# A survey of current challenges in partitioning and processing of graph-structured data in parallel and distributed systems

**Article** *in* Distributed and Parallel Databases · June 2020

DOI: 10.1007/s10619-019-07276-9

CITATIONS
19

READS
1,104

**5 authors**, including:

Hamilton Wilfried Yves Adoni
Helmholtz-Zentrum Dresden-Rossendorf
**43** PUBLICATIONS **251** CITATIONS
SEE PROFILE

Tarik Nahhal
Faculty of Science, Hassan II University
**43** PUBLICATIONS **259** CITATIONS
SEE PROFILE

Moez Krichen
Albaha University, Albaha, Saudi Arabia
**210** PUBLICATIONS **3,046** CITATIONS
SEE PROFILE

Brahim Aghezzaf
Université Hassan II de Casablanca
**54** PUBLICATIONS **623** CITATIONS
SEE PROFILE

Distributed and Parallel Databases
https://doi.org/10.1007/s10619-019-07276-9

# A survey of current challenges in partitioning and processing of graph-structured data in parallel and distributed systems

**Hamilton Wilfried Yves Adoni[1] · Tarik Nahhal[1] · Moez Krichen[2,3] · Brahim Aghezzaf[1] · Abdeltif Elbyed[1]**



**Abstract**
One of the concepts that attracts attention since entering of big data era is the graph-structured data. Suitable frameworks to handle such data would face several constraints, especially scalability, partitioning challenges, processing complexity and hardware configurations. Unfortunately, although several works deal with big data issues, there is a lack of literature review concerning the challenges related to query answering on large-scale graph data. In this survey paper, we review current problems related to the partitioning and processing of graph-structured data. We discuss existing graph processing systems and provide some insights to know how to choose the right system for parallel and distributed processing of large-scale graph data. Finally, we survey current open challenges in this field.

**Keywords** Large-scale graph · Big Data · Graph processing system · Graph partitioning · Distributed computing

✉ Hamilton Wilfried Yves Adoni
adoniwilfried@gmail.com

Tarik Nahhal
t.nahhal@fsac.ac.ma

Moez Krichen
moez.krichen@redcad.org

Brahim Aghezzaf
b.aghezzaf@fsac.ac.ma

Abdeltif Elbyed
abdeltif.elbyed@univh2c.ma

1 Faculty of sciences, Hassan II University of Casablanca, Casablanca, Morocco

2 Faculty of CSIT, Albaha University, Albaha, Saudi Arabia

3 ReDCAD Laboratory, University of Sfax, Sfax, Tunisia

Published online: 12 November 2019

## 1 Introduction

Big graph data (big data + graph algorithms) refers to NoSQL technology whose data are stored in a graph so called "large-scale graph". It is a connection of vertices (information) and edges (relationships) that connect pair of vertices together. They serve in particular to represent a complex network such as social networks, road networks, migratory flux, Very-Large-Scale Integration of circuit (VLSI), DNA or protein interaction networks. These data are heterogeneous because of the variety of information associated to the vertices.

Another important point is that, the graph can be dynamic or be a subject of frequent modifications: add, delete or update of vertices and edges. This high volume of changes can be detected in social networks. For example, in 2013 much of data that powering Facebook database changed continuously with a ratio of more than 86 thousands per second [1]. This highlights the importance of the velocity and the scalability of the graph algorithms. Indeed, we talk about large-scale graph when it becomes difficult to process the data into the graph in reasonable time, or when the data flow is faster than the velocity of the computing program. The graph size is not the only criterion in deciding if a graph is a large-scale graph. In paper presented in 1998 [2], the authors looked at three other metrics: the average distance, the clustering coefficient and the power-law degree distribution.

***Average distance*** in large-scale graph, the average path length between all connected vertices is small. This is not surprising since Jeffrey Travers and Stanley Milgran [3] showed by experimental study that it was possible to connect random people in the world network by using on average five connections. For example, in 2011 the expected distance between two vertices of Facebook's social graph was 4.74 [1].

***The clustering coefficient*** big graph has a very high clustering coefficient. The clustering measures the number of *3-edge triangles* in the graph. Three vertices form a triangle if two of them are connected and share a common neighbor. Graphs with high clustering coefficient have a larger number of triangle count.

***The power-law degree distribution*** the most notable characteristic in large-scale graph, is the observation of *scale-free network* which can be approximated by the power-law degree distribution [4,5]. An important feature of some scale-free network is the presence of strongly connected components between them and relatively less connected to the rest of the network. In big graph, there is a larger number of vertices with small degree and very few probability $P(k)$ of vertices in the graph with high degree of $k$ connections to other vertices following $P(k) \sim k^{-\lambda}$ such that $2 \leq \lambda \leq 3$. Figure 1 highlights the power-law of three complex networks. Top 0.5% of the vertices are adjacent or link a large number of vertices in the networks. The power-law distribution degree underlines the impact of vertices (hub-vertices) with degrees that greatly exceed the average degree on the topology of the graph.

The concept of graph-structured data solves several problems because it provides a capability to manipulate objects and the interactions between them. Graphs are ubiquitous and can represent connections between people, social network, DNA network and protein interactions. Indeed the concept of graphs is not new, and was introduced for

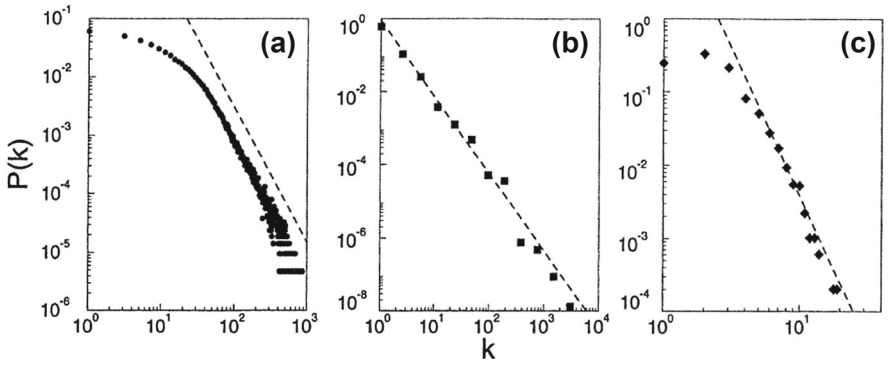


**Fig. 1** The power-law degree distribution of various scale-free networks from [4]: **a** actor collaboration network; **b** world wide web network; **c** power grid network

the first time in 1735 by the Swiss mathematician Leonhard Euler[1] to propose a mathematical solution to the problem of the Seven Bridges of Königsberg. Since, graphs stimulate more interests for many problems such as vehicle routing problem, traveler salesman problem and other graph traversal problems. But the field of applications was limited to small dimension problems.

With the coming of web, social networks and the emergence of new technologies like big data, Internet of Thing (IoT), 5G and Industry 4.0, the data explosion attracts more interest for graph databases because they also support agility. The treatment of very large datasets with traditional approach is very complex because of the 4V challenges (Volume, Velocity, Veracity and Variety) related to the phenomenon of big data. They are asked as soon as the volume of data becomes unmanageable and responding to simple queries takes hours. In this sense, a new way to store naturally and manage effectively the data consists of considering them as graph data whose vertices represent information, and edges the different interactions between them. The ability to easily link different kinds of related information makes the graph data a most promising approach for handling and analyzing of big data.

Many big data applications based on graph data models have emerged in the last decade [6–11]. An obvious example is the shortest path computation in large-scale graph [12], the authors proposed a parallel and distributed graph traversal algorithm based on Hadoop MapReduce framework in order to reduce considerably the time complexity. Aridhi et al [13] presented an efficient graph mining algorithms to detect the approximate number of subgraphs in large-scale graph. Another example is the PageRank's algorithm [14,15], the network of the world wide web can be presented as a web graph in which the web pages refer to the vertices and the links are the edges [16]. PageRank's algorithm computes the scoring of each web page and assigns a rank based on the topology of the web graph in order to measure the popularity of web pages. This algorithm is the key success of Google search engine which allows to classify the results of the web pages.

[1] http://eulerarchive.maa.org/.

Other example is the clustering data, given a set of data, the main objective is to group them into subsets of data that share common characteristics, which most often corresponds to the criteria of similarity or proximity that is defined by introducing measures and classes of distance between objects. In the context of graph data, the clustering consists of finding subgraphs of related vertices that share same properties. In case of social graph, by applying the clustering algorithms such as the well-known k-means based clustering and centrality based clustering [17], we can easily compute the task of community detection within graph [18,19]. Another case is the mining of scientific paper citations [20], the set of documents can be modeled as a citation graph (or citation network) in which each vertex corresponds to a paper and the edges refer to the citation relationships from one paper to another. By clustering the data, we can detect a set of subgraphs or communities of relevant scientific works for a given paper of the graph.

Big graph applications play an important role in the analysis of complex phenomenons emanating from various domains [12,21–23]. For example, we can build, visualize and understand large-scale Bayesian networks [24]. In biologic [11], the regulation of gene expression, the metabolic pathway linked to series of chemical reactions and the protein-to-protein interaction study to reach an important discovery and development can generate a large-scale dataset that needs to be processed and understood. On the other hand, we can compute the shortest path in large road network by considering heterogeneous data as a graph data [9].

According to recent study [25], the total amount of world data produced was 4.4 zettabytes in 2014 and that is set to reach to 44 zettabytes by 2020. In this context, the size of the network will increase and their analysis will became problematic. This great challenge highlights the importance of partitioning the large-scale networks.

Indeed, the graph partitioning problem brings together strategies emanating from graph theory allowing to partition the graph data under subgraphs of data based on various criteria. Generally, the criteria may vary from one partitioning method to another, but the main objectives are known. The graph partitioning problem is NP-Hard [26,27]: (1) the partitioning approach does not guarantee the optimality of the solution, (2) the problem can not be solved effectively in a polynomial time, finally (3) when the graph grows bigger, it becomes difficult and cost-ineffective to compute in a single machine because the computation task is memory-intensive. This can result in serious performance bottlenecks and takes impracticable time for task achievement when the graph size is too large to fit into memory. To face these challenges, many works have been performed to propose a parallel and distributed frameworks for large-scale graphs processing [28]. In this paper, we will focus on the problem of large-scale graph partitioning and different graph partitioning algorithms. In the second half, we will show how this new category of distributed graph processing systems can be used to implement graph partitioning algorithms.

**Contributions** In this paper, we formalize the graph partitioning problem and we draw up a comparative study of existing methods and systems to perform efficiently on large-scale graphs. In particular, the main contributions are as follows.

Concepts and problem formulation We first identify the main criteria allowing identification of big graph data. We highlight the complexity and main challenges related

to the partition of large-scale graphs. Then we proposed a mathematical model of the graph partitioning problem. The proposed model is based on an objective function that satisfies the constraints of load balancing and cross-edges between partitions.

`Existing methods and systems` We provide useful information in the state-of-the-art about the graph partitioning algorithms and the existing graph processing systems. The graph partitioning algorithms are classified into 8 major categories.

`Experimental evaluation` We evaluate the performances and draw up a comparative study of each partitioning method and graph system listed in this paper. The experimental evaluation was driven on a cluster of ten machines with various types of real-world graphs. For each graph partitioning method, we evaluate the runtime, the network bandwidth and the load balancing of the cluster.

`Suggestions` Based on the experimental results, we give some suggestions to choose the right methods or systems in different use cases. This can inspire users who want to implement develop applications designed to run on graph processing systems.

Open challenges We explain the current challenges and open problems remaining to be explored such as benchmarking of graph systems, integration of graph-structured data, visual analysis of graph data and analysis of dynamic graph data.

**Organization** The remainder of this paper is structured as follows. In Sect. 2, we give some necessary background knowledge and explain the problem formulation. In Sect. 3, we discuss existing graph partitioning methods. Then in Sect. 4, we establish a comparative study of the partitioning methods and provide some insights to choose the right methods. Furthermore we give an overview of graph processing systems in Sect. 5 followed by a comparative study of existing systems in Sect. 6. The future directions and open challenges of large-scale graph are discussed in Sect. 7. Finally, we conclude this paper in Sect. 8.

## 2 Problem statement

In this section, we introduce some notations and formal definitions related to the problem statement. Next, we will give a mathematical formulation of the graph partitioning problem.

