Streaming Graph Partitioning: An Experimental Study

Zainab Abbas*, Vasiliki Kalavri†, Paris Carbone*, Vladimir Vlassov*
*KTH Royal Institute of Technology †Systems Group ETH Zurich
Stockholm, Sweden Zurich, Switzerland

*{zainabab, parisc, vladv}@kth.se †kalavriv@inf.ethz.ch

ABSTRACT

Graph partitioning is an essential yet challenging task for massive graph analysis in distributed computing. Common graph partitioning methods scan the complete graph to obtain structural characteristics offline, before partitioning. However, the emerging need for low-latency, continuous graph analysis led to the development of online partitioning methods. Online methods ingest edges or vertices as a stream, making partitioning decisions on the fly based on partial knowledge of the graph. Prior studies have compared offline graph partitioning techniques across different systems. Yet, little effort has been put into investigating the characteristics of online graph partitioning strategies.

In this work, we describe and categorize online graph partitioning techniques based on their assumptions, objectives and costs. Furthermore, we employ an experimental comparison across different applications and datasets, using a unified distributed runtime based on Apache Flink. Our experimental results showcase that model-dependent online partitioning techniques such as low-cut algorithms offer better performance for communication-intensive applications such as bulk synchronous iterative algorithms, albeit higher partitioning costs. Otherwise, model-agnostic techniques trade off data locality for lower partitioning costs and balanced workloads which is beneficial when executing data-parallel single-pass graph algorithms.

PVLDB Reference Format:
Zainab Abbas, Vasiliki Kalavri, Paris Carbone, and Vladimir Vlassov. Streaming Graph Partitioning: An Experimental Study. *PVLDB*, 11(11): 1590-1603, 2018.
DOI: https://doi.org/10.14778/3236187.3236208

1. INTRODUCTION

Graph partitioning, the process of dividing a graph into a predefined number of subgraphs, is essential for graph analysis using distributed algorithms. Distributed graph processing has been widely adopted in recent years and enables knowledge extraction from large and medium-scale graph-structured datasets using commodity clusters [36, 21, 22, 14]. In such settings, each cluster node operates on one partition in parallel and communicates with other nodes through message-passing. Hence, partitioning quality directly affects communication and computation costs and is crucial for graph application performance [21, 45].

The problem of graph partitioning has been thoroughly studied and several methods have been proposed in the past few decades, each with a particular graph type, ingestion model or application objective in mind. Among existing methods, we study *streaming* graph partitioning algorithms. As opposed to *offline* methods, which first load the complete graph in memory and then divide it into partitions, streaming graph partitioning operates *online*, while ingesting the graph data as a stream [45].

We examine two practical use-cases of streaming graph partitioning. First, in the context of the *load-compute-store* computational model (e.g., MapReduce [15], Spark [52, 22], Giraph [14]), partitioning can be performed in a streaming manner during the *load* phase, by treating the bounded input graph dataset as a stream of vertices or edges. Second, it is appropriate for distributed streaming and semi-streaming algorithms [38, 37, 46, 5] that compute graph summaries and perform aggregations on possibly unbounded edge streams, and systems supporting native graph-as-a-stream computations [12, 28, 9, 27, 40].

Partitioning methods vary significantly in terms of their heuristics, assumptions, and respective performance, thus making it difficult for developers to compare them and assess their characteristics. Choosing the right technique for the computational problem at hand is non-trivial, especially because each method adopts a limited set of application objectives and constraints. As in offline graph partitioning, streaming partitioning typically defines two main objectives: *load balancing* and *minimum cuts* (vertex or edge). These correspond to aiming for fair load distribution and minimized communication overhead, respectively. Optimizing for both objectives, also known as the *balanced graph partitioning* problem, is an NP-hard problem [4].

Past studies [49, 24] have focused on offline graph partitioning techniques or heuristics used for streaming graph partitioning [45]. However, in the context of the stream ingestion model, the question of identifying factors that influence performance and quantifying their effects is still open. We specifically examine sensitivity to stream ingestion order, the number of partitions, suitability for unbounded processing, and cost amortization of applications, includ-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were invited to present their results at The 44th International Conference on Very Large Data Bases, August 2018, Rio de Janeiro, Brazil.
Proceedings of the VLDB Endowment, Vol. 11, No. 11
Copyright 2018 VLDB Endowment 2150-8097/18/07.
DOI: https://doi.org/10.14778/3236187.3236208

ing bulk synchronous iterative processing [36] and stream or semi-streaming graph approximations [37, 38].

Contributions. In this study, we first define the domain of online graph partitioning and its role in graph processing workflows, decoupling the partitioning step from the application logic computation, whether staged or pipelined. Second, we propose a uniform analysis framework to evaluate and compare partitioning features and characteristics of streaming partitioning methods. We classify algorithms with regards to their data model, strategy, constraints, complexity, state requirements, and objectives. To provide an unbiased performance comparison, we implement all studied methods on top of a common evaluation framework based on Apache Flink [8, 9], a distributed stream processing engine. Finally, we use bulk synchronous and single-pass graph streaming algorithms to evaluate distributed graph application performance in terms of partitioning cost amortization.

Main Findings. Our results indicate that the majority of streaming graph partitioning algorithms are *unsuitable* for continuous processing of unbounded streams due to their reliance on a priori knowledge of graph properties and their global state requirements. All studied algorithms except hash-based partitioning are stateful and maintain summaries and other global information, such as current vertex assignment, partition capacities, or degree distributions. With regards to performance, we note the advantage of low-cut partitioning methods for iterative applications, while low-cost partitioning mechanisms seem preferable for streaming applications. Overall, we identify several open research challenges in the area of streaming graph partitioning that aim at developing new, scalable online partitioning algorithms, with relaxed constraints on the graph properties and fewer state requirements. We believe that further efforts are necessary to make algorithms practical for modern stream processors, including achieving distributed execution independent of global graph state and developing adaptive partitioning methods that can adapt to workload or other changes.

The survey is structured as follows. Section 2 presents preliminaries and necessary notation. Section 3 describes the streaming graph partitioning algorithms and their categorization based on the selected criteria. Section 4 outlines the applications used for evaluation. Section 5 presents the experimental setup and Section 6 provides the experimental results. Section 7 discusses related work. We conclude and highlight open challenges for future work in Section 8.

2. PRELIMINARIES

Here, we provide the necessary background and introduce the data and computation models we target. Table 1 contains the notation used throughout this paper.

DEFINITION 1. *Given a graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, and k machines of capacity C, so that the total capacity kC is sufficient to store the whole graph, a partitioning algorithm splits G into k partitions, P_i, so that $P_1 \cup \ldots \cup P_k = G$.*

For convenience, we often refer to each individual partition by its index i. An *offline* graph partitioning algorithm accepts the complete graph G as input and typically computes the partitioning in multiple passes. For example, iterative clustering and community detection methods are often used to compute high-quality partitions. In contrast, a *streaming* graph partitioning algorithm processes the graph as a *stream*, a sequence of edges or vertices, and maps each element to a partition index i on-the-fly.

Table 1: The notation used in this paper

Symbol	Description
G	input graph
$m = \|E\|$	number of edges in G
$n = \|V\|$	number of vertices in G
k	number of partitions, $k \in \mathbb{N}$
P_i	set of vertices or edges in a partition i, $i \in [1, k]$
$N(v)$	set of neighbors of a vertex v
$S(v)$	set of partitions containing vertex v
C	partition capacity

2.1 Data Model

There exist two main approaches to graph partitioning (in a broad spectrum and not limited to stream ingestion), namely *vertex* partitioning and *edge* partitioning. Both approaches aim to minimize cross-partition dependencies by defining a *minimum-cut* optimization objective.

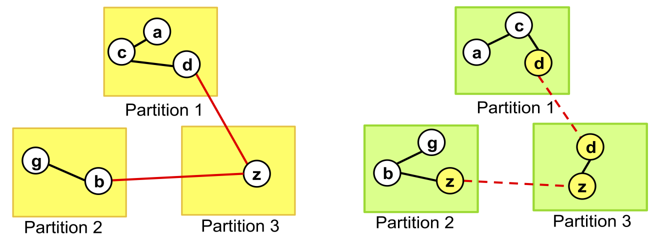


Figure 1: Vertex partitioning (left) assigns vertices to partitions, possibly creating edge-cuts; Edge partitioning (right) places edges to partitions, possibly creating vertex-cuts.

Vertex Partitioning. Vertex partitioning [4] operates on the vertex set V, assigning each vertex to a partition i. Edges can cross partition boundaries, as in Figure 1 (left), where edges (d, z) and (b, z) are *cut* by the partitioning.

