The Journal of Supercomputing (2025) 81:1418
https://doi.org/10.1007/s11227-025-07907-2

Distributed framework for high-quality graph partitioning

Chayma Sakouhi[1] · **Abir Khaldi**[1] · **Henda Ben Ghezala**[1]

Received: 27 April 2025 / Accepted: 23 September 2025

Abstract
The graph partitioning problem is increasing with the emergence of Big Data. Handling tremendous volumes of graph data requires an efficient graph processing system and especially a high-quality graph partitioning approach(s) to cope with graph application needs. However, all graph partitioning algorithms do not consider graph data volumes during graph partitioning. As a result, graph processing systems experience an imbalance in their workload and a decrease in system performance. For this purpose, we designed our distributed framework for high-quality graph partitioning including the volume metric. Also, it is created for scalable, high-availability, and fault tolerance. Using real-world datasets, we show that VF-Hammer performs a good graph partitioning quality and achieves better performance results against state-of-the-art graph partitioning.

Keywords Graph databases · Property graph · Graph partitioning · Balance volume · Balance size

1 Introduction

The usage of graphs has become ubiquitous. The versatility of the graph is because of the interesting problems that require a graph model to understand, resolve, or extract insights from real-world data. Many real-world applications including Social networks, Machine Learning, data mining, analytics, and Artificial Intelligence have

A. Khaldi, H. B. Ghezala have contributed equally to this work.

✉ Chayma Sakouhi
sakouhichayma@gmail.com

Abir Khaldi
abirakaldi@yahoo.fr

Henda Ben Ghezala
hhbg.hhbg@gmail.com

[1] ENSI, RIADI, Compus, Manouba, 2010 Manouba, Tunisia

Published online: 07 October 2025

recourse to using graphs to determine communities, identify shared connections, make recommendations, find the minimum and maximum flow, and solve route transportation problems. The ultimate need for real-world graphs is the primary reason for the emergence of graph processing systems, graph databases, and graph partitioning problems.

Many graph processing systems, including Google's Pregel [1], GraphLab, Powergraph [2], GraphX [3], Giraph, and Blogel, have been designed to meet the challenge of large-scale graph processing. The fast and immense increase of the graph size makes its graph processing very expensive in terms of high response time, resources (high CPU usage, a lot of memory, disk capacity), and even loss of query response that yields to system crashes. Hence, the graph processing systems need to partition the graph over a cluster of nodes to distribute and parallelize the workload and optimize the system performance for efficient graph processing.

There are many graph partitioning algorithms proposed to meet the challenge of large-scale graphs. They are proposed to be integrated as a pre-processing stage before graph processing. In the literature, there are two modes for graph partitioning methods: batch and streaming partitioning algorithms. The batch graph partitioning is used for batch graph processing. There is a wide range of batch partitioning algorithms such as [4–8]. It consists to partition entire graphs offline regardless of the time of their creation. Streaming graph partitioning methods are recently proposed. Several algorithms [9–12] have been proposed to meet the challenge of the balanced graph partitioning problem in streaming mode. Their principle is to partition a received stream of graph data (vertices/edges) in real time.

The challenge of high-quality graph partitioning is defined by the need to simultaneously optimize for three, often competing, objectives. First, balance is paramount to prevent workload skew and ensure efficient resource utilization across a cluster. This necessitates balancing not only the cardinality of entities (the size of a partition, measured by its number of vertices and edges) but also its volume, the total physical memory footprint of its data. An imbalance in either metric can lead to stragglers that degrade overall system performance. Second, it is critical to minimize communication cost, which is directly proportional to the number of cut-edges (i.e., edges spanning different partitions). A high number of cuts generates expensive network communication during graph computation, becoming a primary bottleneck. Finally, the partitioning process itself must maintain low computational overhead to avoid becoming a bottleneck, particularly in streaming environments where low-latency processing is essential. While extensive research has focused on balancing size and minimizing cuts, the crucial aspect of volume balance has been largely overlooked, leading to suboptimal performance in modern data-intensive applications.

In this paper, we address three important problems:

- With the emergence of Big Data, the volume of data generated every day is enormous. Consequently, the size of graphs grows over time, while their volumes continue to increase without limit. Existing graph partitioning methods distribute the number of vertices/edges equally to produce balanced partitions of graph data, but they do not consider the volume of vertices and edges. This can result in imbalanced volumes of partitions, leading to an imbalance in workload and

inefficient usage of cluster resources. However, our recent study [13] highlights the importance of considering volume in graph partitioning to improve the quality of the partitioning. As a result, graph partitioning algorithms must consider graph volumes during graph partitioning to handle the large volume of graphs in real-world applications.

- Graph processing systems have traditionally used techniques such as Hashing, random partitioning, and clustering to partition input graphs for batch processing. While these methods are fast and capable of generating partitions with balanced sizes, they often result in high communication costs and do not guarantee high-quality partitioning. To improve system performance and balance the workload, efficient graph partitioning algorithms are required.
- A distributed graph partitioning framework should be able to implement partitioning algorithms that produce partitions with balanced sizes and volumes, while also decreasing the amount of communication between nodes in the system.

In this paper, we introduce our proposed framework for graph partitioning, VF-Hammer. It can be integrated as a pre-processing system to create partitions suitable for use by real-world applications, distributed graph processing systems, or distributed graph databases. VF-Hammer is a distributed system built for high availability and scalability, capable of handling large volumes of graph data and workload.

As a first step, we will employ our proposed streaming algorithm, Hammer[14], in our distributed framework VF-Hammer. Hammer will be transformed into a distributed algorithm to distribute and parallelize graph partitioning tasks between nodes. What distinguishes Hammer from existing graph partitioning algorithms is that it considers the volume metric during graph partitioning. This allows it to generate partitions with balanced sizes and volumes.

The remaining part of this paper is organized as follows. Section 2 introduces the problem formulation and describes the impact of the graph volume on partitioning results. Section 3 discusses related work in this field, while Sect. 4 presents our proposed framework VF-Hammer. This section details the different components of the framework and their roles. In Sect. 5, we present the distributed algorithm Hammer. In Sect. 6, we report the evaluation results of VF-Hammer over a set of real datasets. Finally, Sect. 7 concludes the paper.

2 Problem formulation

The purpose of this section is to formally define the multi-objective optimization problem of high-quality graph partitioning, with a specific focus on the novel challenge of volume balancing. A high-quality partition must simultaneously achieve three critical, and often competing, goals: **load balance**, **minimized communication cost**, and **low computational overhead**. We first introduce the core notation, summarized in Table 1, before elaborating on these objectives and formalizing our problem.

Table 1 Table of notation used for graph partitioning problem

Attribute	Definition
G	Graph
v_i	vertex i, $\forall i \in \mathbb{N}$
e_j	edge j, $\forall j \in \mathbb{N}$
n	total number of vertices
d_{v_i}	degree of vertex
m	total number of edges
V	set of vertices of graph $V = \{v_1, v_i..v_n\}\, i, n \in \mathbb{N}$
E	set of edges in graph $E = \{e_1, e_j..e_m\}\, j, m \in \mathbb{N}$
$N(v_i)$	list neighbors of vertex v_i
σ	set of edges cut
θ	set of vertices cut
K	total number of partitions (machines)
α	size imbalance factor $\in [1, 2[$
λ	volume imbalance factor $\in [0, 1[$
$\beta(v_i)$	volume of vertex v_i
$\beta(e_j)$	volume of edge e_j
$\beta(P_k)$	volume of partition $P_k \forall k \in \mathbb{N}$
$S(P_k)$	size of partition $P_k . \forall k, S(P_k) \in \mathbb{N}$

Graph notation. Let $G = (V, E)$ be a graph with $n = |V|$ vertices and $m = |E|$ edges. Each vertex $v_i \in V$ is defined by a unique identifier, a set of properties p, its degree d_{v_i}, and a volume $\beta(v_i)$ representing its physical storage footprint in bytes. Each edge $e_j = (u, v) \in E$ is defined by its source and target vertices, a set of properties, and a volume $\beta(e_j)$

The partitioning problem. The objective is to partition G into K disjoint subsets (partitions) $P = \{P_1, P_2, ..., P_k\}$ such that $V = \cup_{k=1}^{K} P_k$ and the edge-cut between partitions is minimized. The quality of a partitioning is measured against three core objectives:

1. **Load balance:** The computational workload and storage must be distributed evenly across all partitions to prevent resource bottlenecks. This requires balancing two distinct metrics:
   - **Size balance:** The number of entities per partition must be balanced. The target size for a partition P_k is $S(P_k) \approx \frac{n}{K}$
   - **Volume Balance:** The total memory footprint of each partition must be balanced. The target volume for a partition P_k is $\beta(P_k) = \sum_{v_i \in P_k} \beta(v_i) + \sum_{e_j \in P_k} \beta(e_j) \approx \frac{\beta(G)}{K}$ where $\beta(G)$ is the total graph volume.
2. **Minimized communication cost:** Inter-partition communication during graph processing is expensive and is directly proportional to the number of **cut-edges** (σ)

edges whose incident vertices reside in different partitions. The primary objective is to minimize this metric: $min|\sigma|$, where $\sigma = \{e_j = (u, v) \in E | u \in P_i, v \in P_j, i \neq j\}$

3. **Low Computational Overhead:** The partitioning algorithm itself must be efficient to avoid becoming a system bottleneck. This is especially critical in **streaming** settings, where the algorithm must process incoming graph data with low latency.

The volume-aware partitioning challenge. The emergence of Big Data exacerbates this problem, as graph volumes grow rapidly. Traditional partitioners focus on objective 1 (size) and objective 2, but neglect **volume balance**. As illustrated in Figure 1, a partition can be balanced by entity count ($S(P_1) \approx S(P_2)$) yet be severely imbalanced by volume ($\beta(P_1) \ll \beta(P_2)$). This leads to straggling machines, inefficient resource usage, and degraded system performance, as the partition with larger volume requires more memory and processing time.