### 2.1 Definitions and notations

**Definition 1** (*Set*) A set $S = \{s_1, \ldots, s_n\}$ is a collection of distinct elements.

We suppose that $S$ has a finite number of elements such that the cardinality is $|S| = n$ and, we note $\emptyset$ the empty set.

**Definition 2** (*Subset*) Let $S$ be a set, the set $S_i$ is a subset of $S$ if and only if $S_i \subseteq S$ and $|S_i| \leq |S|$.

**Definition 3** (*Partition*) Let $S$ be a set, the partition $P_k = \{S_1, S_2, \ldots, S_k\}$ is a k-partition of $S$ if and only if:

- $S_i \neq \emptyset, \forall i \in [1, k]$
- $S_i \cap S_j = \emptyset, \forall i \neq j \in [1, k]$
- $\bigcup_{i=1}^{k} S_i = S$

Thus the subsets of $P_k$ are not empty, two-by-two disjoint and they have no element in common.

**Definition 4** (*Sub-partition*) Let $S$ be a set and two partitions $P_k$, $SP_k$ of $S$. $SP_k$ is a sub-partition of $P_k$ if and only if for all subsets $S_i \in P_k$, there exists a subset $S_j \subseteq SP_k$ such as $S_j \subseteq P_k$.

**Definition 5** (*Multi-level partition*) Let $S$ be a set, the set $P = \{S_1, S_2, \ldots, S_l\}$ is a multi-level partition of $S$ if and only if:

- $S_1$ is a partition of $S$
- $S_i$ is a sub-partition of $S_{i-1}, \forall i \in [2, l]$

In this case, an element $S_i$ of $P$ is the $i^{th}$ level of partitioning. A partition is therefore a special case of multi-level partition, it is a partition of level 1.

### 2.2 Graphs

**Definition 6** (*Graph*) A graph $G = (V, E)$ is a structure composed of a set $V$ of vertices together with a set of edges $E = \{(u, v) | u, v \in V\}$, which connect a pair of vertices of $V$.

We denote $n = |V|$ the number of vertices and $m = |E|$ the number of edges of the graph. We note that an weight $w$ can be assigned to each edge of the graph. A weighted graph is a graph $G = (V, E, W)$ with a weight function $W : E \rightarrow \mathbb{R}$ associated to the edge of $E$.

It is important to mention that graphs present different structures regarding edge characteristics. Figure 2 shows the comparison of various weighted graphs. Firstly, the weighted edges are either directed or undirected. In a directed (resp. undirected) graph, the edges have orientations (resp. no orientations). The edge $(u, v)$ of undirected graph is identical to edge $(v, u)$. If more than two edges connect a pair of vertices, the graph is referred to as multigraph. This type of graph is suited to represent complex networks and widely used by NoSQL graph database systems [5,29]. A particular case of the graphs is a hypergraph, it is a graph with hyperedges that can link more than two vertices.

**Definition 7** (*Subgraph*) Let $G = (V, E)$ be a graph, $G_i = (V_i, E_i)$ is a subgraph of $G$ if and only if $V_i$ is a subset of $V$ and $E_i$ is a subset of $E$.

**Definition 8** (*Cut-vertex*) Let $S_1$ and $S_2$ be two subsets of the graph $G = (V, E)$. A vertex $v \in V$ is a cut-vertex or frontier-vertex if and only if, $u \in S_1$ and $u \in S_2$.

**Definition 9** (*Cut-edge*) Let $S_1$ and $S_2$ be two subsets of the graph $G = (V, E)$. An edge $(u, v) \in E$ is a cut-edge if and only if $\forall u, v \in V, u \in S_1$ and $v \in S_2$.

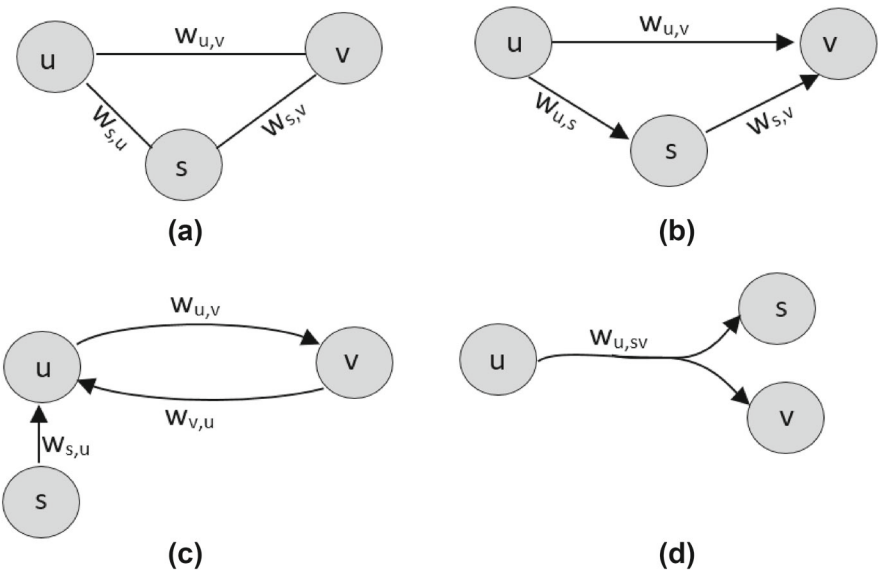


**Fig. 2** The various structures of graphs: **a** undirected graph; **b** directed graph; **c** multigraph; **d** hypergraph

### 2.3 The partitioning problem

The study of complex networks is a difficult task because the search domain increases exponentially compared to data scalability. Furthermore, the need of storage backends that supports concurrency, scalability and huge volumes of data led to the technique of divide-and-conquer. This approach is widely used by distributed database systems. It consists of logically splitting the data. Considering a set of data represented in form of graph $G = (V, E)$ where $V$ is the set of information and $E$ the relationships between information, the problem of graph partitioning consists of finding a k-partition of $G$ that respect a number of criteria. For $k = 2$, it is a graph bisection. For large-scale graph data, the problem becomes NP-Hard [26,27] if we seek to partition the graph into $k \geq 3$, there is no algorithm to solve this problem in polynomial time. The partition can relate to the set of vertices $V$: vertex-partition, or to the set of edges $E$: edge-partition. Generally, by graph partition we mean the partitioning of the set $V$. The graph partition problem can be classified into two categories: (1) partitioning with no-constraints which consists of finding a k-partition of $G$ that minimizes an objective function and (2) the partitioning constraints based on two mains constraints.

The first constraint related to the problem of graph partition with constraints is the partition balancing. It consists of finding a k-partition $P_k = \{S_1, S_2, \ldots, S_k\}$ of $G$ such that a specific balanced $(k, 1+\epsilon)$ partition, the size of each subset $S_i \; \forall i \in [1, k]$ contains a maximum number of $(1+\epsilon).\frac{n}{k}$ vertices. Let $W_i = |V_i|$ be the weight of the $ith$ partition of $G$, the average weight $W_{avg}$ and balance $B(P_k)$ of the partition $P_k$ are formulated as follow:

$$W_{avg} = \frac{\sum_{i=1}^{k} W_i}{k} \tag{1}$$

$$B(P_k) = \frac{\max\{W_1, \ldots, W_k\}}{W_{avg}} \tag{2}$$

The partition $P_k$ is well balanced if the constraint $B(P_k) \leq (1 + \epsilon)$ is satisfied, this means that the partitioning is distributed evenly by considering an error $\epsilon$.

The second constraint is the cut-edge, it consists of finding a k-partition $P_k$ that minimizes the cost of all external edges $(s_i, s_j)$ connecting two partitions $S_i$ and $S_j$. It allows reducing communication overhead for for high performance computing. The cost of the cut-edges $cut(S_i, S_j)$ between two partitions $S_i$ and $S_j$ is calculated as follows:

$$cut(S_i, S_j) = \sum_{s_i \in S_i, s_j \in S_j} W(s_i, s_j) \tag{3}$$

where $W(s_i, s_j)$ corresponds to the weight of the edge $(s_i, s_j)$. In this case, the global cost of the cut-edges between the k-partition of $G$ is defined as:

$$cut(P_k) = \sum_{i,j \leq k} cut(S_i, S_j) \tag{4}$$

The mathematical model of graph partitioning problem with constraints can be described as follows:

$$\begin{cases} \min & cut(P_k) \\ \text{subject to:} & B(P_k) \leq (1 + \epsilon) \\ & S_i \cap S_j = \emptyset \\ & \bigcup_{i=1}^{k} S_i = P_k \\ & S_i, S_j \neq \emptyset \\ & i \neq j \in [1, k] \end{cases} \tag{5}$$

## 3 Existing partitioning methods

Since graph partitioning is a NP-Hard problem, various algorithms have emerged. The principle of these algorithms relies on two search strategies: local and global. The local search strategies start from arbitrary initial partition to converge to the final graph, the main drawback is that the choice of the initial partition has a great impact on the result quality. On the other hand, the global search strategies do not rely on initial partition but on the entire graph. The choice of the partition algorithm depends on the characteristics of the chosen system. For example, if the graph system runs only on edge-centric [30], the choice of partition methods in the pre-processing phase will obviously be restricted to edge-partitioning algorithms. Moreover, the efficiency of partition methods can relate to the velocity or veracity of the final solution. There

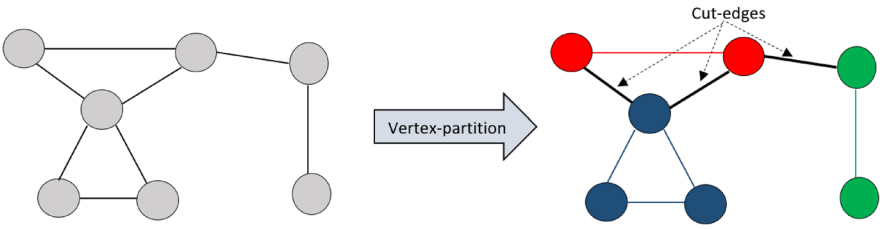


**Fig. 3** Example of vertex-partition algorithm: each vertex is assigned to one partition and the cut-edges connect the different partitions. The three colors represent the three subsets of partitioning (Color figure online)

are extremely faster algorithms that can process big graph data in seconds without guaranteed high-quality of partitioning and slow algorithms that are closen to the optimal solution [30]. We classified the graph partitioning algorithms into 8 major categories:

- Classical methods
- Spectral methods
- Structural clustering
- Partition via exchange
- Multi-level methods
- Heuristic and Metaheuristic methods
- Streaming partitioning
- Distributed partitioning

### 3.1 Classical methods

Classical methods are the most known graph partition algorithms and consist of three main methods: vertex-partition, edge-partition and hypergraph-partition. Vertex-partition method is based on the partitioning of the set of vertices $V$ of the original graph data $G$ such that each vertex belongs exactly to one partition, as shown in Fig. 3. The edges whose vertices appear in two partitions $S_i$ and $S_j$ are the cut-edges, and are used for the communications channels [30]. The main challenge of vertex-partition is the minimization of the cut-edges $cut(P_k)$.

Contrary to vertex-partition, the partitions can be defined as subsets of edges such that each edge belongs to exactly one partition. The frontier-vertices that have their edges in different partitions are called cut-vertices, as shown in Fig. 4. Such an approach of partitioning is defined as edge-partition. The edge-partition problem is similar to vertex-partition, but tries to minimize the cut-vertices [30].

The two approaches have their respective advantages and disadvantages. Vertex-partition is more adapted if we tend to get balanced partitions $B(P_k)$, if we suppose that the size of each partition is proportional to the amount of vertices. The main disadvantage is that it does not take account of the impact of hub-vertices. Having partitions with same amount of vertices does not imply having the same workload per partition. Given that each vertex has different degree of connectivity in particular the hub vertices with very high degree (power-law degree distribution). By using

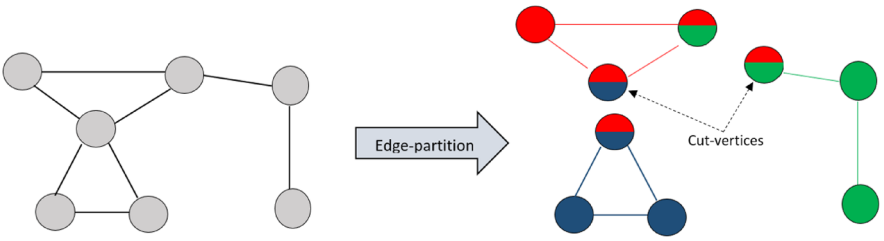


**Fig. 4** Example of edge-partition algorithm: each edge is assigned to one partition and the cut-vertices may belong to more than one partition

**Table 1** Comparison of the three main classical methods [30]

| | Vertex-partition | Edge-partition | Hypergraph-partition |
|---|---|---|---|
| Partition by | Vertices | Edges | Hyperedges |
| Partitions connected via | Cut-edges | Cut-vertices | Cut-hyperedges |
| Advantages | $B(P_k) \leq (1+\epsilon)$ | Reduce $cut(P_k)$ | $B(P_k) \leq (1+\epsilon)$ |
| | | Workload balancing | Reduce $cut(P_k)$ |

edge-partition, we can cut the hub-vertices into different partitions, thus reducing the cut-edges and balancing the workloads between the partitions.