DEFINITION 2. *For a graph G, the* edge-cut *$E' \subseteq E$ is a set of edges, such that $G' = (V, E \backslash E')$ is disconnected.*

The fewer edges crossing partition boundaries the lower the communication overhead, considering distributed graph processing using a vertex-centric model with message-passing along edges. Thus, the main optimization objective of vertex partitioning methods is the **minimum edge-cut**. Vertex partitioning is also known as *edge-cut partitioning.*

Edge Partitioning. Analogously, edge partitioning [6] operates on the edge set E, assigning each edge to a respective partition i. In this case, references to the same vertex can potentially co-exist across multiple partitions. In Figure 1 (right), vertices d and z are *cut* by the partitioning. Their references in individual partitions are also known as *mirrors.*

DEFINITION 3. *For a graph G, the* vertex-cut *$V' \subseteq V$ is a set of vertices such that $V \backslash V'$ along with $E' \subseteq E$, the set of incident edges, make $G' = (V \backslash V', E \backslash E')$ disconnected.*

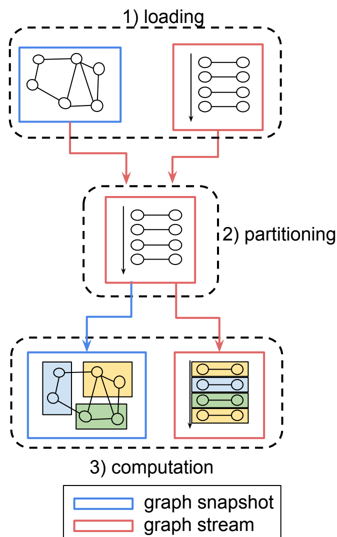


Figure 2: A general workflow for graph snapshot and stream loading, partitioning, and computation.

The fewer mirror vertices the lower the communication overhead, considering a distributed graph processing with an edge-centric programming model. Thus, the main optimization objective is now **minimum vertex-cut**. Edge partitioning is also known as *vertex-cut partitioning*.

2.2 Computational Model

Streaming graph partitioning is applicable to both the *load-compute-store* (batch) graph processing model, used in systems such as Pregel [36], GraphX [22] and Giraph [14], and the stream processing model, used in systems such as Flink [9], Storm [1] and Naiad [40]. Figure 2 shows the workflow of staged and pipelined phases for these models. In the case of batch processing, graph loading, partitioning, and computation, happen in separate consecutive stages. For stream processing, stages are pipelined and data is continuously passed as a stream from one stage to the next.

Loading. During the loading phase, graph is read from the disk or other external source and placed onto the computation cluster. In batch processing, the graph data is bounded and once loaded it represents a graph *snapshot* (e.g. the Facebook social network at a given time). In stream processing, graph data is continuously read from an external source and can be potentially unbounded (e.g. live user interactions on Twitter). A graph stream can be represented either as a sequence of edges (edge stream) or as a sequence of vertices with their adjacency lists (vertex stream). In essence, the streaming model subsumes the batch model, since a graph snapshot is merely a bounded graph stream. Hence, graph properties, such as the number of vertices n, the number of edges m, and the degree distribution can be computed before partitioning for a graph snapshot, while these properties continuously evolve for an unbounded stream.

Partitioning. During the partitioning phase, the partitioner takes a graph stream as input and assigns each vertex or edge to a partition. The decision is made on-the-fly by processing each element only once. In many cases, the partitioning logic can be implemented inside the graph loader, so that loading and partitioning happen in a single phase. Partitioners can base their decision on the current element or they can maintain *state*. Stateful partitioners consider the history of the stream seen so far. For example, in order to properly balance the number of elements per partition, a partitioner might store the current available capacity per partition. In principle, the state can be distributed among parallel partitioner instances, where each instance has a partial view of the stream, or a global view shared across parallel instances. As our analysis reveals, existing stateful streaming partitioning methods require shared state, which is a feature not available in modern distributed stream processors. Thus, partitioning logic needs to be executed by a single instance. Moreover, many methods often assume that global graph metrics are available before partitioning. These characteristics pose a major challenge in adapting existing methods for distributed processing of unbounded graph streams.

Computation. Computation takes place after loading and partitioning. In the batch model, computation starts after the whole graph has been loaded and partitioned, in a subsequent stage, and it operates in one or multiple passes (e.g. bulk synchronous model with fixpoint termination). In contrast, in the streaming model, application logic is triggered on-the-fly, per graph element, in a pipelined fashion after the partitioning step. In this case, graph elements are only accessed once. Therefore, applications that employ on-the-fly processing are also referred as single-pass streaming applications (the term semi-streaming [38] is also used to describe a constant number of graph stream passes).

3. ONLINE PARTITIONING METHODS

Streaming graph partitioning algorithms are quite diverse in their objectives, assumptions, and runtime complexities. We summarize eight partitioning algorithms that can be used in the streaming model and categorize them based on the following criteria: 1) *data model*, 2) *partitioning strategy*, 3) possible *constraints* (e.g., regarding input boundness or a priori knowledge), 4) *computational and space complexities*, 5) *state* requirements while partitioning, and 6) *optimization objectives*. Table 2 summarizes the algorithms across all criteria. In Section 3.4 we highlight the main findings.

3.1 Model-Agnostic Methods

Data-model agnostic partitioning algorithms can be employed in both vertex and edge-centric models. Hash partitioning is probably the most representative and widely-used method in this category.

3.1.1 Hash Partitioning

The idea of using a consistent hashing function to map elements with distinct keys to different partitions is widespread outside the domain of graph processing (e.g., for load balancing content in distributed key-value stores [16] and managed stream state [8]). In the context of vertex partitioning, a consistent hashing function can be used to assign vertices with unique identifiers to a physical partition index $V \rightarrow \mathbb{N}$ uniformly at random. Similarly, in the case of an edge data model hashing maps a set of edges to partitions $E \rightarrow \mathbb{N}$. For brevity, if we assume a vertex-centric model, hash-based partitioning can be defined as the mapping function $f(v) = hash(v) \bmod (k)$.

Table 2: Features and characteristics of the chosen streaming partitioning methods for this study.

Algorithm	Data model	Strategy	Constraints	Space	Time	State	Objective
Hash	Agnostic	Hash	None	None	$O(n)$/ $O(m)$	None	Load balance
LDG	Vertex stream	Neighbors	Bounded stream	$O(n)$	$O(kn+m)$	Vertices, partition assignment	Load balance, edge-cuts
Fennel	Vertex stream	Neighbors / non-neighbors	Bounded stream	$O(n)$	$O(kn+m)$	Vertices, partition assignment	Load balance, edge-cuts
Greedy	Edge stream	End-vertices	None	$O(n)$	$O(km+n)$	Vertices, partition assignment	Load balance, vertex-cuts
HDRF	Edge stream	Degree	None	$O(n)$	$O(km+n)$	Vertices, degree, partition assignment	Load balance, vertex-cuts
DBH	Edge stream	Degree and hash	None	$O(n)$	$O(m+n)$	Vertices, degree, partition assignment	Load balance, vertex-cuts
Grid	Edge stream	Hash	Perfect square partitions	$O(n+k)$	$O(m+n)$	Vertices, partition assignment	Load balance, vertex-cuts
PDS	Edge stream	Hash	$p^2+p+1=k$	$O(n+k)$	$O(m+n)$	Vertices, partition assignment	Load balance, vertex-cuts

Discussion: Hash partitioning is simple and does not require any a priori knowledge of the graph structure (only the number of partitions needs to be known), making it generally applicable to unbounded streams. Hashing is also stateless since it requires no history synopsis during partitioning, thus it can be trivially parallelized and be used to partition large-scale graphs.

3.2 Vertex Partitioning Methods

In the category of vertex partitioning algorithms, we analyze Linear Deterministic Greedy [44, 45] and Fennel [47, 48] as good representatives of partitioning mechanisms that can be applied online on a stream of vertices.

3.2.1 Linear Deterministic Greedy (LDG)

Linear Deterministic Greedy partitioning (LDG) tries to place neighboring vertices to the same partition, in order to yield fewer edge-cuts [45]. It uses a greedy heuristic that assigns a vertex to the partition containing most of its neighbors while respecting certain capacity constraints.

More specifically, given a range of partitions in $[1,k] \in \mathbb{N}$, let P_i represent the set of vertices placed in partition $i \in \{1,...,k\}$. For $N(v)$, the known set of neighbors of v, the LDG heuristic is given by $f(v)$ in the following Equation:

$$\begin{aligned} f(v) &= arg\,max_{i\in[1,k]}\,\{g(v,P_i)\} \\ g(v,P_i) &= |P_i \cap N(v)| w(i) \\ w(i) &= 1 - \frac{|P_i|}{C} \end{aligned} \tag{1}$$

LDG selects the partition that maximizes $|P_i \cap N(v)|$, the number of neighbors already assigned to a partition while enforcing the capacity constraint $C = \frac{n}{k}$.