Therefore, our goal is to design a distributed framework that effectively manages this triple objective. We propose **VF-Hammer**, a framework that natively integrates volume balancing to produce partitions that are balanced in both size and volume while simultaneously minimizing the number of cut-edges and maintaining the low overhead required for scalable streaming processing.

3 Related work

Graph partitioning problems belong to NP-hard problems [15]. Various graph partitioning methods have been proposed to distribute the large-scale graphs in a set of partitions, the aim of which is to minimize the processing cost and balance the workload. Our review of the literature is structured into two parts: first, we survey the key partitioning algorithms proposed in the field, categorized by their processing

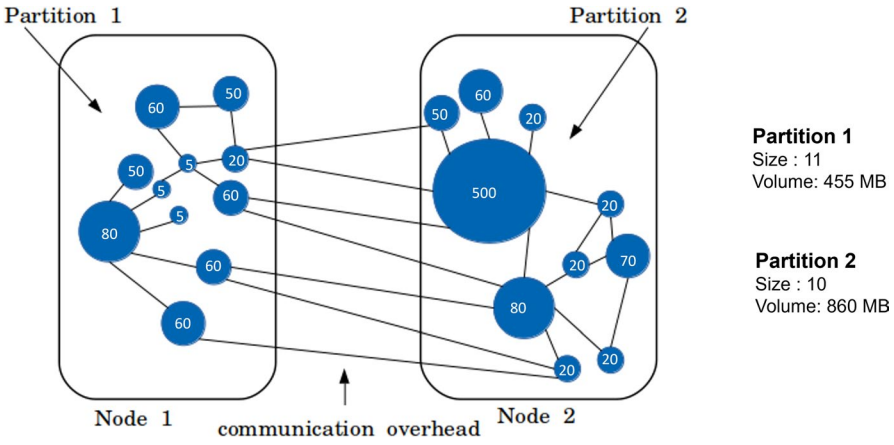

Fig. 1 Graph partitioning based on size

mode; second, we examine existing partitioning frameworks that integrate these algorithms into usable systems.

3.1 Partitioning algorithms

The primary goal of proposed methods is to distribute large-scale graphs into a set of partitions to minimize processing cost and balance workload, typically following one of two paradigms: batch or streaming processing.

Batch Algorithms require the entire graph to be loaded into memory before processing. The Multilevel Graph Partitioning (MGP) model [16] is a widely used and influential paradigm for this purpose. It operates in three stages: coarsening the graph, partitioning the smaller version, and then uncoarsening it while refining the partition. Seminal algorithms in this category include Metis [17], renowned for its speed, and its variants hMetis [18] for hypergraphs, kMetis [19], and the parallel implementation ParMetis [20].

However, MGP is far from the only approach. Spectral methods leverage the eigenvectors of the graph Laplacian matrix to find partitions [21]. Geometric partitioners use vertex coordinates for spatial partitioning [22]. Other distributed batch algorithms include JaBeJa [5], a distributed balancing algorithm, and its vertex-cut extension JaBeJa-VC [6], which reduces communication cost. DFEP (Distributed Funding-Based Edge Partitioning) [23] is another vertex-cut method designed for distributed environments. For specialized domains, algorithms like HipMCL [24] implement distributed Markov Clustering for massive biological networks.

More recently, machine learning-based approaches have emerged for learning partitioning strategies, including reinforcement learning [25] and graph neural networks [26], reflecting a broader trend of using ML for combinatorial optimization [27].

Streaming Algorithms process graph data incrementally as it arrives, making them suitable for dynamic graphs. Early work includes FENNEL [9], a generalized framework for streaming partitioning. Recent efforts have focused on improving quality and adaptability. HDRF [28] is a highly effective algorithm designed for power-law graphs. Akin [10] and IOGP (Incremental Online Graph Partitioner) [29] are designed for distributed graph databases and OLTP workloads. However, a common limitation of these methods is their focus on balancing partition sizes based solely on entity count, neglecting the memory footprint of the data. To address this, our recent work introduced Hammer [14], a streaming algorithm that incorporates vertex and edge volume as a first-class metric during partitioning. By explicitly optimizing for balanced partition volumes, Hammer prevents memory imbalance and reduces the risk of stragglers in distributed graph processing systems.

To mitigate the limited information in pure streaming, semi-streaming approaches have emerged. WStream [11], AdWise [12], and WSGP [30] use a window-based model to buffer a portion of the stream before making assignment decisions. HeiStream [31] combines MGP and Fennel, first creating a model from a batched subgraph and then applying it to the stream. While faster, a key limitation of streaming algorithms is their reliance on partial information, which often results in lower

partition quality compared to offline methods. Furthermore, they traditionally focus on balancing entity counts, neglecting the memory volume of data.

3.2 Partitioning frameworks

Beyond individual algorithms, several frameworks integrate these methods into comprehensive systems for end-users. It is important to note that while these frameworks are publicly available as research prototypes or open-source software, their architectural availability (a systems term referring to a design that ensures continuous operation through features like redundancy and failover) is often lacking.

General-purpose batch partitioning frameworks provide a suite of algorithms. Chaco [32] offers classic algorithms like spectral bisection, Kernighan-Lin, and MGP. KaHIP (Karlsruhe High Quality Partitioning) [8] provides more advanced algorithms, including KaFFPa [33] (Karlsruhe Fast Flow Partitioner), its evolutionary version KaFFPaE [34], and distributed variants [35–38]. XtraPuLP [39] is a distributed memory framework using label propagation designed for massive graphs.

For dynamic graph processing, DynamicDFEP [40] is a distributed framework that extends DFEP, offering complete, partial, and unit-based insertion methods to maintain balance under graph changes. Hermes [41] is another system for dynamic graphs, which uses Metis for an initial partition and a lightweight repartitioner for streaming updates.

However, these existing frameworks have notable limitations regarding scalability and robustness, as identified in their respective evaluations. For instance, scaling KaHIP effectively beyond a few dozen cores is challenging due to complex communication patterns [33, 36]. Similarly, frameworks like Chaco and the standard versions of Metis are designed for shared-memory systems, which inherently limits their scalability to a single node [17, 32]. Furthermore, their design often lacks the availability (e.g., no backup master nodes) and fault tolerance (e.g., an inability to recover from worker node failures without restarting the entire job) required for robust deployment in modern, unreliable distributed environments. These shortcomings are particularly evident when compared to the robust architectures of general-purpose distributed data systems like Apache Spark or Flink.

3.3 Comprehensive comparison and discussion

To clearly illustrate the landscape and position our contribution, Table 2 provides a comprehensive comparison of the aforementioned frameworks based on key criteria such as supported processing modes, algorithmic approach, and native volume support.

The table reveals a clear gap: while numerous high-quality algorithms and frameworks exist for both batch and streaming processing, none natively integrate the volume metric to prevent memory imbalance. Moreover, as the literature shows, existing distributed frameworks face scalability limits and lack the built-in fault tolerance and high availability required for production settings.

Table 2 A unified comparison of graph partitioning frameworks

Partitioning Framework	Supported Modes	Primary Algorithms	Volume Metric	Scalability	Availability	Fault Tolerance	Graph Support
Chaco	Batch	MGP, Spectral, Inertial, KL	–	Shared Memory	–	–	Static
ParMETIS	Batch	Parallel MGP, Multi-constraint	✓	Distributed Memory	–	–	Static
KaHIP	Batch	KaFFPa, KaFFPaE, dSPAC, MGP	–	Multi-core / Distributed	–	–	Static
XtraPuLP	Batch	Label Propagation (Multi-constraint)	✓	Distributed Memory	–	–	Static
HipMCL	Batch	Markov Clustering (MCL)	–	Distributed Memory	–	–	Static
FENNEL	Streaming	FENNEL	–	Single Machine	–	–	Dynamic
CUTTANA	Streaming	Learning-based Heuristics	–	Single Machine	–	–	Dynamic
DynamicDFEP	Streaming	Com-Ins, Part-Ins, UB-Ins	–	Distributed Memory	–	–	Dynamic
IOGP	Streaming	IOGP	–	Distributed Memory	–	–	Dynamic
VF-Hammer	**Batch, Streaming**	**dist-Hammer, (Other Batch)**	✓	**Distributed**	✓	✓	**Dynamic Static**

To address these limitations, we propose VF-Hammer, a distributed framework designed for both batch and streaming modes. Unlike existing solutions, VF-Hammer's partitioning decisions are based on both entity count and memory volume. Furthermore, it is architected with scalability, fault tolerance, and high availability as first-class concerns, offering a more robust solution for large-scale graph processing.

4 VF-Hammer framework

In this section, we provide the details of the design of VF-Hammer Framework. We will describe the principal components of VF-Hammer, their roles, and their interactions.

4.1 Framework design

VF-Hammer is a volume-based framework designed for high availability, scalability, and fault tolerance in graph partitioning. It can accommodate different types of graph partitioning algorithms, whether in batch mode or streaming partitioning, to handle a large number of applications.

It aims to produce high-quality graph partitioning, defined as partitions that are highly balanced in both size (number of entities) and volume (memory footprint in bytes), while reducing total cut-edges and replicated vertices to decrease communication costs between nodes during graph processing. This optimization helps to reduce computation bottlenecks and balance the load for distributed and parallel graph processing systems.

