IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 6 October 2022, accepted 25 October 2022, date of publication 4 November 2022, date of current version 15 November 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3219422

SURVEY

Graph Computing Systems and Partitioning Techniques: A Survey

TEWODROS ALEMU AYALL[1,2], HUAWEN LIU[1,3], CHANGJUN ZHOU[1], ABEGAZ MOHAMMED SEID[2,4], (Member, IEEE), FANTAHUN BOGALE GEREME[5], HAYLA NAHOM ABISHU[6], (Graduate Student Member, IEEE), AND YASIN HABTAMU YACOB[6]

[1]Department of Computer Science, Zhejiang Normal University, Jinhua, Zhejiang 321004, China
[2]Department of Computer Science, Dilla University, Dilla, Ethiopia
[3]Department of Computer Science, Shaoxing University, Shaoxing, Zhejiang 312000, China
[4]Information and Computing Technology, Hamad Bin Khalifa University, Doha, Qatar
[5]Institute of Fundamental and Frontier Sciences, University of Electronic Science and Technology of China, Chengdu 611731, China
[6]School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chengdu 611731, China

Corresponding authors: Tewodros Alemu Ayall (ayalltewodros@zjnu.edu.cn) and Huawen Liu (liu@usx.edu.cn)

This work was supported in part by the Postdoctoral Foundation of Zhejiang Normal University under Grant ZC304021941, in part by the National Science Foundation of China under Grant 61976195 and Grant 62272418, in part by the National Science Foundation of Zhejiang Province under Grant Z23F020009, and in part by the Basic Public Welfare Research Program of Zhejiang Province under Grant LGG18E050011.

ABSTRACT Graphs are a tremendously suitable data representations that model the relationships of entities in many application domains, such as recommendation systems, machine learning, computational biology, social network analysis, and other application domains. Graphs with many vertices and edges have become quite prevalent in recent years. Therefore, graph computing systems with integrated various graph partitioning techniques have been envisioned as a promising paradigm to handle large-scale graph analytics in these application domains. However, scalable processing of large-scale graphs is challenging due to their high volume and inherent irregular structure of the real-world graphs. Hence, industry and academia have been recently proposing graph partitioning and computing systems to process and analyze large-scale graphs efficiently. The graph partitioning and computing systems have been designed to improve scalability issues and reduce processing time complexity. This paper presents an overview, classification, and investigation of the most popular graph partitioning and computing systems. The various methods and approaches of graph partitioning and diverse categories of graph computing systems are presented. Finally, we discuss main challenges and future research directions in graph partitioning and computing systems.

INDEX TERMS Distributed computing, graph computing systems, graph partitioning, graph processing systems, graph databases, graph algorithm, large-scale graph analysis.

I. INTRODUCTION

Graphs are a significant and powerful data representations to model the relationships of entities in many application domains in the form of vertices and edges. In general, vertices represent the entities in the graph, while edges indicate the relationships among the entities in the graph. Graphs are used in search engines to model the relevance of web pages recommended to users [1], [2], and the segment of road networks is modeled by graph [3]. In computational biology, graphs are applicable to represent the interaction of protein-to-protein [4], [5], [6] and the layout of infectious diseases [7]. The interactions of users and groups in social networks are also represented by graph [8], [9], [10], [11]. For example, social networks are made up of social ties, which include relationships between people or groups based on friendship, interest, kinship, likes/dislikes, and various other factors. Those relationships can be visualized as a graph representation. Fig.1 illustrates how to represent a social network using the friendship of 34 karate club members. Each vertex

The associate editor coordinating the review of this manuscript and approving it for publication was Massimo Cafaro.

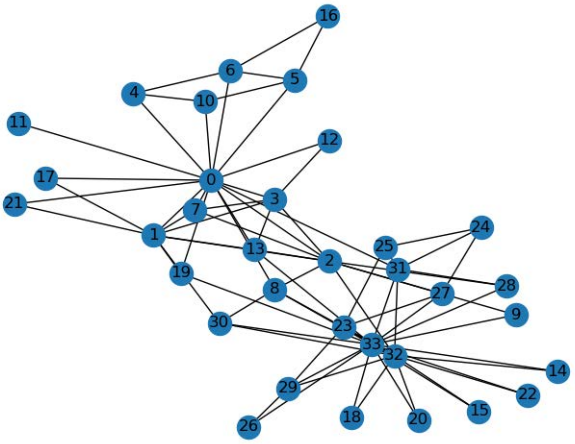


FIGURE 1. An example of social network model of relationships in the Karate Club [15].

represents an individual, and the links/edges show individuals who interact outside of the karate club setting (e.g., meeting up for a coffee or spending social time together).

The study of network analysis has become not only essential but also interdisciplinary in nature since graphs can appear in such a wide variety of settings. The study of these complex systems requires an understanding of their characteristics, as well as their structure and their dynamics. Therefore, the academia and big technology companies like Facebook, Google, and Microsoft have proposed different solutions for organizing and analyzing the rising prevalence of big graphs [12]. Furthermore, the size of these graphs has rapidly increased, with hundreds of billions of nodes and trillions of edges being possible [13], [14]. As the graph size scales up, graph analysis can be performed in a distributed environment. However, graph computing has become a challenging problem due to access irregularity, lack of locality, and intrinsic load imbalance distribution of graphs in different computing clusters [14]. Thus, researchers highlight the critical role of design computing systems in our society today [12].

Graph computing systems are becoming increasingly significant to deal with graphs-based analytics such as graph traversal [16], random walk [1], graph aggregation [17], motifs discovery [18] etc. The design of graph computing systems focuses on two major categories, graph processing systems (GPS) [19], [20], [21], [22], [23], [24] and graph database systems (GDBS) [25], [26], [27], [28] based on their graph analytics nature. GPS execute large-scale batch analytics using a variety of computationally intensive graph algorithms. Google introduced Pregel [19], the pioneer distributed GPS, to process interconnected data since 2010. After that many graph computing systems have recently been proposed in distributed [20], [21], [29] and single-machine [30], [31], [32], [33], [34], [35] computing architecture to improve scalability issues and reduce the systems' processing time complexity. On the other hand, GDBS are designed for high-throughput data retrieval and transaction processing. Before the graph databases systems, relational database management systems (RDBMS) were widely used to store, process, and analyze large-scale graphs [36]. However, there are two issues with analysis of graph in RDBMS. First, the vertices (nodes) and edges (relationships) are stored in separate tables. Therefore, it requires complex join operations to perform a query [37]. Second, RDBMS are ineffective when the data model changes over time, which means they rely on a fixed schema and make it difficult to build new object relationships [38]. Hence, due to these limitations of RDBMS, GDBS [25], [26], [27], [28] have been proposed to store, process, and analyze large-scale graphs.

Graph partitioning is a technique to cut graph into distinct subgraphs based on different heuristic techniques by minimizing cuts and maximizing load balance. Solving the graph partitioning problem with the minimum cut and maximum load balance is a well-known NP-hard problem [39], [40]. Graph partitioning is used as a significant preprocessing step for large-scale graph computing systems. Integrating graph partitioning techniques with computing systems can solve many graph problems in data mining, graph machine learning and pattern discovery. Researchers have proposed many graph partitioning algorithms in the last decade. The methods of these graph partitioning can be categorized into three: vertex partitioning [41], [42], [43], [44], edge partitioning [23], [45], [46], [47], [48], [49], [50], and hybrid partitioning [22], [51], [52], [53]. These methods can further be classified as offline (in-memory), online (stream), offStream, and dynamic approaches. The offline approach loads the whole graph in memory and exploits the graph's global information to allocate edges or vertices to the partitions. Many offline algorithms have been proposed in sequentially, shared, and distributed memory. Before the offline approach starts partitioning, the input graph is loaded in memory. Therefore, it can quickly gather the global graph structure to solve the optimization problem. This case leads to obtaining a higher partitioning quality. However, it does not support large-scale graph partitioning. This issue motivated the design of an online approach to scalable graph partitioning [43]. The online approach loads vertices or edges one by one to directly assign them to the partitions. Online approach is very fast and consumes little memory; yet, it yields a low-quality partitioning. Therefore, offStream approach has been proposed to fill the gap between offline and stream approaches by slitting the edges of a graph into two edge sets. One edge set is partitioned in the offline approach, and another edge set is partitioned in the stream [54], [55]. Sometimes real-world graphs are not only static but also dynamic in that their topologies are dynamically changed because some vertices and edges may be removed or added from the graphs over time. Therefore, the dynamic approach has been proposed for repartitioning when the graphs' topology is dynamically changed [56], [57], [58].

Many research works exist on graph partitioning and computing systems in the current literature. These research works motivated us to provide a structured review of the extensive

TABLE 1. Related surveys.

No.	GP			GCS			Basic benchmarks and evaluation metrics	Source of graph-datasets
	VP	EP	HP	GPS		GDBS		
				DS	SMS			
[59]	✗	✗	✗	✓	✗	✗	✗	✗
[60]	✗	✗	✗	✗	✓	✗	✗	✗
[61]	✗	✗	✗	✗	✓	✗	✗	✗
[62]	✗	✗	✗	✓	✗	✓	✗	✗
[63]	✓	✗	✗	✗	✗	✗	✗	✗
Ours	✓	✓	✓	✓	✓	✓	✓	✓

literature, outlining essential concepts and presenting recent research works that have not been included in prior overviews. This systematic survey paper aims to guide fellow researchers and practitioners to understand the concepts and evolution of large-scale graph partitioning and computing systems.

There exist several experimental and comprehensive studies on graph partitioning and computing systems. The experimental study of stream edge partitioning was performed in [64], [65], and [66]. The experimental analysis of both stream edge and vertex partitioning was studied in [67] and [68]. Pothen [69] discussed the traditional graph partitioning by grouping into three, geometric, algebraic, and multilevel. The evolutionary approach of graph partitioning was presented in [70]. The bipartile and hypergraph model for graph partitioning were surveyed in [71]. Arora et al. [72] discussed the relationship between geometric and flow-based graph partitioning. The traditional multilevel graph partitioning has three main phases, coarsening, partitioning and uncoarsening. These various coarsening phase algorithms were discussed and compared in [73]. Schloege et al. [74] reviewed static vertex partitioning for scientific simulations on high performance parallel computers. An empirical study of RDF (Resource Descriptor Framework) graph partitioning techniques and benchmarks were discussed in [75] and [76]. The empirical evaluation of GPS was analyzed in [59], [77], [78], and [79]. Tran et al. [60] reviewed GPU based large-scale GPS. Authors in [61] discussed the essential features and challenges of multi-core and out-of-core large-scale GPS. The participants' awareness for the usage of GPS and their challenges were conducted in [80]. Gui et al. [81] reviewed the key core graph processing accelerators, preprocessing, parallel graph computation, and run-time scheduling. The experimental evaluation of the graph databases was performed in [82] and [83]. As described in Table 1, there are a limited number of comprehensive works on modern graph partitioning and computing systems. This survey investigates, classifies, and reviews graph partitioning and computing systems. The main contributions of this work are summarized as follows:

- Optimization problems of graph partitioning, graph partitioning methods, approaches, and algorithms are reviewed and discussed. First, we classify the graph partitioning methods into three: vertex partitioning, edge partitioning, and hybrid partitioning. These graph partitioning methods can be further categorized as offline, online, offStream, and dynamic approaches. Then, the representative graph partitioning algorithms in each approach are listed and discussed.
- We discuss the major computational models of graph computing systems. These computational models of graph computing systems can be categorized into two: the computational models of GPS and GDBS. The GPS computational models, including programming, communication, and execution models, are discussed. Also, the data model, partitioning techniques, and query language of GDBS are described.
- We provide a detailed review of the graph computing systems and classify them into GPS and GDBS based on their graph analytics nature. These systems are further classified into several subcategories based on their architecture. For each subcategory, various systems with detailed computational models are listed and discussed.
- Challenges and future research directions in graph partitioning and computing systems are highlighted.

The rest of this paper is organized as follows. Section II explains the basic concepts of graph algorithms, partitioning, and computing systems. Section III describes types of graph partitioning, and Section IV discusses the computational models of graph computing systems. The taxonomy of graph computing systems is presented in Section V. The future challenges and research directions are indicated in Section VI. Finally, the conclusion is summarized in Section VII.

II. CONCEPTS OF GRAPH ALGORITHMS, PARTITIONING AND COMPUTING SYSTEMS

A. GRAPH ALGORITHMS

Graph algorithms are used to solve various real-world problems. These algorithms can be classified into the random walk, graph traversal, and graph aggregation. They are the primary benchmark for testing the performance of graph computing systems. These algorithms can be implemented in various ways based on the principles of the programming model of the graph computing systems [19], [23], [84].

1) RANDOM WALK

A random walk is a techniques that starts at one vertex, selects a neighbor to traverse at random or based on a probability

distribution, and then repeats the process from that vertex, saving the resulting path in a list [85]. PageRank [1], HITS [86], and ObjectRank [87] are examples of random walks. PageRank is the most common algorithm that can be used to check the performance of graph computing systems. PageRank is an iterative vertex ranking algorithm that weights vertices based on their relevance and connectedness to other well-ranked vertices. It starts by assigning a uniform rank to all vertices. After that, in each iteration, a vertex changes its rank by the new rank, then spreads evenly to outgoing neighbors along outgoing edges. When the difference between the vertex rank from the current iteration and the previous iteration is less than a defined threshold, the algorithm converges by adding the partial ranks of its arriving neighbor vertices.

2) GRAPH TRAVERSAL

Graph traversal entails visiting all of a graph's vertices in a specific order while checking and updating the vertices' values. Connected Components [88], Single Source Shortest Path [16], Approximate Diameter [89], Triangle Counting [90], and Bipartite Matching [91] are examples of graph traversal algorithms. These algorithms frequently use graph search. Connected Components finds subgraphs in which each vertex can be reached from every other vertex. Single Source Shortest Path calculates the shortest path from the source vertex to all associated vertices. At the start, it assigns a zero value to the source vertex and infinity to all other vertices. Then, each vertex changes its path length to the source until it does not observe a new update value across two consecutive iterations. Approximate Diameter uses probabilistic counting to estimate an approximation of a graph's diameter, which is the longest and shortest path between each pair of vertices. Triangle Count calculates the number of triangles in each vertex in graph. A triangle is made up of three vertices joined by three edges. It is utilized to detect communities and measure the cohesiveness of those communities. Bipartite matching takes two distinct sets of vertices as input, with edges solely connecting them, and returns a subset of edges with no common endpoints as output.

3) GRAPH AGGREGATION

Graph aggregation condenses the graph into a structurally identical but smaller graph by crumpling edges and vertices. Graph sparsification [92], Graph summarization [93], and graph coarsening [94] are some of the most common types of graph aggregation. Graph sparsification approximates a given graph to a sparse graph with fewer edges but the same number of vertices. Graph summarization represents the input graph into a smaller graph by keeping structural patterns. It facilitates the identification of structural and informative summaries of the input graph. Graph coarsening reduces the number of vertices of a graph by contracting disjoint sets of connected vertices. It is frequently used as an initial step in a graph partitioning algorithms.

B. GRAPH PARTITIONING PROBLEM

To easily understand graph partitioning problems, let's define a graph a bit more formally. A given undirected graph G is defined as $G = (V, E)$, where $V = \{v_1, \ldots, v_n\}$ and $E = \{e_1, \ldots, e_m\}$ are a group of vertices and edges, respectively. $E \subseteq V \times V$, the size of V and E are denoted as n and m, respectively. The undirected graph can be classified as weighted or unweighted. If a graph is a weighted graph, $e \in E$ can have a positive weight associated with them. On the other hand, if a graph is an unweighted graph, there is no weight associated with edges. However, it is possible to interpret the unweighted graph as a weighted graph in which each edge has a weight of 1. Graph partitioning can be classified as vertex, edge, and hybrid partitioning.

1) VERTEX PARTITIONING

Vertex partitioning (VP) is also called edge-cut, as depicted in Fig. 2. It divides the big graph into many subgraphs by assigning vertices to the different partition sets while minimizing edge cuts concerning load balance constraint. Let V_1 and V_2 be two vertex sets of the graph G. An edge-cut is defined as an edge$(u, v) \in E$, if and only if $\forall u, v \in V, u \in V_1$ and $v \in V_2$. Balanced $k-$way VP problem is defined as G is partitioned into k partitions set $\{V_1, V_2, \ldots, V_k\}$ such that $\bigcup_{i=1}^{k} V_i = V$. The vertex set of each partition is not duplicated, i.e, $V_i \cap V_j = \emptyset$, where $(i, j \in \{1, 2, \ldots k\}, i \neq j)$. The objective of VP is finding a k-partition set that minimizes the cost of all external edges (weighted or unweighted) connecting two partition vertex sets V_i and $\overline{V_i} = V - V_i$ with respect to a balance constraint. The edges-cut $\Gamma(V_i, \overline{V_i})$ between two partition vertex sets V_i and $\overline{V_i}$ is calculated as follows:

$$\Gamma(V_i, \overline{V_i}) = \sum_{(v_i, v_j) \in e, v_i \in V_i, v_j \in \overline{V_i}} \omega(v_i, v_j), \tag{1}$$

where $\omega(v_i, v_j)$ is the weight of the edges (v_i, v_j). The overall cost of the edge cut k-partitions $\Gamma(P_k)$ is expressed as:

$$\Gamma(P_k) = \sum_{i \in k} \Gamma(V_i, \overline{V_i}). \tag{2}$$

Therefore, the optimization problem of VP is given by:

$$\begin{aligned} \min \quad & \Gamma(P_k) \\ \text{s.t.} \quad & \max_{i \in k} |V_i| \leq (1 + \epsilon) \frac{n}{|k|}, \end{aligned} \tag{3}$$

where $|V_i|$ and $|k|$ are the size of the vertex set of the partition and the number of partitions, respectively. And $\epsilon \geq 0$ is an imbalance factor.

The k-way vertex partitioning problem can commonly be extended to graphs that contain weights associated with the edges [95]. This scenario aims to divide the vertices into k disjoint subsets where the sum of the edge weights whose incident vertices belong to different subsets is minimized. The basic implementation of distributed graph processing systems usually needs the solution of graph partitioning, where vertices represent computational tasks and edges consider data

exchange. Therefore, graph partitioning significantly impacts these systems' workload balance and communication costs. In VP, computing nodes (machines) that hold the partition set preserve local replicas of the vertices and edge data for the cut edges. These cut edges can act as a bridge to communicate with other machines. The machines' communication and workload costs are determined by the number of edge cuts and load balance.