The algorithm is shown in Pseudocode 1. The heuristic is continuously applied until the load of a partition reaches the threshold $g(v,P_i) < g(v,P_j)$, $j \in \{1,...,k\}$ and $j \neq i$. The load penalty enforces load balancing to avoid the extreme case where all vertices end up in the same partition.

Discussion: LDG requires the number of vertices n to be known a priori for calculating the capacity constraint C. Hence, it is generally unsuitable for unbounded processing. The algorithm requires keeping track of all partitioning decisions made so far, saved as the partitioning assignment state, which is accessed for every vertex in the input stream.

Pseudocode 1 LDG

Input: v, $N(v)$, k
Output: partition ID

```
1: procedure partition(v, N(v), k)
2:     for all partitions i = 1 to k do
3:         P_i ∩ N(v)                    ▷ neighbors in partition i
4:         w(i) = 1 − |P_i|/C            ▷ load penalty
5:         g(v, P_i) = |P_i ∩ N(v)|w(i)  ▷ partition scoring
       end for
6:     for all partitions i = 1 to k do
7:         ind = arg max_i{g(v, P_i)}
       end for
8:     Return ind
```

3.2.2 Fennel

Fennel [48] is a partitioning strategy whose heuristic combines locality-centric measures (low edge-cut) [45] with balancing goals [43]. Fennel's core idea is to interpolate between maximizing the co-location of neighbouring vertices and minimizing that of non-neighbours. Pseudocode 2 presents the Fennel logic in more detail. As in LDG, Fennel computes the number of neighbors present in each partition for every input vertex. In addition, the load limit per partition which sets a threshold for the maximum number of assigned vertices. The score $\delta g(v,P_i)$ is computed according to Equation 2 for each partition whose load is below the threshold and the input vertex is assigned to the partition with the maximum score.

$$\begin{aligned} f(v) &= arg\,max_{i\in[1,k]}\,\{\delta g(v,P_i)\} \\ \delta g(v,P_i) &= |P_i \cap N(v)| - \alpha\gamma|P_i|^{\gamma-1} \end{aligned} \tag{2}$$

Here, $\alpha = \sqrt{k}\frac{m}{n^{3/2}}$ and the load limit $= \nu\frac{n}{k}$. The parameters α, γ, and ν are tunable and control the weights associated with maximizing the number of neighbors and minimizing the number of non-neighbors for the input vertex during partitioning. In our experiments (Section 6) we picked the values used in the original evaluation [48], $\gamma = 1.5, \nu = 1.1$.

Discussion: Similar to LDG, Fennel requires the parameters n and m to be known a priori, and maintaining a persistent state of the assigned partitions during execution, that makes it unsuitable for partitioning unbounded streams.

Pseudocode 2 Fennel

```
Input: v, N(v), k
Output: partition ID
1: procedure partition(v, N(v), k)
2:     load limit = ν n/k                          ▷ load limit
3:     for all partitions i = 1 to k do
4:         if |P_i| < load limit then
5:             P_i ∩ N(v)          ▷ neighbors in partition i
6:             δg(v, P_i) = |P_i ∩ N(v)| − αγ|P_i|^(γ−1)
           end if
       end for
7:     for all partitions i = 1 to k do
8:         ind = arg max_i{δg(v, P_i)}
       end for
9:     Return ind
```

3.3 Edge Partitioning Methods

Many partitioning mechanisms operate on the edge-centric model. We select the following techniques that can operate online on an edge stream: Greedy [21], HDRF [41], DBH [50], and Grid [30], which are presented next.

3.3.1 Greedy

Greedy is a rule-based partitioning mechanism introduced in PowerGraph [21]. It aims to minimize vertex-cuts while also assigning balanced load across partitions. Let $S(v_i)$ denote the set of partitions containing the vertex v_i. Pseudocode 3 presents the rules employed by the Greedy algorithm, where the $leastLoad(S(v))$ method returns the least loaded partition ID form the set $S(v)$.

Pseudocode 3 Greedy

```
Input: v1, v2, k
Output: partition ID
1: procedure partition(v1, v2, k)
2:     if S(v1) ∩ S(v2) ≠ ∅ then
3:         partitionID = leastLoad(S(v1) ∩ S(v2))          ▷
       leastLoad() returns least loaded partition ID
       end if
4:     if S(v1) ∩ S(v2) = ∅ && S(v1) ∪ S(v2) ≠ ∅ then
5:         partitionID = leastLoad(S(v1) ∪ S(v2))
       end if
6:     if S(v1) = ∅ && S(v2) ≠ ∅ then
7:         partitionID = leastLoad(S(v2))
       end if
8:     if S(v1) ≠ ∅ && S(v2) = ∅ then
9:         partitionID = leastLoad(S(v1))
       end if
10:    if S(v1) = ∅ && S(v2) = ∅ then
11:        partitionID = leastLoad(k)
       end if
12:    Return partition ID
```

For each edge in the input stream, Greedy examines the participation of the endpoint vertices to existing partitions by applying the following rules: 1) Rule 1: If both endpoint vertices have been previously assigned in any common partition pick the least loaded common partition. 2) Rule 2: If both endpoint vertices have been previously assigned in different partitions pick the least loaded from the union of all assigned partitions. 3) Rule 3: In case either vertex has been previously assigned, pick the least loaded partition from the assigned partitions of that vertex. 4) Rule 4: If none of the vertices has been previously assigned, then pick the least loaded partition overall.

Discussion: Greedy does not require any knowledge of graph properties before processing the stream. Therefore, it can potentially process an unbounded edge stream. However, it requires maintaining the current partition assignment as a synopsis, which, in case of an unbounded stream would also grow without bound.

3.3.2 HDRF

HDRF [41] is particularly tailored for power-law graphs. It is based on PowerGraph's heuristic [21], which targets workloads with highly skewed graphs. The key idea that since power-law graphs have few high degree nodes and many low degree nodes, it is beneficial to prioritize cutting the high degree nodes to radically reduce the number of vertex-cuts. Pseudocode 4 summarizes the logic of HDRF.

Pseudocode 4 HDRF

```
Input: v1, v2, k
Output: partition ID
1: procedure partition(v1, v2, k)
2:     δ1 = getDegree(v1)
3:     δ2 = getDegree(v2)       ▷ getting partial degree
   values
4:     θ(v1) = δ(v1)/(δ(v1)+δ(v2)) = 1 − θ(v2)  ▷ normalizing the
   degree values
5:     for all partitions i = 1 to k do
6:         C^HDRF_BAL(i) = λ × (maxsize−|i|)/(ε+maxsize−minsize)
7:         C^HDRF_REP(v1, v2, i) = g(v1, i) + g(v2, i)
8:         C^HDRF(v1, v2, i) = C^HDRF_REP(v1, v2, i)+C^HDRF_BAL(i)
       end for
9:     for all partitions i = 1 to k do
10:        ind = arg max_i{C^HDRF(v1, v2, i)}
       end for
11:    Return ind     ▷ returning id of the partition
12: procedure g(v, i)
13:    if partition i ∈ S(v) then
14:        Return 1 + (1 − θ(v))
15:        else
16:        Return 0
       end if else
```

In more detail, for an input edge $e = (v_1, v_2)$, the partial degrees of its endpoint vertices are recorded as δ_1 and δ_2 . These values are then normalized using:

$$\theta(v_1) = \frac{\delta(v_1)}{\delta(v_1) + \delta(v_2)} = 1 - \theta(v_2) \tag{3}$$

HDRF works using Equation 4. Each edge is assigned to the partition i with highest value of $C^{HDRF}(v_1, v_2, i)$.

$$C^{HDRF}(v_1, v_2, i) = C^{HDRF}_{REP}(v_1, v_2, i) + C^{HDRF}_{BAL}(i) \tag{4}$$

$$C^{HDRF}_{REP}(v_1, v_2, i) = g(v_1, i) + g(v_2, i) \tag{5}$$

$$g(v, i) = \begin{cases} 1 + (1 - \theta(v)) & \text{if } i \in S(v) \\ 0 & \text{otherwise} \end{cases}$$

$$C^{HDRF}_{BAL}(i) = \lambda \times \frac{maxsize - |i|}{\epsilon + maxsize - minsize} \tag{6}$$

Here, $maxsize$ is the size of partition with maximum load, $minsize$ is the size of partition with minimum load, $S(v)$ here is set of partitions containing vertex v, ϵ is a constant value, and λ controls load imbalance [41]. When $\lambda \leq 1$, the algorithm behaves similarly to Greedy. When the input stream is ordered in breadth-first or depth-first search order, each incoming edge is placed in the partition containing most of its endpoint vertices' neighbors. As a result, the algorithm can yield imbalanced partitions. Setting $\lambda > 1$ solves this issue by accommodating for load balance. If λ approaches ∞, then the algorithm behaves like random hash partitioning. In our experiments (Section 6), we set the value of $\lambda = 1$ to optimize for minimum vertex-cuts.