The VF-Hammer framework follows a master–slave architecture, consisting of four key components (see Fig. 2). The architecture is designed for scalability by allowing horizontal scaling through the addition of slave nodes; its performance under increased load is empirically evaluated in Sect. 6. The master node acts as the central point for managing the graph partitioning process, distributing jobs, and monitoring the progress of the slaves. The slave nodes are responsible for receiving and executing jobs, as well as storing the resulting graph partitions. Finally, the broker node is responsible for monitoring the status of the cluster.

This architecture ensures high availability and fault tolerance. For instance, in the scenario of a master node failure, the backup master node is ready to take over the management and ensure uninterrupted operations, preventing a single point of failure. Similarly, if a slave node fails, the system can reassign its tasks to other available nodes, ensuring continued processing despite individual component interruptions. The broker node continuously checks the health of all components and reports any anomalies to the master node.

4.2 VF-hammer components functions

The high-level components of VF-Hammer system are described below:

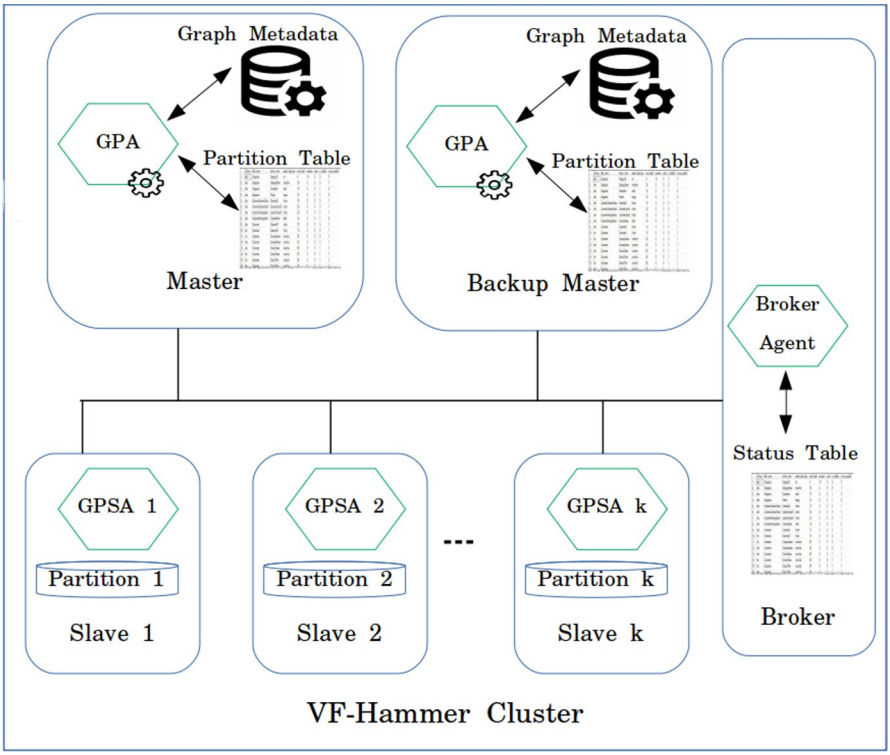

Fig. 2 Global architecture of VF-Hammer framework

1. ***The Master*** performs an important role in the VF-Hammer framework. It handles the smooth and scalable process of the system. It is responsible for receiving the graphs from different sources (HDFS, Kafka, Flume, APIs). Also, it is engaged in implementing a graph partitioning method on an input graph data through the Graph Partitioning Agent (GPA). Moreover, it oversees the jobs of the partitioning tasks by sending them as messages to the slaves. The master frequently updates the state of the Backup Master through synchronous communication to ensure strong consistency and a perfect replica in case of failure. This synchronous update is crucial for maintaining state integrity after a failover event. This process occurs under the supervision of the Broker Agent.
2. ***The master metadata*** is essential in solving graph partitioning problems as it maintains the global structure of the graph. Since vertices and edges are often streamed into the system, the Graph Partitioning Agent (GPA) must distribute them on-the-fly, relying on the graph metadata. The master metadata contains various information such as the list of vertices and edges for each graph, including their identifiers, volumes (their physical storage size in bytes), degrees, neighbors, relationships, and locations. In addition, it also stores additional information related to the partitions, such as their sizes, volumes, and assigned slaves. The

partitioning algorithm utilizes all this information to assign vertices and edges to suitable partitions, which are stored under the corresponding Slave node. By leveraging this information, VF-Hammer can achieve highly balanced partitions and reduce communication costs during graph processing, ultimately improving the efficiency of the distributed and parallel graph processing system.

3. ***Partition Table*** plays a crucial role in the VF-Hammer framework by creating a new table for each graph that stores the locations of every vertex and edge, as per the assignment by the Graph Partitioning Agent (GPA). The Partition Table also responds to requests from the GPA, such as determining the location of an existing vertex stored in a particular partition. Additionally, it assists the graph processing algorithms in accessing the relevant partition directly and carrying out graph querying and updates, rather than having to search through all the partitions. As a result, the Partition Table significantly reduces the time taken for graph partitioning, computation, and processing. This optimization of the graph processing pipeline translates to faster and more efficient graph processing, which is particularly important for real-time applications that require rapid processing of large volumes of data.
4. ***The Slave*** is a critical component in the VF-Hammer framework, as it is responsible for executing graph partitioning jobs and storing the resulting graph partitions. The Slave node is initialized and managed by the Master node, and it deploys a Graph Partitioning Slave Agent (GPSA) that interacts with the storage systems, including HDFS, local file systems, and NoSQL databases, to store the partitions of graphs. The Slave node plays a crucial role in enabling parallel and distributed graph processing by executing partitioned graph computations on different subsets of the graph data. By leveraging the computing power of multiple Slave nodes, VF-Hammer can process large-scale graphs efficiently and effectively.

4.3 VF-Hammer agent functions

The GPA (Graph Partitioning Agent) is a versatile component that supports various types of graph partitioning algorithms. When a new graph is received, GPA extracts its structure (vertices and edges) and stores it in its metadata to create a global representation of the graph. It then applies a distributed graph partitioning algorithm, converting the process into a set of parallel jobs distributed among the slave nodes. Throughout the partitioning process, GPA utilizes the graph structure stored in the metadata and updates the Partition Table by assigning each entity (vertex/edge) to its corresponding partition.

The GPSA (Graph Partitioning Slave Agent) is a critical component of the VF-Hammer framework. It executes the partitioning jobs assigned by the GPA and is responsible for storing the assigned vertices and edges in their respective target partitions. The GPSA interacts with the storage systems (e.g., HDFS, local file system, NoSQL database) to access the partitions and write the output of the partitioning jobs. By executing the partitioning jobs in parallel across multiple slave nodes, the GPSA enables the VF-Hammer framework to achieve high scalability and fault

tolerance. Its contribution to the overall performance and efficiency of the system is significant.

The Broker Agent is responsible for monitoring the status of the cluster nodes by regularly sending heartbeat requests. In the event of a failure of the Master node, the Broker Agent selects the Backup Master to take over and continue the graph partitioning process. Additionally, the Backup Master alerts the Master when a slave node fails or is back online.

4.4 High availability and fault tolerance

When the Master node fails, the system does not experience a complete shutdown. Instead, the Backup Master replaces it and continues the graph partitioning process seamlessly. To ensure that the Backup Master is always up-to-date, the Master periodically updates the metadata and Partition Table in the Backup Master node and sends a log file that contains the job description launched by the graph partitioning agent. This synchronous state replication between the Master and Backup Master is crucial to maintain strong consistency, ensuring the backup has a perfect replica of the critical system state. This design choice prioritizes data integrity and a seamless failover process over the potential latency overhead of synchronous communication. This ensures that the Backup Master has the latest information and can take over without any delay in case of failure.

In the case of a failed slave node, its partition becomes unreachable, which can potentially disrupt the graph partitioning process. However, the VF-Hammer system is designed to handle such situations efficiently. The Master does not exclude the unavailable partition from the partitioning process. Instead, the graph partitioning agent continues to work normally, and the failed slave's jobs are stored temporarily in the message dispatcher until the slave node is back online. This approach ensures fault tolerance at the slave level. The system maintains overall progress and automatically recovers the failed slave's workload upon its reactivation, preventing a single node failure from halting the entire partitioning pipeline. This ensures that the graph partitioning process is not affected, and the system can continue to operate smoothly even in the presence of failures.

After describing the different components of the VF-Hammer system and their functions, we will present the interaction between them.

4.5 Component-component interaction

The VF-Hammer framework relies on a message-driven architecture, which enables communication between its components. As depicted in Fig. 3, VF-Hammer nodes (i.e., master and slaves) exchange messages with each other. Immutable messages are used for communication within the VF-Hammer system. The general communication between operational agents (e.g., Master-to-Slave, Slave-to-Slave) is asynchronous and non-blocking. This design choice was made to maximize system throughput, scalability, and resource utilization, as senders are not