2) EDGE PARTITIONING

Edge partitioning (EP) is also named vertex-cut, as shown in Fig. 2b. It divides a big graph into many subgraphs by assigning edges to the different partition sets while considering a maximum load balance and minimum vertex cut. Let E_1 and E_2 be two edge sets of the graph G. A vertex-cut is defined as a vertex $u \in V$, if and only if $u \in E_1$ and $u \in E_2$. A balanced k-way EP problem is defined as G is partitioned into k partitions $\{E_1, E_2, \ldots, E_k\}$ such that $\bigcup_{i=1}^{k} E_i = E$. The edge set of each partition is not duplicated, i.e, $E_i \cap E_j = \emptyset$, where $(i, j \in \{1, 2, \ldots k\}, i \neq j)$. Let $P(v)$ be the set of partitions that each vertex $v \in V$ is replicated. The replication factor (RF) is calculated as the summation of the number of replicas (copied versions of vertices) divided by the number of vertices:

$$RF = \frac{1}{n} \sum_{i \in k, v \in V} |P_i(v)|. \tag{4}$$

Therefore, the optimization problem of $k-$way EP is expressed as:

$$\begin{aligned} &\min \quad RF \\ &\text{s.t.} \quad \max_{i \in k} |E_i| \leq (1+\epsilon)\frac{m}{|k|}, \end{aligned} \tag{5}$$

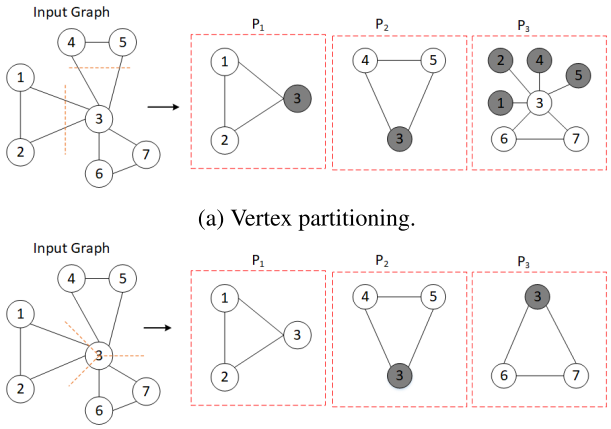
where $|E_i|$ is the size of the edge set.

In the case of distributed and parallel computation with edge partitioning, all machines holding cut vertices should preserve a mirror (local replica) of the vertex. These mirror vertices can act as a bridge communicator between the partitions. The number of mirror vertices and edges determines the communication and workload costs, respectively.

3) HYBRID PARTITIONING

The EP evenly allocates edges to machines and only replicates vertices to construct a local graph within each partition. Therefore, the EP mainly focuses on minimizing the overall RF. However, Hybrid partitioning (HP) considers that instead of reducing RF of all vertices, it distinguishes vertices as a lower and higher degree. Then, VP or EP is applied for better cuts.

HP is a hybrid of VP and EP methods. It exploits the interior structure of the graph to perform partitioning [22]. Most of the real-world graphs are power-law graphs, where a relatively small percentage of vertices have a higher degree, and most vertices have a lower degree [23]. HP differentiates the vertices as low-degree and high-degree. Then, it evenly



(a) Vertex partitioning.

(b) Edge partitioning.

FIGURE 2. Vertex vs Edge partitioning: (a) Input Graph is partitioned into three partitions P_1, P_2, and P_3 by cutting four edges; (b) Input Graph is partitioned into three partitions P_1, P_2, and P_3 by cutting one vertex, and the shaded circle vertices are replicas.

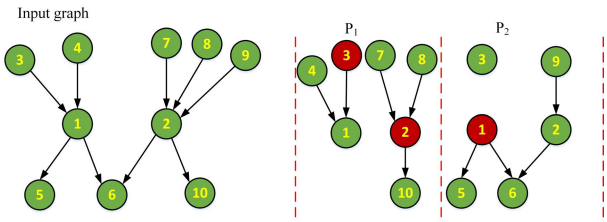


FIGURE 3. Hybrid partitioning: The red colored circle of vertices are mirrors and others vertices are masters.

distributes the edges of a high-degree vertex among partitions (using vertex-cut) to disseminate the computation load and allocates all the in-edges (or out-edges) of a low-degree vertex to the same partition (using edge-cut) to reduce communication among partitions.

For example, consider the input graph in Fig. 3, how to apply HP using Hashing [22]. Suppose that degree threshold is 3. Therefore, if a vertex in-degree is ≥ 3, it is considered to be high-degree vertex. As shown in Fig. 3, the vertex 2 is high-degree and all the other vertices are low-degree. Assume that vertices 1, 4, 7, 8 and 10 are hashed to p_1 and vertices 2, 3, 5, 6, and 9 are hashed to p_2. Then, the in-edges of vertex 2, namely, (7, 2), (8, 2), and (9, 2) will be assigned with their source vertices (source hashing). Therefore, the edges (7, 2) and (8, 2) are assigned to p_1 and the edge (9, 2) is assigned to p_2. Then, the in-edges of other vertices, namely, (3, 1), (4, 1), (1, 5), (1, 6), (2, 6), and (2, 10) will be assigned with their target vertices (target hashing). Therefore, based on their target vertices, edges (3, 1), (4, 1) and (2, 10) are assigned to p_1 and (1, 5), (1, 6) and (2, 6) are assigned to p_2. The partitioned result is depicted in Fig. 3 in p_1 and p_2.

C. GRAPH COMPUTING SYSTEMS

Recently, there has been an increase in the demand for large-scale graph computing systems. Because graphs can

describe a diverse set of objects, the computations performed on graph-based data structures are at the heart of many applications, such as machine learning, data mining, and pattern recognition. The requirement to process large graphs has led to the development of various frameworks that can handle the processing of large graphs in different computing architectures. Graph computing systems, also known as graph analytic systems, process graph-based computation. Existing graph computing systems can be classified into two; GPS [20], [21], [29] and GDBS [25], [26], [27], [28]. The GPS, known as offline graph analytics systems, process an iterative computation on the whole graph until a convergence criterion is satisfied. The GDBS, also called online graph analytics systems, perform analysis on subgraphs or entire graphs and require a fast response time.

D. PERFORMANCE EVALUATION METRICS

1) METRICS OF GRAPH PARTITIONING

Load balance, locality (the number of cut vertices or edges), run-time, and scalability [47], [96] are used to measure the performance of the graph partitioning. Among these metrics, partitioning quality is measured by the number of cut vertices or edges and load balance.

a: LOAD BALANCE (ρ)

It indicates that how well the number of vertices or edges is distributed across partitions. For vertex and edge partitioning methods, the two metrics are calculated differently. The ρ is calculated as:

$$\rho = \frac{\max\limits_{i=1,\ldots,N} |P_i|}{\frac{\psi}{N}}, \tag{6}$$

where ψ is the input size (the number of vertices for vertex partitioning or the number of edges for edge partitioning) and $|P_i|$ is the size of vertices for VP or the size of edges for EP in each partition. Partitions with a good load balance reduce processing latency and enhance the resource utilization of distributed graph computing.

b: LOCALITY

The fraction of edges cut (τ) from balanced constraint vertex partitioning can be calculated as:

$$\tau = \frac{\sum_{i=1}^{k} \Gamma(V_i, \overline{V_i})}{m}. \tag{7}$$

However, other versions of the vertex partitioning problem do not have a fixed balance constraint but encode balance directly in the objective function. Conductance [97], ratio cut [98], and normalized cut [99] are used to measure non balanced constraint vertex partitioning. The conductance of a set of vertices $\Phi(V_k)$ can be expressed as:

$$\Phi(V_k) = \sum_{i=1}^{k} \frac{\Gamma(V_i, \overline{V_i})}{min(vol(V_i), vol(\overline{V_i}))}. \tag{8}$$

The ratio cut of a set of vertices $\Delta(V_k)$ can be expressed as:

$$\Delta(V_k) = \sum_{i=1}^{k} \frac{\Gamma(V_i, \overline{V_i})}{|V_i|}. \tag{9}$$

The normalized cut of a set vertices $\Theta(V_k)$ can be defined as:

$$\Theta(V_k) = \sum_{i=1}^{k} \frac{\Gamma(V_i, \overline{V_i})}{vol(V_i, V)}, \tag{10}$$

where $vol(V_i, V) = \sum_{u_i \in V_i, v_j \in V} w(u_i, v_j)$ is total degree of the vertices V_i in a graph G. The lower value of $\Phi(V_k)$, $\Delta(V_k)$, and $\Theta(V_k)$ indicate that the vertices set are in a good cluster. The balanced constraint vertex partition is more applicable to graph computing systems due to the equal distribution of edges or vertices to computing nodes. However, the non-balanced constraint vertex partition metrics are more applicable to graph clustering [100].

For edge partitioning, the number of cut vertices are called replicas. It is measured by a replication factor (σ). The σ is calculated as:

$$\sigma = \frac{1}{n} \sum_{i \in k} |P_i(v)|, \tag{11}$$

where $P_i(v)$ is the total number of replicas of vertices in each partition. A good partitioner must minimize the value of σ and τ. The number of cut vertices indicates the external communication overhead between different computing machines because communication in such systems coexists with vertices.

c: RUN-TIME

It indicates the elapsed time to partition the graph. The run-time includes ingress (loading the input graph to the memory) and partitioning time of the graph.

2) METRICS OF GRAPH COMPUTING SYSTEMS

The following metrics are used to check the performance of graph computing systems.

a: TOTAL-TIME

It is a time that requires the overall running time from the beginning to the end of graph computation. It can be divided into preprocessing and computation time. The preprocessing time is the time to load the input graph into memory, partition it, and write the output. The computation time is how long it takes to perform barrier local synchronization, vertex computation, and communication.

b: COMMUNICATION COSt

It is the sum of per-machine network usage across all worker machines, with total sent (outgoing) and total incoming (received) network usage. It is influenced by the amount and distribution of data transmitted across servers.

TABLE 2. List of abbreviations.

Abbreviations	Meaning
VP	Vertex partitioning
EP	Edge partitioning
HP	Hybrid partitioning
Sync	Synchronous execution model
Async	Asynchronous execution model
Hsync	Hybrid execution model
PGM	Property graph model
RDF	Resource descriptor framework
GPS	Graph processing systems
GDBS	Graph database systems
MR	MapReduce programming model
VC	Vertex centric porgamming model
GAS	Gather-Apply-Scatter programming model
SC	Subgraph centric programming model
MP	Message passing model
SM	Shared memory model
DF	Dataflow model
SMSM	Single-machine shared memory system
SMOC	Single machine-out-of-core system
DS	Distributed system
OSSMVP	Offline sequential single machine vertex partitioning
OSMSMVP	Offline shared memory single machine vertex partitioning
ODSVP	Offline distributed vertex partitioning
OSMEP	Offline single machine edge partitioning
ODSEP	Offline distributed edge partitioning

c: MEMORY USAGE

It is the total of memory allocations for computing tasks. The memory footprint of each server must be kept to a minimum. This ensures that fewer servers may be utilized for processing large-scale graphs, which is useful when resources are constrained.

d: SCALABILITY

It measures a system's capacity to adjust its performance and cost in response to shifting application and system processing requirements. Thus, a large graph must be loaded and processed by smaller clusters. Communication and computing must become cheaper as the cluster grows, and the overall job must run faster. In the same way, graph partitioning algorithms also give strict guarantees about locality and balance while also scaling to large graphs. Thus, providing such assurances necessitates costly coordination or global views of the graph. This limits scalability [96].

E. GRAPH DATASETS

The generated synthetic graphs of varying sizes and real-world datasets are the main benchmarks for testing the performance of graph partitioning and computing systems. RMAT (Recursive Matrix) [101] is used to generate a synthetic skewed degree distribution graph. The main sources of real-world graph datasets repository are found in SNAP (Stanford Network Analysis Project) [102], Online Network Research Web Portal [103], KONECT (Koblenz Network Collection) [104], LAW (Laboratory for Web Algorithmics) [31], Twitter [105], Friendster [106] and MovieLens 10M datasets [107]. The SNAP[1] data set repository was founded in 2004 as a result of a study into the analysis of significant information and social networks. These datasets on the website were primarily collected for the objectives of the research works in July 2009. The KONECT[2] is a project that aims to collect massive network data sets to aid network mining research. The collection's website also includes statistics, charts, and code for generating all network data sets from the internet. The LAW[3] was founded in 2002 at the University of Milan Department of Information Sciences and has since integrated with the Computer Science Department. The research at LAW focuses on all algorithmic aspects of web and social network researches.

III. TYPES OF GRAPH PARTITIONING

Based on how the input graph is processed, the graph partitioning method can be further classified into four approaches, offline, online (stream), offStream, and dynamic, as depicted in Fig. 4. How well the graph partitioning can be scaled is based on how the input graph is accessed.

A. OFFLINE APPROACH

Offline approach is a traditional graph partitioning approach that exploits the graph's global information to allocate edges or vertices to the partitions. The graph is loaded into memory before it applies the partitioning algorithms. In this approach, many algorithms have been proposed via single and distributed machines. Offline single machine partitioning uses a single machine to perform its partitioning and has a high partitioning accuracy; however, it can not support large-scale graph partitioning due to a lack of memory that can accommodate the entirety of the graph [41]. The two main challenges of graph partitioning are quality and scalability. First, high-quality partitioning is evaluated by total cuts and load balance. However, it is difficult to obtain since graph partitioning is proved to be an NP-hard problem. Second, graph partitioning is required to scale up and deal with large graphs since the size of real-world graphs has been increasing quickly. Therefore, distributed memory graph partitioning has been proposed to support scalability with compromised quality partitioning. In the distributed approach, the graph is already distributed in a distributed memory application. However, to preserve scalability, not every processor stores the whole graph. As a result, distributed-memory partitioning algorithms frequently rely on their partitioning choices on partial views of local graph data rather than having an overall view of the entire graph. Each processor communicates with the other to minimize the cut and maximize load balance. The distributed approach supports large-scale partitioning; however, its partitioning quality is less than the single machine approach. In this approach, offline sequential single machine vertex partitioning (OSSMVP), offline shared memory single machine vertex partitioning (OSMSMVP), offline distributed vertex partitioning (ODVP), Offline single machine edge partitioning (OSMEP) and Offline distributed edge partitioning (ODEP) have been proposed.

[1] https://snap.stanford.edu/

[2] http://konect.cc/

[3] http://law.di.unimi.it/

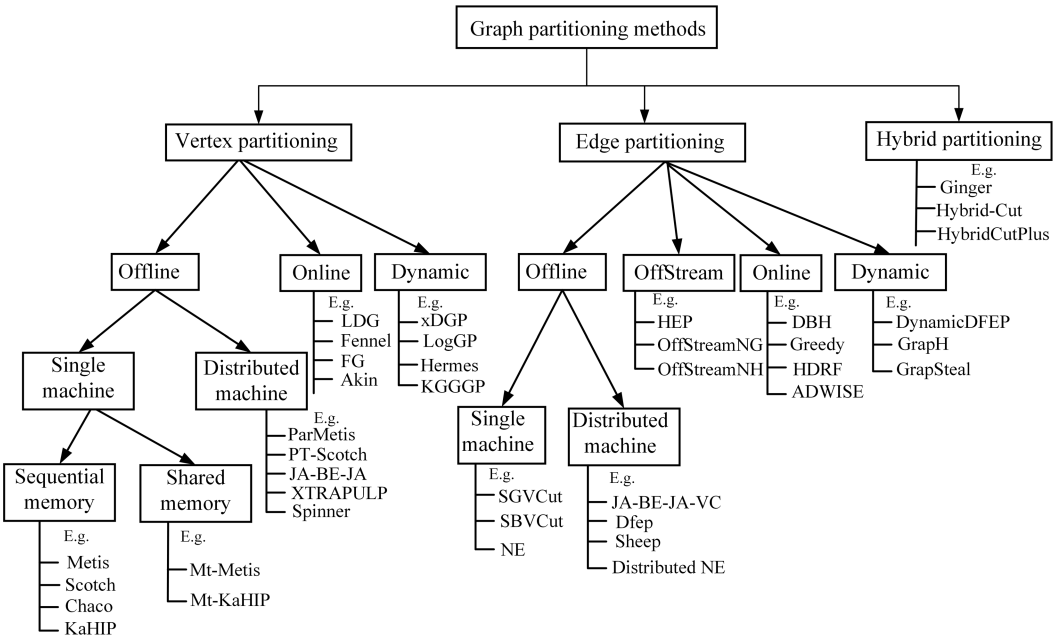


FIGURE 4. Graph partitioning methods, approaches and algorithms.

1) OFFLINE SEQUENTIAL SINGLE MACHINE VERTEX PARTITIONING

Initially, the input graph is loaded into a single machine; then, various iterative techniques are applied to improve the partitioning quality. Most algorithms were proposed based on the multilevel partitioning model. The multilevel graph partitioning model [108], [109] is the most successful heuristic for partitioning a graph. It consists of three phases: coarsening, initial partitioning, and refinement (uncoarsening) as depicted in Fig. 5. During the graph coarsening phase, a sequence of graphs $G_1, G_2, \ldots G_m$ are created by compressing selected vertices of the input graph into a related coarser graph. This newly built graph is then used as the input graph for another round of graph coarsening until the graph is small enough. Coarsening phase is often accomplished by computing matching algorithms [95], [108], [110]. During initial partitioning phase, a partition P_i of the much smaller graph G_i is created using spectral bisection or graph growing heuristic [108]. Local search approach KL [111] and FM [112] are frequently used for refinement phase. KL [111] is the pioneer offline vertex partitioner. To partition the graph, initially, vertices are randomly assigned to one of the partitions; then, it tries to improve partitioning efficiency by evaluating the cut-vertex function's gain, if necessary, exchanging the vertices between partitions. This process is continued until there are no possible exchanges that optimize the final partition's cut vertices. FM [112] begins by calculating the gain values for each vertex, where gain refers to the difference in edge cut if a vertex was shifted to the other partition. The algorithm works in rounds, with a subset of vertices being shifted from one partition to the other in each round. The vertex with the highest gain value is chosen to be moved.

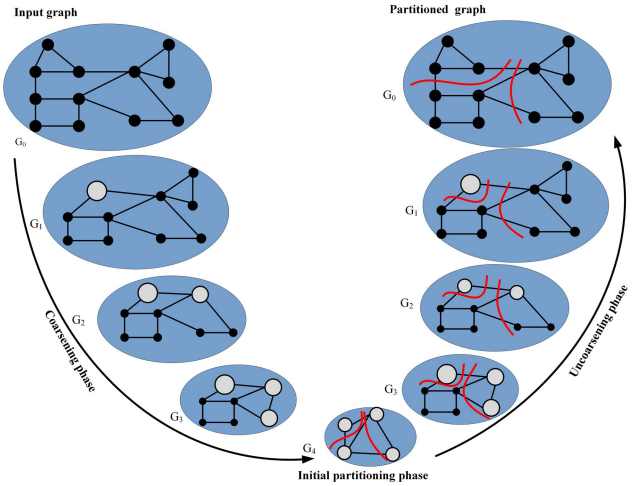


FIGURE 5. Multilevel graph partitioning. The gray-colored vertices are formed by applying a coarsening phase, which contains groups of vertices. After initial partitioning is done, uncoarsening is performed to get a partitioned graph.

Hence, its neighbors' gain values are updated appropriately, and the procedure is repeated with the remaining unmoved vertices until all vertices have been moved precisely once. Metis [41], Scotch [113], Chaco [114], and KaHIP [115] are examples of well-known OSSMVP software packages.

2) OFFLINE SHARED MEMORY SINGLE MACHINE VERTEX PARTITIONING

Recently, the number of cores per chip has increased dramatically. As a result, offline shared-memory single machine vertex partitioning efficiently utilizing available computer cores are highly demanded. Mt-Metis [116] and

Mt-KaHIP [117] have been proposed in this category. Mt-Metis is a multi-threaded implementation of the Metis algorithms by avoiding message passing overhead and modifying existing parallel algorithms implemented in ParMetis. The Mt-Metis has less memory overhead than either PT-Scotch or ParMetis. Because Mt-Metis stores information for each vertex just once, PT-Scotch and ParMetis need to communicate and store the information of remote neighbor vertices. Mt-KaHIP is a multilevel SM partitioning that adopts KaHIP. It uses label propagation for coarsening and refinement and a cache-aware hash table to limit memory consumption and enhance locality. Mt-KaHIP has better partitioning quality and less memory overhead than Mt-Metis. However, Mt-Metis is faster than Mt-KaHIP [116].

