceptualized the problem of improving the initial partitioning by relocating vertices and presented a coarsening and refinement strategy capable of improving the quality of the initial output of any partitioner. The refinement algorithm demonstrated theoretical efficiency, with time complexity independent of graph size. Our partitioner, Cuttana, significantly improved the partitioning quality of its core streaming counterpart, surpassing state-of-the-art vertex partitioners in various scenarios, considering different quality metrics and balance conditions. With a parallel implementation and leveraging the power-law property of large graphs, Cuttana's parallel implementation incurs negligible partitioning latency overhead relative to a simple streaming partitioner. Our application study confirmed that using Cuttana almost always leads to lower network overhead and even load distribution, resulting in better runtimes and throughputs for both graph analytics and database applications. Consequently, Cuttana emerges as the preferred option for graph partitioning. Looking ahead, we envision further advances in the form of developing a new scoring function for buffering and extending our generalized trade concept to address more complex moves. In cases where moving a single sub-partition fails to enhance quality, however, relocating two or more vertices simultaneously can maintain the balance condition and improve overall quality.

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.

[2] Réka Albert, Hawoong Jeong, and Albert-László Barabási. 2000. Error and attack tolerance of complex networks. *nature* 406, 6794 (2000), 378–382.

[3] Amel Awadelkarim and Johan Ugander. 2020. Prioritized restreaming algorithms for balanced graph partitioning. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1877–1887.

[4] Florian Bourse, Marc Lelarge, and Milan Vojnovic. 2014. Balanced graph edge partition. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 1456–1465.

[5] Chiranjeeb Buragohain, Knut Magne Risvik, Paul Brett, Miguel Castro, Wonhee Cho, Joshua Cowhig, Nikolas Gloy, Karthik Kalyanaraman, Richendra Khanna, John Pao, et al. 2020. A1: A distributed in-memory graph database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 329–344.

[6] Ümit Çatalyürek, Karen Devine, Marcelo Faraj, Lars Gottesbüren, Tobias Heuer, Henning Meyerhenke, Peter Sanders, Sebastian Schlag, Christian Schulz, Daniel Seemaier, et al. 2023. More recent advances in (hyper) graph partitioning. *Comput. Surveys* 55, 12 (2023), 1–38.

[7] Jingji Chen and Xuehai Qian. 2023. Khuzdul: Efficient and Scalable Distributed Graph Pattern Mining Engine. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 413–426.

[8] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Transactions on Parallel Computing (TOPC)* 5, 3 (2019), 1–39.

[9] Mark De Berg. 2000. *Computational geometry: algorithms and applications*. Springer Science & Business Media.

[10] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. 2015. The LDBC social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 619–630.

[11] Wenfei Fan, Tao He, Longbin Lai, Xue Li, Yong Li, Zhao Li, Zhengping Qian, Chao Tian, Lei Wang, Jingbo Xu, et al. 2021. GraphScope: a unified engine for big graph processing. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2879–2892.

[12] Wenfei Fan, Muyang Liu, Chao Tian, Ruiqi Xu, and Jingren Zhou. 2020. Incrementalization of graph partitioning algorithms. *Proceedings of the VLDB Endowment* 13, 8 (2020), 1261–1274.

[13] Wenfei Fan, Ruiqi Xu, Qiang Yin, Wenyuan Yu, and Jingren Zhou. 2023. Application-driven graph partitioning. *The VLDB Journal* 32, 1 (2023), 149–172.

[14] Marcelo Fonseca Faraj and Christian Schulz. 2022. Buffered streaming graph partitioning. *ACM Journal of Experimental Algorithmics* 27 (2022), 1–26.

[15] Charles M Fiduccia and Robert M Mattheyses. 1988. A linear-time heuristic for improving network partitions. In *Papers on Twenty-five years of electronic design automation*. 241–247.

[16] Swapnil Gandhi and Anand Padmanabha Iyer. 2021. P3: Distributed deep graph learning at scale. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*. 551–568.