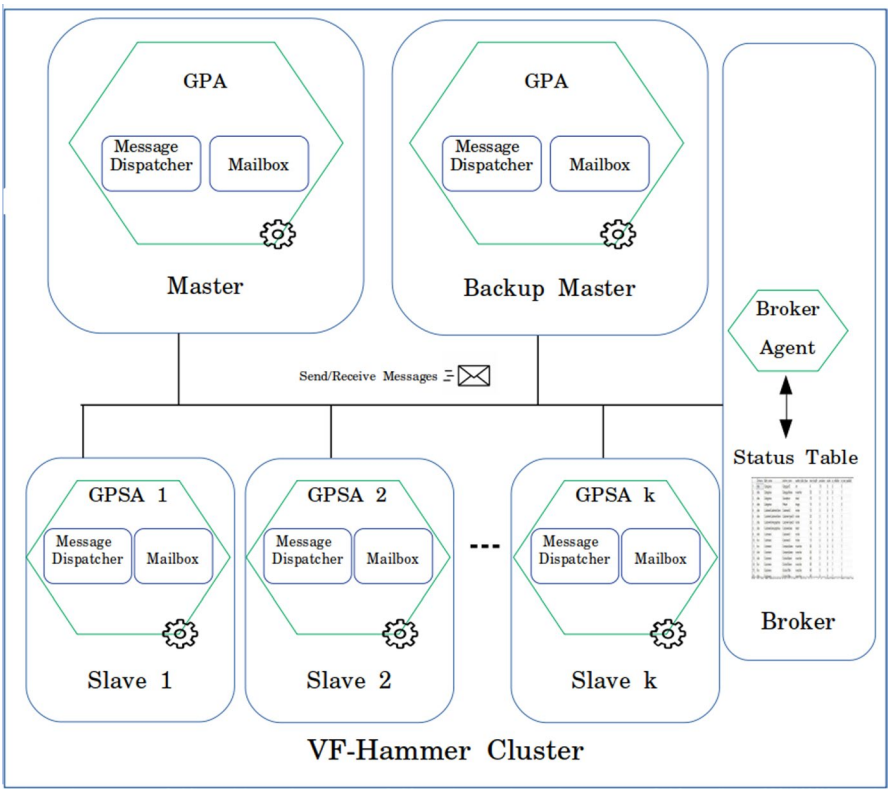

Fig. 3 VF-Hammer system communications

blocked waiting for receivers to process messages. This allows senders to send messages and continue their tasks without being blocked, while receivers can react to incoming messages and return execution when they finish processing their current jobs.

Each agent in the VF-Hammer cluster acts as a sender or receiver and has two types of mailbox: a Message Dispatcher and a Mailbox. The Mailbox component enqueues incoming messages in FIFO (First In First Out) order and serves one message at a time. The Message Dispatcher distributes messages to their target nodes simultaneously and instantly. By leveraging this message-driven architecture, VF-Hammer achieves efficient and scalable communication between its components.

To ensure efficient message exchange, VF-Hammer adopts an event-driven model, where agents communicate through asynchronous and non-blocking messages. When a message is sent from one agent to another, it is first dispatched to the message dispatcher, which distributes it to the appropriate agent's mailbox. The agent's mailbox then processes the message in a FIFO order, allowing for smooth and uninterrupted execution. This approach not only ensures timely

message delivery but also enables the agents to carry out their priority tasks without being blocked by incoming messages. Moreover, encapsulating jobs into message objects allows for better organization and management of the graph partitioning process, making it more scalable and adaptable to changing requirements.

4.6 Framework data flow

Now we will describe in general the graph partitioning process in particular its jobs distribution between the VF-Hammer agents.

The communication between the agents is ensured through message exchange, where each message contains a job to be executed by another agent. Figure 4 describes the distribution of the jobs between the two agents: the sender and receiver. The Sender and Receiver could be the GPA or GPSA agent. The GPA implements the graph partitioning algorithm as a sequence of jobs and the GPSA agents execute these jobs between them.

The lifecycle of any job is described through Fig. 4: The Sender prepares a job and identifies the Receiver who is responsible for executing the job. Then, the Sender writes the job to its Message dispatcher. The job passes throughout the Message Dispatcher as an instance of the message to the Mailbox of the Receiver. The mailbox serves the messages to the agent Receiver where the jobs are processed. When the Receiver finishes its task, it could send an acknowledgment message to the Sender.

To sum up, we have detailed the VF-Hammer architecture. We have presented the different components of the VF-Hammer system meanwhile their interactions. In addition, we have exposed the communication method used in our system (driven-message). Next, we will reveal the distributed algorithm used by the VF-Hammer framework for streaming graph partitioning in the next section.

5 Distributed hammer algorithm

In our VF-Hammer framework, we employ our proposed streaming algorithm, Hammer [14]. Hammer is a graph partitioning algorithm designed specifically for streaming graph partitioning problems. Unlike existing methods, Hammer is

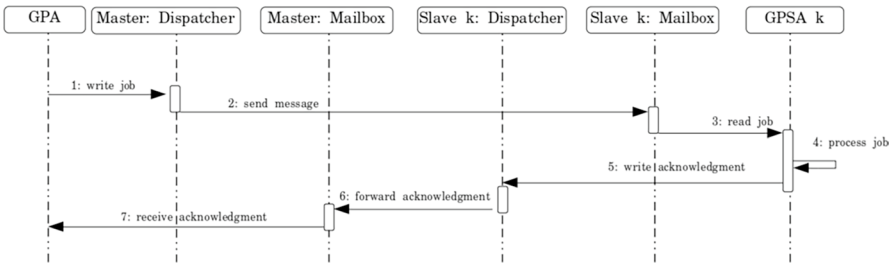

Fig. 4 Job distribution

based on the graph data volumes, including vertices and edges, during the graph partitioning process. It produces high-quality partitions, with balanced volumes and sizes. This motivated us to implement Hammer as the primary partitioning algorithm in our VF-Hammer framework. Moreover, Hammer is designed to run in a distributed and parallel environment, where it is divided into multiple jobs that are executed by the GPA and GPSA agents. As a result, our distributed streaming algorithm is referred to as dist-Hammer. The list of possible jobs derived from dist-Hammer is summarized in Table 3.

We divided the dist-Hammer algorithm into two parts in order to distribute and parallelize the partitioning process. The first part is executed on the master node by the Graph Partitioning Agent (GPA) component, while the second part is executed on the slave nodes by the Graph Partitioning Slave (GPS) component.

5.1 Dist-Hammer algorithm in GPA

The principle of the dist-Hammer is to distribute the stream of edges and vertices to their optimal partition P_k hosted on the slave S_k. The principal aim is to maximize the balance of partitions (in volume and size) while minimizing the total cut edges and replicated vertices.

The dist-Hammer algorithm in GPA is triggered when a fresh stream of graphs data (edges and vertices) are received. To refer to an edge or vertex, we will appoint it as an entity. The first thing to do is, the GPA checks the existence of each entity (edge /vertex) in its metadata if it is already assigned or not, since the metadata stores detailed information about an entity. If yes, then it queries the Partition Table for the locations of the entity. The GPA exploits this valuable information to partition properly the graphs.

Algorithm 1 describes the graph partitioning logic to assign the incoming entities (edges and vertices $e(u, v)$) toward their hosted slaves for final storage. However, the assignment of edges and vertices satisfies one of the following cases:

Table 3 List of main jobs of the distributed algorithm dist-Hammer

Job Title	Input	Output	Sender	Receiver	Description
assignVertex	v, P	vertex assigned	GPA	GPSA	assign the vertex v to its partition P
assignEdge	e, P	edge assigned	GPA	GPSA	assign the edge e to its partition P
splitedVertex	v	copy vertex created	GPSA	GPSA	ask the other GPSA to create a copy vertex and release the cut edges

Algorithm 1 Hammer

```
1:                                                  ▷ arrived of new edge e(u, v)
2: if u , v ∃ in metadata then
3:     P = (P(u) ∩ P(v))
4:     if P! = null then
5:                                                  ▷ case 1
6:         P* ← P
7:     else
8:                                                  ▷ case 2
9:             ▷ return the partition with minimum volume among the partition of u
   and v
10:        P* ← getminpartitionVolume(u , v)
11:        totalcut = totalcut + 1
12:    end if
13:                                                 ▷ send job to the slave of P*
14:    job: assignEdge(e(u, v), P*)
15:                                                 ▷ case 3
16: else if (u |v ∃! metadata ) then
17:    assignEdgeCase3(e(u, v))
18: else
19:                                                 ▷ case 4
20:    P* ← getminpartitionVolume()
21:    job : assignVertex(u, P*)
22:    assignEdgeCase3(e(u, v))
23: end if
```

Case 1: If *u* and *v* are assigned, and the partition of *u* and *v* intersect. The first case indicates that the vertices are assigned before and they exist in the same partition. Therefore, dist-Hammer will assign the edge to a partition in the intersection (see Algorithm 1 case 1). In this case, the edge *e* will be stored into a partition near to its connected vertices (*u*, *v*) which decreases the number of cut edges.

Case 2: If *u* and *v* are assigned, and the partition of *u* and *v* doesn't intersect. Unlike the first case, the two vertices *u* and *v* are already assigned to two different partitions (stored in different slaves). Therefore, dist-Hammer selects among the two partitions the optimal one P^* that has the minimum volume *getminpartitionVolume* **()** to store the edge *e*. In this case, the total number of cut edges rises because the endpoint vertices (*u*, *v*) of the edge belong to different partitions (in different slaves).

If either case 1 or case 2 is satisfied, dist-Hammer will send a message to the target slave of P^*, containing the job *job* : *assignEdge*$(e(u, v), P^*)$, instructing the GPSA to assign the edge *e* to the partition P^*.

In **Case 3**, when only one of the two vertices is assigned (e.g., vertex *u* is assigned and *v* is the new vertex), dist-Hammer attempts to assign the new vertex *v* close to its assigned neighbor *u* to reduce the number of cut edges, while maintaining balanced partition sizes and volumes. Algorithm 2 outlines the approach for this case, which can be summarized as follows:

First, dist-Hammer uses the metadata to identify the partition P^* with the minimum volume and extracts the volume of the partition (P_u) that stores the assigned vertex u. Second, it checks if the assigned vertex u belongs to the partition P^*, i.e., if $P^* = P_u$. If this condition is met, dist-Hammer assigns the new vertex v to P^*. Otherwise, dist-Hammer attempts to assign v close to its neighbor u in P_u, estimating the impact on the partition volumes. Using a score function that computes the difference between two volumes: $|\beta(P_u) - \beta(P^*)|/min(\beta(P_u), \beta(P^*))$, it compares the result with the defined imbalance factor λ. If the score is less than λ, then the two volumes are similar, and dist-Hammer assigns v to P_u with its neighbor u, since it will not impact the partition balance and reduce cut costs. Otherwise, dist-Hammer assigns v to P^* to avoid partition volume imbalance.