3) OFFLINE DISTRIBUTED VERTEX PARTITIONING

The input graph is loaded into different machines, then various optimization techniques are applied to improve the partitioning quality. Most of the distributed partitioning apply the label propagation method [118]. This method assigns k labels to represent partitions. First, each vertex chooses a random label and sends its label to neighbors. Then, each vertex ranks the labels based on neighbors' labels, choosing the label with the highest rank for itself, and sending it to its neighbors again. These steps are iterated until the label of vertices ceases modifying and the algorithm converges. ParMETIS [119], PT-Scotch [120], KaPPa [121], JOSTLE [122], JA-BE-JA [123], Blp [124], BS [125], XTRAPULP [126], and Spinner [96] are examples of ODVP.

ParMETIS [119] is MPI-based parallel partitioning that implements several methods for partitioning unstructured graphs and computing sparse matrices fill-reducing orderings. It adopts the popular multilevel partitioning METIS [41] by including routines explicitly designed for parallel computations and large-scale numerical simulations. PT-Scotch [120] extends Scotch to parallelize the nested dissection method to compute efficient ordering of very large graphs. Unlike ParMETIS, PT-Scotch does not have a limit on the number of processors. PT-Scotch outperforms ParMETIS in terms of graph ordering quality. KaPPa is a parallel match-based multilevel graph partitioning. It uses either Scotch or pMetis [133] for initial partitioning and FM for refinement. JOSTLE [122] uses the MPI and single program multiple data paradigms to parallelize multilevel graph partitioning by enhancing multiphase mesh partitioning, heterogeneous mapping, and partitioning to improve subdomain shape. ParHIP [127] adopts the label propagation clustering algorithm for multilevel graph partitioning phases of coarsening and refinement. First, it computes the cluster of a graph via size-constrained label propagation. The clustering is shrunk by replacing each cluster with a single node, and the process is continued recursively until the graph is small enough to compute a graph hierarchy. Then it uses a coarse-grained distributed memory parallel evolutionary algorithm to perform partitioning. ParHIP has achieved a higher partitioning quality and scalable than either

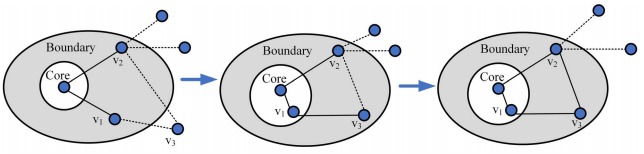


FIGURE 6. Edge partitioning by Expansion. The broken line edges are unallocated, and the solid line edges are allocated. Initially, vertices v_1 and v_2 are in boundary sets. Therefore, v_1 is selected to be included in a core set because v_1 has fewer external neighbors than v_2. Then, edge allocation is performed. This step is continued until all edges are allocated.

ParMetis or PT-Scotch. However, multilevel-based partitions can only scale to a few hundred processors [134]. JA-BE-JA [123] considers a partial view of the graph information and uses Simulated Annealing optimization techniques to avoid becoming terminated in local optima. Each vertex is a processing unit, contains information of its neighboring vertices and a few subsets of random vertices. Initially, every vertex chooses a random partition. Through time, vertices swap their partition to improve a locality value based on the number of neighbors they have in the same partition. Blp [124] partitions large-scale graphs based on label propagation by maximizing edge locality, the total of edges that are allocated to a similar shard of the partition. BS [125] uses a scatter-gather local search strategy, the simulated annealing techniques, and the Bulk Synchronous Parallel computation model. XTRAPULP [126] extends PULP [135] which is multiple objective and constraint partitioning based on label propagation to improve partitioning quality with minimal computational time. Spinner [96] exploits label propagation algorithm (LPA) and vertex-centric programming model. It executes on top of Giraph and exploits a recursive node migration approach using LPA to deal with scalability and changing partitions. Comparison of offline approach graph partitioning is described in Table 3.

4) OFFLINE SINGLE MACHINE EDGE PARTITIONING

Initially, the input graph is loaded into single machine memory. Then, the partitioners get complete information of the graph and evenly assign edges to the partition via structure-aware of vertices relationship. Offline single machine edge partitioning include, SBVCut [136], SGVCut [128], and NE [49]. SBVCut [136] works to get a structurally balanced cut. First, it identifies a set of balanced vertices that can be exploited effectively bisect a direct graph. The graph is then further divided by an iterative application of structurally balanced cut to get the graph's hierarchical partitioning. SGVCut [128] performs a workload-aware block-based partitioning strategy. First, it groups edges into blocks based on their connectivity scores to different predefined seeds. Next, if the blocks are too large, it splits the blocks by considering connectivity values. Finally, it merges all these blocks into balanced partitions.

NE [49] is the state-of-the-art edge partitioning algorithm which partitions based on neighborhood expansion

TABLE 3. Summary of offline approach graph partitioning.

Algorithms	Partitioning methods	Years	No.of Times Cited (as of 20/09/2021)	Programming models	Strategies	Multi Constriant	Scalability
Mt-Metis [116]	OSMSMVP	2013	196	Shared Multilevel	Matching	No	Not scalable
Mt-KaHIP [117]	OSMSMVP	2020	32	Shared Multilevel	Parallel label propagation	No	Not scalable
ParMetis [119]	ODVP	1997	523	Distributed multilevel	Matching	No	Limited
PT-Scotch [120]	ODVP	2008	496	Distributed multilevel	Nested dissection	No	Limited
JA-BE-JA [123]	ODVP	2013	120	VC	Label Exchange	Yes	Scalable
Blp [124]	ODVP	2013	219	VC	Label propagation	Yes	Limited
XTRAPULP [126]	ODVP	2017	63	VC	Label propagation	Yes	Scalable
ParHIP [127]	ODVP	2017	146	Multilevel	Label propagation	No	Limited
Spinner [96]	ODVP	2017	97	VC	Label propagation	No	Scalable
SGVcut [128]	OSMEP	2017	4	Random walks	Local access pattern	No	Not scalable
NE [49]	OSMEP	2017	42	Neighborhood Expansion	Expansion Boundary vertices	No	Not scalable
JA-BE-JA-VC [129]	ODEP	2014	58	VC	Label Exchange	Yes	Scalable
Dfep [130]	ODEP	2015	22	VC	Currency distribution	No	Limited
Sheep [131]	ODEP	2015	61	MapReduce	Elimination tree	No	Scalable
Distributed NE [132]	ODEP	2019	24	Neighborhood Expansion	Parallel Expansion	No	Scalable

heuristics with two stages, edge expansion and edge allocation as depicted in Fig. 6. First, one edge set is generated from the given graph then that edge set is allocated to the partitions during the edge allocation stage. In NE algorithm, partitioning is performed in iterative manner. To build partition k_i, first, NE establishes the core C and boundary B sets. The B begins to expand, and then the relevant vertices are selected as participants in C. A seed vertex is chosen before the expansion. The seed vertices are placed in C. All neighboring vertices of each seed vertex in k_i are placed in a boundary set B_i. Edges that link vertices between or within C and B_i are assigned to the current partition k_i. In the expansion step, the vertex form B_i with the external degree d_{ext} and the fewest neighbors who are neither in B_i nor in C is chosen. Then, the vertex was relocated from B_i to C, and the external degree for each vertex v in B_i was calculated. Finally, NE allocates edges between v and vertices in B_i and C to the current partition k_i and removes the edges from the graph. The vertex in B_i with the lowest d_{ext} is then determined and moved to C using the following expansion phase. The remaining edges of a partition will overflow into the next partition if the partition reaches its capacity limit. When the partition is complete, all of the edges in the graph will be eliminated, and the algorithm will begin again at the seed vertex. The method comes to a halt once the entire graph has been partitioned.

5) OFFLINE DISTRIBUTED EDGE PARTITIONING

All edges of a graph are resigned in different machines and it employ global placement heuristics to optimize edge allocation. Sheep [131], JA-BE-JA-VC [129], Dfep [130], dSPAC+X [137], and DNE [132] are examples of offline distributed edge partitioning. Sheep [131] converts the graph near to a smaller elimination tree using a distributed MapReduce operation. It sorts the vertices, reduces the input graph into an elimination tree, and partitions the elimination tree. Finally, it translates the partitioned tree into edge partition. JA-BE-JA-VC [129] randomly assigns the edges to the partitions and applies edge coloring. Then, vertices perform edge-color exchange to reduce the vertex cut. It uses simulated annealing to improve the partitioning quality iteratively. dSPAC+X is a scalable distributed edge partitioning via split and connect graph construction method. First, the input graph G is changed to a hypergraph (Hg) via the split and connect method, and then the Hg is partitioned via vertex partitioning. dSPAC+X partitions billions of edges by integrating parallel vertex partitionings like ParMETIS [119] and ParHIP [127]. DNE [132] is a distributed version of NE [49] and introduces a parallel expansion heuristic. It divides edges into disjoint sets and minimizes the number of replicated vertices. Dfep [130] assigns random vertices and an equal amount of funds to each partition. In each round, each partition makes an offer to obtain an edge based on its neighbors vertices.

B. ONLINE APPROACH

The offline approach loads the complete graph in memory before it begins partitioning. This loaded graph in memory helps it quickly gather the global graph structure to solve the optimization problem. Thus, it has a higher partitioning quality. However, it does not support large-scale graph partitioning. This issue motivated the design of an online approach to scalable graph partitioning. The online approach is also known as stream approach. The vertices with edge sets arrive in a pipeline fashion to a partitioner as shown in Fig. 7. The online approach performs partitioning based on partial view graph data and needs to save a partitioned state for further decisions. This state is crucial for the online partitioners to assign the incoming edges to the appropriate partitions. However, once edges or vertices are allocated, they will never be reassigned again. Because the edges does not need to be retained in memory entirely at any time, the online approach allows graphs to be partitioned with minimum memory overhead. Therefore, lower capacity workstations can be utilized to partition massive graphs, which reduces the monetary expense of graph partitioning. However, in the beginning, the online approach does not have enough partition state to allocate the incoming edges, but over time, it accumulates the partition state. Early edges or vertices

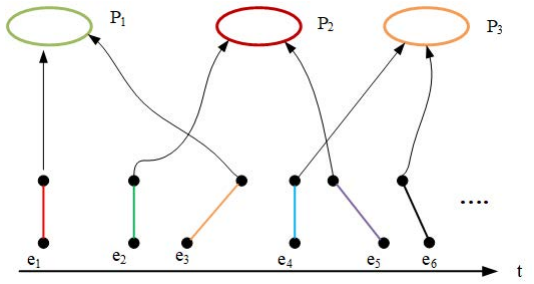


FIGURE 7. Vertices with edge sets arrive online (one edge at a time), and then target partition is determined after each edge arrives.

in the stream are allocated to partitions with little partitioning state available, leading to poor quality of such allocations. Therefore, its partitioning quality is worse than the offline approach. However, it supports big graph partitioning. Furthermore, the graph data may reach the partitioner either in Random, DFS (Depth First Search), or BFS (Breadth First Search) orders. These arrival orders affect the performance of the partitioning methods [65].

1) ONLINE VERTEX PARTITIONING

When vertices with edge sets arrive in stream fashion, a partitioner chooses one of the partition to allocate the vertices. The aim of the partitioner is to discover a balanced partitioning that close to optimal as possible with as little computation. An example of online vertex partitioning includes Hashing, LDG [43], Fennel [138], FG [42], and Akin [139]. Hashing is used for both vertex or edge partitioning. It allocates edges or vertices to the partitions by mapping the hashing function to edges or vertices. LDG [43] assigns the incoming vertices to most of its neighborhood found and controls load balance by a penalty multiplier. Fennel [138] extends the idea of LDG to formulate graph partitioning problem as modularity maximization in streaming settings, and it relaxes hard cardinality constraints into an element that accounts for the cost of edges cut and the sizes of individual clusters. It assigns incoming vertices to the partition which holds the highest neighborhood and a minimum of none-neighborhood. FG [42] introduces a hybrid streaming mode that considers partial restreaming on the graph's portion several times, applying one pass for the rest of the portion. Akin [139] performs stream vertex partitioning by allowing the migration of vertices between partitions over time. It uses the Jaccard similarity measure to determine which vertices are similar and puts them in the same partition as possible. It constructs a fixed neighbor list sorted by the degree to access every vertex easily. It takes the stream of edges and vertices as input. Vertices are assigned by deterministic hashing during vertex stream, and edges are assigned by vertices' similarity during edge placement. During edge placement, it assigns an edge based on maximizing a similarity score via migration vertices of an edge to the partition. Nishimura and Ugander [140] proposed a restreaming partitioning model to extend existing online vertex partitioning. The restreaming partitioning model is driven by circumstances in which the same dataset is consistently streamed, allowing streaming partitioning algorithms to be transformed into an iterative approach. reFennel and reLDG are extended versions of Fennel and LDG, respectively, via the restreaming partitioning model. They retain linear memory bounds as single-pass online vertex partitioning and present comparable results with METIS. This model can also support parallelization without inter-stream communication.

2) ONLINE EDGE PARTITIONING

Each edge of the input graph is loaded one at a time, and as soon as it is loaded, it is assigned to a partition. The decision about where to put an edge is made by a scoring function that looks at graph properties, either degree, cluster information, or the state of the partition (information about where edges have already been allocated). Online edge partitioning algorithms have been proposed in a single-pass (e.g. DBH [48], Grid [46], PDS [45], Greedy [23], HDRF [47], CLDA [141] and Deter [142], Quasi-streaming [143]), window-based (e.g. ADWISE [144], RBSEP [145], and WSGP [146]), restreaming (e.g. 2PS-L [50], 2PS-HDRF [147], and CLUGP [148])

DBH [48] assigns the incoming edge based on vertices' degree. It compares the degree of the paired value of edge vertices and gives a hash value of the vertex with a smaller degree to the edge. Grid [46] organizes all the partition into a square matrix. This constraint set for any vertex v is the group of all the partitions in the row and column of the partition v hashes. The Grid works for only none prime numbers of partitions. PDS [45] uses a Perfect Difference Sets to generate a constraint set and applies for only prime numbers of partitions. Greedy [23] assigns the incoming edge by checking the previously allocated partition state and considering a minimum load balance among each partition. HDRF [47] (Higher Degree Replicated First) takes the Greedy heuristic advantage and adds a degree of vertices information to calculate the sore function. This degree information helps to partition a skewed power-law degree distribution. When it comes to replication factor, HDRF is better than competitor stream partitioning, even though it takes a little more memory. CLDA [141] is a hybrid of two edge partitioning techniques, Greedy and HDRF, and considers a lower degree edge assignment. The lower and higher degree edges are partitioned by Greedy and HDRF, respectively. It has the same replication factor with HDRF but achieves a better load balance than HDRF. Deter [142] extends the idea of HDRF by considering both degree and cluster information into account when assigns an edge to the partition. This cluster information helps to allocate high dense subgraph into the same partition to reduce the communication cost. Quasi-streaming [143] divides incoming edges into batches of a fixed size (a constant multiple of partitions) and assigns edges to partitions using a game theory model. All edges in each batch make up the players in a gaming process. The reasonable strategy in the game is the edge's partition selection. The edge partitioning for this batch is completed when the game process of each batch finds a Nash Equilibrium. Quasi-streaming reduces memory

overhead and achieves a lower replication factor than online single pass edge partitioning.

Window-based edge partitioning have been proposed to overcome the uninformed assignment problem of state-based single-pass online edge partitioning by storing some edges in a window to get more knowledge of the graph or postponing them in a buffer window. The buffered window helps to gather enough information about two end vertices of incoming edge to determine edge allocation. ADWISE [144] performs edge partitioning by storing and selecting the best edge among multiple edge lists in the buffered window. It controls window size at run-time and considers adaptive balance score, degree aware window score, and clustering score to calculate the score function. It determines the best edge from the buffered window via the high value of score function, assigns it to the best partition, and refills the window with edges from the edge stream. RBSEP [145] introduces a buffer window, postponing, and reassigning edges. If the incoming edge incident vertices neighborhood has not been visited yet, the edge will be stored in the buffer window and postponed edge allocation. Otherwise, the edge allocation is made using HDRF procedures. Later, edges stored in the buffer will be considered for reallocation. WSGP [146] adapts edge allocation from Greedy and delays the incoming edge, which does not fit to be assigned in the current iteration to a fixed-bounded buffered window. After the buffered window has been filled, the edge is popped and allocated to a partition. The assignment is decided by looking at the edges that have already been settled and the edges that are still in the buffer window. ADWISE, RBSEP, WSGP have a lower replication factor than HDRF, however, they have memory and run-time overhead.

Mayer et al. [147] proposed a two-phase stream edge partitioning model via streaming vertex clustering and restreaming partitioning. A lightweight streaming clustering technique [149] is used in the initial phase to begin separating vertices into clusters. In the second phase, the graph is re-streamed, and the vertex clustering that was done in the first phase is exploited to achieve a lower replication factor. The model checks that the edges are pre-partitioned via adjacent vertices in the same cluster or in the cluster mapped to the same partition during restreaming. If the conditions for pre-partition are satisfied, the edge is skipped because it has already been allocated. Otherwise, a score is performed to allocate the edges. Based on this model, 2PS-HDRF and 2PS-L are proposed. They used the same clustering algorithm in the first phase. However, they considered different scoring functions in the second phase. 2PS-HDRF exploits the same score function as HDRF. However, 2PS-L considers three things to calculate the score function: the degree of a vertex, the cluster of a vertex, and the volume of a vertex. Unlike the 2PS-HDRF, the 2PS-L calculates score functions for only two partitions to determine the highest score partition. They have a lower replication factor and a good runtime than HDRF. 2PS-HDRF outperforms 2PS-L in terms of replication factor. However, 2PS-L has a shorter runtime than 2PS-HDRF. CLUGP [148] is a restreaming edge partitioning that consisting of three phases: stream clustering, cluster partitioning, and partition transformation. The streaming clustering phase uses relationship between clustering and edge partitioning to generate fine-grained clusters to decrease the number of vertex replicas. In this phase, CLUGP improves the stream clustering [149] to fit for edge partition via a split operation (when a cluster's volume reaches its max, it splits higher degree vertices to generate a new cluster). The cluster partition phase converts the clusters to partitions by considering balancing and edge cutting as a cost function. This problem is solved using game theory. Finally, to get edge partitioning, it combines the output of the two phases to map vertex to partition in the partition transformation phase. CLUGP outperforms online single-pass edge partitioning in terms of replication factor and run-time in web graphs [148].

3) ONLINE HYBRID PARTITIONING

It targets reducing the cuts of low-degree vertices. First, it distinguishes low-degree and high-degree vertices. Then, it applies various techniques for the lower-degree and high-degree vertices to get optimal partitioning quality. Hybrid-Cut [22], Ginger [22], and HybridCutPlus [52] are examples of online hybrid partitioning. Hybrid-Cut differentiates the vertices as the lower and higher degree based on the user-defined threshold. Then, the vertex partitioning and edge partitioning are applied for the lower and higher degree vertices, respectively. The lower degree vertices are evenly assigned vertices along with in-edges to partition by hashing their target vertices. And for the higher degree vertices, it distributes all in-edges by hashing their source vertices. Ginger differentiates the lower and higher degree vertices similar to Hybrid-Cut. Then, the lower degree vertices are partitioned like Hybrid-Cut. However, for the higher-degree vertices, it employs a Fennel-like heuristic to allocate the vertex and its in-edges to the partition that minimizes the expected replication. Unlike Fennel [138], Ginger includes both the size of edges and vertices into its objective function. By distinguishing higher and lower vertices, HybridCutPus employs the Hybrid-Cut, and Grid [46] partitioners. It uses Hybrid-Cut, if one vertex of an edge is a higher degree and another vertex is a lower degree; otherwise, it performs similar to Grid partitioner. Table 4 describes the comparison of online partitioning.