Discussion: Similar to Greedy, HDRF does not require any graph parameters to be computed before partitioning, and thus, it can potentially process an unbounded edge stream. On the other hand, it requires maintaining the state of partitions for computing the load and $S(v)$. HDRF uses degree information for making partitioning decisions, however, instead of pre-computing degrees offline before partitioning, it can maintain partial degree information and update it while processing the input stream.

3.3.3 DBH

Degree Based Hashing (DBH) [50] is quite similar to HDRF, because it prioritizes cutting those vertices that have the highest degree. However, unlike HDRF, DBH employs hashing for partitioning. Pseudocode 5 presents the algorithm in more detail. For an input edge e, DBH computes the partial degree of its endpoint vertices v_1 and v_2, δ_1 and δ_2. After that, e is assigned to the partition ID computed as the hash of the vertex with the lowest degree.

Pseudocode 5 DBH

Input: v_1, v_2, k
Output: partition ID

```
1: procedure partition(v1, v2, k)
2:     δ1 = getDegree(v1)
3:     δ2 = getDegree(v2)
4:     if δ1 < δ2 then
5:         Return Hash(v1)mod(k)
6:     else
7:         Return Hash(v2)mod(k)
       end if else
```

Discussion: DBH algorithm keeps partial degree information of vertices as a state synopsis. Since it uses hashing, it can compute the current partitioning assignment on-the-fly, thus reducing the state requirements. DBH can potentially process unbounded streams because no global graph properties are required prior to partitioning.

3.3.4 Grid and PDS

The Grid [30] algorithm also uses hashing for partitioning. Prior to employing hashing, all partition IDs are arranged in a square matrix, termed the *Grid*. For each incoming edge $e = (v_1, v_2)$, a constrained set of partitions $S(v)$ for each end vertex v is formed by taking all the partitions in the row and column of the partition where v hashes to in the grid. The edge is assigned to the least loaded partition in the set $S(v_1) \cap S(v_2)$. The main limitation of Grid is that it limits the possible number of partitions to logarithmic degrees for constructing a square matrix ($rows \times columns = N$), where N is the total number of partitions. An alternative algorithm to Grid is PDS [30] that computes the set of partitions using Perfect Difference Sets. It requires $(p^2 + p + 1)$ number of partitions, where p is a prime number.

Discussion: Both Grid and PDS place a constraint on the number of partitions, but, they require no pre-computation on the input graph and they can both potentially sustain unbounded streams. With regards to the state, both algorithms need to maintain the partitioning assignment to compute the least loaded partitions.

3.4 Comparison Summary

The streaming graph partitioning algorithms presented so far in this section have various objectives and characteristics. We summarize their main features and design choices and point the reader to their categorization in Table 2.

Strategy: Except for hash-based partitioning which is the only stateless algorithm we consider, the strategy used by a partitioning method generally also defines the state it needs to maintain during partitioning. The current partitioning assignment and degree information are used across algorithms to reduce cuts. Vertex partitioning algorithms check the partitioning assignment to compute the number of existing neighbors and non-neighbors of a vertex, while few edge partitioning algorithms also use degree-based strategies.

Constraints: All stateful vertex partitioning algorithms considered are designed for partitioning bounded streams of vertices, while edge partitioning algorithms are more flexible in general. Greedy, HDRF, and DBH have no constraints whatsoever and could potentially process unbounded edge streams. However, this is challenging in practice, due to state requirements discussed next.

State: The state kept and accessed for decision making by the partitioning algorithms affects their computational and space complexities, as well as their applicability to processing unbounded graph streams. All stateful algorithms considered require a way to inspect the current partition assignment or degree, whether for load balancing or for minimizing cuts. As a result, they need to maintain a synopsis that can be queried for vertex or edge membership and current partition size at the very least. Such synopses can grow beyond memory limits for unbounded streams, and also complicate distributed implementations, as they need to be consistent and accessible by all parallel instances of the partitioner.

4. APPLICATIONS

We evaluate the partitioning methods with applications that operate on graph snapshots (batch) and graph streams. We have chosen bulk iterative algorithms and single-pass streaming summaries which we briefly describe next.

4.1 Iterative Applications

Connected Components: The connected components algorithm identifies subgraphs within which every vertex is reachable from every other vertex [26]. In the iterative, vertex-centric implementation of this algorithm [31, 33], each vertex is initially assigned a value equal to its own ID. Then, in every iteration, the vertex gathers values from its neighbors and picks the lowest value, which it then scatters back to its neighbors. When the algorithm converges, vertices with the same ID belong to the same component.

PageRank: The PageRank algorithm is an iterative vertex ranking algorithm that assigns weights to vertices based on their importance and their connectivity to other well ranked vertices [7]. The algorithm assigns an initial uniform rank to all vertices. Then, in each iteration, a vertex updates its rank by summing up the partial ranks from its incoming neighbor vertices. The new rank is then evenly distributed across outgoing edges to outgoing neighbors. The algorithm converges when the difference between the vertex rank form the current iteration and the rank in the previous iteration is less than a specified threshold.

Single Source Shortest Paths: The SSSP algorithm finds the shortest path between the source vertex and all connected vertices [17]. It initially assigns a zero value to the source vertex and ∞ to all other vertices. Then, each vertex updates its distance or path length to the source, until it does not change anymore across two consecutive iterations.

4.2 Single-Pass Applications

Bipartiteness: The bipartiteness algorithm continuously checks whether a graph stream forms a bipartite graph [19]. As long as the vertices seen so far can be divided into two groups such that there are no edges within those groups, then the graph is bipartite. The single-pass implementation maintains the current groups as state and assigns them a positive or negative sign. Then, the algorithm tries to place the vertices of each incoming edge to the existing groups by maintaining or flipping the signs. The distributed implementation maintains a partial state per processing instance and periodically merges the states into a combined state reflecting the history of the graph stream. The number of operations during the merge of partial state corresponds to cross-partition communication and thus we expect good partitioning algorithms to result into smaller partial states and more efficient merging.

Connected Components: The single-pass Connected Components [20] algorithm, also known as union-find, operates online over an edge stream. The algorithm maintains a disjoint set data structure to keep track of components. For each incoming edge, it checks whether the endpoint vertices belong to an existing component and merges components accordingly if the endpoints already exist in disjoint components. The distributed implementation maintains a partial disjoint set per partition and periodically merges states similarly to the bipartiteness check implementation.

5. EVALUATION METHODOLOGY

We design our experimental analysis by separating concerns among the system runtime, partitioning algorithms, and application on top. With that goal, we implement a comparison framework that can isolate partitioning costs and application performance under different iterative and pure streaming workloads. More concretely, in our experiments, we seek answers to the following questions:

Q1 What are the benefits, if any, of using more complex, data-centric partitioning methods compared to a generic hash-based strategy?

Q2 What is the partitioning overhead for an application using each partitioning algorithm?

Q3 How does the partitioning quality affect the application performance?

Next, we describe the implementation, datasets, parameters, metrics and experimental setup we use.

5.1 Implementation

Apache Flink [9, 8] is a streaming-first distributed analytics platform which executes complex applications by generating a DAG of logical operators and connecting the data streams to it. Bounded computation in Flink (in this case a graph snapshot) is also ingested internally as a stream, which makes Flink a convenient platform to build any graph as a stream ingestion scenario and implement complex partitioning logic. For the purposes of our evaluation framework, we have effectively implemented on Flink the general graph processing workflow presented in Figure 2. For staged and iterative tasks, we use the *DataSet* API to implement all respective transformations and application logic (i.e. iterative algorithms). Similarly, we use the *DataStream* API to implement all pipelined workflow steps. That includes all partitioning algorithms presented, as well as the single-pass stream processing applications.

5.2 Datasets

Table 3 shows the characteristics of the datasets we use in our experiments. We have generated synthetic graphs of varying sizes using the RMAT (Recursive Matrix) model [11] implemented in Gelly [29], Flink's graph processing API. RMAT produces skewed graphs that follow a power-law degree distribution. Such graphs commonly appear in social network problems [21] and are thus interesting for our analysis. We also use real-world datasets, including the Flickr graph [10] from the Online Social Networks Research web portal [3] produced by [39], several graphs from SNAP [35] (DBLP and Skitter), the MovieLens 10M datasets from GroupLens [2], large Twitter graph [34] and Friendster [51].