Once dist-Hammer determines the target slave to store v, it sends two messages: one to assign the vertex v and another to assign the edge. dist-Hammer also updates the metadata with the partition size and volume of the target slave and adds the vertex v to the Partition Table with its new partition.

Algorithm 2 assignEdgeCase3(*edge* $e(u, v)$)

```
1:  P* ← getminpartitionVolume()
2:
3:  if P_u != P* then
4:      score = |β(P_u) − β(P*)|/min(β(P_u), β(P*))
5:      if (score ≤ λ) then
6:          P* ← P_u
7:      else
8:          totalcut = totalcut + 1
9:      end if
10: end if
11: update metadata
12: job : assignVertex(v, P*)
13: job : assignEdge(e(u, v), P*)
```

Case 4: If neither vertex has been assigned, dist-Hammer employs a two-step process to determine the optimal partition for each vertex and the affiliated edge. First, it uses the metadata to identify the partition P^* with the minimum volume among all available partitions. It then assigns one of the vertices, e.g., u, to P^*. For the second vertex v, dist-Hammer applies the same approach as in **Case 3**, considering u as the assigned neighbor vertex in a partition P_u and P^* as the optimal partition with minimum volume. Dist-Hammer calculates the score function $|\beta(P_u) - \beta(P^*)|/min(\beta(P_u), \beta(P^*))$ to estimate the difference in partition volumes and compares it to the defined imbalance factor λ. If the score is less than λ, dist-Hammer assigns v to P_u to minimize the cut edges while maintaining the balance of partition volumes. Otherwise, it assigns v to P^* to prevent any significant imbalance in partition sizes. Finally, dist-Hammer sends two messages to the target slave, the first to assign u and v to their respective partitions and the second to assign the affiliated edge. Dist-Hammer also updates the metadata and the Partition Table to reflect the changes in partition sizes and locations of the vertices and edges.

5.2 Dist-Hammer algorithm in GPSA

The GPA plays a crucial role in the graph partitioning process by determining where to store each received entity, be it a vertex or an edge. On the other hand, the GPSA is responsible not only for storing received entities in their final partitions but also for applying local changes to its partition. Additionally, the GPSA splits vertices with a large degree to reduce communication costs during graph processing. The GPSA receives two types of jobs from the GPA: one to assign a vertex and the other to assign an edge.

Assigning an edge using the job "*assignEdge*($e(u, v), P^*$)" is more complicated than assigning a vertex. It involves saving an edge to the partition and executing a sequence of updates related to the affiliated vertices u and v. When a vertex reaches its maximum number of edges, it enters the splitting stage. During this stage, the large vertex is split over partitions that hold its neighbors, resulting in the reduction of cut edges across partitions and a decrease in the total number of cuts.

Algorithm 3 defines the procedure for assigning an edge $e(u, v)$ to the target partition, which involves a series of updates to its associated vertices u and v. Initially, the algorithm designates the new partition for e and increases the size and volume of its partition accordingly. Then, each vertex on the edge $e(u, v)$ adds its adjacent vertex to the list of neighbors, resulting in an increase in the degrees of both vertices u and v. The algorithm then checks whether each vertex of the edge $e(u, v)$ belongs to the local partition, and if the vertex has not been split and reaches the maximum number of edges (MAX_EDGES), then dist-Hammer executes a Vertex Splitting operation. This operation involves sending a message to each slave S that stores the remote neighbors of the vertex to create a copy vertex. Subsequently, dist-Hammer proceeds to release the cut edges associated with the splitting stage and retains only the list of local neighbors for each slave. This ensures that the communication costs during graph processing are minimized.

Algorithm 3 $job : assignEdge(e(u, v), P^*)$

```
1: store e into partition
2: β(P*) = β(P*) + β(e)
3: for each (vertex ∈ e) do
4:     if vertex ∃ in P* then
5:         if (vertex.split == false)and(vertex.degree ≥ MAX_EDGES) then
6:             vertex.split = true
7:             for each (slave holds neighbors of vertex) do
8:                 job : splitedVertex(vertex)
9:             end for
10:            releaseCutEdges()          ▷ original vertex releases its remote edges
11:
12:        end if
13:    end if
14: end for
```

In the next section, we will present a comprehensive evaluation of the performance of VF-Hammer using the distributed streaming algorithm dist-Hammer, in comparison with several state-of-the-art partitioning frameworks. We will use a variety of real-world and synthetic datasets to assess the effectiveness and scalability of VF-Hammer in different scenarios. Additionally, we will measure and compare the performance of VF-Hammer using several metrics such as partitioning time, balance, and communication cost. By conducting this rigorous evaluation, we aim to demonstrate the superiority of VF-Hammer and its potential for large-scale graph processing applications.

6 Evaluation

We conducted experiments to assess the performance of our graph partitioning method VF-Hammer on various real and synthetic datasets. We compared VF-Hammer to other partitioning methods and evaluated it based on several performance metrics.

6.1 Experimental setup

All evaluations were conducted on Amazon EC2 using m3.medium instances (1 virtual 64-bit CPU, 8GB of main memory, 1TB of a local hard disk). A pool of 21 instances was provisioned to accommodate all experimental configurations.

For the primary performance and quality evaluation, a fixed cluster topology was used to ensure a fair and consistent comparison between all partitioning frameworks. This cluster consisted of 1 master node and 10 slave nodes. Each slave node was configured to host a single partition, resulting in a fixed number of partitions $K = 10$ for these experiments.

For the scalability analysis (Section 6.4.5), the cluster size was varied from 4 to 20 slave nodes (plus the master node) to evaluate the strong scaling efficiency of the distributed frameworks. In this setup, the number of partitions K was set to equal the number of available slave nodes.

All algorithms have been implemented in Java.

6.2 Dataset selection

We evaluated the performance of VF-Hammer using popular real-world graph datasets from the SNAP collection (Table 4), which provides diverse graph topologies from various domains. Since these datasets only contain the graph structure without storage volume information, we incorporated synthetic volumes to evaluate our volume-aware partitioning framework. This approach is necessary to model the variability in memory footprint found in real-world property graphs, where vertices and edges can have properties of vastly different sizes (e.g., text profiles, images, sensor readings).

Table 4 Summary description of used datasets

Dataset	Type	Vertices	Edges
com-amazon	undirected	334 863	925 872
dblp	undirected	317 280	1 049 866
amazon0302	undirected	262 111	1 234 877
twitter	directed	81 306	1 768 149
web-Stanford	directed	281 903	2 312 497
com-youtube	undirected	1 134 890	2 987 624
roadNet-CA	undirected	1 965 206	2 766 607
web-Google	directed	875 713	5 105 039
in-2004	undirected	1 382 908	13 591 473
eu-2005	undirected	862 664	16 138 468
com-LiveJournal	undirected	3 997 962	34 681 189
orkut	undirected	3 072 441	117 185 083

The synthetic volumes were generated according to the following rationale, designed to simulate a realistic and challenging scenario:

Vertex Volume Range [1, 150]: This range models entities with highly variable storage needs. A vertex with a volume of 1 could represent a minimal object with an identifier, while a vertex with a volume of 150 could represent a complex object with multiple large attributes (e.g., a user profile with a picture, biography, and metadata).

Edge Volume Range [1, 14]: This smaller, constrained range models relationship entities, which typically require less storage than the nodes they connect (e.g., storing a timestamp, weight, or type). The lower bound ensures no entity has zero volume.

This configuration creates a challenging partitioning problem where balancing the total partition volume is distinct from balancing the partition size (vertex/edge count). The total graph volume $\beta(G)$ for each dataset was calculated as the sum of the volumes of all its vertices and edges. The structural properties of the selected graphs are outlined in Table 4.

6.3 Performance metrics

There are various metrics available to measure the quality of partitioning results. The most straightforward metrics include execution time, i.e., the time taken to partition the graph, and the cost of edge-cuts, which refers to the communication cost between machines. However, the most crucial metric is balance, which we use to evaluate the partitioning quality of each algorithm. To assess balance, we employ a set of criteria, which are defined as follows:

6.3.1 Balance

Balance is the most significant metric we use to evaluate the quality of partitioning for each algorithm. We use the standard deviation to measure the distribution of graph data over machines. A low standard deviation indicates that the servers tend to be close to the mean (expected value) server size, while a high standard deviation indicates that the servers are spread out over a wide range of values. In fact, there are two metrics of balance.

The first metric of balance compares the sizes of the servers. In this context, a server's size refers to the set of vertices or edges that are distributed equally between machines. This metric measures the sizes of the servers against the average size, which is determined using Eq. 1. The mean size of one machine is obtained by dividing the total number of vertices n in the graph equally between the k machines. Typically, the mean size represents the ideal size for one server. By comparing the actual server sizes to the mean, we can determine if the partitioning is balanced or not. A balanced partitioning distributes the vertices or edges uniformly across all machines, resulting in server sizes that are closer to the mean, and consequently, a low standard deviation. On the other hand, an imbalanced partitioning leads to servers with significantly different sizes, and therefore, a high standard deviation.

$$mean_s = \frac{n}{K} \tag{1}$$

To evaluate the balance size of servers, we utilize the standard deviation (SD) of size, as shown in Eq. 2. The SD of size provides a measure of the difference between the sizes of the servers $\chi(S_k)$, which is the number of vertices in each server S_k, and the mean size $mean_s$. Additionally, we also utilize the relative standard deviation (RSD) of size, as shown in Eq. 3, which measures the disparity of the servers' sizes around the mean. The RSD of size provides a more accurate value for evaluating the balance size of servers since it takes into account the mean size.