C. OffStream APPOACH

OffStream partitioning approach was proposed by hybriding the offline and stream approaches. It Overcome the gap between pure in-memory and pure streaming algorithms. The main idea is that if a graph is too large to partition in memory, the algorithm instead reads only some input graph scale to memory, runs a good partitioning method for the offline and stream parts. OffStreamNG [53], OffStreamNH [54], and HEP [55] are examples of offstream edge partitioning. Initially, OffStreamNG and OffStreamNH randomly split edge set in two subsets; then, it applies online and stream edge partitioners for each subset. OffstreamNG uses NE [49] and

TABLE 4. Summary of online graph partitioning algorithms.

Algorithms	Partitioning Methods	Years	Cited (as of 20/09/2022)	Strategies	State	Time Complexity	Space Complexity
LDG [43]	VP	2012	466	Neighbors	Vertices and partitions	$O(kn+m)$	$O(n)$
Fennel [138]	VP	2014	346	Neighbor/Non neighbor	Vertices and partitions	$O(kn+m)$	$O(n)$
Akin [139]	VP	2018	18	Vertices similarity	Vertices, degree and partitions	-	-
FG [42]	VP	2014	10	Partial stream	Vertices and partitions	-	$O(n)$
Grid [46]	EP	2013	99	Hash	Vertices, and partitions	$O(m)$	$O(1)$
Greedy [23]	EP	2012	2160	End vertices	Vertices and partitions	$O(km)$	$O(n*k)$
DBH [48]	EP	2014	98	Degree, hash	Vertices, degree and partitions	$O(m)$	$O(n)$
HDRF [47]	EP	2015	121	End vertices, degree	Vertices, degree and partitions	$O(km)$	$O(n*k)$
ADWISE [144]	EP	2018	35	End vertices, degree, window	Vertices, degree and partitions,cluster information	$O(km)$	$O(n*k+w)$
2PS-HDRF [147]	EP	2022	4	End vertices, degree	Vertices, degree and partitions, cluster information	$O(km)$	$O(n*k)$
2PS-L [50]	EP	2022	-	End vertices, degree	Vertices, degree and two partitions, cluster information	$O(m)$	$O(n*k)$
CLUGP [148]	EP	2022	-	End vertices, degree	Vertices, degree and cluster information	$O(m)$	$O(n)$

Greedy [46] heuristic for the offline and stream components with minor modifications of both algorithms, respectively. OffStreamNH uses NE and HDRF [47] for the offline and stream parts, respectively. HEP reduces the memory overhead by splitting the graph's edge set into two subsets, low-degree, and high-degree vertices. The average degree of the graph is used to figure out which vertices are high-degree and which are low-degree. Edges connecting two high-degree vertices are partitioned in the stream (using HDRF), and edges with one of the low-degree vertices are partitioned in-memory (using NE) partitioning.

D. DYNAMIC APPROACH

Graphs are inherently dynamic. The graphs' topology is dynamically changed because some vertices and edges may be removed or added from the graph over time [150], [151]. As these graphs' topology evolves, the partitioning quality of partitioners would be constantly degraded due to unbalanced load distribution in each partition and communication overhead. Therefore, the dynamic approach was proposed to overcome this challenge.

1) DYNAMIC VERTEX PARTITIONING

Dynamic vertex partitioning regulates the communications and load of computing nodes based on a selection of vertices to migrate. The main differences among dynamic vertex partitioning are how to choose vertices for migration, selecting a target partition, and how to exchange vertices. xDGP [56], X-Pregel [57], Mizan [58], GPS [29], and LogGP [152] graph processing systems integrate their own dynamic vertex partitioning. xDGP uses adaptive iterative partitioning, which performs an iterative vertex migration, relying only on local information. At every iteration, after initial partitioning, each vertex will decide whether to remain in the present partition or migrate to other partitions, which have the highest number of neighbor vertices to minimize edge cut. GPS uses Large Adjacency-List Partitioning (LALP). To dynamic repartition the graph, it considers only external communication of vertices. Migrations of vertices are performed from vertex v, at worker w_i to new worker w_j, if v has more incoming/outgoing message from w_j than any other workers. X-Pregel uses dynamic repartition by considering both internal and external communications of vertices. It proposed two options before migrating vertices to each worker, sharing and without sharing adjacent lists of the vertices to the workers. Mizan uses a migration planner to find the most substantial cause of workload imbalance based on three metrics, an outgoing message, incoming message, and response time. Each machine computes the correlation between each metric and selects the factor with the highest correlation as the objective factor for moving vertices. LogGP introduces a log-based graph partitioning that records, analyzes, and reuses the previous partition and calculates statistical information to improve partitioning quality. It uses hypergraph repartitioning and superstep repartitioning. Hermes [153] was developed as a fork for Neo4j [25] graph database. Hermes uses a multi-level partitioning method like Metis [41] to partition the graph across numerous servers. Metis was designed for offline; however, Hermes introduced the lightweight repartitioner, which maintains high-quality partitions while adapting to graph changes. The lightweight repartitioner algorithm tries to improve an existing partitioning by reducing edge-cuts while keeping divisions nearly balanced. KGGGP [154] is a dynamic vertex partitioning that can be easily implemented into a multilevel structure with some minor adjustments to the fixed vertices at the start. To begin, an extra restriction is imposed during the coarsening step, preventing fixed vertices from belonging to distinct portions from being matched together, whereas they can be directly matched with free vertices.

2) DYNAMIC EDGE PARTITIONING

DynamicDFEP [155], GrapH [156], and GraphSteal [157] are example of dynamic edge partitioning. DynamicDFEP leverages Dfep [130] algorithm to make initial partitioning and introduces three update strategies, a complete partitioning method, partial partitioning method, and unit-based insertion. It updates the partition of a large graph when new vertices and edges are included or removed. GrapH uses H-adapt strategies to migrate a set of bag-of edges after GAS iteration. It selects two arbitrary partitions after each superstep and migrates nominee edges between them. To avoid inconsistency, it exploits locking techniques on the vertices adjacent nominee edges. GraphSteal is a dynamic edge partitioner that dynamically re-partition graph based on the job's runtime characteristics. It migrates edges from slow nodes to fast nodes to avoid computational imbalance in the cluster.

IV. COMPUTATIONAL MODELS OF GRAPH COMPUTING SYSTEMS

We classify the computational models of existing graph computing systems into two general categories; computational models for graph processing and graph database systems. Both platforms have used different computational models to process graph analytic on large-scale graphs. The computational models of GPS and graph databases are discussed in this section.

A. COMPUTATIONAL MODELS OF GRAPH PROCESSING SYSTEMS

The graph processing systems' design explores a new model to compute large-scale graphs efficiently due to the explosive graph size and the inherent complex structure of graphs. GPS's principal computational models include programming, communication, execution, and graph partitioning methods.

1) PROGRAMMING MODELS

Programming models are a higher-level programming interface that users quickly write graph applications. They provide a set of methods that allow users to read and modify their graph data. Therefore, users can focus on their algorithms' logic and not bother about communication patterns, data representation, and the underlying architecture of the computing system. Algorithms for graph processing usually require a sequence of iterative operations. Hence, several programming models have been proposed to improve iterative computation. The programming models of GPS include MapReduce [158], Vertex-centric [19], Gather-Apply-Scatter [23], and Subgraph-centric [84].

a: MapReduce PROGRAMMING MODEL

Jeffrey and Sanjay [158] proposed the MapReduce (MR) programming model. It is a distributed programming framework for large-scale data computing on commodity clusters. MR has two components: Map and Reduce functions. Both the Map and Reduce functions are written by the users. The Map function accepts a batch of data and changes it into other intermediate data called key-value pairs. The Reduce function gets the Map function output as input and combines them to form possibly smaller key-value pairs. Apache Hadoop [159] implements the MR for the distributed analysis of large-scale data across clusters. Many real-world tasks are represented in this model, as well as graph algorithms. However, the MR paradigm can't process graph data efficiently because graphs don't have good locality of memory access and do little work per vertex. Hence, the vertex-centric programming model was proposed by [19].

b: VERTEX-CENTRIC PROGRAMMING MODEL

The vertex-centric (VC) programming model is called Think-Like-A-Vertex (TLAV). VC is the most mature model for large-scale GPS which users express computational tasks from the point of a single vertex. Each vertex consists of a unique id, local state, outgoing edges, and optional vertex and edge value. The computation of the VC model is represented as an order of supersteps. In each superstep, vertices can be active or inactive, and messages are exchanged among vertices synchronously. The VC model exploits the vertex partitioning method to compute large-scale graphs [19].

c: GATHER-APPLY-SCATTER PROGRAMMING MODEL

PowerGraph [23] introduced the Gather-Apply-Scatter (GAS) programming model and applied edge partitioning to avoid the imbalanced workload distribution when using the VC programming model on power-law graphs. To eliminate the influence of higher-degree vertices in VC, the GAS programming model decomposes the vertex program into three stages: Gather, Apply, and Scatter. In the Gather stage, data about adjacent edges and vertices are collected using a derived sum over the vertex neighborhood. In the Apply stage, the accumulated sum is updated on the central vertex. Finally, in the Scatter stage, the adjacent edges' values are updated by the central vertex's new value.

d: SUBGRAPH-CENTRIC PROGRAMMING MODEL

Both the VC and GAS models work on the focus of the scope of a single vertex computation. This characteristic brings simplicity and scalability. But because these models use supersteps which are single hops in iterations, it may take a while to talk to the node you want to reach. Moreover, communication comes with the cost of network messaging, and it may become problematic if there are many large messages to exchange. Therefore, the Subgraph-centric (SC) [84] model was proposed to address communication latency issues by offering a scope of subgraph computation. Instead of storing different vertices on each partition, it suggests keeping their subgraphs.

2) COMMUNICATION MODELS

During graph computation, the vertices send messages through edges to their neighbors. Therefore, plenty of messages are exchanged among partitions of subgraphs for coordination and data synchronization. Communication models play a critical role in coordinating the data transfer among the cluster of computing machines. The communication models can be classified as message-passing, shared memory, and dataflow based on how data is transferred.

a: MESSAGE PASSING MODEL

In message passing (MP) model, information is dispatched from one vertex program to another using a message-based communication. The message has local vertex data and Id of the target vertex. In MP model, the graph entities have their own local and non-local states. These states are partitioned and distributed across different workers. These workers have read-only access to the local state and can not access and modify other workers' states. The update is performed by sending and receiving messages explicitly or implicitly

within the graph entities. Message passing interface is commonly used in GPS [19], [20], [21], [22], [23].

b: SHARED MEMORY MODEL

Vertex data is exposed as shared variables in shared memory (SM), which can be read or updated directly by other vertex programs. SM eliminates the additional memory overhead caused by messages. Communication through the SM model allows tasks in different worker machines to communicate by mutating a shared state. The framework that employs this model uses lock or semaphore to handle race conditions and data consistency [160].

c: DATAFLOW MODEL

A distributed application is represented by a Distributed Acyclic Graph (DAG) of operations in dataflow (DF) model, which is a generalization of the MR model. The DF model [161] is a DAG that consists of operators, sources, and target. The data sources, targets, and intermediate data sets that pass through operators. Vertices represent data-parallel tasks, whereas edges represent data flowing from one task to another in the DAG. In DF model [52], the data flows through the systems towards the next computation phase. The framework deploys this model that provides explicitly or automatic caching mechanisms and integrate general-purpose operators (e.g., map, reduce, join, filter) to load and transform graphs.

3) EXECUTION MODELS

In GPS, distributed coordination of graph entity is an essential task to perform iterative computation. Execution models deal with how a specific implementation of a program model leads to convergence. There are three types of execution models in the existing GPS: synchronous, asynchronous, and hybrid.

a: SYNCHRONOUS EXECUTION MODEL

Synchronous (Sync) [19] execution refers to concurrent workers that process their task one iteration followed by other iteration based on global barriers as shown in Fig.8. Initially, a graph computation has an input. Then, the graph is initialized and followed by a series of supersteps separated by global barriers until the overall graph computation terminates with the desired output. At the end of each superstep i, changes to the vertex and edge data are committed and visible in the next superstep $i + 1$. In each superstep, active vertices are executed. Regardless of the number of machines, the Sync execution model assures deterministic execution. The frequent barriers that reduce the efficiency distributed execution and algorithm convergence [23]. Most single machine or distributed GPS use the Sync execution model.

b: ASYNCHRONOUS EXECUTION MODEL

In the asynchronous (Async) execution model, computation is performed immediately after its current iteration. As shown in Fig. 9, it does not use any global barriers.

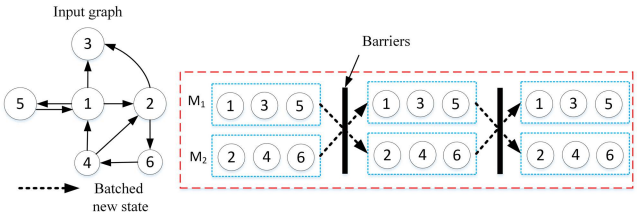


FIGURE 8. Execution flow of Sync model. Within each iteration, all vertices in the input graph are performed in a fixed order.

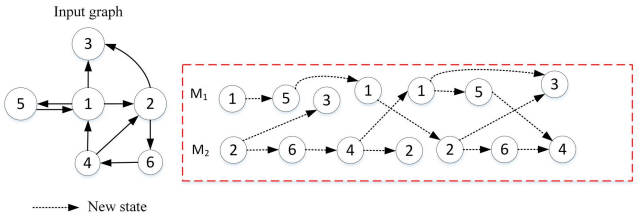


FIGURE 9. Execution flow of Async model. As quickly as possible, the update of each vertex is accessible to neighboring vertices.

Synchronization can be applied either through shared memory or through local barriers and distributed coordination. In the Async execution model, computing engines execute active vertices as processors and allocate network resources immediately. During computation, changes to the edge and vertex data are automatically committed to the graph and accessible to subsequent computation on neighboring vertices. The Async execution model can make better use of resources while increasing the algorithm convergence rate.

c: HYBRID EXECUTION MODEL

The hybrid execution model (Hsync) is a hybrid of the Sync and Async models that changes from the Sync and Async mode based on the current situation vice versa as shown in Fig. 10. Recently, several GPS have used this model to overcome the shortcoming of existing systems. PowerSwitch [162] adapts a Hsync that allows dynamic switching from the Async to Sync model to gain performance. PowerSwitch captures execution statistics such as active vertices, throughput and convergence speed on a continuous basis and uses online sampling, offline profiling, and a set of algorithms to reliably forecast ideal mode transition points. GoFFish [163] and Giraph++ [84] also uses hybrid execution model. These frameworks apply the Async execution model for local vertices and the Sync execution model for remote vertices.

B. COMPUTATIONAL MODELS OF GRAPH DATABASE SYSTEMS

Graph databases design mainly focus on general architecture, data model and organization, data distribution, and transaction queries. This sub-section describes the computational model of graph databases, including the data models, partitioning techniques, and query languages.

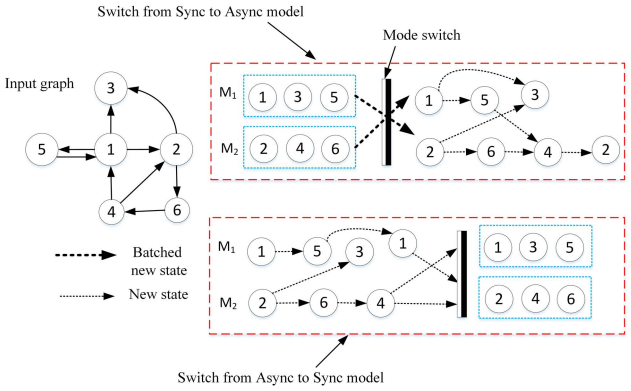


FIGURE 10. Execution flow of Hsync model. Based on a statistical analysis of algorithms, the Hsync switches from Sync to Asysn model vice versal.

1) DATA MODELS

Data models are essential to represent information and knowledge, depend on application areas and user requirements. The data models of graph database can be classified as graph and nongraph data.

a: GRAPH DATA MODELS

Graph data models set a new standard for visualization of data in the form of vertices (nodes) and edges (relations). There are four types of graph data models: simple graph, hypergraph, property graph model (PGM), and RDF.

Simple graph model is used to represent the group of vertices and edges that form the graph and is frequently applied in graph processing platform [19]. However, it doesn't seem applicable in graph databases. Hypergraph model is extends version of the simple graph model that an edge (called a hyperedge) can connect multiple nodes. It can be applied when data sets contain a plenty number of many-to-many relationships [25]. PGM is a broadened version of the simple graph model that contains the property of nodes and relationships. The PGM has three components, nodes, relationships, and properties (data stored on the relationships or nodes) [25], [37]. Nodes represent real-world entities. They can store any number of attributes. Relationships represent the relation type of the start and end nodes, with distinct properties just like nodes. A property is a key-value pair that key identifies a property name, and value is actual data. The PGM is the most popular data model for graph database [25]. Fig. 11 illustrates property graph model. RDF [164] is a framework for modeling information on the Web. The RDF is also named as triples store. It can be intuitively considered as a semantic network. The RDF contains three elements to represent data, subject (resource), predicate (attribute), and object (attribute value). Each element expresses a logical relationship between the subjects and objects. The RDF triples can be represented the subjects and objects as nodes, and the predicates are denoted as edges. Fig. 12 illustrates an example of the RDF model. For more comprehensive reviews on RDF, readers can refer to [76].

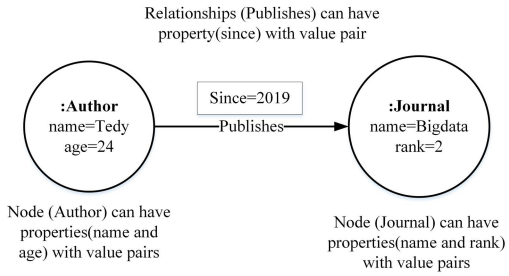


FIGURE 11. An example of PGM representation of author and journal relationships.

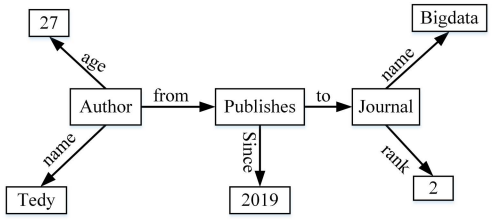


FIGURE 12. An example of RDF data model representation of author and journal relationships. Author, publishes and journal as the subjects and age, name, since, from, to and rank as the predicates and "27", "Tedy", "2019", "2", and "bigdata" as the objects.

b: NONGRAPH DATA MODELS

There exist data models that are not specific to graphs; however, they are used in various systems to design and store graphs. Those data models [165] include key-value, wide-column, and document stores. Key-value-store contains key-value pairs with unique keys. It helps easy partitioning and efficient querying data with high scalability. In the key-value-store, vertices and edges are stored as values and are indexed by unique keys. Wide-column-store is also called column-family stores [166] that presents data in tabular form of rows and columns. This storage combines the nature of relational tables and key-value pairs. Each row can have an arbitrary number of columns, and every column consists of key-value pairs. Each vertex is stored in a row and is indexed by a unique key. The vertex value, labels, properties, and adjacent edges are stored in row columns (cells). Document-store [167] extends the key-value-store that encodes the values via semi-structured formats such as XML or JSON documents. The values have a flexible schema, which consists of an attribute with one or more values. Document-store queries entire document by key and also fetches only some part of the documents. The vertices and edges are encoded in documents and linked via document Ids.