5.3 Order

For the real datasets, we have used the original order in which they were generated by their source, usually ordered by IDs. Otherwise, we consider three stream orderings for iterating through our datasets: 1) *BFS* [13], in a breadth-first search traversal, a vertex of the graph is selected at random, then the neighbors of that vertex are processed first. After that, the next level neighbors (the neighbors of the neighbors) are processed. 2) *DFS* [13], similar to *BFS*, after selecting a vertex at random, depth-first search is performed starting from that vertex. 3) *Random* [23], this order assumes that the vertices or the edges arrive at random from the streaming source. All partitioning algorithms behave similarly for *BFS* and *DFS* orderings, thus we only present results with *DFS* in Section 6.2.1.

5.4 Metrics

We use the following metrics for evaluation:

Partitioning Performance. We measure the **throughput** of a partitioning algorithm as the number of edges or vertices it can process (assign to a partition) per second.

Partitioning Quality. We evaluate partitioning quality using three metrics. **Load balancing** indicates how well the computation load is divided across partitions. Specifically, we calculate the normalized load for the highest loaded partition using the following formula:

Table 3: Datasets information

Dataset	Vertices	Edges	Category
RMAT	500,000	9,127,486	Synthetic
RMAT	1,000,000	18,540,007	Synthetic
RMAT	1,500,000	28,181,948	Synthetic
RMAT	2,000,000	37,547,390	Synthetic
RMAT	2,500,000	47,411,497	Synthetic
RMAT	4,000,000	80,000,000	Synthetic
RMAT	5,000,000	100,000,000	Synthetic
DBLP	317,080	1,049,866	Collaboration
Flickr	1,715,255	15,551,250	Social
Skitter	1,696,415	11,095,298	Computer
MovieLens	80,555	10,000,054	Rating
Twitter	41,652,230	1,468,365,182	Social
Friendster	65,608,366	1,806,067,135	Social

$$\rho = \frac{Load\ on\ highest\ loaded\ partition}{\frac{n}{k}} \tag{7}$$

where n is the input size (number of edges for edge stream partitioning or number of vertices for vertex stream partitioning) and k is the total number of partitions.

Edge-cut measures the fraction of edges cut from the resulting partitions. We calculate it using the following formula:

$$\lambda = \frac{No.\ of\ edges\ cut\ by\ the\ partitions}{Total\ no.\ of\ edges} \tag{8}$$

This metric applies to vertex partitioning algorithms only.

Finally, the **replication factor** indicates how many vertex copies an edge partitioning algorithm creates. We calculate it as follows:

$$\sigma = \frac{Total\ vertex\ copies}{Total\ no.\ of\ vertices} \tag{9}$$

Application Performance. We evaluate the partitioning quality and performance for both vertex and edge partitioning methods, but the application performance is evaluated using edge partitioning because it partitions power-law graphs better in terms of low communication cost than vertex partitioning. Also, some vertex partitioning algorithms require a priori knowledge, i.e, $|V|$ and $|E|$, making them unsuitable for processing continuous streams.

Next, we evaluate the effect of partitioning algorithms on the performance of graph analysis applications. Particularly, for iterative applications we measure complete application **execution time** which consists of partitioning time spent during the stream ingestion phase and computation time spent during the compute phase of the staged workflow. We report the ratio of partitioning time over total application execution time. This metric provides a good indication of the impact a partitioning method can make to the performance of an application. We also report the ratio of total application execution time when using a partitioning method over the execution time when using hash partitioning as a baseline. It is a meaningful metric to infer the cases where the partitioning cost is amortized. In the case of single-pass stream processing applications, we measure **the number of edges processed** during the whole pipelined workflow. This indicates which algorithm yields lower end-to-end latency. Additionally, for both these applications, we measure the **communication cost** as the ratio of the network traffic when using a partitioning method over the network traffic when using hash partitioning as a baseline.

5.5 Environment Setup

We deployed our experiments both on-premises on a university cluster as well as using a virtualized environment at Amazon EC2. The specs of the physical on-premises nodes are `2x Intel(R) Xeon(R) CPU @ 2.80GHz`, 44GB of RAM and Linux OS. This setup applies to all experiments in Sections 6.1 and 6.2. For the experiments in Section 6.3 we used up to 17x `r3.2xlarge` EC2 instances. The exact number of virtual instances in the latter case is experiment-specific and depends on the dataset size and the application type.

For our on-premises deployment, we set up Flink with one Job Manager (master node) and two Task Managers (workers). We further shared equally the amount of slots (allocated tasks) throughout workers. Regarding the virtual EC2 deployment we used one instance as the JobManager and the rest as TaskManagers. Finally, we used Flink v1.2.0 with Java 8 (Oracle JVM).

6. RESULTS

In order to answer **Q1** proposed in Section 5, we present partitioning performance results in Section 6.1 and partitioning quality (edge-cuts, replication factor and load balancing) results in Section 6.2. We give answers to **Q2** and **Q3** by results related to application performance in Section 6.3.

6.1 Partitioning Performance

We measure the throughput of vertex and edge partitioning algorithms with varying graph sizes. We use synthetic RMAT graphs and set the number of partitions to 4.

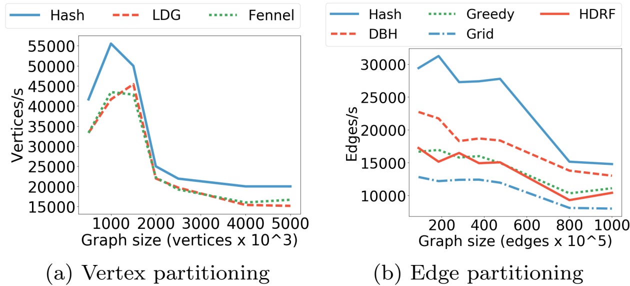
Vertex Partitioning. Figure 3(a) shows throughput measurements in vertices processed per second for vertex partitioning methods. Throughput initially increases with the graph size for all algorithms and then drops sharply for graphs with 20×10^5 vertices or larger. Overall, Hash partitioning demonstrates superior performance, while Fennel and LDG behave worse but similar to each other.

Edge Partitioning. Figure 3(b) plots throughput for edge partitioning algorithms. Hash partitioning shows the highest throughput as compared to all other methods. DBH has the second best throughput, followed by Greedy and HDRF which show almost identical performance. Finally, the Grid partitioner ranks last in terms of throughput.

Findings. Our results so far demonstrate that Hash partitioning outperforms all other evaluated methods in terms of throughput. However, the difference in performance is not dramatic. In both experiments, Hash shows at most 2x higher throughput than that of the second best partitioning method and the gap shrinks for larger graphs.

6.2 Partitioning Quality

A good partitioning method is not only fast but also produces high-quality partitions. We evaluate partitioning quality by first measuring the edge-cut for vertex partitioners and the replication factor for edge partitioners using different datasets. Then we measure the load balancing for these datasets. Next, we evaluate how stream order affects these



(a) Vertex partitioning (b) Edge partitioning

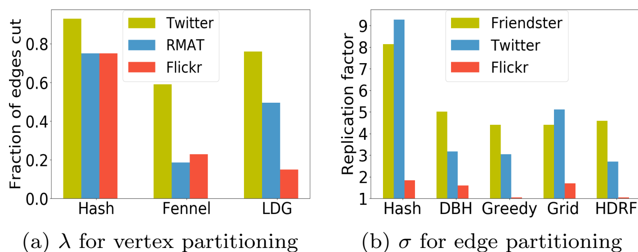
Figure 3: Throughput of partitioning algorithms using 4 partitions. Input: RMAT graphs.

results in Section 6.2.1. Finally, we examine how the number of partitions affects these results in Section 6.2.2.

We measure the partitioning quality using different input graphs. We use the Friendster, Twitter, largest RMAT, and Flickr graphs. We set the number of partitions to 16 for Friendster and Twitter and to 4 for the two smaller graphs. We stream all graphs in the order in which they are generated by their source. The power-law exponent that controls skewness is 1.7 for Flickr and has a very low value for RMAT making it highly skewed.

Edge-Cuts. Figure 4(a) shows the fraction of edges cut, i.e., λ for Twitter, RMAT, and Flickr. Hash has the highest λ value for all three datasets, since it does not take into account vertex locality. In fact, more than 70% of edges are cut for RMAT and Flickr, while on Twitter Hash generates 90% edge-cut. Fennel has the lowest cuts for Twitter and RMAT and it is only slightly outperformed by LDG for Flickr. In the case of RMAT, LDG produces more cuts than Fennel because RMAT is highly skewed compared to the other datasets.