The SD of size indicates the disparity between the sizes of the servers, with a higher SD value indicating greater imbalance between the servers. The RSD of size, on the other hand, represents the SD of size as a percentage of the mean size, making it easier to compare the balance across different graphs or algorithms. A lower RSD of size indicates a better-balanced partitioning, while a higher RSD of size indicates a more imbalanced partitioning. By using both the SD and RSD of size, we can gain a more comprehensive understanding of the balance size of servers in a partitioned graph.

$$SDsize = \sqrt{\frac{1}{K}\sum_{k=1}^{K}(S(S_k) - mean_s)^2} \tag{2}$$

$$RSDsize = \frac{SDsize}{mean_s} \tag{3}$$

The second metric we use to evaluate the balance is the balanced volume. In our previous work [13], we introduced a novel metric of balance that evaluates the balance of partition volumes. Specifically, it measures the total volume of graph data, including vertices, edges, and replicated entities, distributed among different machines and compares it to the mean volume, as shown in Eq. 4.

The mean volume of one server is the ideal server volume, where it is equal to the volume of graph data $\beta(G)$, which is defined in Eq. 4, distributed equally among k machines. By comparing the actual volumes of servers to the mean, we can determine the balance of the partitioning. A balanced partitioning distributes the graph data uniformly across all machines, resulting in volumes that are closer to the mean and therefore, a low standard deviation. Conversely, an imbalanced partitioning results in servers with significantly different volumes, leading to a high standard deviation.

$$mean_v = \frac{\beta(G)}{K} \tag{4}$$

In order to assess the balance volumes of servers, we utilized the standard deviation (SD) of volume. The SD value indicates the variation in volume of the servers $\beta(S_K)$ (as described in Eq. 5), in comparison with the mean volume. To provide a more precise measure of the distribution of the volumes, we also used the relative standard deviation (RSD) of volume (as shown in Eq. 6). This allowed us to analyze the disparity of servers' volumes around the mean.

$$SDvolume = \sqrt{\frac{1}{K}\sum_{k=1}^{K}(\beta(S_k) - mean_v)^2} \tag{5}$$

$$RSDvolume = \frac{SDvolume}{mean_v} \tag{6}$$

6.3.2 Communication cost

The communication cost is directly linked to the number of edge-cuts. The edge-cut method partitions the graph's vertices among multiple machines while permitting edges to span across two servers (cuts). Each edge-cut signifies a communication link between two machines. As the number of edge-cuts increases, the communication cost during graph processing or querying also increases. Thus, the primary objective of graph partitioning is to minimize communication cost by minimizing the number of edge-cuts.

To evaluate the effectiveness of a partitioning scheme, we use the edge-cut ratio σ, which measures the ratio of the number of edge-cuts to the total number of edges in the graph (as displayed in Eq. 7). Let S_k denote the set of vertices assigned to partition P_k and let $N(v_i)$ be the set of neighbors of vertex v_i The number of edge-cuts can be formally defined as the number of edges where the connected vertices belong

to different partitions. By minimizing σ, we can reduce the communication cost and enhance the performance of graph processing or querying.

$$\sigma = \frac{1}{2} \sum_{k=1}^{K} \sum_{i=1} N(v_i) \setminus S_k \tag{7}$$

6.3.3 Time

The time metric measures the total runtime required to complete the graph partitioning process. For our evaluations, we measured the total wall-clock time, recorded on the master node, from the initiation of the partitioning job until the entire graph was processed and all entities were assigned to their respective partitions. This includes the full computation, communication, and coordination overhead for the entire framework. For streaming experiments, this represents the total latency to process the entire graph stream. Each experiment was repeated 5 times, and the average runtime is reported to ensure measurement stability. By analyzing the time required for each algorithm to partition the graph, we can evaluate its efficiency and practical applicability. This allows us to select the optimal algorithm for graph partitioning based on performance and runtime requirements.

6.4 Evaluation results

To thoroughly evaluate the efficiency of our proposed VF-Hammer framework, we selected three state-of-the-art partitioning systems for comparison. This selection is designed to benchmark VF-Hammer against leading representatives from key categories of partitioning strategies:

KaHIP (Karlsruhe High Quality Partitioning) [8] is included as a gold-standard offline, batch partitioner. It produces very high-quality partitions with low edge-cuts and serves as a quality baseline, representing the best achievable result when computational time is not a primary constraint.

DynamicDFEP [40] is chosen as a representative distributed streaming partitioner designed for dynamic graphs. It shares VF-Hammer's target environment (large-scale, evolving graphs) and distributed nature, providing a direct comparison of performance within the same domain.

IOGP (Incremental Online Graph Partitioner) [29] is selected as a highly efficient streaming partitioner for graph databases and OLTP workloads. It represents systems that prioritize low-latency assignment, providing a contrast to volume-aware strategies.

This diverse set of baselines allows us to demonstrate VF-Hammer's advantages across multiple dimensions: it achieves higher quality than streaming alternatives (DynamicDFEP, IOGP) and does so with the scalability and speed that offline methods (KaHIP) lack.

We used KaHIP with its default parameters (3% balance parameter and no time limit) to establish a quality-oriented benchmark. For a fair comparison in

the streaming context, we configured the systems as follows: the vertex reassigning threshold for IOGP was set to REASSIGN_THRSH=100, and both IOGP and VF-Hammer employed a maximum threshold of MAX_EDGES=1000 to determine when to split a vertex. We enforced a strict volume imbalance threshold, defined as $\lambda = 0.1$. This means the volume of any partition could not exceed the volume of the smallest partition by more than 10%.

6.4.1 Balance size

Figure 5 presents the results for partition size balance, measured by the relative standard deviation (RSD) of partition sizes. A lower RSD value indicates a more balanced distribution of vertices and edges across partitions.

We observe that VF-Hammer's performance exceeded that of the existing systems DynamicDFEP, IOGP, and KaHIP. As shown in Fig. 5, VF-Hammer produced balanced partition sizes with less than 2% imbalance between partitions sizes in all graph datasets used.

During graph partitioning, VF-Hammer manages the size of partitions in each slave node to keep them balanced and avoid creating overloaded or underloaded partitions, thanks to the master metadata component. In contrast, DynamicDFEP and IOGP produced almost similar results in balanced partition sizes for most datasets. KaHIP produced approximately the same results for all datasets used because it uses a strict balance parameter that does not exceed 3% of the imbalance size.

Overall, VF-Hammer outperformed the existing systems in terms of balanced partition sizes. This demonstrates the effectiveness of our proposed framework in achieving balanced partitioning in various graph datasets.

6.4.2 Balance volume

The results for partition volume balance are presented in Fig. 6. This figure shows the relative standard deviation (RSD) of the total memory volume (in bytes) for each partition. A lower value indicates superior balance in terms of physical storage footprint, which is crucial for preventing memory-based stragglers.

As expected, VF-Hammer produces partitions with well-balanced volumes. Our framework is uniquely designed to manage the storage volume of different nodes in a cluster, and it is the first framework to implement a graph partitioning algorithm that is natively based on graph data volume. Remarkably, VF-Hammer achieves an imbalance in partitions volumes of no more than 0.5% for all graph datasets used, except for the graph eu-2005, which was 1%. This impressive result is thanks to the careful consideration of volume balance in our algorithm, which ensures that each slave node has a similar amount of data to process. In contrast, KaHIP produces partitions with imbalanced volumes, as their results are scattered around 20% far from the mean volume. As shown in Fig. 5, all the balance sizes of partitions are relatively close for all datasets due to the strict imbalance factor employed by these methods. Therefore, their partition volumes are approximately similar, but they still exhibit a significant imbalance compared to the results achieved by VF-Hammer. In particular, DynamicDFEP and IOGP show a great imbalance in their partitions'

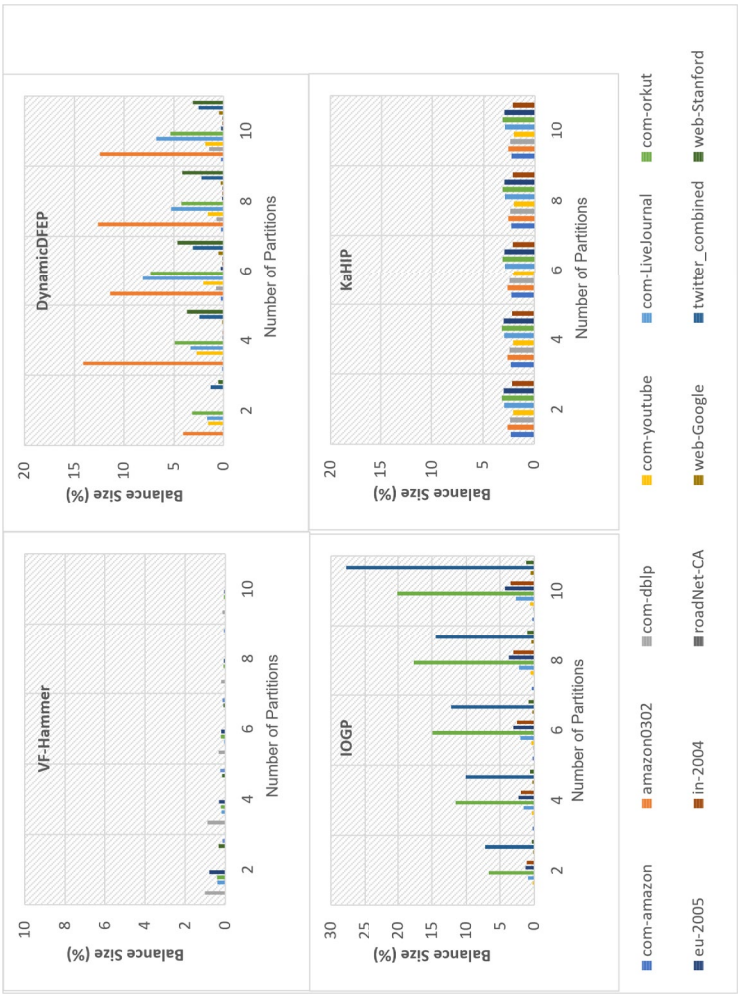

Fig. 5 Evaluation of partition size balance across all datasets. Lower values indicate more balanced partitions