2) PARTITIONING TECHNIQUES

Graph partitioning and sharding are the essential data partitioning techniques for large-scale data. The former and the latter are used to partition graphs and tabular data, respectively. As we have seen in section II, graph partitioning is utilized for GPS and GDBS to divide large-scale graphs into subgraphs. Some parts of these subgraphs are replicated before it starts processing.

Sharding involves splitting large-scale data into many partitions that are distributed across several database instances [168]. Its primary purpose is to speed up query processing and extend the system as needed. The sharding process comprises a database server that handles the burden of the requests that are delivered to it. The database server must have a user id, and each database is served by one server. Unlike graph partitioning, sharding does not use a requirement for load balance and splits rows or columns of a large database table into multiple smaller tables without replication [169]. The server can use lookup, hash, and rang sharding strategies. The sharding is commonly practiced for relational database systems [170] and NoSQL [171] databases; however, it is rarely applied to graph databases [172].

3) GRAPH QUERY LANGUAGES

Graph query languages are designed for the manipulation of GDBS. The most widely used graph query languages for graph databases include, SPARQL [173], Cypher [174], Gremlin [175] and GraphQL [176]. Each query language has its functionality to navigate the data. SPARQL and Cypher are designed to operate for RDF graphs and property graphs, respectively. Gremlin and GraphQL are designed towards graph traversal and APIs for fulfilling those queries with existing data. Some graph databases can support two or more than two query languages.

SPARQL [173] is a standard declarative query language recommended by the W3C[4] for querying RDF. SPARQL supports all of the complicated graph patterns. Triple patterns of RFD (the subject, object, or predicate) are the core building blocks of SPARQL queries. Both SPARQL and Cypher contain graph pattern matching styles that can be composed via SQL-ish keywords.

Cypher [174] is a high-level, well-established declarative query language for the PGM, initially invented and implemented as part of the Neo4j graph database project. It gets a property graph as input and displays a table as output. Cypher is designed similar to SQL to make the transition between the two languages as smooth as possible. For many functions, it uses the same clause syntax structure and implements the existing semantics. It includes new features to the language to support multiple graphs and query composition. Many commercial products like Memgraph, HANA Graph, Redis Graph, and Agens Graph have recently implemented Cypher as a core query language. Cypher is now being defined as a fully specified standard under the auspices of the openCypher,[5] which can be independently implemented utilizing various architectures, storage and query optimization techniques.

Gremlin [175] is a low-level language that offers imperative and declarative query language within the same framework. TinkerPop[6] project designed, and distributed this Gremlin query language. It is more imperative in nature and focuses on graph traversal instead of pattern matching. Gremlin supports pattern matching features in a declarative pattern style. These two features help to execute the query on graph database and graph processing system.

GraphQL[7] [176] is an open-source graph query language for application programming interfaces and is initially created by Facebook. GraphQL is more popular as an alternative to REST-based interfaces, which have influenced the Web-API scenario by giving the decision to clients instead of servers. Like Germlin, GraphQL supports imperative and declarative query processing. For more comprehensive reviews on graph query languages, readers can refer to [177].

V. TAXONOMY OF GRAPH COMPUTING SYSTEMS

Graph computing systems are developed for processing, and analyzing large-scale graphs. Based on their graph analytics nature, the graph computing systems can be classified into two categories, GPS and GDBS. The various classification of these platforms are discussed in this Section. Fig.13 illustrates the detailed taxonomy of graph computing systems.

A. GRAPH PROCESSING SYSTEMS

Based on the architecture they are designed, GPS also can be classified into two, distributed graph (DS) and single machine graph processing systems [31].

1) DISTRIBUTED GRAPH PROCESSING SYSTEMS

Distributed GPS are a group of multiple processing nodes and each node participates during graph computations. They use various computing model to improve their performance. We classify these systems into two, MapReduce and Non-MapReduce family based on their computing model.

a: MapReduce FAMILY SYSTEMS

MapReduce family systems are used MR model with a minor modification of the stage of the MR model. Hadoop [159] uses MR model to enable users to easily build scalable parallel algorithms and processes large-scale data on clusters machines. However, Hadoop does not give direct support for iterative data analysis tasks. To solve this, several MapReduce family graph analysing systems have been proposed with modification of of MR model to improve the efficiency. These systems include Pegasus [178], HaLoop [179], Twister [180], iMapReduce [181], and Surfer [182]. Pegasus [178] implements GIM-V(Generalized Iterated Matrix-Vector multiplication) as a two-stage MapReduce algorithm. It represents the input graph as two files, vertices as vector and edges as matrix. To operate, it provides three function *combine2()*, *combineAll()*, and *assign()*. In the first stage, the map phase converts the input edges to set destination vertex as the key, and the reduce phase performs *combine2()* to multiplicate the matrix element with the vector element. The second stage accepts the output of the first stage. In this

[4]https://www.w3.org/TR/rdf-sparql-query/
[5]https://opencypher.org/Group
[6]http://tinkerpop.apache.org/
[7]https://graphql.org/

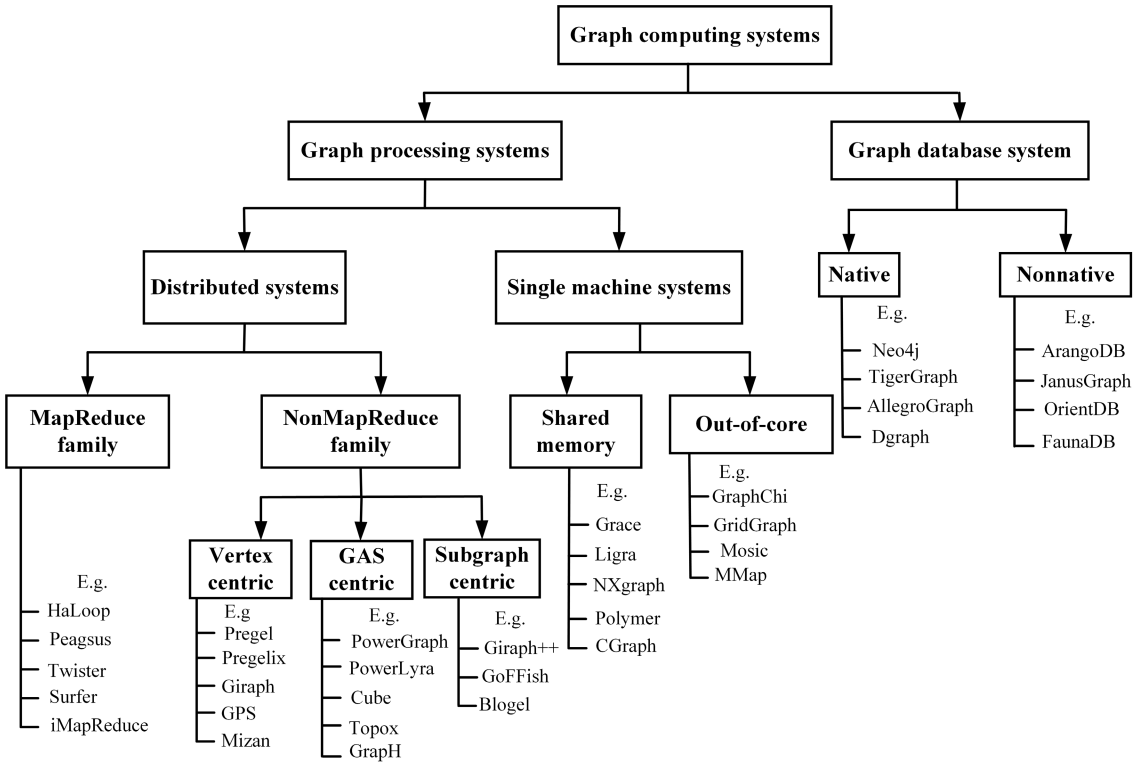


FIGURE 13. Taxonomy of graph computing systems.

second stage, *combineAll()* and *assign()* perform summation of partial multiplication and write the new result, respectively. HaLoop [179] is a modified variant of the MapReduce framework that supports an iterative computation. It uses task scheduler loop-aware and caching mechanisms to avoid reloading iteration-invariant data and to reduce communication costs. Twister [180] extends MapReduce API to support an iterative computation. It provides broadcast and scatters data transfers. Its communication and data transfer are performed through publish/subscribe messaging. Surfer [182] is designed to handle large-scale graph analytic based on two principal primitives for users: MapReduce and Propagation. In this system, MapReduce performs different key-value pair in parallel while propagation is an iterative computation that transfer information along the edges from a vertex to its neighbors in the graph. iMapReduce [181] allows for programmers to specify the iterative processing with a map and reduce functions. It explicitly provides model, iterative algorithm, and the concept of persistence task to accomplish recursive computation by avoiding frequently destroying, creating,and scheduling tasks. It also provides to load input data to the persistence task once and never needs to be shuffled between the map and reduce the job.

b: NonMapReduce FAMILY

MapReduce family GPS are inefficient for the graph processing because the efficiency of graph computations depends heavily on inter-processor bandwidth as graph elements are transferred over the network after each iteration [19]. To solve this inherent performance degradation, many NonMapReduce based graph processing system have been proposed. In 2010, Google has proposed a novel scalable platform using vertex centric programming model called Pregel [19]. Recently, many graph processing have been proposed by extending this framework. The NonMapReduce family systems can be classified into, Vertex-centric, Gather-Apply-Scatter, and Subgraph-centric based on the programming model they are operated.

c: VERTEX-CENTRIC SYSTEMS (VCS)

VCS execute a user-defined program over the vertices of a graph iteratively. The vertex program is written from the point of view of a vertex, and it accepts data from neighboring vertices and incident edges as input. The VCS include Pregel [19], Giraph [20], HAMA [21], Pregelix [183], GPS [29], Mizan [58], and Cyclops [184]. Pregel [19] is a pioneer GPS. It uses the vertex-centric programming model, bulk synchronization parallel model, and vertex partitioning method. Giraph[8] [20] is an open-source implementation of Pregel and adds several characteristics beyond the principal Pregel model such as edge-oriented input, shared aggregator, out-of-core computation and master computation. HAMA[9] [21] is a distributed system on top of Hadoop

[8]https://giraph.apache.org/
[9]https://hama.apache.org/

for graph computations and massive matrix computations. It supports three computation engines, BSP, MapReduce, and Microsoft Dryad [185]. MapReduce is used for matrix multiplication, BSP and Drayd are used for graph computation. Pregel+[10] [186] supports vertex mirroring and request-respond paradigm for the reduction of message exchange through a network. Mirroring is needed to create a copy of vertex for the higher degree vertex on a different machine. In the request-respond paradigm, each vertex requests another vertex to send a message. All machine request from the same target vertex merged together into one single request. Pregelix[11] [183] supports in-memory and out-of-core workloads. It is an open-source implementation on top of the Hyracks (parallel dataflow engine based). It represents messages and vertices data as a tuple, then applies join operation for message exchange between vertices. GPS[12] [29] introduces many built-in system optimizations such as message objects, single canonical vertex, and using per-worker rather than per-vertex message buffering (which improves network usage), Large Adjacency List Partitioning (LALP), and dynamic migration. Mizan[13] [58] identifies the runtime characteristics of the system and provides a dynamic migration scheme. Cyclops [184] combine the best feature from other GPS. It takes the BSP from Pregel [19], direct memory access from Graphlab [187], and distributed activation from PowerGraph [23]. It uses a distributed immutable view that permits a vertex alongside read-only access to every its neighboring vertices and provides read-only replication of vertices for the edges spanning during a graph cut.

d: GATHER-APPLY-SCATTER SYSTEMS (GASS)

GASS improve power-law graph processing by combining the GAS model with vertex-cut partitioning. GASS systems include PowerGraph [23], PowerLyra [22], GraphA [188], Cube [189], SympleGraph [190] and Topox [191]. PowerGraph[14] is designed to compute large scale power-law graphs. It supports GAS Programming model, edge partitioning, synchronous and asynchronous serializable timing. PowerLyra[15] extends the PowerGraph system and introduces a hybrid graph partition method to reduce replication by separate lower and higher degree vertices. It uses the GAS programming model, synchronous execution model. The higher-degree vertex computes as same as PowerGraph. However, the lower-degree vertex limit from bidirectional flow to unidirectional computations. GraphA [188] introduced an adaptive and uniform graph partitioning algorithm that partitions graphs using an incremental number of mapping functions. To achieve fine-grained and low-cost graph storage, GraphA leverages the Adaptive Radix Tree adjacency list [192]. It uses the GAS model and synchronous timing. SympleGraph [190] observes user-defined functions and identifies the loop-carried dependency. This system enforces the precise semantics by performing dependency propagation dynamically. Circulant scheduling and double buffering is proposed to improve performance. Topox [191] utilizes GAS Model, hybrid-BL partitioner and topology refactorization (TR). TR transforms the power-low graph into a further communication efficiency topology through the fusion and fission method. The fusion organizes a group of neighboring lower-degree vertices into a super-vertex while the fission makes splitting a higher-degree vertex into a group of siblings-vertices. The hybrid-BL partitions the new topology.

e: SUBGRAPH-CENTRIC SYSTEMS (SCS):

SCS extend the view of the vertex as specified subgraph. SCS include Giraph++ [84], GoFFish [163], and Blogel [193]. Giraph++ uses SC programming model to open partitions structure to users and allows information to flow freely inside the partitions. It contains two groups of vertices, internal and boundary. Internal vertices contain vertex value, edge values, and incoming message; however, boundary vertices have only vertex value. It is implemented based on Apache Giraph. GoFFish uses SC programming model with a distributed steady graph storage for large-scale graphs analytics on commodity clusters, providing natural flexibility of SM sub-graphs computation. Blogel is a block centric framework via SC programming model. A block represents to connected subgraph, and message exchanges occur within the blocks. It uses graph Voronoi diagram partitioner to create a block.

2) SINGLE-MACHINE GRAPH PROCESSING SYSTEMS

Plenty of distributed GPS have recently been proposed to support the large-scale graph, such as Pregel, PowerGraph, etc. However, these systems have suffered from load balance [194], synchronization overhead [195] and fault tolerance overhead [196]. Moreover, the programmers face challenges to easily use and optimize the graph algorithm in distributed than single-machine systems. Therefore, single-machine GPS have been introduced to tackle large-scale graphs by extending multi-core, Solid State Drive (SSD) or Hard Disk Drive (HDD). The design issue of single-machine graph processing must consider four rules: (i) ensure the locality of graph data; (ii) exploit the parallelism of multi-thread CPU; (iii) minimize the size of disk data transfer and (iv) streamline the disk Input/Output. We classify the single-machine graph processing into single-machine shared memory (SMSM) and single-machine out-of-core (SMOC) systems based on memory usage.

a: SINGLE-MACHINE SHARED MEMORY SYSTEMS

They consist of one processing unit, physical memory, and one or more CPU cores that share the graph entities across all the cores. The SMSM with multicore can handle surpassing terabytes of memory, which can fit graphs alongside

[10] http://www.cse.cuhk.edu.hk/pregelplus/
[11] http://pregelix.ics.uci.edu/
[12] http://infolab.stanford.edu/gps/
[13] https://github.com/khayyatzy/Mizan
[14] https://github.com/jegonzal/PowerGraph
[15] https://github.com/realstolz/powerlyra

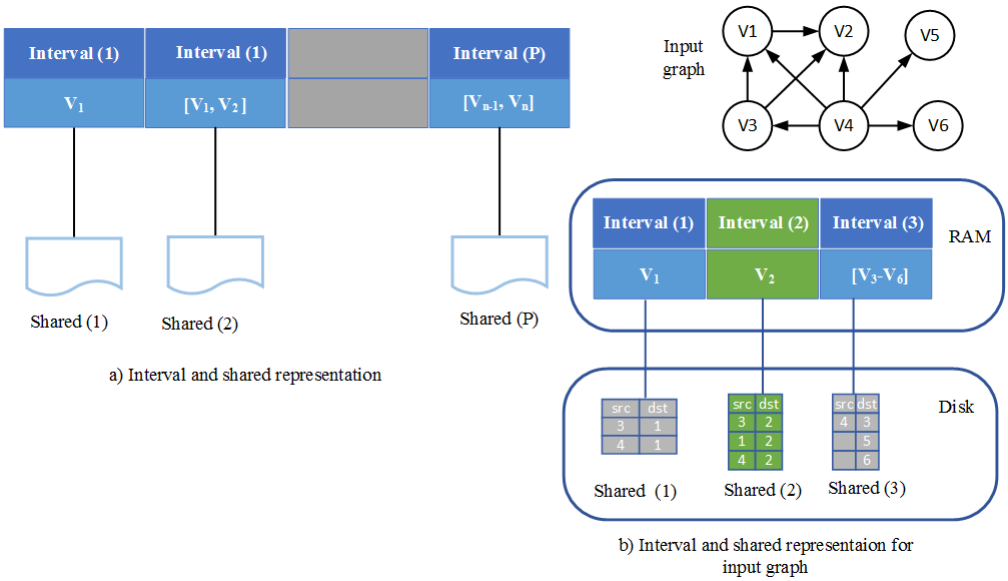


FIGURE 14. Out-of-core graph representation in GraphChi. a) A given vertices of graph are divide into intervals and each interval has a shard, b) Input graph is split into 3 intervals and 3 shards.

tens or even hundreds of billions of edges [33]. The SMSM include Grace [32], Ligra [33], Polymer [199], NXgraph [35], and CGraph [200]. Grace [32] introduces block-oriented computation by separating application logic and execution. It operates similar to the VC programming model; however, it executes a block of highly connected vertices at a time. It applies block-level and vertex level scheduling policies. Ligra[16] [33] is a lightweight framework that is applicable for graph traversal. It provides two routines for mapping vertices and edges. Polymer [199] adapts non-uniform memory access (NUMA) architecture by co-locating graphs and computation inside NUMA-nodes as far as possible. To minimize random and remote memory access, it uses hierarchical scheduling, edge partitioning and adaptive data structure. NXgraph [35] offers a destination-sorted sub-shard structure to store graphs. It splits vertices and edges into intervals and sub-shards, respectively. Edges in each shard are sorted according to their destination vertices to ensure graph data access locality and enable fine-grained scheduling. CGraph [200] uses a correlation-aware execution model, together with a core-subgraph-based scheduling algorithm, and achieves improvement on concurrent recursive graph processing (CGP) jobs. SMSM systems are mainly characterized by simple programming and computing models, low hardware overhead, and limited computing power.

b: SINGLE-MACHINE OUT-OF-CORE SYSTEMS

With the advent of big graph data, the intuition of another approach is required to store a graph out-of-core in the external memory, such as SSD and HDD to tackle the challenge of scalability. The primary consideration for Out-of-core GPS is that the size of the graphs is larger than the main memory. However, it can fit the storage size of the HDD or SSD. However, computing capacity and data exchange bandwidth of external memory are hard to process large-scale graphs under acceptable conditions because of random disk access memory. The SMOC systems include, GraphChi [30], MMap [202], GridGiraph [31], Mosaic [203], and GraphQ [201].

GraphChi [17] is a pioneer in single machine out-of-core GPS. It performs preliminary processing on the graph data before beginning the actual computation. It introduces the parallel sliding windows (PSW) method, which represents graph properties to efficient processing from disk. It uses the VC programming model, PSW (to load data for computing), and selective scheduling to accelerate convergence. GraphChi divides the graph into several vertex intervals and keeps each vertex interval's incoming edges as a shard. Each shard contains all the input edges of the corresponding vertex set and sorts them according to the Id of their source vertices. Fig. 14a depicts graph representation as intervals and shards in GraphChi. For example, Fig. 14b shows the shard structure for input graph. Assume the vertices of the input graph are partitioned as V_1, V_2 and $V_3 - V_6$ in interval (1), interval (2), and interval (3), respectively. The shard (1) saves every incoming edge of the vertex interval V_1, shard (2) stores every incoming edge of the vertex interval V_2, and shard (3) stores every incoming edge of the vertex interval $V_3 - V_6$, respectively. As shown in Fig. 14b, when the vertex set in interval 2 is active (the green colored vertex), the shard (2) (the green edge list) is loaded to memory. After the computation is completed, the result is written to

[16]https://github.com/jshun/ligra

[17]http://graphlab.org/projects/graphchi.html

TABLE 5. Summary of graph processing systems.