Another definition of the partitioning problem concerns the hypergraphs partitioning. It uses similar idea as vertex-partition but uses the hyperedges to connect more than two partitions. This subproblem is used in the partitioning of VLSI. Table 1 shows the summary of the three partitioning strategies.

### 3.2 Spectral clustering

Given a set of $n$ data $\{x_1, \ldots, x_n\} \in \mathbb{R}^n$, we associate an affinity graph $G = (V, E)$ such that each vertex $s_i \in V$ corresponds to the $i$th data point [31,32]. The edges represent the affinities of the data and the weight associated to each edge $(s_i, s_j)$ encodes the similarity value between two data points $i$ and $j$ [31]. The goal of spectral method is to cluster the data such that each data $x_i$ belongs to one and only one cluster. The basic idea of spectral cluster consists of 4 main steps [32].

**Step 1** compute the affinity matrix $A$.

An affinity is a metric that evaluates the similarity or how close two data points are. In the literature review, various methods allow the measure of data similarity. However, the most common and used function metric is the Gausal Kernel [33]. Given two vertices $s_i, s_j \in V$, we define the affinity $a_{i,j} \in [0, 1]$ between the two points as [31–33]:

$$a_{i,j} = \begin{cases} exp\left(\frac{-W^2(s_i,s_j)}{2\sigma^2}\right) & \text{if } i \neq j \\ 0 & \text{else} \end{cases} \tag{6}$$

where $W(s_i, s_j)$ depends on a heuristic distance (e.g Euclidean, Manhattan,etc) and $\sigma$ a scale parameter fixed by the user.

The affinity matrix $A$ can be defined as follow [31,32]:

$$A = (a_{i,j})_{\substack{1\leq i\leq n \\ 1\leq j\leq n}} = \begin{bmatrix} 0 & a_{1,2} & \cdots & & a_{1,n} \\ & \ddots & & & \\ a_{i,1} & & 0 & & a_{i,n} \\ & & & \ddots & \\ a_{n,1} & & \cdots & & 0 \end{bmatrix} \tag{7}$$

Two vertices $s_i, s_j$ are close if $a_{i,j} \to 1$, and are so far if $a_{i,j} \to 0$ [32]. Generally the vertices belonging to the same cluster are close and those in different clusters are far apart. However, in some cases the vertices in the same cluster may also be even farther away than vertices in different clusters [32]. This highlights the importance of the Laplacian graph to achieve the regularization of the spectral partitioning.

**Step 2** compute the Laplacian matrix $L$ from $A$.
There are various normalization techniques for computing the Laplacian matrix $L$. They all use the diagonal matrix $D$. The diagonal matrix is a matrix obtained from $A$ that measures the degree $d_i$ of each vertex $s_i \in V$, such as [31,32]:

$$d_i = \sum_{j=1}^{n} a_{i,j} \tag{8}$$

The diagonal matrix $D$ can be calculated as follow:

$$D = (d_i)_{i\leq n} = \begin{bmatrix} d_1 & & & & \\ & \ddots & & & \\ & & d_i & & \\ & & & \ddots & \\ & & & & d_n \end{bmatrix} \tag{9}$$

Now, from ( 7) and ( 9) we compute the Laplacian matrix $L$ taking account of the type of normalization techniques [31–36]:

$$L = \begin{cases} D^{-1/2}(D-A)D^{-1/2} & \text{Normalized Laplacian [31]} \\ D^{-1}(D-A) & \text{Generalized Laplacian [35]} \\ D^{-1/2}AD^{-1/2} & \text{Andrew et al. Laplacian [34]} \\ (A + d_{max}I_n - D)/d_{max} & \text{Symmetric and stochastic Laplacian [36]} \end{cases} \tag{10}$$

where $d_{max}$ is the maximum degree of $D$ and $I_n$ the identity matrix.

**Step 3** extract the eigenvectors from $L$.

This step consists of selecting the $k$ eigenvectors $v_i$ corresponding to the $k$ eigenvalues $\lambda_i$ of $L$ [34]. If $k$ major clusters are formed, then the Laplacian $L$ is approximately identified as a diagonal matrix:

$$L = \begin{bmatrix} L_{1,1} & \cdots & L_{1,i} & \cdots & L_{1,k} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ L_{i,1} & \cdots & L_{i,i} & \cdots & L_{i,k} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ L_{k,1} & \cdots & L_{k,i} & \cdots & L_{k,k} \end{bmatrix} \approx \begin{bmatrix} L_{1,1} & & & & \\ & \ddots & & & \\ & & L_{i,i} & & \\ & & & \ddots & \\ & & & & L_{k,k} \end{bmatrix} \tag{11}$$

where $L_{i,i}$ is the $i$th subsets of the cluster $S_i$.

The $k$ eigenvectors of $L$ are solution of the equation $L \times v_i = \lambda_i \times v_i$, this allows for obtaining a projection space in $k$ dimensions. The vector matrix is then constructed by storing these eigenvectors in columns [34].

**Step 4** cluster the data.
The data partition is performed one the eigenvectors matrix. The first stage consists of considering each line $i$ of the matrix as spectral space. The second stage consists of applying an unsupervised k-means learning algorithms [37] on the matrix. Thus, the data partitioning under k-partition amounts to assigning a point $x_i$ to partition $j$ if and only if the line $i$ of the eigenvectors matrix has been affected to partition $j$.

### 3.3 Structural clustering

Structural clustering methods are similar to spectral methods but are applied on probabilistic graphs to find densely-connected subgraphs, hub vertices and outliers. Traditional clustering algorithms are designed for static graphs. Moreover current real-world are not deterministic but probabilistic because of the connections between edges can often inferred using statistical models [38]. The graph clustering methods are typically grouped into two categories [39]: vertex clustering and graph clustering.

Vertex clustering is a multi-dimensional clustering algorithm based on an objective function that minimizes the distance between two vertices of the graph. Each vertex is considered as a data point and the edges correspond to their distance values. The clustering algorithm consists of creating cluster of vertices with respect to the minimization of the inter-cluster similarity for a given number of clusters.

In the case of graph clustering, we have a large-scale graph which contains multi-graph. We need to cluster the graph based on the topology of each sub-graph within the graph. This task is challenging because it involves to match each sub-graph topology and use them for clustering purposes.

### 3.4 Partition via exchange

The complexity of this problem leads to most approaches focusing on algorithms without guaranteed approximate rate [30]. Partition via exchange is based on the

optimization of an objective function and is widely influenced by the initial solution and therefore might eventually fall in local minima.

The most well know partition algorithm was developed in 1970 by Kernighan and Lin [40]. As shown in Algorithm 1, the basic idea of this approach consists in first assigning randomly each vertex $v_i$ to one of the k partitions. Afterward, the algorithm tries to improve the solution by evaluating the gain on the cut-vertex function and exchange if possible the vertices between partitions. The process is repeated until there is no possible exchanges that optimizes the cut-vertices of the final partition. Similarly, Fiduccia et al. [41] proposed an extended version of Kernighan and Lin algorithm's for hypergraph partitioning. For each iteration, the algorithm computes the cost of exchanges between partitions, chooses the best and locks the vertices [30]. The operation is repeated until all vertices are locked.

**Algorithm 1:** Kernighan and Lin strategy

```
1  function Kernighan_Lin_Partition
2  input
3  G(V, E) : original graph data
4  output
5  P_k: k-partition
6  Step 1: initialization
7  S_j ← ∅, ∀j ∈ [1, k];
8  for each vertex v_i ∈ V do
9  |   v_i --move--> random(S_j ∈ P_k);
10 end
11 Step 2: optimization via exchange
12 for each vertex v_i ∈ S_i and v_j ∈ S_j do
13 |   if ∃ gain(S_i, S_j) < 0 then
14 |   |   v_i --move--> S_j;
15 |   |   v_j --move--> S_i;
16 |   end
17 end
18 P_k = {S_1, ··· , S_k};
19 return P_k;
```

### 3.5 Multi-level methods

The multi-level partitioning is an highly successful local partitioning strategy. It allows at any given level to partition each partition into sub-partitions. As shown in Fig. 5, the basic idea consists of firstly reducing the original graph size by recursively collapsing vertices and edges until a smaller graph is obtained. Then, it preforms the partitioning on the smaller graph. In the refinement phase the partition results are projected on a larger graph until the whole graph is covered.

The algorithm that uses widely such an approach is Metis [42,43]. In the coarsening phase, the authors introduced the heavy edge matching strategy to collapse the edges [30]. In the partitioning phase, as simple Breadth-First Search algorithms is running

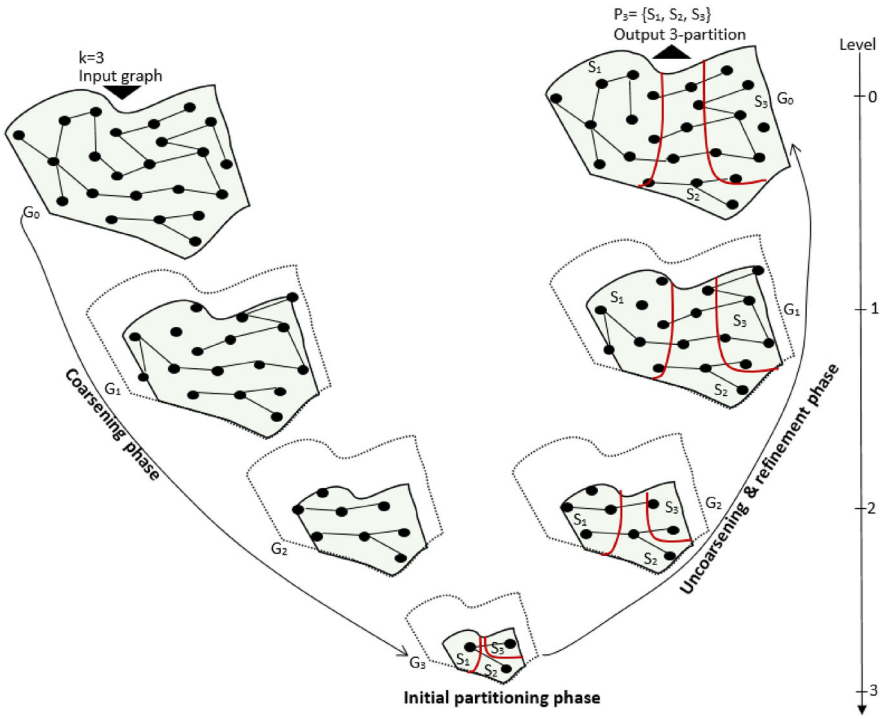


**Fig. 5** Multi-level graph partitioning overview [45]. In the coarsening phase, the graph size is successively reduced. A 3-partition is computed during the initial partition phase. In the uncoarsening phase, the partitioning result is successively projected on each level of the graph and refined until it reaches the original graph $G_0$

from a random initial vertex to frontier-vertices that generate less cut-edges. Finally in the refinement phase, the result is projected back through the graph by refining it with respect to each partition border.

The authors quickly developed hMetis [44], an extended version of Metis for hypergraph partitioning. Followed by ParMetis [45], a parallel version designed for running on multi-core processor.

### 3.6 Heuristic and metaheuristic methods

Heuristic partitioning strategy is a fast solving approach that employs a practical method not guaranteed for optimal partitioning, but satisfactory for immediate solutions. As representative examples, we present the quick heuristic approaches: EdgePartition1D and EdgePartition2D implemented by Apache Spark [46,47] in GraphX [48].