Replication Factor. Figure 4(b) shows the replication factor, σ, for Friendster, Twitter, and Flickr. Hash has the highest σ of all methods across all datasets. The rest of the algorithms perform quite similarly, with Grid performing worst for Twitter and DBH for Friendster. We also observe that overall Greedy and HDRF perform better than other algorithms for all the datasets by giving lower σ. Specifically, for Flickr, Greedy and HDRF have $\sigma \approx 1$.



(a) λ for vertex partitioning (b) σ for edge partitioning

Figure 4: Fraction of edges cut λ and replication factor σ for different types of graphs.

Vertex Partitioning Load Balance. Table 4 shows the normalized load, ρ, defined in Section 5.4, using vertex partitioning algorithms. Hash gives perfectly balanced partitions because of its random placement. Fennel and LDG have nearly perfect load balance for Twitter and Flickr. However, LDG does not balance RMAT well compared to Fennel because LDG uses a greedy placement of vertices, and RMAT is highly skewed.

Table 4: Normalized maximum load ρ for vertex partitioning algorithms.

Dataset	Hash	Fennel	LDG
Twitter	1.0	1.1	1.13
RMAT	1.0	1.1	1.5
Flickr	1.0	1.0	1.0

Edge Partitioning Load Balance. Table 5 shows the results for ρ using edge partitioning algorithms. All algorithms yield almost perfectly balanced partitions with $\rho \approx 1$ for all graphs except Flickr. When partitioning Flickr with HDRF and Greedy the result has a lower replication factor compared to others, hence generating unbalanced partitions.

Table 5: Normalized maximum load ρ for edge partitioning algorithms.

Dataset	Hash	DBH	Greedy	Grid	HDRF
Friendster	1.0	1.001	1.0	1.0	1.0
Twitter	1.0	1.001	1.0	1.0	1.0
Flickr	1.001	1.002	3.98	1.0	3.98

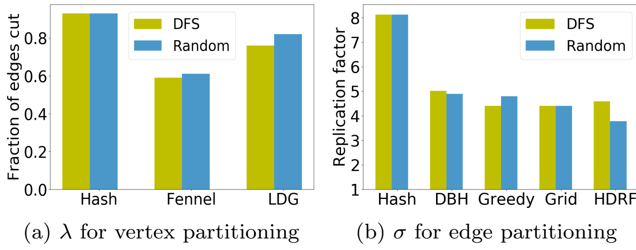
6.2.1 *Sensitivity to Order*

We now investigate how stream ordering affects the partitioning quality in terms of cuts and load balance. Some algorithms are particularly sensitive to the order in which they receive edges and vertices for processing. For example, *BFS* order is bad for Greedy because all neighboring vertices that arrive together might end up in the same partition. As a matter of fact, neighboring nodes do often arrive together in a graph stream. For instance, nodes grouped based on location in social graphs and links connecting web pages. We simulate this locality using *BFS* and *DFS* ordering and also use *Random* order as a baseline. To measure the effect of order on the partitioning quality in terms of λ, σ and ρ, we stream the Twitter and Friendster graphs in different orders and set the number of partitions to 16.

Edge-Cuts. Figure 5(a) shows the results for λ using ordered streams. Hash performs worst with highest λ for all orders. Moreover, λ remains the same using Hash despite of the change in order. LDG has higher λ for *Random* order compared to the *DFS* order. In the case of Fennel, we see that λ is not affected by the order. This can actually be controlled by changing the γ parameter values. In this experiment, we have set $\gamma = 1.5$, which according to [48] makes the algorithm less sensitive to the order. Overall, Fennel also has lower λ values than both Hash and LDG.

Replication Factor. Figure 5(b) contains the results for σ using ordered streams. Hash has the highest σ, while the other algorithms perform better and similar to each other. Hash and Grid are unaffected by order. HDRF has lower σ using *Random* order compared to the *DFS* order.

Vertex Partitioning Load Balance. Table 6 displays results for ρ using vertex partitioning algorithms on different stream orders. Hash and Fennel give well balanced partitions for all orders. LDG has slightly better results for *Random*



(a) λ for vertex partitioning (b) σ for edge partitioning

Figure 5: Fraction of edges cut λ and replication factor σ for different input orders, using 16 partitions. Input: Twitter (vertex partitioning) and Friendster (edge partitioning).

order. Overall, we conclude that none of the studied algorithms is highly sensitive to order when balancing partitions.

Table 6: Normalized maximum load ρ values for vertex partitioning algorithms using 16 partitions. Input: Twitter.

Order	Hash	Fennel	LDG
Random	1.0	1.1	1.12
DFS	1.0	1.1	1.13

Edge Partitioning Load Balance. Table 7 contains the result for ρ using edge partitioning algorithms on different stream orders for the MovieLens graph. We omit the results for the Friendster graph, as they were almost identical for both orders and all algorithms produced well-balanced partitions. For MovieLens, Greedy and HDRF show imbalance because they greedily place the neighboring edges arriving in the stream, together in the same partition.

Table 7: Normalized maximum load ρ values for edge partitioning algorithms and 4 partitions. Input: MovieLens.

Order	Hash	DBH	Greedy	Grid	HDRF
Random	1.001	1.005	1.0	1.0	1.0
DFS	1.001	1.006	4.0	1.0	4.0

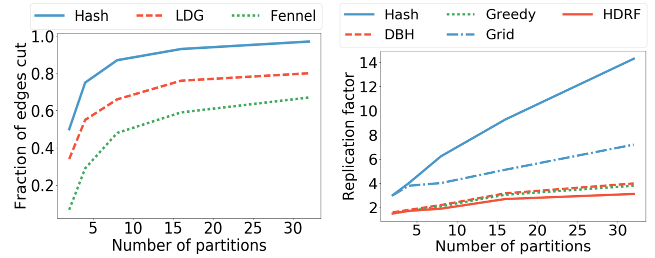
6.2.2 *Sensitivity to The Number of Partitions*

We evaluate the effect of the increase in the number of partitions on λ, σ and ρ by taking the Twitter graph and partitioning it across a range of partitions from 2 to 32.

Edge-Cuts. Figure 6(a) plots λ for vertex partitioning algorithms with increasing number of partitions. λ increases with more partitions for all the algorithms. While Hash performs poorly for a high number of partitions, LDG approaches 0.8 for 32 partitions and Fennel produces few edge-cuts even with many partitions, with λ slightly above 0.6.

Replication Factor. Figure 6(b) plots σ for edge partitioning algorithms using different number of partitions. For all algorithms, σ increases with the increase in the number of partitions. Hash shows a steep increase, while Grid shows the second worst behavior. σ for DBH, HDRF, and Greedy does not exceed a value of 4 even for 32 partitions.

Vertex Partitioning Load Balance. When examining how the number of partitions affects load balancing, we find



(a) λ for vertex partitioning (b) σ for edge partitioning

Figure 6: Fraction of edges cut λ and replication factor σ for different number of partitions (2 to 32). Input: Twitter.

that both Hash and Fennel have $\rho \approx 1$, thus we omit the results for these methods. Figure 7 plots ρ for LDG on Twitter, which is the only algorithm with different behavior. We see that LDG is affected by the number of partitions and its load factor increases, reaching a value of 1.15 for 32 partitions.

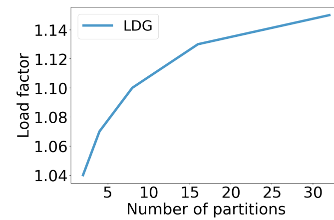


Figure 7: Normalized maximum load ρ for different number of partitions (2 to 32). Input: Twitter.

Edge Partitioning Load Balance. Increasing the number of partitions does not significantly affect load balancing for the Twitter graph regardless of the edge partitioning algorithm. All methods produce almost perfectly balanced partitions with $\rho \approx 1$.

Findings. We can summarize our results so far into the following observations: 1) Hash gives well-balanced partitions in all cases but produces many edge-cuts and high vertex replication. It is not sensitive to order and it behaves worse in terms of cuts when increasing the number of partitions. In the next section, we investigate how the low partitioning quality it provides in some cases affects application performance and when its perfect load balancing proves to be beneficial. 2) Fennel and LDG provide low edge-cuts; LDG is sensitive to order; Fennel is tunable. 3) HDRF and Greedy give low vertex-cuts, but both are sensitive to order. 4) Grid and DBH give moderate vertex-cuts; Grid and DBH give almost perfectly balanced partitions.