[17] Michael R Garey, David S Johnson, and Larry Stockmeyer. 1974. Some simplified NP-complete problems. In *Proceedings of the sixth annual ACM symposium on Theory of computing*. 47–63.

[18] 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.

[19] GraphX Apache Spark September 8, 2024. https://spark.apache.org/graphx/

[20] Masatoshi Hanai, Toyotaro Suzumura, Wen Jun Tan, Elvis Liu, Georgios Theodoropoulos, and Wentong Cai. 2019. Distributed edge partitioning for trillion-edge graphs. *arXiv preprint arXiv:1908.05855* (2019).

[21] Jiewen Huang and Daniel J Abadi. 2016. Leopard: Lightweight edge-oriented partitioning and replication for dynamic graphs. *Proceedings of the VLDB Endowment* 9, 7 (2016).

[22] Anand Padmanabha Iyer, Qifan Pu, Kishan Patel, Joseph E Gonzalez, and Ion Stoica. 2021. {TEGRA}: Efficient {Ad-Hoc} Analytics on Evolving Graphs. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 337–355.

[23] JanusGraph September 8, 2024. https://janusgraph.org/

[24] 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.

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

[26] Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan. 2015. Scalable SIMD-Efficient Graph Processing on GPUs. In *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques (PACT '15)*. 39–50.

[27] Jon M Kleinberg, Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew S Tomkins. 1999. The web as a graph: Measurements, models, and methods. In *Computing and Combinatorics: 5th Annual International Conference, COCOON'99 Tokyo, Japan, July 26–28, 1999 Proceedings 5*. Springer, 1–17.

[28] 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)*. IEEE, 593–606.

[29] Jérôme Kunegis. 2013. KONECT – The Koblenz Network Collection. In *Proc. Int. Conf. on World Wide Web Companion*. 1343–1350. http://dl.acm.org/citation.cfm?id=2488173

[30] Changji Li, Hongzhi Chen, Shuai Zhang, Yingqian Hu, Chao Chen, Zhenjie Zhang, Meng Li, Xiangchen Li, Dongqing Han, Xiaohui Chen, et al. 2022. ByteGraph: a high-performance distributed graph database in ByteDance. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3306–3318.

[31] Dongsheng Li, Yiming Zhang, Jinyan Wang, and Kian-Lee Tan. 2019. TopoX: Topology refactorization for efficient graph partitioning and processing. *Proceedings of the VLDB Endowment* 12, 8 (2019), 891–905.

[32] Lock-free queue September 8, 2024. https://github.com/cameron314/readerwriterqueue

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

[34] Daniel Margo and Margo Seltzer. 2015. A scalable distributed graph partitioner. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1478–1489.

[35] Daniel Wyatt Margo. 2017. *Sorting Shapes the Performance of Graph-Structured Systems*. Ph.D. Dissertation. Harvard University.

[36] Claudio Martella, Dionysios Logothetis, Andreas Loukas, and Georgos Siganos. 2017. Spinner: Scalable graph partitioning in the cloud. In *2017 IEEE 33rd international conference on data engineering (ICDE)*. Ieee, 1083–1094.

[37] 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.

[38] Ruben Mayer and Hans-Arno Jacobsen. 2021. Hybrid edge partitioner: Partitioning large power-law graphs under memory constraints. In *Proceedings of the 2021 International Conference on Management of Data*. 1289–1302.

[39] Robert Ryan McCune, Tim Weninger, and Greg Madey. 2015. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)* 48, 2 (2015), 1–39.

[40] Frank McSherry, Michael Isard, and Derek G Murray. 2015. Scalability! but at what {COST}?. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*.

[41] Vasimuddin Md, Sanchit Misra, Guixiang Ma, Ramanarayan Mohanty, Evangelos Georganas, Alexander Heinecke, Dhiraj Kalamkar, Nesreen K Ahmed, and Sasikanth Avancha. 2021. Distgnn: Scalable distributed training for large-scale graph neural networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[42] Joel Nishimura and Johan Ugander. 2013. Restreaming graph partitioning: simple versatile algorithms for advanced balancing. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. 1106–1114.