EdgePartition1D improves the Edge-partition algorithm by optimizing the random edge placement. The random placement creates a large number of cut-vertices, see Fig. 6a. To solve this issue, EdgePartition1D uses a hash function to assign edges to

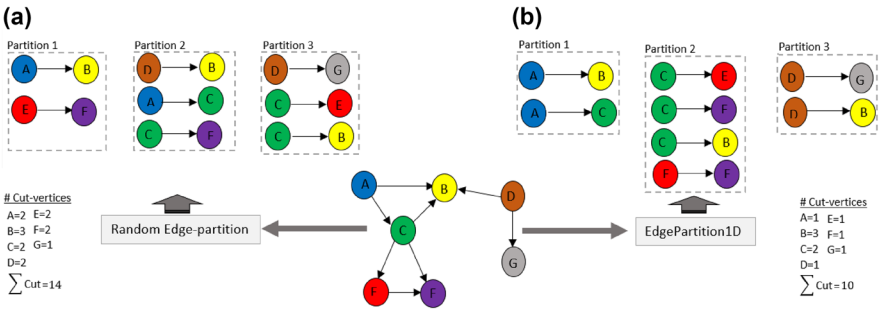


**Fig. 6** Edge placement scenario: **a** Edge-partition strategy assigns randomly the edges to the three partitions, thus generating 14 cut-vertices; **b** EdgePartition1D optimizes the cut-vertices to ten by using the hash function of source vertex id for the derandomization of edge placement

different partitions such as edges that share the same source id will end in the same partition, as shown in Fig. 6b. The vertices that have both edges will become a cut-vertices and they will be replicated between fewer partitions. The main advantage of this strategy is the derandomization of edge placement scenario and the higher quality of solution with fewer cut-edges.

EdgePartition2D strategy is based on a 2D partitioning using a $n \times n$ sparse adjacency matrix of the graph such as $n = |V|$. For a $k$-partition of a given graph, the matrix is split into $\sqrt{k} \times \sqrt{k}$ sub-matrices and the edges are assigned across partitions using their position in the sub-matrices. This guarantees that all vertices are replicated to at most $2 \times \sqrt{k}$ partitions and ensures effectively the split of hubs. However the main limitation of this strategy is that if the number of partition $k$ is not a perfect square, we can lead to imbalanced partitions.

Contrary to heuristic methods, metaheuristic-based approaches focus on high-quality solutions without guarantee partitioning in a reasonable time. Most of representative methods that highlight metaheuristic-based approaches include optimization by simulated annealing [49], tabu search [50], ant colony optimization [51] and genetic algorithms [52,53]. Generally these algorithms incorporate local strategies to accelerate the convergence to the optimal solution. The experimental tests performed on different families of graphs show that metaheuristic approaches outperform other graph partitioning methods and are very close to the best solutions [54].

### 3.7 Streaming partitioning

The large size of the graph makes the partitioning very difficult, especially in the case of streaming scenario where the data is continuously arriving and updating. The classical partitioning algorithms designed for static data are not applicable for graph streams. For example in Facebook's social graph, vertices and edges change every seconds because of user interactions (e.g likes, comments, shares, etc.). Due to the streaming nature, more specialized algorithms have emerged. Aggarwal et al. [55] proposed a technique for clustering dynamic graphs. They proposed a hash-compressed based on compression of the coming edges to create microclusters onto a smaller domain space.

Two heuristic based approaches are widely used for streaming graph partitioning: place the new vertex in the partition with more neighbors or in the partition with fewer number of non-neighbors. Charalampos et al. [56] introduced Fennel, a novel framework which unifies the two heuristics and define an objective function that measures the interpolation between them. This improves the load balancing between partitions. Moreover, they provided a one-pass streaming algorithm that runs in $O(\frac{log(k)}{k})$ time, where $k$ is the number of partitions. Despite the fact that the proposed algorithm runs in one-pass, it is surprising that it outperforms Metis [42,43]. For instance, with Twitter graph composed of more than 1.6 Billion edges, Fennel partitions the graph in 40 minutes with balanced partition composed of 6.8% of cut-edges, whereas Metis took 81/2 hours for balanced partition composed of 11.98% of cut-edges.

For large-scale graph, the topology impacts greatly on the partitioning because of power-law degree of distribution. The Greedy vertex-cut method introduced by Joseph et al. [57] leverages the power-law graphs to distribute the edge placement. If both endpoints $v_i, v_j$ of the coming edge $(v_i, v_j)$ is already inside a common partition ($v_i \in S_i$ and $v_j \in S_i$) or only one vertex is already in a partition ($v_i \in S_i$ and $v_j \notin S_i$), the edge will be assigned to that partition $S_i$. In case that the endpoints $v_i, v_j$ of the edge $(v_i, v_j)$ have no common partitions ($v_i \in S_i$, $v_j \in S_j$ and $v_i, v_j \notin S_i \cap S_j$), if degree of $v_i$ $d(v_i) > d(v_j)$ then the edge will be assigned to partition $S_i$ else it will be added to partition $S_j$. Finally, if both vertices are free, the edge will be affected to the smallest partition.

### 3.8 Distributed partitioning

The computation of graph partitioning becomes challenging when the graph grows in size. The use of centralized algorithms is very expensive, incurs high computation and communication cost quickly becomes a limiting factor for large graphs. For example spectral methods do not scale to partition big data. Distributed partitioning is a model in which the partitioning task is distributed across a cluster of computers networked in distributed architecture. The computers communicate and coordinate by passing messages to achieve quickly the partitioning of large graph data. Works performed on distributed partitioning are few [30,58–60,60,61] and they are not reliable because they typically involve global knowledge on the graph topology.

A successful example and main competitor of distributed partition is JA-BE-JA [58,59], a fully distributed algorithm that uses local search and simulated annealing method. JA-BE-JA is designed for big graphs and it provides two types of graph partitioning: vertex-partition [59] and edge-partition [58]. The choice between both partition strategies depends on the context of application and JA-BE-JA is the only algorithm that can use these two strategies. The algorithm is completely decentralized: each vertex of the graph is processed with local information related from its direct neighbors and a small set of vertices chosen randomly in the graph. This allows the JA-BE-JA's program to be easily incorporated into any distributed graph processing frameworks. Initially, each vertex/edge is assigned to random partition, and, iteratively the initial assignment is improved through message exchanges between vertices. The authors showed that JA-BE-JA's version of edge-partition [58] outperforms Metis

**Table 2** Comparison of DFEP [30] against JA-BE-JA [59] and Greedy [57]

| DFEP versus JA-BE-JA | DFED versus Greedy |
|---|---|
| DFEP achieves more balanced partitions in small datasets than JA-BE-JA | Greedy creates remarkably balanced partitions compared to DFEP |
| DFEP needs less iterations to converge than JA-BE-JA | Greedy is extremely fast than DFEP |
| JA-BE-JA's communication costs are ten times higher than DFEP | Greedy needs only one iteration |
| | DFEP needs more iterations |
| | Greedy's partitions are less connected than DFEP's partitions |

[42,43] with very low vertex-cuts, in particular on large social networks. For JA-BE-JA version of vertex-partition, the results outperform Alessio Guerrieri's algorithm [30] and the size of the partitions are well balanced and also have better vertex-cut.

The main drawback of JA-BE-JA is the implementation of simulated annealing method which needs several hundred iterations to converge to the optimal solution. This large number of iterations requires very costly communication overhead because there is a synchronization at the end of each iteration. To overcome this limitation, Alessio Guerrieri proposed [30] a Distributed Funding-based Edge Partitioning (DFEP) that requires less iterations to complete. DFEP is based on an amount of funding assigned to partitions for buying the edges of each partition. Initially, each partition receives the same amount of funding with initial vertex selected randomly. During each iteration, each partition tries to buy the edges that are adjacent to those already buying. A coordinator monitors all transactions, it balances the sizes of each partition and sends additional amount of funding to the smaller partitions, to help in buying other edges. DFEP is also suited for running on other big data platforms such Hadoop MapReduce [62], GraphX [48] and Amazon EC2 cloud. Table 2 shows a comparative study of DFEP against JA-BE-JA and distributed version of Greedy algorithm.

Instead of static graph data, Stanton et al. [60] presented a streaming and distributed method for dynamic graph partitioning. The vertices assignment is based on three main orders: random, breadth-first search and depth-first search. The partitioner places the incoming vertex in one of the $k$ machines of the cluster. After vertex is placed, it will not move to another machine. The algorithm is based on local search and access only to the subgraph formed by all previous vertices. Moreover, the use of this distributed algorithm on Spark [47] allowed to improve PageRank's computation, by 18% to 39% on big social networks.

## 4 Comparison of partitioning methods

This section starts by introducing the detailed cost analysis of each method, and provides the comparison among all the partitioning methods. Then we provide some

**Table 3** SNAP datasets used for the experiments

| Name | Type | $\|V\|$ | $\|E\|$ | D | ACC |
|---|---|---|---|---|---|
| ASTRO-PH | Undirected | 18772 | 198110 | 14 | 0.6306 |
| ENRON-EMAIL | Undirected | 36692 | 183831 | 11 | 0.4970 |
| USROAD-NET | Directed | 126146 | 161950 | 617 | 0.0145 |
| EGO-TWITTER | Directed | 81306 | 1768149 | 7 | 0.5653 |

suggestions for users how to choose the right partitioning methods in different cases. We evaluate in detail the behavior of each method through a simulation engine of grid'50000.

### 4.1 Datasets

We used four different datasets for the experiments. All the networks have been taken from SNAP datasets.[2] Table 3 presents all information about the datasets. For each dataset, we list the graph size, the diameter (longest shortest path) $D$ and the average clustering coefficient $ACC$.

ASTRO-PH is collaboration network which covers scientific collaborations between authors papers submitted to Astro Physics category, while ENRON-EMAIL covers all the email communication within a dataset of around half million emails. The USROAD-NET dataset is a road networks of US. Intersections and endpoints are represented by nodes, and the roads connecting these intersections or endpoints are represented by undirected edges. Finally, EGO-TWITTER is a social circles from Twitter. This dataset consists of 'circles' (or 'lists') from Twitter. Twitter data was crawled from public sources. The dataset includes node features (profiles), circles, and ego networks.

### 4.2 Analysis

Each algorithm has been executed ten times and the presented values are the average values of the executions. Each SNAP dataset is partitioned under $k = 25$ partitions. The metrics considered to evaluate the partitioning algorithms are the following:

- **Runtime** the computation time to achieve the partitioning.
- **Communication** the number of cross-edges or cut-edges between partitions.
- **Balance** the standard deviation of the normalized partitions. It measures how each partition is close as possible to the same size. It is calculated as follows:

$$StD = \sqrt{\frac{\sum_{i=1}^{k}(\frac{|E_i|}{|E|/k} - 1)^2}{k}} \tag{12}$$

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

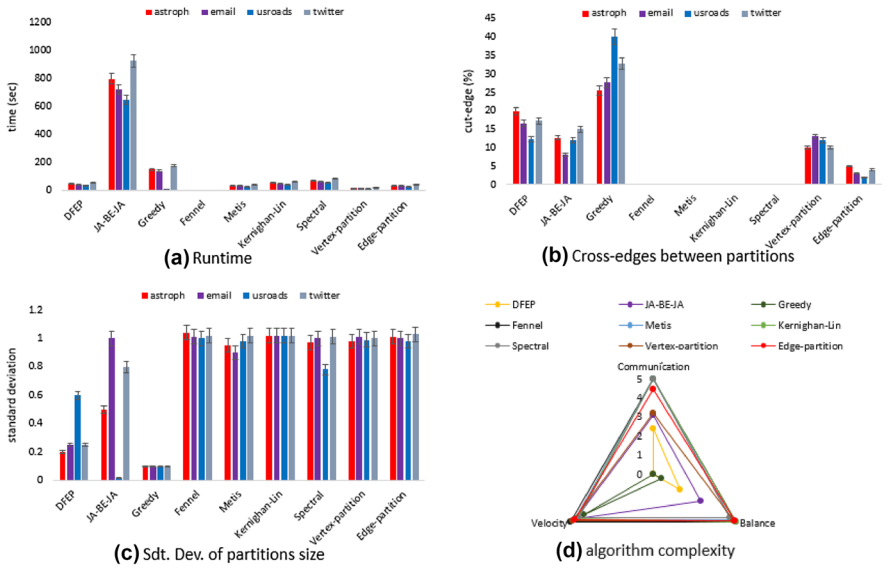


**Fig. 7** Comparative study of graph partitioning methods (k = 25)

Figure 7 shows the performance of the partitioning algorithms against the k-partition of the four SNAP datasets. In term of time complexity we remark that JA-BE-JA takes lot of times to converge to the optimal solution. On the other hand the other algorithms are faster. Regardless of the topology of the datasets, the traditional methods provide a good partition that optimize the cross-edges between partitions. This allows the avoiding of network overhead. Moreover the partitioning results provided by streaming and distributed algorithms are not balanced according the standard deviation.