6.3 Application Performance

Considering our previous observations regarding partitioning algorithms, we would like to understand how the partitioning quality of different algorithms affects the performance of applications. We examine two factors that can have an impact: (a) the partitioning performance, i.e, the one-time overhead of the partitioning algorithm when the graph stream is ingested and (b) the partitioning quality, i.e, the load balance and cuts that the partitioning method

produces. Good load balancing is important for distributed execution because it lowers the probability of straggler workers and computation skew. On the other hand, a low number of cuts usually enables distributed algorithms to perform as much computation as possible locally and minimize cross-partition communication. To evaluate these factors we use the Twitter and Friendster graphs and partition them across 16 partitions. We examine the effects of partitioning on performance both on the batch, iterative applications, as well as on single-pass distributed streaming applications.

6.3.1 *Iterative Applications*

We first measure the ratio of network traffic as the communication cost for different partitioning methods. Next, we measure the application execution time and report the ratio of partitioning time over the total application execution time. We also compare the total application execution time for different partitioning methods with that of Hash. We run 10 iterations for the considered applications.

Communication Cost. Figure 8 plots the results for the ratio of network traffic using a partitioning method over the network traffic using the baseline Hash partitioning. In the case of SSSP and Connected Components (CC), data exchanged between the partitions using other partitioning algorithms is lower compared to Hash. All algorithms perform similarly with Greedy and HDRF being slightly better. With regards to PageRank on Twitter, DBH shows poor behavior and even exchanges more data than the baseline.

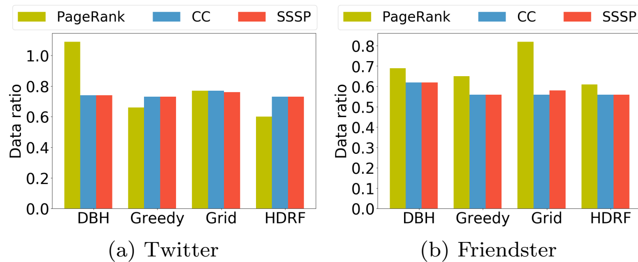


Figure 8: Communication cost of partitioning algorithms as compared to Hash for iterative applications on Twitter and Friendster.

Execution Time. Figure 9 plots the ratio of partitioning time over the total execution time (partitioning time and computation time) for iterative applications. Here, we want to compare the effect of different partitioning algorithms on the execution time. Greedy, Grid and HDRF have high ratio for all applications; whereas the ratio of DBH is almost as low as of Hash. The ratio is much lower for PageRank than for SSSP and CC across all methods.

Figure 10 shows the ratio of total execution time for applications using different partitioning algorithms over the total execution using the baseline Hash. Here, we want to examine whether the partitioning time for expensive partitioning methods can be amortized by improved application performance. For both datasets, Grid results in the highest total execution time for all applications. After Grid, Hash yields higher execution time compared to others followed by DBH, for SSSP and Connected Components; whereas, for PageRank, the total execution time using DBH is higher than that of Hash. Finally, Greedy and HDRF result in lower total execution time for all applications. Overall, HDRF and Greedy improve the performance of iterative applications by reducing computation times.

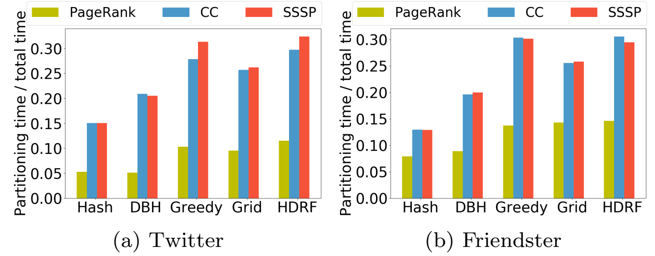


Figure 9: Partitioning time over execution time ratio for iterative applications on Twitter and Friendster.

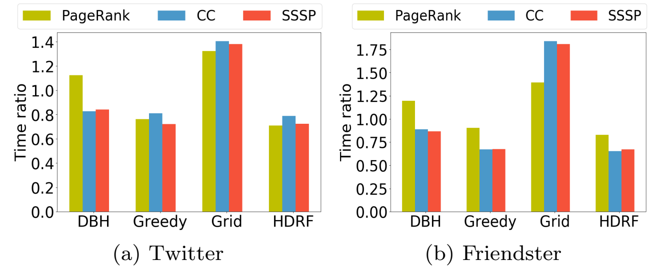


Figure 10: Total application execution time of partitioning algorithms as compared to Hash for iterative applications on Twitter and Friendster.

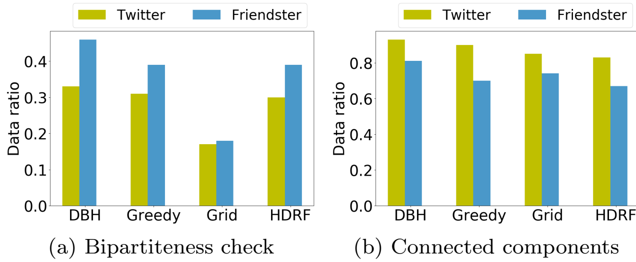
6.3.2 *Single-Pass Applications*

We ingest the Twitter and Friendster graph streams without using any a priori information to emulate the behaviour of unbounded streams and partition them. Next, we run the applications over the incoming partitioned streams for an interval of 15 min. After this, we measure: 1) the communication cost as the ratio of network traffic, and 2) the number of edges processed per second during the execution of the application to indicate which partitioning algorithm improves the latency of edges.

Communication Cost. Figure 11 shows the ratio of network traffic using a partitioning method over the network traffic using the baseline Hash. Grid minimizes network traffic for Bipartiteness check, while none of the partitioning algorithms provides impressive results for Connected components. For this application, HDRF has low communication cost for both Friendster and Twitter.

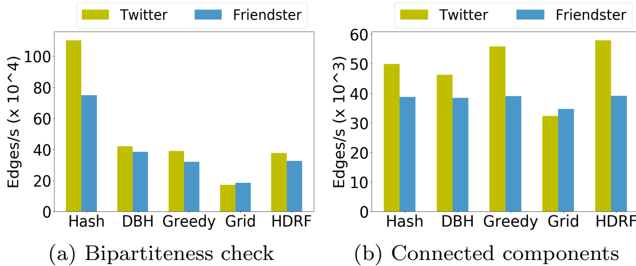
Number of Edges Processed. Figure 12 plots the average throughput in edges per second for Bipartiteness check and Connected components. Hash results in significantly superior performance for Bipartiteness check, where the state requirements are lower. In the case of Connected components, HDRF and Greedy either match or exceed the performance of Hash. Grid partitioning results in poor throughput for both applications and both datasets.

Findings. Our last results demonstrate that HDRF and Greedy, which have lower replication factor, yield higher partitioning cost that is amortized by lower computation time



(a) Bipartiteness check (b) Connected components

Figure 11: Communication cost of partitioning algorithms as compared to Hash for streaming applications on Twitter and Friendster.



(a) Bipartiteness check (b) Connected components

Figure 12: Average throughput (edges/s) for streaming applications on Twitter and Friendster.

for iterative applications and better end-to-end latencies for single-pass stream processing applications where data locality significantly affects the communication cost. Hash, with lowest partitioning time, improves the performance of applications when data locality does not significantly affect communication cost. In most cases, Hash results in higher communication cost; however, its partitioning overhead is negligible. HDRF and Greedy are effective at minimizing communication costs; however, they are more beneficial for computation and communication-intensive applications, since their partitioning costs cannot be easily amortized.

6.4 Summary of Results

We have arranged the partitioning algorithms based on our findings in Figure 13. The partitioning cost is indicated by shades of gray. Hash has the lowest partitioning cost; it is shown in the darkest color. Grid with highest partitioning cost is shown in lightest shade. For load balancing, DBH, Grid, and Hash give well-balanced partitions. Finally, HDRF provides the lowest vertex-cuts. Overall, the trade-off between balancing and reducing cuts remains. None of the studied algorithms provides both low cuts and perfect load balancing.

Among these algorithms, the low-cut partitioning methods, such as HDRF and Greedy, improve the performance of iterative applications, which have frequent communication between partitions during the compute phase. On the other hand, the impact seems to be less significant for streaming graph applications. Low-cost, partitioning algorithms, such as Hash appear more beneficial, probably because they have a pipelined compute phase and no state requirements.