GPS	Architectures	Storage	Years	No.of Times Cited (as of 20/09/2022)	Programming Models	Partitioning Methods	Communication Models	Execution Models	Implemented Languages
Peagsus [178]	DS	RAM	2009	930	MR	VP	MP	Sync	Java
Surfer [182]	DS	RAM	2010	119	MR	VP	MP	Sync	Java
HaLoop [179]	DS	RAM	2010	1137	MR	VP	DF	Sync	Java
Twister [180]	DS	RAM	2010	1253	MR	VP	DF	Sync	Java
iMapReduce [181]	DS	RAM	2011	307	MR	VP	DF	Async	Java
Pregel [19]	DS	RAM	2010	4783	VC	VP	MP	Sync	C++
Apache Hama [21]	DS	RAM/Disk	2016	321	VC	VP	MP	Sync	Java
Apache Giraph [20]	DS	RAM/Disk	2011	-	VC	VP	MP	Sync	Java
GPS [29]	DS	RAM	2013	694	VC	VP	MP	Sync	Java
Mizan [58]	DS	RAM	2013	386	VC	VP	MP	Sync	C++
Pregelix [183]	DS	RAM/Disk	2014	386	VC	VP	DF	Sync	Java
Cyclops [184]	DS	RAM	2014	66	VC	VP	MP/SM	Sync	Java
Pregel+ [186]	DS	RAM	2015	133	VC	EP	MP	Sync	C++
PowerGraph [23]	DS	RAM	2012	2160	GAS	EP	SM	Async,Sync	C++
PowerLyra [22]	DS	RAM	2015	447	GAS	HP	SM	Sync	C++
Cube [189]	DS	RAM	2016	69	GAS	EP	MP	Sync	C++
GraphA [188]	DS	RAM	2017	4	GAS	EP	MP	Sync	C++
GrapH [156]	DS	RAM	2018	25	GAS	EP	MP	Sync	Java
L-PowerGraph [197]	DS	RAM	2020	2	GAS	EP	MP	Sync	C++
SympleGraph [190]	DS	RAM	2020	9	GAS	EP	SM	Sync	C++
Topox [191]	DS	RAM	2020	5	GAS	HP	SM	Sync	C++
Giraph++ [84]	DS	RAM	2013	404	SC	VC	MP	Hsync	Java
GoFFish [163]	DS	RAM	2014	123	SC	VP	MP	Hsync	Java
Bloge [193]	DS	RAM	2014	239	SC	VP	MP	Hsync	C++
Grace [198]	SMSM	RAM	2013	104	VC	VP	MP	Async	C++
Ligra [33]	SMSM	RAM	2013	877	SC	VP	SM	Async	C++
Polymer [199]	SMSM	RAM	2015	219	VC	EP	SM	Sync	C++
CGraph [200]	SMSM	RAM	2018	23	SC	EP	SM	Sync	C++
GraphChi [30]	SMOC	RAM&Disk	2012	1304	VC	EP	SM	Async	C++
GridGiraph [31]	SMOC	RAM&Disk	2015	356	VC	EP	MP	Async	C++
GraphQ [201]	SMOC	RAM&Disk	2015	66	VC	VP	MP	Async	Java
MMap [202]	SMOC	RAM&Disk	2014	59	VC	VP	SM	Sync	Java
Mosaic [203]	SMOC	RAM&Disk	2017	170	Hybrid	VP	MP	Hsync	C++

the disk. This step continues until it reaches convergence. MMap exploits the memory mapping, which maps the edge list into the virtual memory so that the edge file on the disk is accessed as the same as file is loaded in memory. The memory-mapped edge file minimizes data copy to and from the user-space buffer; thus, improves performance. Mosaic[18] uses Hilbert-order tiles graph representation, hybrid computation and execution model. The hybrid computation model enables the vertex-centric model computation for the fast processor and edge-centric model for massively parallel co-processors. The hybrid execution applies synchronous vertex states update. However, if there are no changes in the current programming abstraction, it will use the asynchronous update to help attain scale-up and scale-out and enabling graph analytic on one trillion edges. GridGraph[19] utilize a 2-level hierarchical method to partition a graph at the preprocessing and run time phase. During the preprocessing phase, vertices and edges are divided into 1D-partitioned vertex chunk and 2D-partitioned edge blocks, respectively. At the run time phase, it uses a dual sliding window method to partition the graph by stream edges and perform vertices update. Table 5 describes the detail comparison of GPS.

B. GRAPH DATABASE SYSTEMS

GDBS are designed to efficiently store, process, and analyze large-scale graphs based on the principle of database management systems such as persistent data storage, data consistency, and integrity, logical or physical data independence. They use various data models to store and retrieve graph elements, vertices, edges, and properties. The fundamental element of GDBS are edges (connections) that are treated as the core component of the model, along with vertices. In contrast with conventional relational databases, connections between data are stored in separate tables; therefore, searching for connections require join operations, which takes much computational time. The GDBS face main challenges due to the nature of irregular graph computations to achieve low latency and high throughput of the graph queries to accessing or modifying a small or a large part of the graph.

Based on the graph storage and processing, graph databases can be classified as native and nonnative graph databases. Graph storage refers to the underlying storage layer of the database that is designed specifically for storing graph data. It is known as native graph storage. Graph processing refers to how the graph databases execute database operations, including both storage and queries.

1) NATIVE GRAPH DATABASES

Native GDBS implement their own underlying data structures and indexing for storing and querying graphs. Native graph databases include Neo4j [37], TigerGraph [26], AllegroGraph [27], Dgraph [28]. Neo4j is a famous standalone graph database based on a property graph model that naively stores nodes (vertices), relationships (edges),

[18]https://github.com/sslab-gatech/mosaic
[19]https://github.com/thu-pacman/GridGraph

TABLE 6. Summary of GDBS.

GDBS	Data models	Years	Partition methods	Query Languages	Types	Implemented Languages	Licenses
AllegroGraph [27]	Multi-model	2004	VP	SPARQL	Native	C++	Commercial
BlazeGraph [204]	Multi-model	2006	Sharding	SPARQL	Nonnative	Java	Open Source
OrientDB [205]	Multi-model	2010	Sharding	Gremlin	Nonnative	Java	Open source
Stradog [206]	Multi-model	2010	No	GraphQL	Nonnative	Java	Commercial
ArangoDB [207]	Multi-model	2012	No	AQL, Gremlin and GraphQL	Nonnative	C++	Open source
FaunaDB [208]	Multi-model	2014	VP	GraphQL	Nonnative	Scala	Commercial
Neo4j [37]	Property graph	2007	No	Cypher	Native	Java	Open source
Dgraph [28]	Property graph	2016	VP	GraphQL	Native	Go	Open source
JanusGraph [209]	Property graph	2017	VP	Gremlin	Nonnative	Java	Open Source
TigerGraph [26]	Property graph	2017	No	GSQL	Native	C++	Commercial

and attributes. It uses pointers to navigate and traverse the graph, supports transactions operation, and fulfills the ACID (Atomicity, Consistency, Isolation, Durability) properties. It is implemented in Java and utilizes Cypher query language to query graphs. TigerGraph is a commercial, native parallel, and distributed graph database based on a property graph model that supports bulk data loading, providing built-in parallel computation and real-time graph updates. It is written in C++ programming language and uses GSQL (TigerGraph Query Language). AllegroGraph is an enterprise, supports a multi-mode(property graph, Document, and RDF), horizontally distributed graph database. It uses a federation function to speed up complex queries across highly and knowledge bases and distributed data sets. It is written in python, Java, and Lips and uses SPARQL query language. Dgraph is an open-source and distributed native graph database based on a property graph model. It provides horizontal scalable, high availability, low-latency arbitrary depth joins, and crash resilience. It is written by Go programming language and uses GraphQL query language.

2) NONNATIVE GRAPH DATABASES

Nonnative GDBS exploit other database systems such as relational or NoSQL [165] to store graph data and design query interfaces to execute graph queries over the back-end system. Nonnative GDBS include, ArangoDB [207], OrientDB [205], Janusgraph [209], FaunaDB [208], Stardog [206], and Blazegraph [204]. ArangoDB is a multi-model (property graph, Document, and key-value) graph database system, and it can scale up vertically and horizontally, fulfills the ACID consistency properties, and supports fault tolerance. It is implemented in C++ and uses its own query language AQL (ArangoDB Query Language), and supports the other two query language, Gremlin, and GraphQL. OrientDB is a multi-model (property graph, Document, and key-value), distributed architecture, and transactions graph database. It is implemented in Java and uses Gremlin for query processing. Janusgraph is an open-source and a distributed graph database. It can scale graph data processing for analytics and traversal across a multi-machine cluster through Hadoop. It is designed based on a property graph data model and is implemented in Java. It supports concurrent transaction and batch graph processing. It uses Gremlin query language as manipulation of the graph data. FaunaDB is a multi-model (property graph, Document, and key-value) and serverless graph database in which the cloud provider dynamically allocates and manages the resource distribution. It is implemented in Scala and uses GraphQL query language. Stardog is a multi-model (property graph and RDF), secure, scalable, and an enterprise graph database and knowledge graph platform. It combines graph storage and visualization capability for cost effective and flexible integration. It is written in Java and uses GraphQL query language. Blazegraph is a multi-model (Property graph and RDF) and high-performance graph database. It is implemented in Java and uses SPARQL query language. Table 6 describes the comparison of GDBS.

VI. CHALLENGES AND FUTURE RESEARCH DIRECTIONS

Although researchers have made significant contributions to graph partitioning and computing systems in the last decade, there are still many challenges, from the algorithms to the system perspectives. This section discusses several research directions in graph partitioning and computing systems.

A. SCALABILITY

Graph partitioning is an NP-hard problem to reduce the cuts and maximize the load balance. This problem and the increased size of graph datasets make the graph partitioning problem more difficult. This problem is an open challenge. Research on the scalability of high-quality parallel graph partitioning is still ongoing. Even on shared-memory machines, scaling to a large number of threads remains challenging. In particular, attaining good scalability and quality on larger distributed memory machines is still a challenging problem. The stream partitioning is more scalable and performs well with minimal resource constraints. Unlike offline partitioning techniques, streaming partitioning produces substantially lower quality because such partitioners do not view a global graph structure. Thus, improving the performance of stream partitioning is an open problem. OffStream partitioning has recently been proposed to trade off the stream and in-memory edge partitioning by distributing one edge set in-memory and another edge set in stream. However, offStream approach was applied for only edge partitioning; thus, applying OffStream partitioning to vertex partitioning is not investigated.

B. DYNAMICITY

Graphs are naturally dynamic because vertices or edges may appear or disappear over time. Dynamic graph partitioning has been proposed to repartition dynamic graphs. Most existing dynamic partitioners are repartitioning the graphs based on the vertex partitioning method. However, many GAS-centric distributed frameworks use edge partitioning models. Therefore, there is a gap in the dynamic edge partitioning approach, which can be exploited in future research.

C. DOMAIN SPECIFIC

Real-world graphs and graph algorithms have unique characteristics. General-purpose graph partitioners have recently been proposed and integrated into computing systems to analyze all graph structures and algorithms. However, these partitioners frequently aim to divide a graph into pieces of equal sizes and minimize the edges and vertices cut to balance workload and lower synchronization overhead. For instance, they do not achieve a deserved performance improvement when computing PageRank and Triangle Count algorithms in the graph computing system with the same partitioning strategy. Due to the variability in algorithms' computation and communication patterns, such criteria do not always capture the bottleneck variables that affect the performance of parallel graph algorithms. Therefore, graph algorithms with computation-aware partitioning should be investigated in the future. In the same manner, real-world graphs have different topological structures. For instance, web and social network graphs do not have the same topology structure. Therefore, instead of designing a general graph partitioning, we suggest a graph structure-aware partitioning as future research direction.

D. ADOPTING MACHINE LEARNING

Recently, many research works on extending deep learning approaches for graph data have emerged [210]. The integration of graph neural networks and federated learning has been applied for graph classification, node classification, and edge classification [211]. However, the adoption of these techniques for graph partitioning has not been investigated. Thus, formulating a graph partitioning problem into a graph neural network and applying federated learning for distributed learning should be investigated in the future. Moreover, formulating a graph partitioning problem into a game theory approach is also envisioned in the future. Hua et al. [143] introduced a game theory for stream edge partitioning. Thus, applying a game theory for future static and dynamic vertex partitioning is a potential research direction.

E. SYSTEM PERSPECTIVES

Most existing graph processing systems have been developed to handle static graphs. However, real-world graphs are dynamic, with new vertices and edges quickly added and removed. Preserving a large amount of updating in dynamic graphs and performing practical real-time computation are challenging tasks. Thus, more study is needed to bring a dynamic large-scale graph processing system. Developing a routing-aware or topology-aware data distribution scheme for graph databases is still not investigated, especially in the context of recently proposed data center and high-performance computing network topologies and routing architectures. Moreover, designing a general-purpose graph computing system that supports both distributed graph processing and graph database could solve problems in this area. Applying a deep learning techniques on transactional aware data partitioning, user-friendly query formulation, high-performance transaction processing, and ensuring security in the form of authentication is significant in graph databases.

VII. CONCLUSION

The graphs have become significant and influential data representations in many application domains in the recent Big data era. To handle the rapid increase in large-scale graph sizes, efficient graph partitioning and computing systems are essential. Thus, graph partitioning methods and graph computing systems have been suggested to address these large-scale graph computing challenges in various architectures and computing models.

In this survey, we have made a comprehensive review of graph partitioning methods and graph computing systems. We have classified and discussed the graph partitioning methods and graph computing systems into several subcategories to understand the subject area. Their approaches, computing, and data models of those algorithms and systems are presented briefly. Finally, we have highlighted promising research directions in graph partitioning and computing systems.

ACKNOWLEDGMENT

The authors would like to thank the anonymous referees for their valuable comments and suggestions, which have improved the article greatly.

REFERENCES

[1] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Comput. Netw. ISDN Syst.*, vol. 30, nos. 1–7, pp. 107–117, Apr. 1998.

[2] S. Kulkarni and S. F. Rodd, "Context aware recommendation systems: A review of the state of the art techniques," *Comput. Sci. Rev.*, vol. 37, Aug. 2020, Art. no. 100255.

[3] L. Cao and J. Krumm, "From GPS traces to a routable road map," in *Proc. 17th ACM SIGSPATIAL Int. Conf. Adv. Geographic Inf. Syst.*, 2009, pp. 3–12.

[4] R. Jansen, H. Yu, D. Greenbaum, Y. Kluger, N. J. Krogan, S. Chung, A. Emili, M. Snyder, J. F. Greenblatt, and M. Gerstein, "A Bayesian networks approach for predicting protein-protein interactions from genomic data," *Science*, vol. 302, no. 5644, pp. 449–453, 2003.

[5] N. B. Turk-Browne, "Functional interactions as big data in the human brain," *Science*, vol. 342, no. 6158, pp. 580–584, Nov. 2013.

[6] H. Bolouri, "Modeling genomic regulatory networks with big data," *Trends Genet.*, vol. 30, no. 5, pp. 182–191, May 2014.

[7] I. Shrier and R. W. Platt, "Reducing bias through directed acyclic graphs," *BMC Med. Res. Methodology*, vol. 8, no. 1, p. 70, Dec. 2008.

[8] L. Harris and A. Rae, "Social networks: The future of marketing for small business," *J. Bus. Strateg.*, vol. 30, no. 5, pp. 24–31, 2009.