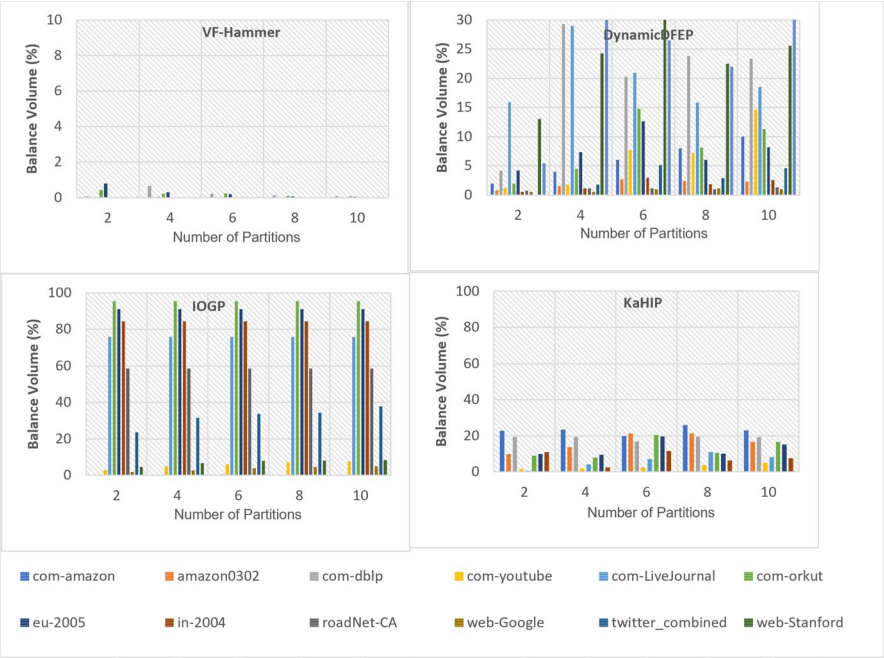

Fig. 6 Evaluation of partition volume balance across all datasets. Lower values indicate more balanced memory usage

volumes, with IOGP producing imperfect partitioning quality, especially for volume balance. In summary, VF-Hammer outperforms existing methods in achieving highly balanced partitions in terms of both size and volume, making it a highly efficient and effective partitioning system for large-scale graph processing.

6.4.3 Communication cost

Figure 7 evaluates the communication cost, represented by the edge-cut ratio. This metric measures the fraction of edges that are cut between partitions; a lower ratio indicates fewer remote communications and thus lower overhead during distributed graph processing.

The results indicate that KaHIP is the most efficient framework in terms of minimizing the number of cuts between partitions. It achieves a significantly lower cut ratio than all the other frameworks, including VF-Hammer. However, VF-Hammer produces a lower cut ratio than DynamicDFEP and IOGP for most of the datasets used. The two latter frameworks generate a large number of cuts, both in terms of edge-cuts and vertex-cuts, which could lead to significant communication overhead during graph processing. On the other hand, VF-Hammer employs a strategy that balances the load among the partitions while minimizing the number of cuts. Nevertheless, it is important to note that VF-Hammer has a relatively high rate of cuts in the com-orkut and com-LiveJournal datasets. Overall, the results suggest that

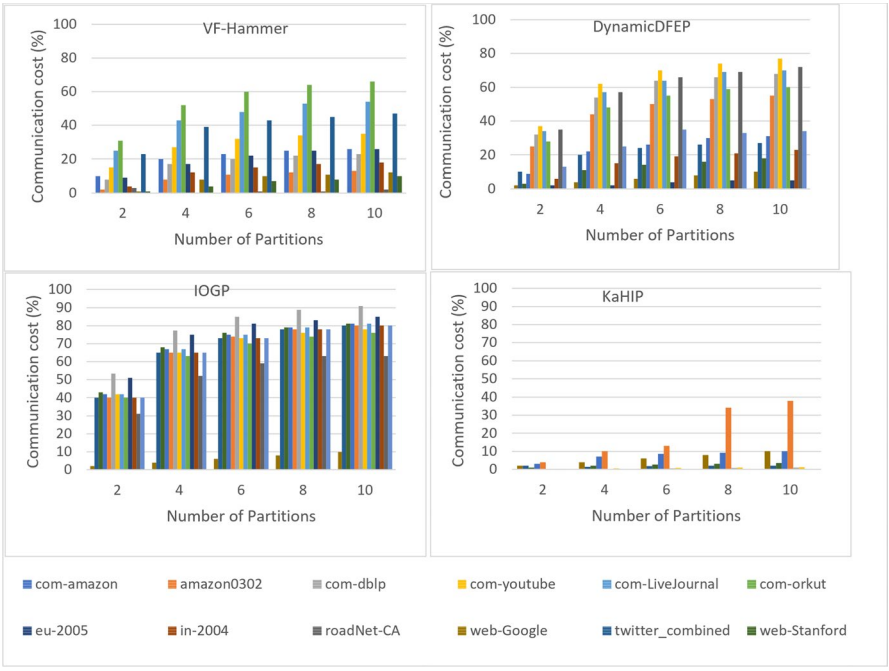

Fig. 7 Communication cost measured by the edge-cut ratio. Lower values are better

VF-Hammer provides a good trade-off between partition balancing and minimizing the number of cuts between partitions, making it a promising approach for large-scale graph partitioning.

6.4.4 Runtime

The partitioning runtime is an important metric, especially for streaming graph processing. Figure 8 plots the total time required to complete the partitioning process for all graphs and frameworks.

VF-Hammer accomplishes the total graphs partitioning in an efficient runtime in order to produce the best balance volume of partitions, with a good balance size of partitions and also with an acceptable cut ratio. Despite the fact that the other systems DynamicDFEP, IOGP and KaHIP have a good partitioning time, VF-Hammer remains faster compared to them.

6.4.5 Scalability assessment

A core claim of our work is that VF-Hammer's distributed architecture is highly scalable. To validate this empirically, we conducted a strong scaling analysis. In this experiment, we partition the fixed, large-scale com-orkut graph while increasing the number of slave nodes from 4 to 20. The objective is to measure the reduction in runtime as more computational resources are added.

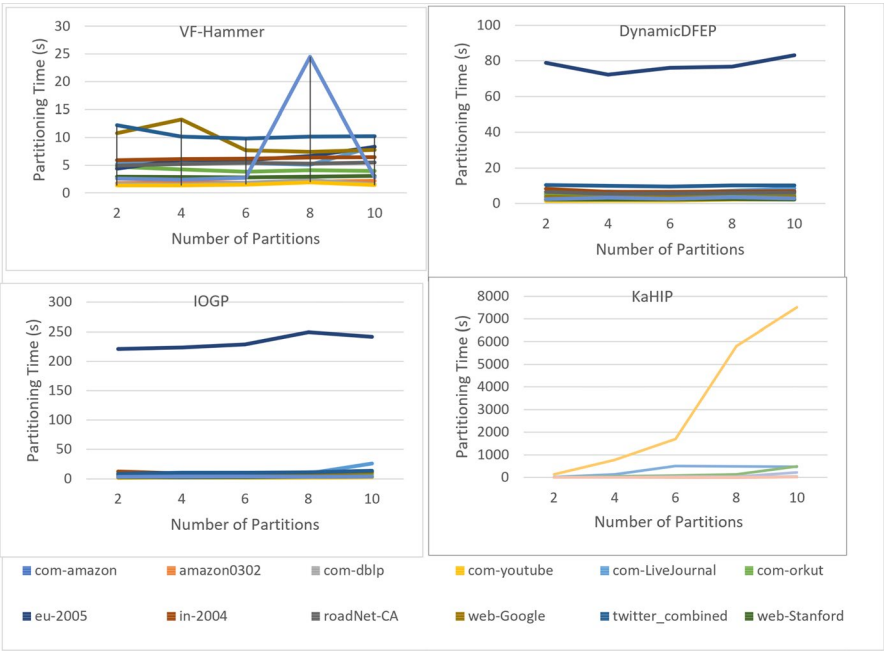

Fig. 8 Total partitioning runtime (in seconds) for each framework and dataset. Lower values are better

For this evaluation, we compare VF-Hammer against DynamicDFEP as a baseline. This choice is deliberate: DynamicDFEP is a distributed streaming partitioner, making it architecturally comparable to VF-Hammer and providing a direct, apples-to-apples comparison of how each framework utilizes additional nodes. Other frameworks are excluded from this specific test for fundamental architectural reasons:

IOGP is also a distributed partitioner but is designed for a different primary goal (OLTP workload handling in graph databases). Its partitioning strategy is less relevant for a strong scaling performance test compared to the more general-purpose DynamicDFEP.

KaHIP is a high-quality but centralized partitioner. It is designed to run on a single machine and cannot leverage multiple slave nodes. Including it in a strong scaling experiment would be invalid, as its runtime would remain constant regardless of the cluster size.

The results of our strong scaling experiment (Figure 9 clearly demonstrates VF-Hammer's scalability advantages.

With 4 nodes, both distributed frameworks have comparable runtimes. However, as we increase the cluster size to 16 nodes, VF-Hammer's runtime drops to 480 s, a 3.75x speedup. In contrast, DynamicDFEP only achieves a 2.44x speedup (from 2200 s to 900 s), indicating higher communication and coordination overhead in its architecture.