### 4.3 Suggestions

Graph partitioning problem is considered as NP-Hard problem i.e there is no algorithm that provides an optimal solution in linear time. Thus a faster algorithm might be slowest for some graphs and vice versa. Therefore we have to first define our use case and then we may consider which parameter of optimization to choose.

Figure 8 shows a decision tree that provides some insights on how to choose the right methods. For smaller graphs, it would be better to opt for traditional graph methods. When the graph is too large to fit in memory, it is more advantageous to opt for distributed methods. DFED and JA-BE-JA are still the best choices because they are based on Hadoop and Spark frameworks. In case that the graph changes continuously, we suggest to use heuristic or streaming algorithms.

## 5 Existing graph systems

The topic of graph processing systems has attracted more interests for large number of academic and industrial institutes. Due to its complexity in graphs composed

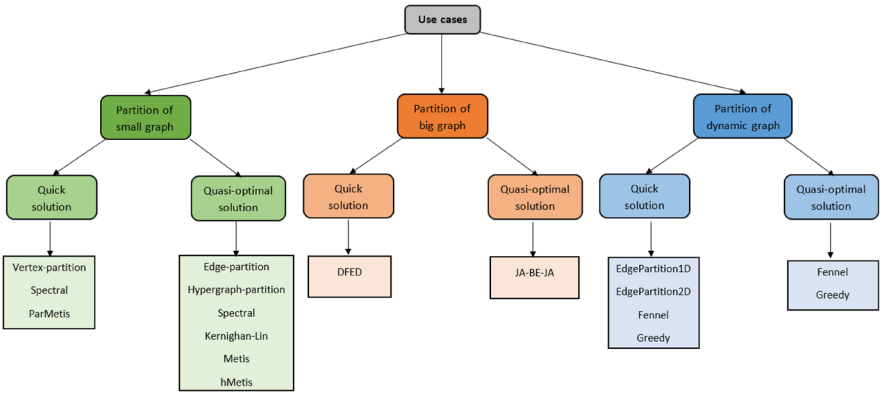


**Fig. 8** Decision tree of graph partitioning algorithms

of million/billion of vertices, it consumes huge computing resources. Therefore, a commodity machine with single thread does not deal with this issue. In this context, substantial efforts have been made to build parallel and distributed frameworks. So the choose of graph processing systems is a headache for developers because it greatly influences on the computation task. Several factors are to be considered: the graph size and topology, the partitioning method, the cluster configuration and the program complexity.

## 5.1 Graph programming models

We will focus on four programming models used to implement graph processing applications. We first introduce MapReduce paradigm, the big player. Followed by an improved models of MapReduce: vertex-centric, GAS and partition-centric.

### 5.1.1 MapReduce

MapReduce is a programming model of Google developed for intensive computation on large-scale data with enough machines. It is designed to run in parallel and distributed environments and can be implemented by big data programs for processing huge datasets [63]. The programming model of MapReduce is designed to run on multi-nodes cluster. A representative use case of MapReduce is Apache Hadoop [62,64], initially designed to process a web crawl. MapReduce model is too simple to express and consists of two phases: the map phase and reduce phase, as shown in Fig. 9. Each phase takes as input a set of key-values pairs and produces another set of key-values as output. The computations performed in the two stages depend on two main functions defined by the user: map function and reduce function. For performance reasons a local aggregate in the map phase can be helpful, this reduces the amount of data transmitted between the map and reduce stage. Thus task is possible through the combiner function specified by the user.

MAP: $(\text{KEY}, \text{VAL}) \rightarrow \text{LIST}(\text{KEY}, \text{VAL})$

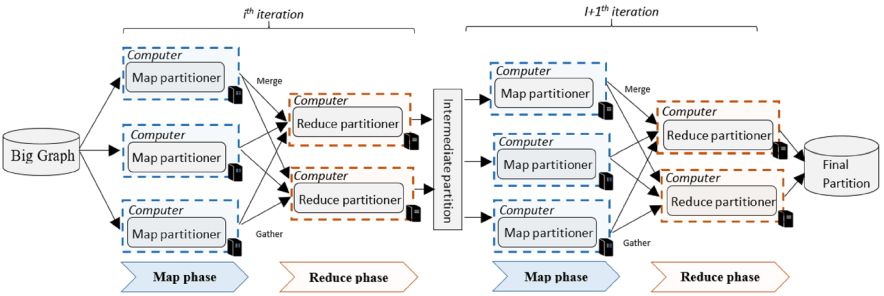


**Fig. 9** Overview of MapReduce programming model

COMBINE: (KEY, LIST(VAL))→(KEY, LIST(VAL))
REDUCE: (KEY, LIST(VAL))→LIST(KEY, VAL)

**Map phase** in this phase, the partition problem is divided into set of sub-partitioning tasks called mappers. This set of mappers is then distributed across the cluster. Each mapper process its task independently in a single machine and sends its intermediate result to the reduce stage.

**Reduce phase** in this phase, the intermediate sub-partitions of the map phase are gathered and then merged to produce another intermediate partitions. Similarly, the set of reduce tasks is distributed across the cluster and each reducer processes its task independently in a single machine.

Initially, the partitioning is achieved on the original graph to produce intermediate result, and, iteratively this result is improved at each iteration. Despite the fact that MapReduce achieves significant gain of time and shows its effectiveness for several large-scale graph problems [65], different linked problems arise mainly:

1. MapReduce is not suited for iterative computation, most of graph algorithms are iterative and require many iterations. Such implementation consumes large bandwidth and involves lot of I/O, so it is very low efficient.
2. MapReduce is not Message-Passing (MPI), some graph algorithms need local information from one vertex or neighborhood during the graph processing, such an operation is not feasible with MapReduce.
3. Graphs can be dynamic or be a subject of frequent modifications: add, delete or update of vertices/edges. This huge volume of changes can be detected on social networks. MapReduce does not provide CRUD operations to create, read, update, and delete informations on graph databases.

### 5.1.2 Vertex-centric

Vertex-centric or "think like a vertex" is a variant of MapReduce model inspired from the Bulk Synchronous Parallel (BSP) paradigm [66]. Vertex-centric has a very simple programming model that allows for easily implementing graph processing algorithms through a *vertex compute function* [21]. It consists of three steps: (1) read all received messages from incoming neighbors; (2) update the states, each vertex has two states:

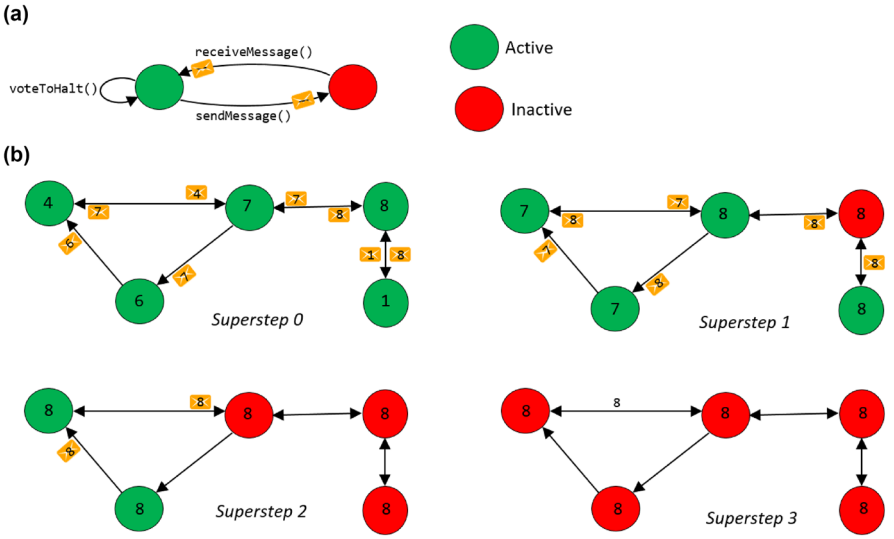


**Fig. 10** Illustration of vertex-centric programming model with the computation of maximum value

*active* or *inactive*; and (3) send messages to its outgoing neighbors. Note that each vertex access locally to information of its direct neighbors and sends messages along the edges. Similarly to MapReduce, vertex-centric model works in multi-node cluster. Each node computes the same *vertex functions* at each *superstep* and controls the states of its associated vertices. Initially all vertices are *active*, then can *vote to halt* (active to inactive) during computation of each *superstep*. Moreover, each node completes its computation if all its vertices are in the *inactive* state. The vertex whose state is *inactive* does not take part in any *superstep* unless it receives message to be *active*.

Figure 10 illustrates a simple example of maximum value computation with vertex-centric model. Initially (*superstep 0*), all vertices are in the *active* state. In each *superstep*, the user-defined *vertex function* reads the values of its incoming neighbors, updates its value to the maximum value if the received values are superior to its current value. Next, it sends this maximum value along all of its outgoing edges. If the maximum value of a given vertex does not change in the next superstep, the vertex then votes to halt. This keeps going on until all vertices vote to halt, then the program terminates the computation.

### 5.1.3 Gather–apply–scatter

Vertex-centric model is efficient for big graph with few neighborhood, it allows to maximize the parallelism and reduce the network communication. However, graphs derived from real-world graphs such as social networks, web networks or road networks have a power-law degree distributions causing a decline in performance because of low parallelism and costly network communication. To overcome this problem, Joseph et al. [57] introduced the Gather-Apply-Scatter (GAS) programming model. It integrates the basic idea of combiner which aggregates local messages. Instead of using *vertex*

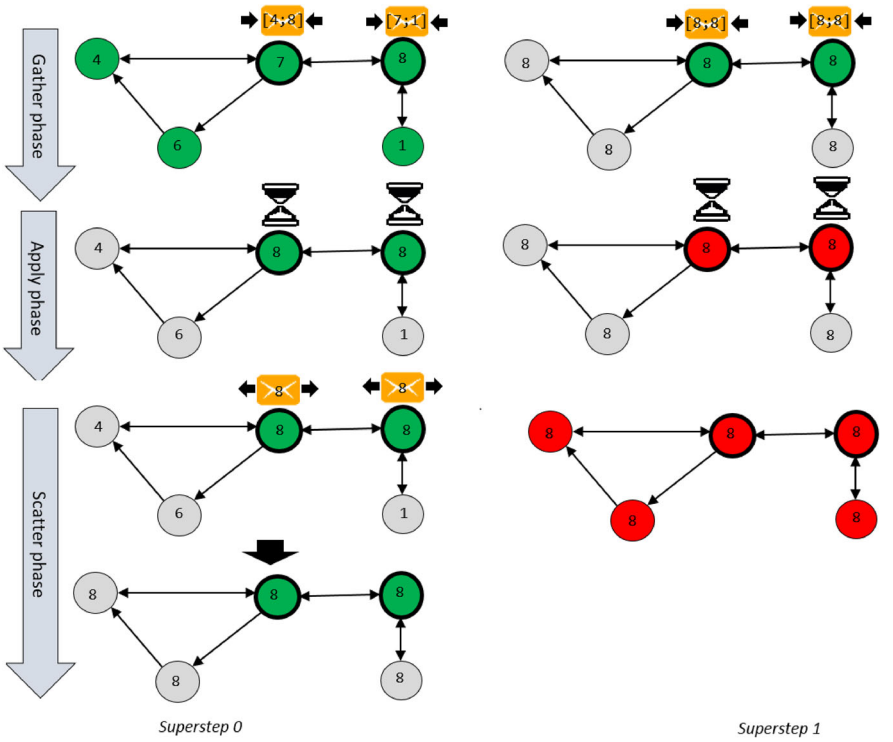


**Fig. 11** Illustration of GAS programming model with the computation of maximum value

*compute function* [21], the user has to *gather*, *apply* and *scatter* functions that need the entire neighborhood of a given vertex. GAS model is too simple to express and consists of 3 phases:

**Gather phase** this phase is similar to combiner phase of MapReduce. All messages addressed to the hub vertex are aggregated through a *sum function.*

**Apply phase** in this phase, the *apply function* takes as input the aggregated message, applies then updates the vertex state.