[43] Anil Pacaci and M Tamer Özsu. 2019. Experimental analysis of streaming algorithms for graph partitioning. In *Proceedings of the 2019 International Conference on Management of Data*. 1375–1392.

[44] François Pellegrini and Jean Roman. 1996. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *High-Performance Computing and Networking: International Conference and Exhibition HPCN EUROPE 1996 Brussels, Belgium, April 15–19, 1996 Proceedings 4*. Springer, 493–498.

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

[46] Wenwen Qu, Weixi Zhang, Ji Cheng, Chaorui Zhang, Wei Han, Bo Bai, Chen Jason Zhang, Liang He, and Xiaoling Wang. 2023. Optimizing Graph Partition by Optimal Vertex-Cut: A Holistic Approach. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 1019–1031.

[47] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M Tamer Özsu. 2017. The ubiquity of large graphs and surprising challenges of graph processing. *Proceedings of the VLDB Endowment* 11, 4 (2017), 420–431.

[48] Peter Sanders and Christian Schulz. 2011. Engineering multilevel graph partitioning algorithms. In *European Symposium on algorithms*. Springer, 469–480.

[49] George M Slota, Cameron Root, Karen Devine, Kamesh Madduri, and Sivasankaran Rajamanickam. 2020. Scalable, multi-constraint, complex-objective

graph partitioning. *IEEE Transactions on Parallel and Distributed Systems* 31, 12 (2020), 2789–2801.

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

[51] Titan DB September 8, 2024. https://github.com/thinkaurelius/titan

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

[53] US Road Dataset September 8, 2024. https://networkrepository.com/road-road-usa.php

[54] John D Valois. 1994. Implementing lock-free queues. In *Proceedings of the seventh international conference on Parallel and Distributed Computing Systems*. Citeseer, 64–69.

[55] Jana Vatter, Ruben Mayer, and Hans-Arno Jacobsen. 2023. The Evolution of Distributed Systems for Graph Neural Networks and their Origin in Graph Processing and Deep Learning: A Survey. *Comput. Surveys* (2023).

[56] Xubo Wang, Dong Wen, Lu Qin, Lijun Chang, and Wenjie Zhang. 2022. ScaleG: A Distributed Disk-based System for Vertex-centric Graph Processing. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 1511–1512.

[57] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. 2021. {GNNAdvisor}: An adaptive and efficient runtime system for {GNN} acceleration on {GPUs}. In *15th USENIX symposium on operating systems design and implementation (OSDI 21)*. 515–531.

[58] Zhigang Wang, Zichao Yang, Ning Wang, Yujie Du, Jie Nie, Zhiqiang Wei, Yu Gu, and Ge Yu. 2023. Lightweight Streaming Graph Partitioning by Fully Utilizing Knowledge from Local View. In *2023 IEEE 43rd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 614–625.

[59] Zhigang Wang, Zichao Yang, Ning Wang, Yujie Du, Jie Nie, Zhiqiang Wei, Yu Gu, and Ge Yu. 2023. Lightweight Streaming Graph Partitioning by Fully Utilizing Knowledge from Local View. In *2023 IEEE 43rd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 614–625.

[60] Da Yan, Guimu Guo, Md Mashiur Rahman Chowdhury, M Tamer Özsu, Wei-Shinn Ku, and John CS Lui. 2020. G-thinker: A distributed framework for mining subgraphs in a big graph. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1369–1380.

[61] Chenzi Zhang, Fan Wei, Qin Liu, Zhihao Gavin Tang, and Zhenguo Li. 2017. Graph edge partitioning via neighborhood heuristic. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 605–614.

[62] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. 2020. Distdgl: distributed graph neural network training for billion-scale graphs. In *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE, 36–44.