VF-Hammer's performance curve closely follows the ideal scaling line, notably at 8 and 12 nodes, confirming that its master–slave architecture and asynchronous

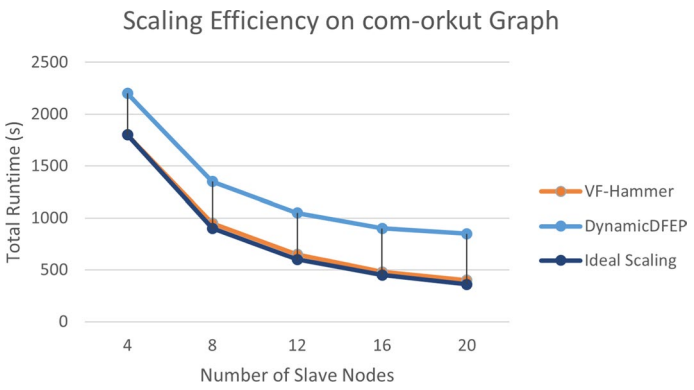

Fig. 9 Strong Scaling Efficiency. VF-Hammer's runtime decreases significantly with added nodes, demonstrating superior scaling efficiency compared to DynamicDFEP

communication model efficiently utilize added resources. The growing performance gap at 16 and 20 nodes shows that VF-Hammer maintains its efficiency better at scale, making it more suitable for large clusters. This provides the empirical evidence that VF-Hammer's distributed design successfully translates into superior scalability.

7 Conclusion

In conclusion, this paper has presented a comprehensive study on streaming graph partitioning for property graph models with large volumes. The study highlighted the impact of volume on partitioning results, which can lead to data storage and workload imbalance. To address this issue, we proposed a novel framework called VF-Hammer that leverages vertex volumes to achieve a balanced distribution of graph data among distributed servers and reduce communication costs. Our experimental results, obtained from real and synthetic datasets, demonstrated that VF-Hammer outperformed state-of-the-art algorithms such as IOGP, DynamicDFEP, and KaHIP in terms of generating highly balanced partitions with a good balance of partition sizes. Therefore, the proposed VF-Hammer framework provides an efficient solution for streaming graph partitioning that can handle large graph volumes with improved performance.

Author Contributions Chayma Sakouhi and Abir Khaldi wrote the main manuscript text. All authors reviewed the manuscript.

Data Availability No datasets were generated or analyzed during the current study.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Malewicz G, Austern MH, Bik AJ, Dehnert JC, Horn I, Leiser N, Czajkowski G (2010) Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, pp 135–146. ACM
2. Gonzalez JE, Low Y, Gu H, Bickson D, Guestrin C (2012) Powergraph: distributed graph-parallel computation on natural graphs. In: OSDI, vol. 12, p. 2
3. Gonzalez JE, Xin RS, Dave A, Crankshaw D, Franklin MJ, Stoica I (2014) Graphx: Graph processing in a distributed dataflow framework. In: OSDI, vol. 14, pp. 599–613
4. Kernighan BW, Lin S (1970) An efficient heuristic procedure for partitioning graphs. Bell Syst Tech J 49(2):291–307
5. Rahimian F, Payberah AH, Girdzijauskas S, Jelasity M, Haridi S (2013) Ja-be-ja: A distributed algorithm for balanced graph partitioning. In: 2013 IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems, pp 51–60. IEEE
6. Rahimian F, Payberah AH, Girdzijauskas S, Haridi S (2014) Distributed vertex-cut partitioning. In: IFIP International Conference on Distributed Applications and Interoperable Systems, pp. 186–200. Springer
7. Karypis G, Kumar V (1999) Parallel multilevel series k-way partitioning scheme for irregular graphs. SIAM Rev 41(2):278–300
8. Sanders P, Schulz C (2019) KaHIP–Karlsruhe High Quality Partitioning. Accessed: Dec
9. Tsourakakis C, Gkantsidis C, Radunovic B, Vojnovic M (2014) Fennel: Streaming graph partitioning for massive scale graphs. In: Proceedings of the 7th ACM International Conference on Web Search and Data Mining, pp 333–342. ACM
10. Zhang W, Chen Y, Dai D (2018) Akin: a streaming graph partitioning algorithm for distributed graph storage systems. In: Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pp 183–192. IEEE Press
11. Patwary MAK, Garg S, Kang B (2019) Window-based streaming graph partitioning algorithm. In: Proceedings of the Australasian Computer Science Week Multiconference, p 51. ACM
12. Mayer C, Mayer R, Tariq MA, Geppert H, Laich L, Rieger L, Rothermel K (2018) Adwise: Adaptive window-based streaming edge partitioning for high-speed graph processing. In: 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), pp 685–695. IEEE
13. Sakouhi C, Khaldi A, Ghezala HB (2018) Volume: Novel metric for graph partitioning. In: Proceedings of the International Conference on Information and Knowledge Engineering (IKE), pp. 174–180. The Steering Committee of The World Congress in Computer Science, Computer
14. Sakouhi C, Khaldi A, Ghezala HB (2021) Hammer lightweight graph partitioner based on graph data volumes. J Parallel Distrib Comput 158:16–28
15. Garey MR, Johnson DS, Stockmeyer L (1976) Some simplified np-complete graph problems. Theoret Comput Sci 1(3):237–267
16. Hendrickson B, Leland RW et al (1995) A multi-level algorithm for partitioning graphs. SC 95(28):1–14
17. Karypis G, Kumar V (1998) A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J Sci Comput 20(1):359–392
18. Karypis G, Kumar V (2000) Multilevel k-way hypergraph partitioning VLSI design 11(3):285–300
19. Karypis G, Kumar V (1998) Multilevelk-way partitioning scheme for irregular graphs. J Parallel Distrib Comput 48(1):96–129
20. Karypis G, Schloegel K, Kumar V (1997) Parmetis: Parallel graph partitioning and sparse matrix ordering library
21. Von Luxburg U (2007) A tutorial on spectral clustering. Stat Comput 17(4):395–416
22. Devine KD, Boman EG, Heaphy RT, Bisseling RH, Ataly Rek V (2006) New challenges in dynamic load balancing. Appl Numer Math 52(2):133–152
23. Guerrieri A, Montresor A (2015) Dfep: Distributed funding-based edge partitioning. In: European Conference on Parallel Processing, pp 346–358. Springer
24. Azad A, Pavlopoulos GA, Ouzounis CA, Kyrpides NC, Buluç A (2018) HIPMCL: a high-performance parallel implementation of the Markov clustering algorithm for large-scale networks. Nucleic Acids Res 46(6):33–33
25. Schwarzer D, Ganev S, Garg A, Ke NR, Blundell C (2020) Learning to partition graphs with aRL. In: International Conference on Learning Representations (ICLR)

26. Li G, Xiong C, Thabet A, Ghanem B (2021) Graph partition neural networks for semi-supervised classification. In: International Conference on Learning Representations (ICLR)
27. Bengio Y, Lodi A, Prouvost A (2021) Machine learning for combinatorial optimization: a methodological tour d'horizon. Eur J Oper Res 290(2):405–421
28. Petroni F, Querzoni L, Daudjee K, Kamali S, Iacoboni G (2015) HDRF: Stream-based partitioning for power-law graphs. In: Proceedings of the 24th ACM International on Conference on Information and Knowledge Management, pp 243–252. ACM
29. Dai D, Zhang W, Chen Y (2017) IOGP: An incremental online graph partitioning algorithm for distributed graph databases. In: Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing, pp 219–230. ACM
30. Li Y, Li C, Orgerie A-C, Parvédy PR (2021) WSGP: A window-based streaming graph partitioning approach. In: 2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid), pp 586–595. IEEE
31. Faraj MF, Schulz C (2021) Buffered streaming graph partitioning. Preprint at arXiv:2102.09384
32. Hendrickson B, Leland R (1993) The chaco users guide. version 1.0. Technical report, Sandia National Labs., Albuquerque, NM (United States)
33. Sanders P, Schulz C (2011) Engineering multilevel graph partitioning algorithms. In: European Symposium on Algorithms, pp 469–480. Springer
34. Sanders P, Schulz C (2012) Distributed evolutionary graph partitioning. In: 2012 Proceedings of the Fourteenth Workshop on Algorithm Engineering and Experiments (ALENEX), pp 16–29. SIAM
35. Sanders P, Schulz C (2013) Think locally, act globally: Highly balanced graph partitioning. In: International Symposium on Experimental Algorithms, pp 164–175. Springer
36. Meyerhenke H, Sanders P, Schulz C (2017) Parallel graph partitioning for complex networks. IEEE Trans Parallel Distrib Syst 28(9):2625–2638
37. Akhremtsev Y, Sanders P, Schulz C (2020) High-quality shared-memory graph partitioning. IEEE Trans Parallel Distrib Syst 31(11):2710–2722
38. Schlag S, Schulz C, Seemaier D, Strash D (2019) Scalable edge partitioning. In: 2019 Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX), pp 211–225. SIAM
39. Slota GM, Rajamanickam S, Devine K, Madduri K (2017) Partitioning trillion-edge graphs in minutes. In: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp 646–655. IEEE
40. Sakouhi C, Aridhi S, Guerrieri A, Sassi S, Montresor A (2016) Dynamicdfep: a distributed edge partitioning approach for large dynamic graphs. In: Proceedings of the 20th International Database Engineering & Applications Symposium, pp 142–147. ACM
41. Nicoara D, Kamali S, Daudjee K, Chen L (2015) Hermes: Dynamic partitioning for distributed social network graph databases. In: EDBT, pp 25–36