**Scatter phase** finally, the scatter function takes as input the vertex state and creates new outgoing messages.

Figure 11 shows an illustration of GAS model with computation of maximum value. Initially (*superstep 0*) the two hub vertices aggregates all incoming messages and computes the maximum values. After, they send the max values to outgoing vertices then volt to halt. The program terminates when all hub vertices are inactive. Compared to vertex-centric model, GAS requires only two supersteps instead of four and uses low multiples messages to achieve the computation.

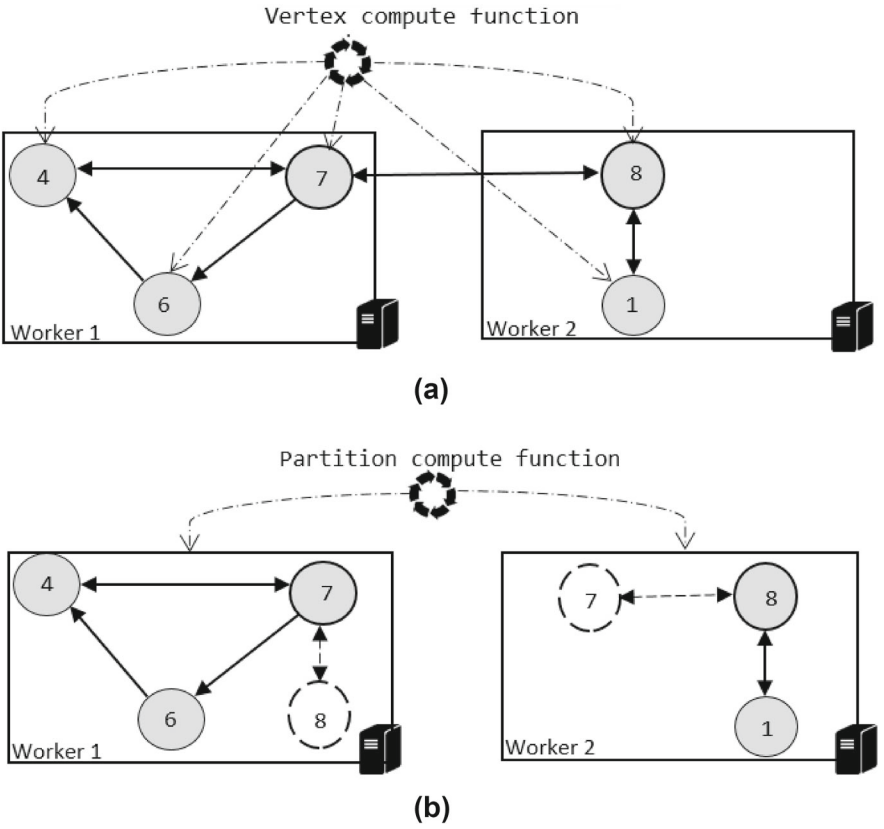


**Fig. 12** Graph partitioning for different programming models: **a** Vertex-centric. **b** Partition-centric

### 5.1.4 Partition-centric

Since the graphs are partitioned and distributed across the cluster, it will be advantageous to define a function that instead of computing on each vertex, compute on each graph partition rather. Such programming model is called: graph-centric or partition-centric [14]. It is an extension of vertex-centric [21] and is based on the "Think like a Graph" approach [67].

In the pre-processing step, the worker machine must create a copy of each adjacent vertex that is not internal to its partition (see Fig. 12). These vertices define the boundary vertices and are used to send messages. During the processing step, each worker executes independently its *partition compute function* then sends messages from the boundary vertices to its internal vertices. This programming model presents more advantages compared to vertex-centric and GAS model: (1) it provides algorithms that converge more quickly; (2) it uses a fewer number of supersteps to achieve the graph processing. For example, the number of supersteps required for the shortest path computation is usually equal to the diameter of the graph, graph-centric uses less number.

**Table 4** Key features of graph systems discussed in this paper

| System | Programming model | Computing model | BSP | Asyn |
|---|---|---|---|---|
| Hadoop MapReduce [62] | MapReduce | Distributed | ✓ | ✗ |
| Pregel [68] | Vertex-centric | Distributed | ✓ | ✗ |
| ExPregel [69] | Vertex-centric | Distributed | ✓ | ✗ |
| GraphChi [70] | Vertex-centric | Parallel | ✗ | ✓ |
| Giraph [70] | Vertex-centric | Distributed | ✓ | ✗ |
| GraphLab [71] | GAS | Parallel | ✓ | ✓ |
| PowerGraph [57] | GAS | Distributed | ✓ | ✗ |
| Giraph++ [67] | Partition-centric | Distributed | ✓ | ✗ |
| GPS [72] | Partition-centric | Distributed | ✓ | ✗ |
| Blogel [73] | Partition-centric | Distributed | ✓ | ✗ |
| GraphX [74] | Edge-centric | Distributed | ✓ | ✗ |
| Neo4j [29] | Cypher query | Parallel | N/A | N/A |
| Gradoop [75] | GrAla query | Distributed | N/A | N/A |

## 5.2 Graph processing systems

All graph systems are designed with a common goal: to provide an easy API for the analysis of complex graphs. Table 4 reviews all the frameworks introduced in this paper. More of these frameworks are an enhanced versions or improvements of existing frameworks.

### 5.2.1 Pregel

Pregel [68] is the big player and most popular framework dedicated especially for graph computation problems, it is Google's proprietary introduced to face the limitations of MapReduce. It is designed to make the graph programs very simple to develop by erasing all technical difficulties with regards to writing complex programs for large graphs. Pregel is vertex-centric based system and was inspired from Bulk Synchronous Parallel (BSP) programming model. Pregel is a scalable, parallel and distributed system, the computation is distributed across the cluster nodes. It provides an API to define functions that execute on all the vertices in each superstep. Each vertex communicates with other vertices through messages passed over the network and votes to halt.

### 5.2.2 ExPregel

To synchronize the distributed computations on the cluster, Pregel requires very time-consuming task at each superstep. This may result in serious bottleneck when the number of communication increases in large-scale graph. To deal with this issue, an asynchronous version of Pregel called ExPregel [69] was introduced. Unlike Pregel,

ExPregel is well-designed to reduce the network traffic by according priority to messages exchanged between vertices resided on the same partitions. The programming model is similar to partition-centric, each worker executes the compute function on its vertices and all intermediate results generated are immediately treated. Once all internal messages are consumed, the external messages are sent to the other workers through the network. ExPregel is much faster than Pregel, the runtime speeds up from 1.2 to 30 times and reduces the number of supersteps from 45% to 96%.

#### 5.2.3 Giraph

Giraph [70] is an open-source implementation of Pregel, it provides a similar API to Pregel and follows the BSP model. Giraph leverages Hadoop components. The input data is stored in the Hadoop Distributed File System (HDFS) [76], Hadoop starts the cluster and Apache Zookeeper manages the cluster and synchronizes the computation states between workers. Since Giraph is based on Hadoop, the computation is executed in memory and can run on existing MapReduce program but it is a single Map-only job.

#### 5.2.4 Giraph++

Giraph++ [67] is an extension of Giraph, but supports the asynchronous computation. It incorporates three programming models: (1) vertex-centric, (2) partition-centric, (3) hybrid setting, similar to vertex-centric model, but the vertices that share same partitions are processed asynchronously, while external vertices are processed synchronously over the network.

#### 5.2.5 GPS

GPS [72] (Graph Processing System) is an open-source system inspired by Pregel, but adds three new features: (1) an extended API that enables a more easy writing and execution of complex graph algorithms. (2) GPS can dynamically reassign vertices across cluster during computation. (3) Finally, GPS integrates an optimization LALP (Large Adjacency List Partition) that partitions adjacency lists of hub-vertices across cluster to improve performance and again to reduce the communication.

#### 5.2.6 GraphLab

GraphLab [71] is a parallel and distributed framework targeted for sparse iterative graph algorithms. It is an open source system originally designed for Machine Learning tasks. GraphLab was implemented independently from Pregel, but shares the same motivation concerning the limitations of MapReduce on large graphs. However, the main difference between both systems is that Pregel targets Google's distributed file system [76] while GraphLab addresses shared memory parallel systems. Compared to Pregel, GraphLab provides new features: (1) A complete API to write and execute complex programs in shared and distributed memory systems. (2) An integration

of HDFS for distributed storage with fault-tolerance. (3) An integration of powerful toolkit for machine learning. (4) Finally, GraphLab integrates new sophisticated algorithms that allows for partitioning the graph data intelligently.

### 5.2.7 PowerGraph

GraphLab's programming model is based on vertex-centric: users must define an update function that takes a vertex as input, then reads, writes and updates all data associated to this vertex and its entire neighborhood. The main drawback is that in scale-free graphs, the hub-vertices may have huge data that fit in memory. This is very expensive and can lead to performance decline. To address this challenge, the developers of GraphLab introduced PowerGraph [57]. PowerGraph is a graph system that takes advantage of Pregel and GraphLab by combining the best features from both systems. It optimizes the computation tasks on power-law graphs. PowerGraph is able to conserve the "Think like a Vertex" paradigm while distributing the vertex compute function from single machine to entire cluster. In order to expand the scope of application, GraphLab and PowerGraph's developers developed a complete framework named *GraphLab Create* which does not exploit only GraphLab and PowerGraph, but enables generic graph processing without the need for external tools.

### 5.2.8 Blogel

Blogel [73] is the most efficient partition-centric framework for distributed computation on real-world graphs. Blogel provides three computing modes: vertex-centric, partition-centric and hybrid. In hybrid mode, during superstep, all vertices execute following firstly vertex-centric mode, and then partition-centric mode for blocks. Blogel integrates HDFS for data I/O and can be deployed on any Hadoop version. In comparison with existing partitioning algorithms, Blogel offers a specialized methods for URL and 2D spatial partitioning. In term of performance, experiments carried out on various graph systems with four classic graph problems (connected components, shortest path, reachability and PageRank) showed that Blogel is significantly faster than Pregel, Giraph, Giraph++, GraphLab and PowerGraph.

### 5.2.9 GraphChi

GraphChi [48] is a disk-based graph system exploiting a low-memory for parallel computations on single machine. It first splits the large graph into set of small parts (shards) and stores them on disk. Then it uses a novel parallel sliding windows method that supports the asynchronous model for computation. In each superstep, the shards are loaded from disk to main memory, then updates vertices and writes the results to disk. If the number of vertices is too large, then the shards will be stored on disk. In this case, Graphchi will be able to compute on large graph using single commodity machine. Compared to other existing systems, GraphChi is faster and runs fewer supersteps than GraphLab, GPS and Spark. On the other hand, if we ignore the graph loading, PowerGraph is faster on large cluster than GraphChi on just one machine.

### 5.2.10 Other existing systems

While most graph systems are centered around basic computation models, there is a new generation of systems that have been researched into in recent years [29,74,75].

The first system in this field is GraphX [74], a Resilient Distributed Graph (RDG) system based on Spark. GraphX extends Spark's Resilient Distributed Dataset (RDD) to RDG. This RDG is composed of two record files, one for the vertices and the other containing the edges. On the execution side, GraphX computation model is based on edge-centric. It consists of defining a *edge compute function* which joins RDDs values of vertices and edges for iterative computation. GraphX provides a complete API that simplifies graph ETL and computations. Since GraphX is based on Spark, it provides streaming algorithms for dynamic graphs.

The second system is Neo4j [29], currently it is the world's leading and most popular graph database. It allows high query performance on huge volume of complex data. It is full ACID transaction that stores NoSQL data as graphs. Neo4j provides a very simple declarative query language for data processing and can be deployed in two modes: (1) Standalone, with only one worker node and (2) Highly-Available (HA) cluster, with multiple workers. A load balancer deployed upstream of the cluster manage directly the read/write operations of the client applications so that all read requests are distributed to the slave workers and all write requests are synchronized to the master worker. To complete a write/load and guarantee consistency, the HA cluster requires a quorum in order to accept write operations. If this quorum is not formed, the cluster will degrade into read-only mode. Neo4j materializes graphs from Hadoop, Hive and Spark. Moreover each machine of the cluster can host at most 34 billion vertices, 34 billion edges and 68 billion properties.