[9] D. F. Nettleton, "Data mining of social networks represented as graphs," *Comput. Sci. Rev.*, vol. 7, pp. 1–34, Feb. 2013.
[10] C. Orellana-Rodriguez and M. T. Keane, "Attention to news and its dissemination on Twitter: A survey," *Comput. Sci. Rev.*, vol. 29, pp. 74–94, Aug. 2018.
[11] A. De Salve, P. Mori, and L. Ricci, "A survey on privacy in decentralized online social networks," *Comput. Sci. Rev.*, vol. 27, pp. 154–176, Feb. 2018.
[12] S. Sakr, A. Bonifati, H. Voigt, A. Iosup, K. Ammar, R. Angles, W. Aref, M. Arenas, M. Besta, P. A. Boncz, and K. Daudjee, "The future is big graphs: A community view on graph processing systems," *Commun. ACM*, vol. 64, no. 9, pp. 62–71, 2021.
[13] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: Graph processing at Facebook-scale," *Proc. VLDB Endowment*, vol. 8, no. 12, pp. 1804–1815, 2015.
[14] H. Cao, Y. Wang, H. Wang, H. Lin, Z. Ma, W. Yin, and W. Chen, "Scaling graph traversal to 281 trillion edges with 40 million cores," in *Proc. 27th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, Apr. 2022, pp. 234–245.
[15] W. W. Zachary, "An information flow model for conflict and fission in small groups," *J. Anthropol. Res.*, vol. 33, no. 4, pp. 452–473, 1977.
[16] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numer. Math.*, vol. 1, no. 1, pp. 269–271, Dec. 1959.
[17] U. Endriss and U. Grandi, "Graph aggregation," *Artif. Intell.*, vol. 245, pp. 86–114, Apr. 2017.
[18] S. Yu, Y. Feng, D. Zhang, H. D. Bedru, B. Xu, and F. Xia, "Motif discovery in networks: A survey," *Comput. Sci. Rev.*, vol. 37, Aug. 2020, Art. no. 100267.
[19] 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 *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 135–146.
[20] (2012). *Giraph*. [Online]. Available: http://giraph.apache.org/
[21] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng, "HAMA: An efficient matrix computation with the mapreduce framework," in *Proc. IEEE 2nd Int. Conf. Cloud Comput. Technol. Sci.*, Nov. 2010, pp. 721–726.
[22] R. Chen, J. Shi, Y. Chen, B. Zang, H. Guan, and H. Chen, "Powerlyra: Differentiated graph computation and partitioning on skewed graphs," *ACM Trans. Parallel Comput.*, vol. 5, no. 3, p. 13, 2019.
[23] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proc. 10th USENIX Symp. Operating Syst. Design Implement. (OSDI)*, 2012, pp. 17–30.
[24] Y. Xing, Z. Chen, N. Xiao, F. Liu, and Y. Lu, "Graph analytics on many-core memory systems," *IEEE Access*, vol. 6, pp. 51429–51439, 2018.
[25] (2007). *Neo4j*. [Online]. Available: https://neo4j.com/
[26] (2017). *Tigergraph*. [Online]. Available: https://www.tigergraph.com/
[27] (2004). *Allegrograph*. [Online]. Available: https://franz.com/agraph/allegrograph/
[28] (2016). *Dgraph*. [Online]. Available: https://docs.dgraph.io/design-concepts
[29] S. Salihoglu and J. Widom, "GPS: A graph processing system," in *Proc. 25th Int. Conf. Sci. Stat. Database Manag.*, 2013, pp. 1–12.
[30] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-scale graph computation on just a PC," in *Proc. 10th USENIX Symp. Operating Syst. Design Implement. (OSDI)*, 2012, pp. 31–46.
[31] X. Zhu, W. Han, and W. Chen, "GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2015, pp. 375–386.
[32] W. Xie, G. Wang, D. Bindel, A. Demers, and J. Gehrke, "Fast iterative graph computation with block updates," *Proc. VLDB Endowment*, vol. 6, no. 14, pp. 2014–2025, Sep. 2013.
[33] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proc. 18th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2013, pp. 135–146.
[34] N. Sundaram, N. R. Satish, M. M. A. Patwary, S. R Dulloor, S. G. Vadlamudi, D. Das, and P. Dubey, "GraphMat: High performance graph analytics made productive," 2015, *arXiv:1503.07241*.
[35] Y. Chi, G. Dai, Y. Wang, G. Sun, G. Li, and H. Yang, "NXgraph: An efficient graph processing system on a single machine," in *Proc. IEEE 32nd Int. Conf. Data Eng. (ICDE)*, May 2016, pp. 409–420.
[36] S. Sumathi and S. Esakkirajan, *Fundamentals of Relational Database Management Systems*, vol. 47. Berlin, Germany: Springer, 2007.
[37] I. Robinson, J. Webber, and E. Eifrem, *Graph Databases: New Opportunities for Connected Data*. O'Reilly Media, 2015.
[38] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins, "A comparison of a graph database and a relational database: A data provenance perspective," in *Proc. 48th Annu. Southeast Regional Conf. (ACM SE)*, 2010, pp. 1–6.
[39] K. Andreev and H. Racke, "Balanced graph partitioning," *Theory Comput. Syst.*, vol. 39, no. 6, pp. 929–939, 2006.
[40] I. Holyer, "The NP-completeness of some edge-partition problems," *SIAM J. Comput.*, vol. 10, no. 4, pp. 713–717, Nov. 1981.
[41] G. Karypis, "METIS: Unstructured graph partitioning and sparse matrix ordering system," Tech. Rep., 1997.
[42] G. Echbarthi and H. Kheddouci, "Fractional greedy and partial restreaming partitioning: New methods for massive graph partitioning," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Oct. 2014, pp. 25–32.
[43] I. Stanton and G. Kliot, "Streaming graph partitioning for large distributed graphs," in *Proc. 18th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining (KDD)*, 2012, pp. 1222–1230.
[44] M. Li, H. Cui, C. Zhou, and S. Xu, "GAP: Genetic algorithm based large-scale graph partition in heterogeneous cluster," *IEEE Access*, vol. 8, pp. 144197–144204, 2020.
[45] H. Halberstam and R. Laxton, "Perfect difference sets," *Glasgow Math. J.*, vol. 6, no. 4, pp. 177–184, 1964.
[46] N. Jain, G. Liao, and T. L. Willke, "Graphbuilder: Scalable graph ETL framework," in *Proc. 1st Int. Workshop Graph Data Manage. Experiences Syst.*, 2013, p. 4.
[47] F. Petroni, L. Querzoni, K. Daudjee, S. Kamali, and G. Iacoboni, "HDRF: Stream-based partitioning for power-law graphs," in *Proc. 24th ACM Int. Conf. Inf. Knowl. Manage.*, Oct. 2015, pp. 243–252.
[48] C. Xie, L. Yan, W.-J. Li, and Z. Zhang, "Distributed power-law graph computing: Theoretical and empirical analysis," in *Proc. Adv. Neural Inf. Process. Syst.*, 2014, pp. 1673–1681.
[49] C. Zhang, F. Wei, Q. Liu, Z. G. Tang, and Z. Li, "Graph edge partitioning via neighborhood heuristic," in *Proc. 23rd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Aug. 2017, pp. 605–614.
[50] R. Mayer, K. Orujzade, and H.-A. Jacobsen, "Out-of-Core edge partitioning at linear run-time," 2022, *arXiv:2203.12721*.
[51] D. Dai, W. Zhang, and Y. Chen, "IOGP: An incremental online graph partitioning algorithm for distributed graph databases," in *Proc. 26th Int. Symp. High-Performance Parallel Distrib. Comput.*, Jun. 2017, pp. 219–230.
[52] 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 *Proc. 11th USENIX Symp. Operating Syst. Design Implement. (OSDI)*, 2014, pp. 599–613.
[53] T. Ayall, H. Duan, C. Liu, F. Gereme, and M. Deleli, "Offstreamng: Partial stream hybrid graph edge partitioning based on neighborhood expansion and greedy heuristic," in *Proc. Eur. Conf. Adv. Databases Inf. Syst.* Lyon, France: Springer, Aug. 2020, pp. 118–128.
[54] T. Ayall, H. Duan, C. Liu, F. Gereme, M. Abegaz, and M. Deleli, "Taking heuristic based graph edge partitioning one step ahead via OffStream partitioning approach," in *Proc. IEEE 37th Int. Conf. Data Eng. (ICDE)*, Apr. 2021, pp. 2081–2086.
[55] R. Mayer and H.-A. Jacobsen, "Hybrid edge partitioner: Partitioning large power-law graphs under memory constraints," in *Proc. Int. Conf. Manage. Data*, Jun. 2021, pp. 1289–1302.
[56] L. Vaquero, F. Cuadrado, D. Logothetis, and C. Martella, "XDGP: A dynamic graph processing system with adaptive partitioning," 2013, *arXiv:1309.1049*.
[57] N. T. Bao and T. Suzumura, "Towards highly scalable pregel-based graph processing platform with x10," in *Proc. 22nd Int. Conf. World Wide Web*, May 2013, pp. 501–508.
[58] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis, "Mizan: A system for dynamic load balancing in large-scale graph processing," in *Proc. 8th ACM Eur. Conf. Comput. Syst. (EuroSys)*, 2013, pp. 169–182.
[59] O. Batarfi, R. E. Shawi, A. G. Fayoumi, R. Nouri, S. M. R. Beheshti, A. Barnawi, and S. Sakr, "Large scale graph processing systems: Survey and an experimental evaluation," *Cluster Comput.*, vol. 18, no. 3, pp. 1189–1213, 2015.
[60] H.-N. Tran and E. Cambria, "A survey of graph processing on graphics processing units," *J. Supercomput.*, vol. 74, no. 5, pp. 2086–2115, May 2018.

[61] J. Huang, W. Qin, X. Wang, and W. Chen, "Survey of external memory large-scale graph processing on a multi-core system," *J. Supercomput.*, vol. 76, no. 1, pp. 549–579, Jan. 2020.
[62] M. Junghanns, A. Petermann, M. Neumann, and E. Rahm, "Management and analysis of big graph data: Current systems and open challenges," in *Handbook of Big Data Technologies*. Cham, Switzerland: Springer, 2017, pp. 457–505.
[63] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz, "Recent advances in graph partitioning," in *Algorithm Engineering*. Cham, Switzerland: Springer, 2016, pp. 117–158.
[64] S. Verma, L. M. Leslie, Y. Shin, and I. Gupta, "An experimental comparison of partitioning strategies in distributed graph processing," *Proc. VLDB Endowment*, vol. 10, no. 5, pp. 493–504, 2017.
[65] T. Ayall, H. Duan, and C. Liu, "Edge property based stream order reduce the performance of stream edge graph partition," *J. Phys., Conf.*, vol. 1395, no. 1, Nov. 2019, Art. no. 012010.
[66] H. Mykhailenko, F. Huet, and G. Neglia, "Comparison of edge partitioners for graph processing," in *Proc. Int. Conf. Comput. Sci. Comput. Intell. (CSCI)*, Dec. 2016, pp. 441–446.
[67] A. Pacaci and M. T. Özsu, "Experimental analysis of streaming algorithms for graph partitioning," in *Proc. Int. Conf. Manage. Data*, Jun. 2019, pp. 1375–1392.
[68] Z. Abbas, V. Kalavri, P. Carbone, and V. Vlassov, "Streaming graph partitioning: An experimental study," *Proc. VLDB Endowment*, vol. 11, no. 11, pp. 1590–1603, 2018.
[69] A. Pothen, "Graph partitioning algorithms with applications to scientific computing," in *Parallel Numerical Algorithms*. Dordrecht, The Netherlands: Springer, 1997, pp. 323–368.
[70] H.-J. Kim and Y.-H. Kim, "Recent progress on graph partitioning problems using evolutionary computation," 2018, *arXiv:1805.01623*.
[71] B. Hendrickson and T. G. Kolda, "Graph partitioning models for parallel computing," *Parallel Comput.*, vol. 26, no. 12, pp. 1519–1534, Nov. 2000.
[72] S. Arora, S. Rao, and U. Vazirani, "Geometry, flows, and graph-partitioning algorithms," *Commun. ACM*, vol. 51, no. 10, pp. 96–105, Oct. 2008.
[73] I. Safro, P. Sanders, and C. Schulz, "Advanced coarsening schemes for graph partitioning," *ACM J. Experim. Algorithmics*, vol. 19, pp. 1–24, Feb. 2015.
[74] K. Schloegel, G. Karypis, and V. Kumar, "Graph partitioning for high-performance scientific simulations," in *Sourcebook of Parallel Computing*. San Mateo, CA, USA: Morgan Kaufmann, 2003, pp. 491–541.
[75] A. Akhter, A.-C. N. Ngonga, and M. Saleem, "An empirical evaluation of RDF graph partitioning techniques," in *Proc. Eur. Knowl. Acquisition Workshop*. Cham, Switzerland: Springer, 2018, pp. 3–18.
[76] T. Chawla, G. Singh, E. S. Pilli, and M. C. Govil, "Storage, partitioning, indexing and retrieval in big RDF frameworks: A survey," *Comput. Sci. Rev.*, vol. 38, Nov. 2020, Art. no. 100309.
[77] M. Han, K. Daudjee, K. Ammar, M. T. Özsu, X. Wang, and T. Jin, "An experimental comparison of pregel-like graph processing systems," *Proc. VLDB Endowment*, vol. 7, no. 12, pp. 1047–1058, Aug. 2014.
[78] Y. Lu, J. Cheng, D. Yan, and H. Wu, "Large-scale distributed graph computing systems: An experimental evaluation," *Proc. VLDB Endowment*, vol. 8, no. 3, pp. 281–292, Nov. 2014.
[79] K. Ammar and T. Ozsu, "Experimental analysis of distributed graph systems," 2018, *arXiv:1806.08082*.
[80] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu, "The ubiquity of large graphs and surprising challenges of graph processing: Extended survey," *VLDB J.*, vol. 29, nos. 2–3, pp. 595–618, May 2020.
[81] C.-Y. Gui, L. Zheng, B. He, C. Liu, X.-Y. Chen, X.-F. Liao, and H. Jin, "A survey on graph processing accelerators: Challenges and opportunities," *J. Comput. Sci. Technol.*, vol. 34, no. 2, pp. 339–371, Mar. 2019.
[82] D. Dominguez-Sal, P. Urbón-Bayes, A. Giménez-Vanó, S. Gómez-Villamor, N. Martínez-Bazan, and J.-L. Larriba-Pey, "Survey of graph database performance on the HPC scalable graph analysis benchmark," in *Proc. Int. Conf. Web-Age Inf. Manage.* Berlin, Germany: Springer, 2010, pp. 37–48.
[83] S. Jouili and V. Vansteenberghe, "An empirical comparison of graph databases," in *Proc. Int. Conf. Social Comput.*, Sep. 2013, pp. 708–715.
[84] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson, "From 'think like a vertex 'to' think like a graph,'" *Proc. VLDB Endowment*, vol. 7, no. 3, pp. 193–204, 2013.
[85] F. Xia, J. Liu, H. Nie, Y. Fu, L. Wan, and X. Kong, "Random walks: A review of algorithms and applications," *IEEE Trans. Emerg. Topics Comput. Intell.*, vol. 4, no. 2, pp. 95–107, Apr. 2019.
[86] J. M. Kleinberg, "Authoritative sources in a hyperlinked environment," in *Proc. SODA*, vol. 98, 1998, pp. 668–677.
[87] A. Balmin, V. Hristidis, and Y. Papakonstantinou, "Objectrank: Authority-based keyword search in databases," *VLDB*, vol. 4, pp. 564–575, Aug. 2004.
[88] J. Hopcroft and R. Tarjan, "Algorithm 447: Efficient algorithms for graph manipulation," *Commun. ACM*, vol. 16, no. 6, pp. 372–378, 1973.
[89] U. Kang, C. E. Tsourakakis, A. P. Appel, C. Faloutsos, and J. Leskovec, "HADI: Mining radii of large graphs," *ACM Trans. Knowl. Discovery Data*, vol. 5, no. 2, pp. 1–24, Feb. 2011.
[90] S. Suri and S. Vassilvitskii, "Counting triangles and the curse of the last reducer," in *Proc. 20th Int. Conf. World Wide Web (WWW)*, 2011, pp. 607–614.
[91] M. Bayati, D. Shah, and M. Sharma, "Maximum weight matching via max-product belief propagation," in *Proc. Int. Symp. Inf. Theory (ISIT)*, Sep. 2005, pp. 1763–1767.
[92] V. Satuluri, S. Parthasarathy, and Y. Ruan, "Local graph sparsification for scalable clustering," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, 2011, pp. 721–732.
[93] Y. Liu, T. Safavi, A. Dighe, and D. Koutra, "Graph summarization methods and applications: A survey," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 1–34, May 2018.
[94] J. Chen, Y. Saad, and Z. Zhang, "Graph coarsening: From scientific computing to machine learning," *SeMA J.*, vol. 79, no. 1, pp. 187–223, Mar. 2022.
[95] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359–392, Aug. 1999.
[96] C. Martella, D. Logothetis, A. Loukas, and G. Siganos, "Spinner: Scalable graph partitioning in the cloud," in *Proc. IEEE 33rd Int. Conf. Data Eng. (ICDE)*, Apr. 2017, pp. 1083–1094.
[97] R. Andersen, F. Chung, and K. Lang, "Local graph partitioning using pagerank vectors," in *Proc. 47th Annu. IEEE Symp. Found. Comput. Sci. (FOCS)*, Oct. 2006, pp. 475–486.
[98] L. Hagen and A. B. Kahng, "New spectral methods for ratio cut partitioning and clustering," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 11, no. 9, pp. 1074–1085, Sep. 1992.
[99] J. Shi and J. Malik, "Normalized cuts and image segmentation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 22, no. 8, pp. 888–905, Aug. 2000.
[100] G. Palubeckis, "Metaheuristic approaches for ratio cut and normalized cut graph partitioning," *Memetic Comput.*, vol. 14, pp. 1–33, Apr. 2022.
[101] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Proc. SIAM Int. Conf. Data Mining*, Apr. 2004, pp. 442–446.
[102] J. Leskovec and A. Krevl. (Jun. 2014). *SNAP Datasets: Stanford Large Network Dataset Collection*. [Online]. Available: http://snap.stanford.edu/data
[103] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and analysis of online social networks," in *Proc. 7th ACM SIGCOMM Conf. Internet Meas. (IMC)*, 2007, pp. 29–42.
[104] J. Kunegis, "Konect: The Koblenz network collection," in *Proc. 22nd Int. Conf. World Wide Web*, 2013, pp. 1343–1350.
[105] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" in *Proc. 19th Int. Conf. World Wide Web (WWW)*, 2010, pp. 591–600.
[106] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *Knowl. Inf. Syst.*, vol. 42, no. 1, pp. 181–213, Jan. 2015.
[107] *Grouplens*. Accessed: Feb. 20, 2022. [Online]. Available: https://grouplens.org/datasets/movielens/
[108] B. Hendrickson and R. Leland, "A multi-level algorithm for partitioning graphs," in *Proc. SC Conf. (SC)*, vol. 95, Dec. 1995, pp. 1–14.
[109] A. Valejo, V. Ferreira, R. Fabbri, M. C. F. D. Oliveira, and A. D. A. Lopes, "A critical survey of the multilevel method in complex networks," *ACM Comput. Surv.*, vol. 53, no. 2, pp. 1–35, Mar. 2020.
[110] B. Monien, R. Preis, and R. Diekmann, "Quality matching and local improvement for multilevel graph-partitioning," *Parallel Comput.*, vol. 26, no. 12, pp. 1609–1634, Nov. 2000.
[111] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell Syst. Tech. J.*, vol. 49, no. 2, pp. 291–307, 1970.
[112] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," in *Proc. 19th Design Autom. Conf.*, 1982, pp. 175–181.