7. RELATED WORK

A number of surveys [18, 42, 32, 25] have focused on offline partitioning algorithms in the past. Besides, several online

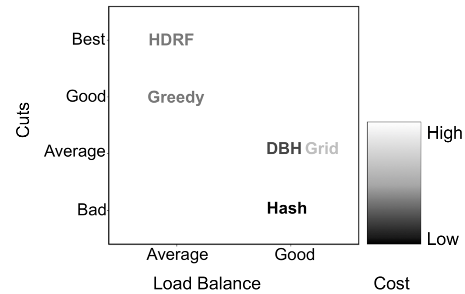


Figure 13: Cuts, load balance and partitioning cost comparison of edge partitioning algorithms.

partitioning methods have been surveyed in the context of the *load-compute-store* model and bounded graphs (snapshots) [49]. A survey by Guo et.al. [24] exclusively covers vertex partitioning algorithms. Our work is, to our knowledge, the first dedicated study to online graph partitioning methods that includes stream-specific properties (e.g., ingestion order) as well as considering single-pass graph stream aggregations, an emerging application domain with increasing system support [12, 28, 9, 27, 40].

8. CONCLUSION AND FUTURE WORK

In this paper, we have studied streaming graph partitioning algorithms and we have empirically compared them using a framework based on Apache Flink [9]. We have evaluated the partitioning quality and performance and we have measured the effect of partitioning on application performance, using both iterative and streaming applications. We conclude that algorithms aiming for optimal cuts, such as HDRF and Greedy, exhibit higher online partitioning cost that is otherwise amortized throughout the graph computation when that computation is sensitive to data locality and associated communication costs. Otherwise, when the computation is not directly affected by data locality, it is preferred to use online partitioning algorithms that aim for load balancing and performance (e.g., Hash and DBH). In the future, we plan to study more graph applications, especially streaming applications, and work to evaluate their performance using various partitioning techniques. Several open challenges remain in the area of streaming graph partitioning. We highlight the need for developing new, scalable online partitioning algorithms, with relaxed constraints on the graph properties and fewer state requirements. We believe that the development of such algorithms is crucial for making graph partitioning practical for applications ingesting continuous streams on top of modern distributed streaming engines.

Acknowledgements: We would like to thank our peer-reviewers and the financial support from the Continuous Deep Analytics project granted by Stiftelsen för Strategisk Forskning (BD15-0006) and the Streamline project of European Union's Horizon 2020 (688191). Furthermore, Vasiliki Kalavri is supported by an ETH Postdoctoral Fellowship and Zainab Abbas is funded by the Erasmus Mundus Joint Doctorate program in Distributed Computing (EACEA of the European Commission under FPA 2012-0030).

9. REFERENCES

[1] Apache Storm project. http://storm.apache.org/.

[2] Grouplens. https://grouplens.org/datasets/movielens/.

[3] Online social networks research at the Max Planck Institute for Software Systems. http://socialnetworks.mpi-sws.org/data-imc2007.html.

[4] K. Andreev and H. Räcke. Balanced graph partitioning. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 120–124. ACM, 2004.

[5] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 16–24. ACM, 2008.

[6] F. Bourse, M. Lelarge, and M. Vojnovic. Balanced graph edge partition. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1456–1465. ACM, 2014.

[7] S. Brin and L. Page. Reprint of: The anatomy of a large-scale hypertextual web search engine. *Computer networks*, 56(18):3825–3833, 2012.

[8] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas. State management in Apache Flink®: consistent stateful distributed stream processing. *PVLDB*, 10(12):1718–1729, 2017.

[9] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[10] M. Cha, A. Mislove, and K. P. Gummadi. A measurement-driven analysis of information propagation in the Flickr social network. In *Proceedings of the 18th International Conference on World Wide Web*, pages 721–730. ACM, 2009.

[11] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *SDM*, volume 4, pages 442–446. SIAM, 2004.

[12] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM European Conference on Computer Systems*, pages 85–98. ACM, 2012.

[13] T.-Y. Cheung. Graph traversal techniques and the maximum flow problem in distributed computation. *IEEE Transactions on Software Engineering*, (4):504–512, 1983.

[14] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: graph processing at Facebook-scale. *PVLDB*, 8(12):1804–1815, 2015.

[15] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[16] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

[17] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[18] U. Elsner. Graph partitioning-a survey. 1997.

[19] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. On graph problems in a semi-streaming model. *Theoretical Computer Science*, 348(2):207–216, 2005.

[20] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. Graph distances in the data-stream model. *SIAM Journal on Computing*, 38(5):1709–1727, 2008.

[21] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, volume 12, page 2, 2012.

[22] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 599–613, Broomfield, CO, 2014. USENIX Association.

[23] S. Guha and A. McGregor. Stream order and order statistics: Quantile estimation in random-order streams. *SIAM Journal on Computing*, 38(5):2044–2059, 2009.

[24] Y. Guo, S. Hong, H. Chafi, A. Iosup, and D. Epema. Modeling, analysis, and experimental comparison of streaming graph-partitioning policies. *Journal of Parallel and Distributed Computing*, 108:106–121, 2017.

[25] B. Hendrickson and T. G. Kolda. Graph partitioning models for parallel computing. *Parallel computing*, 26(12):1519–1534, 2000.

[26] J. Hopcroft and R. Tarjan. Algorithm 447: efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, 1973.

[27] A. Iyer, L. E. Li, and I. Stoica. CellIQ: real-time cellular network analytics at scale. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 309–322, 2015.

[28] A. P. Iyer, L. E. Li, T. Das, and I. Stoica. Time-evolving graph processing at scale. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, page 5. ACM, 2016.

[29] P. C. J. D. Bali, V. Kalavri. Streaming graph analytics framework design, 2015. http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-170425.

[30] N. Jain, G. Liao, and T. L. Willke. Graphbuilder: scalable graph etl framework. In *First International Workshop on Graph Data Management Experiences and Systems*, page 4. ACM, 2013.

[31] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: mining peta-scale graphs. *Knowledge and Information Systems*, 27(2):303–325, May 2011.

[32] J. Kim, I. Hwang, Y.-H. Kim, and B.-R. Moon. Genetic approaches for graph partitioning: a survey. In *Proceedings of the 13th Annual Conference on*

Genetic and Evolutionary Computation, pages 473–480. ACM, 2011.

[33] R. Kiveris, S. Lattanzi, V. Mirrokni, V. Rastogi, and S. Vassilvitskii. Connected components in MapReduce and beyond. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 18:1–18:13, New York, NY, USA, 2014. ACM.

[34] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW '10: Proceedings of the 19th International Conference on World Wide Web*, pages 591–600, New York, NY, USA, 2010. ACM.

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

[36] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146. ACM, 2010.

[37] A. McGregor. Graph mining on streams. *Encyclopedia of Database Systems*, pages 1271–1275, 2009.

[38] A. McGregor. Graph stream algorithms: A survey. *ACM SIGMOD Record*, 43(1):9–20, 2014.

[39] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, IMC '07, pages 29–42, New York, NY, USA, 2007. ACM.

[40] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.

[41] F. Petroni, L. Querzoni, K. Daudjee, S. Kamali, and G. Iacoboni. Hdrf: stream-based partitioning for power-law graphs. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, pages 243–252. ACM, 2015.

[42] A. Pothen. Graph partitioning algorithms with applications to scientific computing. *ICASE LaRC Interdisciplinary Series in Science and Engineering*, 4:323–368, 1997.

[43] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haradasan. Managing large graphs on multi-cores with graph awareness. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 41–52, Boston, MA, 2012. USENIX.

[44] I. Stanton. Streaming balanced graph partitioning algorithms for random graphs. In *Proceedings of the Twenty-fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '14, pages 1287–1301, Philadelphia, PA, USA, 2014. Society for Industrial and Applied Mathematics.

[45] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '12, pages 1222–1230, New York, NY, USA, 2012. ACM.

[46] J. Thaler. Semi-streaming algorithms for annotated graph streams. *arXiv preprint arXiv:1407.3462*, 2014.

[47] C. Tsourakakis. Streaming graph partitioning in the planted partition model. In *Proceedings of the 2015 ACM on Conference on Online Social Networks*, COSN '15, pages 27–35, New York, NY, USA, 2015. ACM.

[48] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*, pages 333–342. ACM, 2014.

[49] S. Verma, L. M. Leslie, Y. Shin, and I. Gupta. An experimental comparison of partitioning strategies in distributed graph processing. *PVLDB*, 10(5):493–504, 2017.

[50] C. Xie, L. Yan, W.-J. Li, and Z. Zhang. Distributed power-law graph computing: Theoretical and empirical analysis. In *Advances in Neural Information Processing Systems*, pages 1673–1681, 2014.

[51] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. In *2012 IEEE 12th International Conference on Data Mining*, pages 745–754, Dec 2012.

[52] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.