The last system in this field is Gradoop [75], a scalable graph data management and analytics with Hadoop. Gradoop's framework is based on the so-called Extended Property Graph Data Model (EPGM) that supports semantic and schema-free graphs. It provides high-level operators for analyzing multiple graphs. The architecture of Gradoop is build on top of Hadoop ecosystem. The users can define the analytical programs by using a specific declarative language GrALa (Graph Analytical Language) which is based on EPGM graph data model. Gradoop is easily integrated into Apache Flink, this way it benefits from Flink capabilities for graph mining in distributed environment. Additionally, it allows for reading and writing any graph supported by Flink such as HBase and HDFS. Gradoop is still in its initial stage and some levels of the framework's architecture need to be completed and optimized.

## 6 Comparison of existing systems

In this section, we evaluated the performance of each graph processing system. Then we provide some suggestions for users how to choose the right systems. We used Grid'5000 simulation engine for experiment-driver. The simulation engine consists of 10 compute-nodes grouped in homogeneous cluster. The test was also performed by varying the cluster size from 1 to 10 nodes. The machines used in the experiments are equipped with 2 x Intel Xeon Gold 6130 (16 cores/ CPU), 192 GB of RAM, 240 GB

**Table 5** BioSNAP dataset statistics

| Parameter | $\|V\|$ | $\|E\|$ | D | ACC | Nbr of triangles |
|---|---|---|---|---|---|
| Value | 1018524 | 24735503 | 15 | 0.4 | 1466910096 |

SSD + 480 GB SSD + 4.0 TB HDD and 10 Gbps + 100 Gbps Omni-Path Ethernet cables.

### 6.1 Datasets

We used the BioSNAP dataset from SNAP. It is a large single-cell RNA-sequencing dataset of embryonic mouse brain cells. Nodes represent cells in the mouse brain and edges represent nearest neighbor similarities between the cells. We used traditional vertex-partition algorithm to equitably partition the vertices. Each partition is assigned to each machine of the cluster. Table 5 presents the dataset statistics.

### 6.2 Analysis

The experimental tests was performed on each system by running five times the triangle counting algorithm [77] for community detection in the BioSNAP network. The principle of this graph algorithm consists of determining the number of triangles passing through each vertex of the graph. We used Ganglia to monitor cluster performance.

Figure 13 shows the performance of each graph processing system. First, we notice that the increasing number of computing-nodes improves the computation time. We observe that Blogel presents better performance. It is faster and consumes less throughput for data transfer across the cluster. We also observe that Hadoop takes long time to achieve the computation and consumes significant network bandwidth. In general, the systems based on Vertex-centric and Edge-centric programming models are faster but to the detriment of the network bandwidth. However systems whose model is based on GAS model or Partition-centric model consume small network bandwidth: the mean is 75 MB/sec.

### 6.3 Suggestions

In this section we are interested in suggesting the right systems that attempt to tackle some issues related to applications of large-scale graph analytic. Figure 14 provides a brief suggestions of the right systems with respect to their properties of graph storage and visualization, support of dynamic graph processing, in-memory processing, graph mining and distributed processing. Neo4j and Gradoop are two graph database platforms that provide a native storage of the graph-structured data. Both systems support real-time analysis on dynamic graphs. The using of in-memory based systems such as GraphX, Giraph, GraphChi, Giraph++ and GraphLab can resolve the random I/O bottleneck. Neo4j, GraphLab and GraphX deliver powerful graph mining algorithms for improving machine learning predictions. Neo4j, GraphLab and GraphChi are centralized systems, and hence, they cannot handle big graphs efficiently because they are

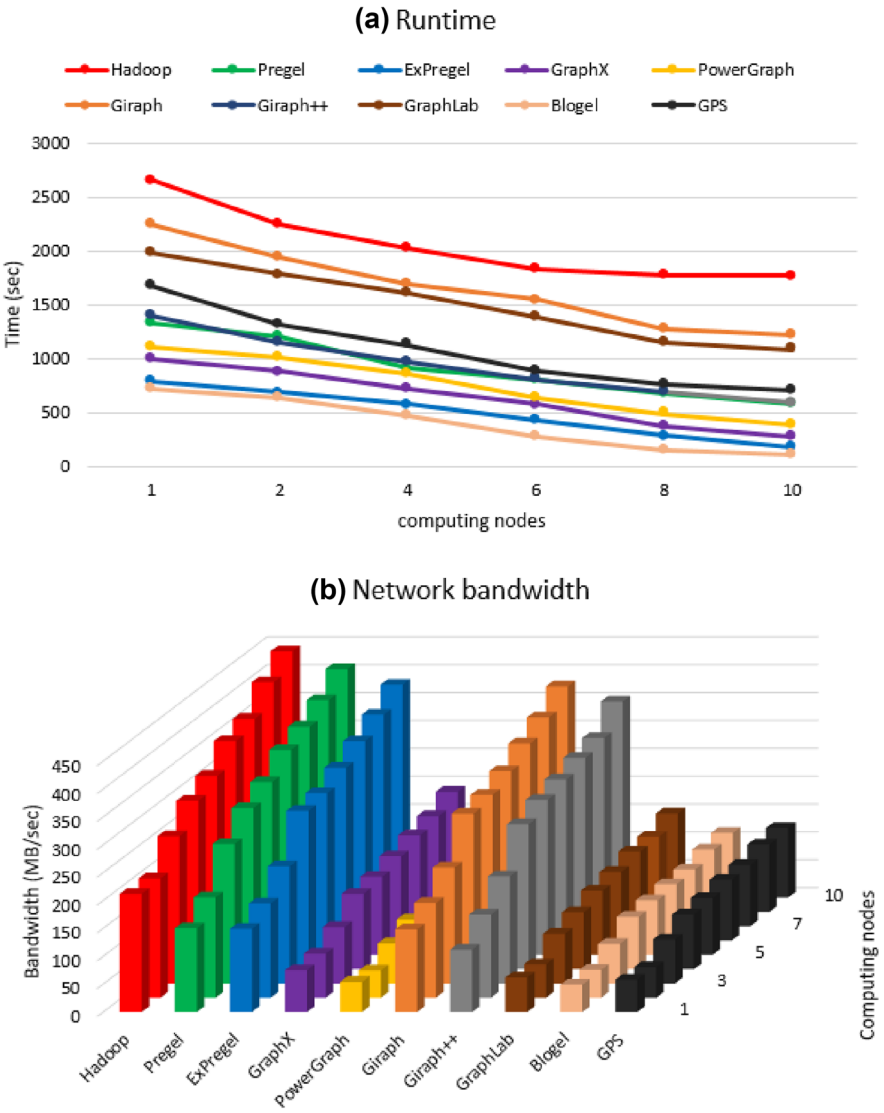


**Fig. 13** Performance of graph systems according to the time complexity and the pairwise network bandwidth between ten machines

limited by the centralized system architecture. Unlike centralized systems, distributed systems can scale the data management to cluster of thousand nodes.

## 7 Future directions and open challenges

Despite recent efforts in graph systems, there are many open problems in harnessing the potential of big data in future trends. These open challenges include the benchmarking

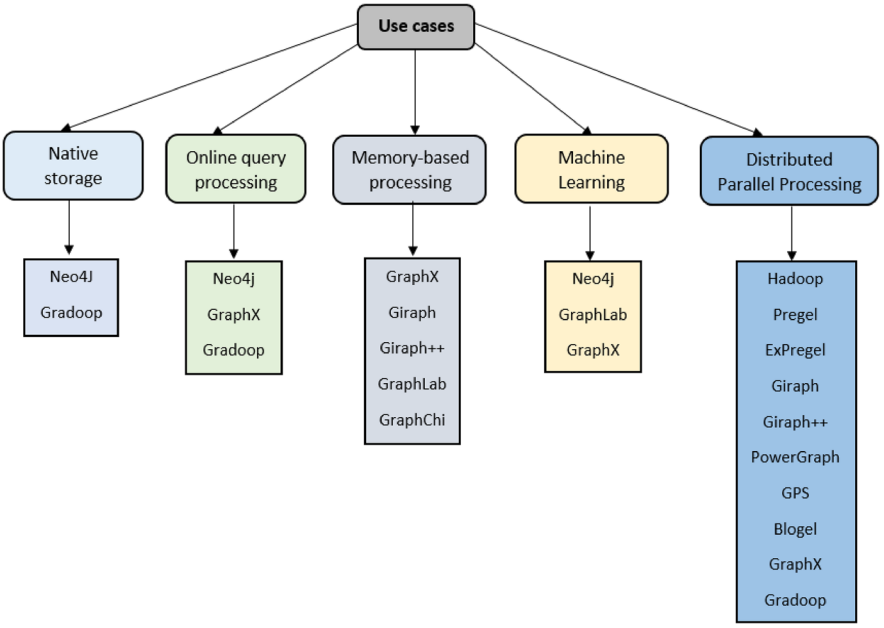


**Fig. 14** Decision tree of graph systems

of graph systems, the integration of graph data, the visualization of graph data and the analysis of dynamic graphs.

### 7.1 Benchmarking of graph systems

The existence of various graph processing systems poses the question of their choice for end-users, because there is no benchmarked model on which the architecture of the different systems is based. This issue opens the challenges of benchmarking the graph systems in order to measure their performance and identify bottlenecks. This depends on various metrics: the complexity of the computation (in pre-processing and processing), the programming model, the size of datasets and the cluster configuration.

### 7.2 Integration of graph data

Before the analysis of graph data, it is necessary to build the graph by integrating heterogeneous data for further processing. Generally, this task passes through the so-called ETL (Extract-Transform-Load) pipeline that consists of merging and synchronizing data from multiple sources into the graph. This task is challenging because the building graphs are based on relationships hidden with large amount of unstructured datasets (not coherent with the notion of structured data) [78]. Due to exponential growth in data, the crucial step is the matching of data from various source. So far, not much work exist on graph ETL. It is a challenging task to integrate huge amounts of

heterogeneous data in graph because it takes into consideration the 4V issues related to big data. Since graph ETL is a data-parallel problem, MapReduce is well-designed for parallel and distributed ETL algorithms. A typical example is GraphBuilder [78], a scalable framework using MapReduce model to offload most of the complexities of graph ETL, including loading, merging and normalization.

### 7.3 Visual analysis of graph data

Visualization of the information contained in big graph is impossible due to limited visual capacity of humans. In fact, the visualization of set of information is based on the cognitive of the human pulling from the bandwidth of the eye and his brain. In this context, the main issue is to propose a visual and interactive representation strategy, drawing on the cognitive capacities of the human, allowing him to see, analyze and understand a large amount of information at a time. The commonly used visual representation is an adjacency matrix [78] or node-link diagram [79]. A good visual representation must satisfy some ergonomic criteria such as the minimization of the number of edge crossings in complex networks (planar graphs), the adaptation of the graph with respect to the screen size and the human recognition capabilities limit (e.g: Neo4j [29] browser user interface).

### 7.4 Analysis of dynamic graph data

Previous strategies for graph mining algorithms focused on static graphs. Currently, most graphs (e.g, web networks, social networks) are dynamic graphs so that the associated vertices and edges change constantly. In this context, existing graph mining algorithms need to be updated to support dynamic graphs such as slowly evolving graphs (e.g road networks, co-authorship networks) and streaming networks (social networks, web networks). More, there is a need for fast analysis on data in motion which power the graph. For example, to quickly identify new posts on Facebook data, products recommendations for users based on their clickstream analysis or to identify on real-time the best path for users mobility taking account of traffic events [12].

## 8 Conclusion

Big data analysis via graph data processing is of great interest in the industry and the research community. Graph data provides flexibility to integrate and handle any kinds of complex data. In this paper, we have surveyed the complexity of large graph partitioning problem and a number of graph partitioning algorithms. Large-scale graph partitioning is NP-Hard problem, which poses the challenges related to the time complexity, workloads, load balancing and the network bandwidth. Moreover, this paper reviewed the well-known graph programming models and established a comparison of graph processing systems. We briefly discussed these many problems remain to be explored in this field.