[113] F. Pellegrini and J. Roman, "Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs," in *Proc. Int. Conf. High-Performance Comput. Netw.* Berlin, Germany: Springer, 1996, pp. 493–498.
[114] B. Hendrickson and R. Leland, "The Chaco users guide. version 1.0," Sandia Nat. Labs., Albuquerque, NM, USA, Tech. Rep. SAND-93-2339, 1993.
[115] P. Sanders and C. Schulz, "KaHIP v3.00—Karlsruhe high quality partitioning–user guide," 2013, *arXiv:1311.1714*.
[116] D. Lasalle and G. Karypis, "Multi-threaded graph partitioning," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process.*, May 2013, pp. 225–236.
[117] Y. Akhremtsev, P. Sanders, and C. Schulz, "High-quality shared-memory graph partitioning," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 11, pp. 2710–2722, Nov. 2020.
[118] S. Gregory, "Finding overlapping communities in networks by label propagation," *New J. Phys.*, vol. 12, no. 10, Oct. 2010, Art. no. 103018.
[119] G. Karypis, K. Schloegel, and V. Kumar, "ParMETIS: Parallel graph partitioning and sparse matrix ordering library," Dept. Comput. Sci., Univ. Minnesota, Minneapolis, MN, USA, Tech. Rep., 1997.
[120] C. Chevalier and F. Pellegrini, "PT-scotch: A tool for efficient parallel graph ordering," *Parallel Comput.*, vol. 34, nos. 6–8, pp. 318–331, Jul. 2008.
[121] M. Holtgrewe, P. Sanders, and C. Schulz, "Engineering a scalable high quality graph partitioner," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. (IPDPS)*, Apr. 2010, pp. 1–12.
[122] C. Walshaw and M. Cross, "JOSTLE: Parallel multilevel graph-partitioning software—An overview," *Mesh Partitioning Techn. Domain Decomposition Techn.*, vol. 10, pp. 27–58, Jan. 2007.
[123] F. Rahimian, A. H. Payberah, S. Girdzijauskas, M. Jelasity, and S. Haridi, "JA-BE-JA: A distributed algorithm for balanced graph partitioning," in *Proc. IEEE 7th Int. Conf. Self-Adaptive Self-Organizing Syst.*, Sep. 2013, pp. 51–60.
[124] J. Ugander and L. Backstrom, "Balanced label propagation for partitioning massive graphs," in *Proc. 6th ACM Int. Conf. Web Search Data Mining*, 2013, pp. 507–516.
[125] T. Chen and B. Li, "A distributed graph partitioning algorithm for processing large graphs," in *Proc. IEEE Symp. Service-Oriented Syst. Eng. (SOSE)*, Mar. 2016, pp. 53–59.
[126] G. M. Slota, S. Rajamanickam, K. Devine, and K. Madduri, "Partitioning trillion-edge graphs in minutes," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2017, pp. 646–655.
[127] H. Meyerhenke, P. Sanders, and C. Schulz, "Parallel graph partitioning for complex networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 9, pp. 2625–2638, Sep. 2017.
[128] Y. Li, C. Constantin, and C. du Mouza, "SGVCut: A vertex-cut partitioning tool for random walks-based computations over social network graphs," in *Proc. 29th Int. Conf. Sci. Stat. Database Manage.*, Jun. 2017, pp. 1–4.
[129] F. Rahimian, A. H. Payberah, S. Girdzijauskas, and S. Haridi, "Distributed vertex-cut partitioning," in *Proc. IFIP Int. Conf. Distrib. Appl. Interoperable Syst.* Berlin, Germany: Springer, 2014, pp. 186–200.
[130] A. Guerrieri and A. Montresor, "DFEP: Distributed funding-based edge partitioning," in *Proc. Eur. Conf. Parallel Process.* Berlin, Germany: Springer, 2015, pp. 346–358.
[131] D. Margo and M. Seltzer, "A scalable distributed graph partitioner," *Proc. VLDB Endowment*, vol. 8, no. 12, pp. 1478–1489, 2015.
[132] M. Hanai, T. Suzumura, W. J. Tan, E. Liu, G. Theodoropoulos, and W. Cai, "Distributed edge partitioning for trillion-edge graphs," *Proc. VLDB Endowment*, vol. 12, no. 13, pp. 2379–2392, Sep. 2019.
[133] G. Karypis and V. Kumar, "A parallel algorithm for multilevel graph partitioning and sparse matrix ordering," *J. Parallel Distrib. Comput.*, vol. 48, no. 1, pp. 71–95, Jan. 1998.
[134] S. Kirmani and P. Raghavan, "Scalable parallel graph partitioning," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2013, pp. 1–10.
[135] G. M. Slota, K. Madduri, and S. Rajamanickam, "PuLP: Scalable multi-objective multi-constraint partitioning for small-world networks," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Oct. 2014, pp. 481–490.
[136] M. Kim and K. S. Candan, "SBV-cut: Vertex-cut based graph partitioning using structural balance vertices," *Data Knowl. Eng.*, vol. 72, pp. 285–303, Feb. 2012.
[137] S. Schlag, C. Schulz, D. Seemaier, and D. Strash, "Scalable edge partitioning," in *Proc. 21st Workshop Algorithm Eng. Exp. (ALENEX)*, 2019, pp. 211–225.
[138] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, "FENNEL: Streaming graph partitioning for massive scale graphs," in *Proc. 7th ACM Int. Conf. Web Search Data Mining*, Feb. 2014, pp. 333–342.
[139] W. Zhang, Y. Chen, and D. Dai, "AKIN: A streaming graph partitioning algorithm for distributed graph storage systems," in *Proc. 18th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput. (CCGRID)*, May 2018, pp. 183–192.
[140] J. Nishimura and J. Ugander, "Restreaming graph partitioning: Simple versatile algorithms for advanced balancing," in *Proc. 19th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2013, pp. 1106–1114.
[141] H. C. Rad and R. Azmi, "CLDA: Vertex-cut partitioning for streaming power-law graphs," in *Proc. 9th Int. Conf. Inf. Knowl. Technol. (IKT)*, Oct. 2017, pp. 104–110.
[142] C. Hu, J. Zhong, Q. Li, and Q. Li, "DETER: Streaming graph partitioning via combined degree and cluster information," in *Proc. Int. Conf. Algorithms Architectures Parallel Process.* Cham, Switzerland: Springer, 2019, pp. 242–255.
[143] Q.-S. Hua, Y. Li, D. Yu, and H. Jin, "Quasi-streaming graph partitioning: A game theoretical approach," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 7, pp. 1643–1656, Jul. 2019.
[144] C. Mayer, R. Mayer, M. A. Tariq, H. Geppert, L. Laich, L. Rieger, and K. Rothermel, "ADWISE: Adaptive window-based streaming edge partitioning for high-speed graph processing," in *Proc. IEEE 38th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jul. 2018, pp. 685–695.
[145] M. Taimouri and H. Saadatfar, "RBSEP: A reassignment and buffer based streaming edge partitioning approach," *J. Big Data*, vol. 6, no. 1, pp. 1–17, Dec. 2019.
[146] Y. Li, C. Li, A.-C. Orgerie, and P. R. Parvedy, "WSGP: A window-based streaming graph partitioning approach," in *Proc. IEEE/ACM 21st Int. Symp. Cluster, Cloud Internet Comput. (CCGrid)*, May 2021, pp. 586–595.
[147] R. Mayer, K. Orujzade, and H.-A. Jacobsen, "2PS: High-quality edge partitioning with two-phase streaming," 2020, *arXiv:2001.07086*.
[148] D. Kong, X. Xie, and Z. Zhang, "Clustering-based partitioning for large web graphs," 2022, *arXiv:2201.00472*.
[149] A. Hollocou, J. Maudet, T. Bonald, and M. Lelarge, "A streaming algorithm for graph clustering," 2017, *arXiv:1712.04337*.
[150] D. Choi, J. Han, J. Lim, J. Han, K. Bok, and J. Yoo, "Dynamic graph partitioning scheme for supporting load balancing in distributed graph environments," *IEEE Access*, vol. 9, pp. 65254–65265, 2021.
[151] A. Zaki, M. Attia, D. Hegazy, and S. Amin, "Comprehensive survey on dynamic graph models," *Int. J. Adv. Comput. Sci. Appl.*, vol. 7, no. 2, pp. 1–10, 2016.
[152] N. Xu, L. Chen, and B. Cui, "LogGP: A log-based dynamic graph partitioning method," *Proc. VLDB Endowment*, vol. 7, no. 14, pp. 1917–1928, Oct. 2014.
[153] D. Nicoara, S. Kamali, K. Daudjee, and L. Chen, "Hermes: Dynamic partitioning for distributed social network graph databases," in *Proc. EDBT*, 2015, pp. 25–36.
[154] M. Predari and A. Esnard, "A K-Way greedy graph partitioning with initial fixed vertices for parallel applications," in *Proc. 24th Euromicro Int. Conf. Parallel, Distrib., Netw.-Based Process. (PDP)*, Feb. 2016, pp. 280–287.
[155] C. Sakouhi, S. Aridhi, A. Guerrieri, S. Sassi, and A. Montresor, "DynamicDFEP: A distributed edge partitioning approach for large dynamic graphs," in *Proc. 20th Int. Database Eng. Appl. Symp. (IDEAS)*, 2016, pp. 142–147.
[156] C. Mayer, M. A. Tariq, R. Mayer, and K. Rothermel, "GrapH: Traffic-aware graph processing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 6, pp. 1289–1302, Jun. 2018.
[157] D. Kumar, A. Raj, and J. Dharanipragada, "Graphsteal: Dynamic re-partitioning for efficient graph processing in heterogeneous clusters," in *Proc. IEEE 10th Int. Conf. Cloud Comput. (CLOUD)*, Jun. 2017, pp. 439–446.
[158] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.
[159] (2006). *Hadoop*. [Online]. Available: http://hadoop.apache.org/
[160] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Distributed GraphLab: A framework for machine learning in the cloud," 2012, *arXiv:1204.6078*.
[161] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl, "Spinning fast iterative data flows," 2012, *arXiv:1208.0088*.

[162] C. Xie, R. Chen, H. Guan, B. Zang, and H. Chen, "Sync or async: Time to fuse for distributed graph-parallel computation," *ACM SIGPLAN Notices*, vol. 50, no. 3, pp. 194–204, 2015.

[163] Y. Simmhan, A. Kumbhare, C. Wickramaarachchi, S. Nagarkar, S. Ravi, C. Raghavendra, and V. Prasanna, "GoFFish: A sub-graph centric framework for large-scale graph analytics," in *Proc. Eur. Conf. Parallel Process.* Cham, Switzerland: Springer, 2014, pp. 451–462.

[164] R. Cyganiak, D. Wood, M. Lanthaler, G. Klyne, J. J. Carroll, and B. McBride, "RDF 1.1 concepts and abstract syntax," *W3C Recommendation*, vol. 25, no. 2, pp. 1–22, 2014.

[165] A. Davoudian, L. Chen, and M. Liu, "A survey on NoSQL stores," *ACM Comput. Surv.*, vol. 51, no. 2, pp. 1–43, Mar. 2019.

[166] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst. (TOCS)*, vol. 26, no. 2, pp. 1–26, 2008.

[167] A. B M Moniruzzaman and S. A. Hossain, "NoSQL database: New era of databases for big data analytics–classification, characteristics and comparison," 2013, *arXiv:1307.0191*.

[168] B. M. Abdelhafiz and M. Elhadef, "Sharding database for fault tolerance and scalability of data," in *Proc. 2nd Int. Conf. Comput., Autom. Knowl. Manage. (ICCAKM)*, Jan. 2021, pp. 17–24.

[169] C. H. Costa, J. V. B. Filho, P. H. M. Maia, and F. Carlos, "Sharding by hash partitioning," in *Proc. 17th Int. Conf. Enterprise Inf. Syst.*, 2015, pp. 313–320.

[170] D. Kuhn and T. Kyte, "Architecture overview," in *Expert Oracle Database Architecture*. Berkeley, CA, USA: Apress, 2022, pp. 83–107.

[171] G. Harrison and M. Harrison, "Sharding," in *MongoDB Performance Tuning*. Berkeley, CA, USA: Apress, 2021, pp. 315–342.

[172] M. Indrawan-Santiago, "Database research: Are we at a crossroad? Reflection on NoSQL," in *Proc. 15th Int. Conf. Netw.-Based Inf. Syst.*, Sep. 2012, pp. 45–51.

[173] J. Pérez, M. Arenas, and C. Gutierrez, "Semantics and complexity of SPARQL," *ACM Trans. Database Syst.*, vol. 34, no. 3, pp. 1–45, Aug. 2009.

[174] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor, "Cypher: An evolving query language for property graphs," in *Proc. Int. Conf. Manage. Data*, May 2018, pp. 1433–1445.

[175] M. A. Rodriguez, "The gremlin graph traversal machine and language (invited talk)," in *Proc. 15th Symp. Database Program. Lang.*, Oct. 2015, pp. 1–10.

[176] O. Hartig and J. Pérez, "Semantics and complexity of GraphQL," in *Proc. World Wide Web Conf.*, 2018, pp. 1155–1164.

[177] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter, and D. Vrgoč, "Foundations of modern query languages for graph databases," *ACM Comput. Surv.*, vol. 50, no. 5, pp. 1–40, Sep. 2017.

[178] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "PEGASUS: A peta-scale graph mining system implementation and observations," in *Proc. 9th IEEE Int. Conf. Data Mining*, Dec. 2009, pp. 229–238.

[179] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "HaLoop: Efficient iterative data processing on large clusters," *Proc. VLDB Endowment*, vol. 3, nos. 1–2, pp. 285–296, 2010.

[180] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: A runtime for iterative mapreduce," in *Proc. 19th ACM Int. Symp. High Perform. Distrib. Comput.*, 2010, pp. 810–818.

[181] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "IMapReduce: A distributed computing framework for iterative computation," *J. Grid. Comput.*, vol. 10, no. 3, pp. 47–68, 2012.

[182] R. Chen, X. Weng, B. He, and M. Yang, "Large graph processing in the cloud," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Jun. 2010, pp. 1123–1126.

[183] Y. Bu, V. Borkar, J. Jia, M. J. Carey, and T. Condie, "Pregelix: Big(ger) graph analytics on a dataflow engine," 2014, *arXiv:1407.0455*.

[184] R. Chen, X. Ding, P. Wang, H. Chen, B. Zang, and H. Guan, "Computation and communication efficient graph processing with distributed immutable view," in *Proc. 23rd Int. Symp. High-Performance Parallel Distrib. Comput. (HPDC)*, 2014, pp. 215–226.

[185] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proc. 2nd ACM SIGOPS/EuroSys Eur. Conf. Comput. Syst.*, 2007, pp. 59–72.

[186] D. Yan, J. Cheng, Y. Lu, and W. Ng, "Effective techniques for message reduction and load balancing in distributed graph computation," in *Proc. 24th Int. Conf. World Wide Web*, 2015, pp. 1307–1317.

[187] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, and C. Guestrin, "GraphLab: A distributed framework for machine learning in the cloud," 2011, *arXiv:1107.0922*.

[188] Y. Zhang, D. Li, C. Zhang, J. Wang, and L. Liu, "GraphA: Efficient partitioning and storage for distributed graph computation," *IEEE Trans. Services Comput.*, vol. 14, no. 1, pp. 155–166, Jan. 2017.

[189] M. Zhang, Y. Wu, K. Chen, X. Qian, X. Li, and W. Zheng, "Exploring the hidden dimension in graph processing," in *Proc. 12th USENIX Symp. Operating Syst. Design Implement. (OSDI)*, 2016, pp. 285–300.

[190] Y. Zhuo, J. Chen, Q. Luo, Y. Wang, H. Yang, D. Qian, and X. Qian, "SympleGraph: Distributed graph processing with precise loop-carried dependency guarantee," *Bioinformatics*, vol. 1, no. 14, p. 29, 2020.

[191] Y. Zhang, H. Wang, M. Jia, J. Wang, D. Li, G. Xue, and K.-L. Tan, "TopoX: Topology refactorization for minimizing network communication in graph computations," *IEEE/ACM Trans. Netw.*, vol. 28, no. 6, pp. 2768–2782, Dec. 2020.

[192] V. Leis, A. Kemper, and T. Neumann, "The adaptive radix tree: ARTful indexing for main-memory databases," in *Proc. IEEE 29th Int. Conf. Data Eng. (ICDE)*, Apr. 2013, pp. 38–49.

[193] D. Yan, J. Cheng, Y. Lu, and W. Ng, "Blogel: A block-centric framework for distributed computation on real-world graphs," *Proc. VLDB Endowment*, vol. 7, no. 14, pp. 1981–1992, 2014.

[194] M. Randles, D. Lamb, and A. Taleb-Bendiab, "A comparative study into distributed load balancing algorithms for cloud computing," in *Proc. IEEE 24th Int. Conf. Adv. Inf. Netw. Appl. Workshops*, 2010, pp. 551–556.

[195] Y. Zhao, K. Yoshigoe, M. Xie, S. Zhou, R. Seker, and J. Bian, "LightGraph: Lighten communication in distributed graph-parallel processing," in *Proc. IEEE Int. Congr. Big Data*, Apr. 2014, pp. 717–724.

[196] P. Wang, K. Zhang, R. Chen, H. Chen, and H. Guan, "Replication-based fault-tolerance for large-scale graph processing," in *Proc. 44th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, Jun. 2014, pp. 562–573.

[197] Y. Zhao, K. Yoshigoe, M. Xie, J. Bian, and K. Xiong, "L-PowerGraph: A lightweight distributed graph-parallel communication mechanism," *J. Supercomput.*, vol. 76, no. 3, pp. 1850–1879, Mar. 2020.

[198] S. Hong, S. Depner, T. Manhardt, J. Van Der Lugt, M. Verstraaten, and H. Chafi, "PGX. D: A fast distributed graph processing engine," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2015, pp. 1–12.

[199] K. Zhang, R. Chen, and H. Chen, "NUMA-aware graph-structured analytics," in *Proc. 20th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, Jan. 2015, pp. 183–193.

[200] Y. Zhang, X. Liao, H. Jin, L. Gu, L. He, B. He, and H. Liu, "CGraph: A correlations-aware approach for efficient concurrent iterative graph processing," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2018, pp. 441–452.

[201] K. Wang, G. Xu, Z. Su, and Y. D. Liu, "GraphQ: Graph query processing with abstraction refinement—Scalable and programmable analytics over very large graphs on a single PC," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2015, pp. 387–401.

[202] Z. Lin, M. Kahng, K. M. Sabrin, D. H. P. Chau, H. Lee, and U. Kang, "MMap: Fast billion-scale graph computation on a PC via memory mapping," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Oct. 2014, pp. 159–164.

[203] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim, "Mosaic: Processing a trillion-edge graph on a single machine," in *Proc. 12th Eur. Conf. Comput. Syst.*, Apr. 2017, pp. 527–543.

[204] (2006). *Blazegraph*. [Online]. Available: https://www.blazegraph.com/

[205] (2010). *Orientdb*. [Online]. Available: https://orientdb.com

[206] (2010). *Stardog*. [Online]. Available: https://www.stardog.com

[207] (2012). *Arangodb*. [Online]. Available: https://docs.arangodb.com/

[208] (2014). *Faunadb*. [Online]. Available: https://fauna.com/

[209] (2017). L. Foundation. *Janusgraph*. [Online]. Available: http://janusgraph.org/

[210] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, "A comprehensive survey on graph neural networks," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 32, no. 1, pp. 4–24, Mar. 2020.

[211] C. He, K. Balasubramanian, E. Ceyani, C. Yang, H. Xie, L. Sun, L. He, L. Yang, P. S. Yu, Y. Rong, P. Zhao, J. Huang, M. Annavaram, and S. Avestimehr, "FedGraphNN: A federated learning system and benchmark for graph neural networks," 2021, *arXiv:2104.07145*.

TEWODROS ALEMU AYALL received the B.Sc. degree in computer science from University Gondar, Ethiopia, in 2010, the M.Sc. degree in computer science from Andhra University, India, in 2015, and the Ph.D. degree in computer Science and technology from the School of Computer Science and Engineering, University of Electronic Science and Technology of China (UESTC), in 2021. He is currently a Postdoctoral Researcher with Zhejiang Normal University. His research interests include distributed graph computing, big data processing, big graph partitioning, and graph machine learning.

HUAWEN LIU received the M.S. and Ph.D. degrees in computer science from Jilin University, Changchun, China, in 2007 and 2010, respectively. He is currently a Visiting Scholar with The University of Texas at San Antonio, San Antonio, TX, USA. He is also a Professor with the Department of Computer Science, Zhejiang Normal University, Jinhua, China. His research interests include feature selection, sparse learning, and machine learning.

CHANGJUN ZHOU was born in Shangrao, China, in 1977. He received the Ph.D. degree in mechanical design and theory from the School of Mechanical Engineering, Dalian University of Technology, Dalian, in 2008. He is currently a Professor with Zhejiang Normal University. He has published 60 articles in his research areas. His research interests include pattern recognition and intelligence computing, and DNA computing.

ABEGAZ MOHAMMED SEID (Member, IEEE) received the B.Sc. degree in computer science from Ambo University, Ethiopia, in 2010, and the M.Sc. degree from Addis Ababa University, Ethiopia, in 2015, and the Ph.D. degree in computer science and technology from the School of Computer Science and Engineering, University of Electronic Science and Technology of China (UESTC), in 2021. He is currently a Postdoctoral Fellow with the College of Science and Engineering, Hamad Bin Khalifa University, Doha, Qatar. His research interests include wireless networks, mobile edge computing, fog computing, UAV networks, the IoT, 5G wireless networks, and graph partitioning.

FANTAHUN BOGALE GEREME received the B.Sc. degree in computer science and IT from Haramaya University, Ethiopia, in 2006, the M.Sc. degree in computer science from Osmania University, India, in 2013, and the Ph.D. degree in computer science and technology from the Institute of Fundamental and Frontier Sciences, University of Electronic Science and Technology of China (UESTC), in 2021. His research interest includes intelligence computing.

HAYLA NAHOM ABISHU (Graduate Student Member, IEEE) received the B.Sc. degree in computer science and information technology from Haramaya University, in 2007, and the M.Sc. degree in computer science and networking from Dilla University, Ethiopia, in 2017. He is currently pursuing the Ph.D. degree in computer science and technology with the University of Electronic Science and Technology of China. His research interests include mobile computing, wireless networks, blockchain, UAV networks, the IoT, network security, and machine learning.

YASIN HABTAMU YACOB received the B.Sc. degree in computer science and information technology from Haramaya University, in 2007, and the M.Sc. degree in computer science and networking from Dilla University, Ethiopia, in 2017. He is currently pursuing the Ph.D. degree in computer science and technology with the University of Electronic Science and Technology of China. His research interests include mobile computing, wireless networks, blockchain, UAV networks, the IoT, network security, and machine learning.