## References


1. Armstrong, T.G., Ponnekanti, V., Borthakur, D., Callaghan, M.: LinkBench: a database benchmark based on the facebook social graph. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD '13, pp. 1185–1196. ACM, New York (2013)
2. Watts, D.J., Strogatz, S.H.: Collective dynamics of 'small-world' networks. Nature **393**(6684), 440–442 (1998)
3. Travers, J., Milgram, S.: An experimental study of the small world problem. Sociometry **32**(4), 425–443 (1969)
4. Barabási, A.-L., Albert, R.: Emergence of scaling in random networks. Science **286**(5439), 509–512 (1999)
5. Albert, R., Barabási, A.-L.: Statistical mechanics of complex networks. Rev. Mod. Phys. **74**(1), 47 (2002)
6. Abeywickrama, T., Cheema, M.A., Taniar, D.: K-nearest neighbors on road networks: a journey in experimentation and in-memory implementation. Proc. VLDB Endow. **9**(6), 492–503 (2016)
7. Beutel, A.: User behavior modeling with large-scale graph analysis. PhD thesis, University of Trento (2016)
8. Czerepicki, A.: Application of graph databases for transport purposes. Bull. Pol. Acad. Sci. Tech. Sci. **64**(3), 457–466 (2016)
9. Miler, M., Medak, D., Odobašióc, D.: The shortest path algorithm performance comparison in graph and relational database on a transportation network. Promet Traffic Transp. **26**(1), 75–82 (2014)
10. Have, C.T., Jensen, L.J.: Are graph databases ready for bioinformatics? Bioinformatics **29**(24), 3107–3108 (2013)
11. Yoon, B.-H., Kim, S.-K., Kim, S.-Y.: Use of graph database for the integration of heterogeneous biological data. Genomics Inform. **15**(1), 19–27 (2017)
12. Adoni, W.Y.H., Nahhal, T., Aghezzaf, B., Elbyed, A.: MRA*: parallel and distributed path in large-scale graph using MapReduce-A* based approach. In: Ubiquitous Networking, Lecture Notes in Computer Science, pp. 390–401. Springer, Cham (2017)
13. Aridhi, S., d'Orazio, L., Maddouri, M., Mephu, N.E.: Density-based data partitioning strategy to approximate large-scale subgraph mining. Inf. Syst. **48**, 213–223 (2015)
14. Lakhotia, K., Kannan, R., Prasanna, V.: Accelerating pagerank using partition-centric processing. In: 2018 USENIX Annual Technical Conference (USENIX ATC 18). USENIX Association, Boston (2018)
15. Plimpton, S.J., Devine, K.D.: MapReduce in MPI for large-scale graph algorithms. Parallel Comput. **37**(9), 610–632 (2011)
16. Kleinberg, J.M., Kumar, R., Raghavan, P., Rajagopalan, S., Tomkins, A.S.: The web as a graph: measurements, models, and methods. In: Computing and Combinatorics, Lecture Notes in Computer Science, pp. 1–17. Springer, Berlin (1999)
17. Maillo, J., Ramírez, S., Triguero, I., Herrera, F.: kNN-IS: an iterative spark-based design of the k-nearest neighbors classifier for big data. Knowl. Based Syst. **117**, 3–15 (2016)
18. Guo, K., Guo, W., Chen, Y., Qiu, Q., Zhang, Q.: Community discovery by propagating local and global information based on the MapReduce model. Inf. Sci. **323**, 73–93 (2015)
19. Moon, S., Lee, J.-G., Kang, M., Choy, M., Lee, J.-W.: Parallel community detection on large graphs with MapReduce and GraphChi. Data Knowl. Eng. **104**, 17–31 (2016)
20. Dhillon, I.S.: Co-clustering documents and words using bipartite spectral graph partitioning. In: Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '01, pp. 269–274. ACM, New York (2001)
21. Junghanns, M., Petermann, A., Neumann, M., Rahm, E.: Management and analysis of big graph data: current systems and open challenges. In: Handbook of Big Data Technologies, pp. 457–505. Springer (2017)
22. Skhiri, S., Jouili, S.: Large graph mining: recent developments, challenges and potential solutions. In: European Business Intelligence Summer School, pp. 103–124. Springer (2012)
23. Adoni, W.Y.H., Nahhal, T., Aghezzaf, B., Elbyed, A.: The MapReduce-based approach to improve the shortest path computation in large-scale road networks: the case of A* algorithm. J. Big Data **5**(1), 16 (2018)
24. Cossalter, M., Mengshoel, O., Selker, T.: Visualizing and understanding large-scale Bayesian networks. In: Proceedings of the 17th AAAI Conference on Scalable Integration of Analytics and Visualization, AAAIWS'11-17, pp. 12–21. AAAI Press, Menlo Park (2011)

25. Gantz, J., Reinsel, D.: Extracting value from chaos. IDC iView **1142**(2011), 1–12 (2011)
26. Alekseev, V.E., Boliac, R., Korobitsyn, D.V., Lozin, V.V.: NP-hard graph problems and boundary classes of graphs. Theor. Comput. Sci. **389**(1), 219–236 (2007)
27. Cameron, K., Eschen, E.M., Hoáng, C.T., Sritharan, R.: The complexity of the list partition problem for graphs. SIAM J. Discret. Math. **21**(4), 900–929 (2008)
28. Yan, D., Tian, Y., Cheng, J.: Systems for Big Graph Analytics. Springer Briefs in Computer Science. Springer, Cham (2017)
29. Goel, A.: Neo4j Cookbook Harness the Power of Neo4j to Perform Complex Data Analysis over the Course of 75 Easy-to-Follow Recipes. Packt Publishing, Birmingham (2015)
30. Guerrieri, A.: Distributed computing for large-scale graphs. PhD thesis, University of Trento (2015)
31. Yan, D., Huang, L., Jordan, M.I.: Fast approximate spectral clustering. In: Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '09, pp. 907–916. ACM, New York (2009)
32. Martin, C.H.: Spectral clustering: a quick overview. PhD thesis (2012)
33. Filippone, M., Camastra, F., Masulli, F., Rovetta, S.: A survey of kernel and spectral methods for clustering. Pattern Recognit. **41**(1), 176–190 (2008)
34. Ng, A.Y., Jordan, M.I., Weiss, Y.: On spectral clustering: analysis and an algorithm. In: Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic, NIPS'01, pp. 849–856. MIT Press, Cambridge (2001)
35. Kong, H., Akakin, H.C., Sarma, S.E.: A generalized Laplacian of Gaussian filter for Blob detection and its applications. IEEE Trans. Cybern. **43**(6), 1719–1733 (2013)
36. Kamvar, S.D., Klein, D., Manning, C.D.: Spectral learning. In: Proceedings of the 18th International Joint Conference on Artificial Intelligence, IJCAI'03, pp. 561–566. Morgan Kaufmann Publishers Inc., San Francisco (2003)
37. Dhillon, I.S., Guan, Y., Kulis, B.: Kernel k-means: spectral clustering and normalized cuts. In: Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '04, pp. 551–556. ACM, New York (2004)
38. Qiu, Y., Li, R., Li, J., Qiao, S., Wang, G., Yu, J.X., Mao, R.: Efficient structural clustering on probabilistic graphs. IEEE Trans. Knowl. Data Eng. **31**, 1555–1568 (2018)
39. Aggarwal, C.C., Wang, H.: A survey of clustering algorithms for graph data. In: Aggarwal, C.C., Wang, H. (eds.) Managing and Mining Graph Data, vol. 40, pp. 275–301. Springer, Boston (2010)
40. Kernighan, B.W., Lin, S.: An efficient heuristic procedure for partitioning graphs. Bell Syst. Tech. J. **49**(2), 291–307 (1970)
41. Fiduccia, C.M., Mattheyses, R.M.: A linear-time heuristic for improving network partitions. In: Proceedings of the 19th Design Automation Conference, DAC '82, pp. 175–181. IEEE Press, Piscataway (1982)
42. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput. **20**, 359–392 (1998)
43. Karypis, G., Kumar, V.: Multilevel algorithms for multi-constraint graph partitioning. In: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing, SC '98, pp. 1–13. IEEE Computer Society, Washington, DC (1998)
44. Karypis, G., Kumar, V.: Multilevel K-way hypergraph partitioning. In: Proceedings of the 36th Annual ACM/IEEE Design Automation Conference, DAC '99, pp. 343–348. ACM, New York (1999)
45. Schloegel, K., Karypis, G., Kumar, V.: Parallel multilevel algorithms for multi-constraint graph partitioning. In: Euro-Par 2000 Parallel Processing. Lecture Notes in Computer Science, pp. 296–310. Springer, Berlin (2000)
46. Apache Spark-Lightning-Fast Cluster Computing. https://spark.apache.org/
47. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. HotCloud **10**(10), 95 (2010)
48. Kyrola, A., Blelloch, G., Guestrin, C.: GraphChi: large-scale graph computation on just a PC. In: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI'12, pp. 31–46. USENIX Association, Berkeley (2012)
49. Johnson, D.S., Aragon, C.R., McGeoch, L.A., Schevon, C.: Optimization by simulated annealing: an experimental evaluation; part I, graph partitioning. Oper. Res. **37**(6), 865–892 (1989)
50. Rolland, E., Pirkul, H., Glover, F.: Tabu search for graph partitioning. Ann. Oper. Res. **63**, 209–232 (1996)

51. Bui, T.N., Strite, L.C.: An ant system algorithm for graph bisection. In: Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation, GECCO'02, pp. 43–51. Morgan Kaufmann Publishers Inc., San Francisco (2002)
52. Maini, H., Mehrotra, K., Mohan, C., Ranka, S.: Genetic algorithms for graph partitioning and incremental graph partitioning. In: Proceedings of the 1994 ACM/IEEE Conference on Supercomputing, Supercomputing '94, pp. 449–457. IEEE Computer Society Press, Los Alamitos (1994)
53. Kim, J., Hwang, I., Kim, Y.-H., Moon, B.-R.: Genetic approaches for graph partitioning: a survey. In: Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation, GECCO '11, pp. 473–480. ACM, New York (2011)
54. Chen, R., Weng, X., He, B., Choi, B., Yang, M.: Network Performance Aware Graph Partitioning for Large Graph Processing Systems in the Cloud. Nanyang Technological University, Singapore (2014)
55. Aggarwal, C.C., Zhao, Y., Yu, P.S.: A framework for clustering massive graph streams. Stat. Anal. Data Min. **3**(6), 399–416 (2010)
56. Tsourakakis, C., Gkantsidis, C., Radunovic, B., Vojnovic, M.: FENNEL: streaming graph partitioning for massive scale graphs. In: Proceedings of the 7th ACM International Conference on Web Search and Data Mining, WSDM '14, pp. 333–342. ACM, New York (2014)
57. Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: PowerGraph: Distributed graph-parallel computation on natural graphs. In: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI'12, pp. 17–30. USENIX Association, Berkeley (2012)
58. Rahimian, F., Payberah, A.H., Girdzijauskas, S., Jelasity, M., Haridi, S.: A distributed algorithm for large-scale graph partitioning. ACM Trans. Auton. Adapt. Syst. **10**(2), 1–24 (2015)
59. Rahimian, F., Payberah, A.H., Girdzijauskas, S., Haridi, S.: Distributed vertex-cut partitioning. In: IFIP International Conference on Distributed Applications and Interoperable Systems, pp. 186–200. Springer (2014)
60. Stanton, I., Kliot, G.: Streaming graph partitioning for large distributed graphs. In: Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '12, pp. 1222–1230. ACM, New York (2012)
61. Tashkova, K., Korošec, P., Šilc, J.: A distributed multilevel ant-colony algorithm for the multi-way graph partitioning. Int. J. Bio-Inspired Comput. **3**(5), 286–296 (2011)
62. White, T.: Hadoop: The Definitive Guide, 3rd edn. O'Reilly, Beijing (2012)
63. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
64. Vavilapalli, V.K., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B., Baldeschwieler, E., Murthy, A.C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H.: Apache Hadoop YARN: Yet Another Resource Negotiator, pp. 1–16. ACM Press, Santa Clara (2013)
65. Al hajj Hassan, M., Bamha, M.: Handling Limits of High Degree Vertices in Graph Processing Using MapReduce and Pregel, Research Report. Université Orléans, INSA Centre Val de Loire, LIFO EA 4022, Orléans (2017)
66. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM **33**(8), 103–111 (1990)
67. Tian, Y., Balmin, A., Corsten, S.A., Tatikonda, S., McPherson, J.: From think like a vertex to think like a graph. Proc. VLDB Endow. **7**(3), 193–204 (2013)
68. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD '10, pp. 135–146. ACM, New York (2010)
69. Sagharichian, M., Naderi, H., Haghjoo, M.: ExPregel: a new computational model for large-scale graph processing. Concur. Comput. Pract. Exp. **27**(17), 4954–4969 (2015)
70. Ching, A.: Giraph: large-scale graph processing infrastructure on Hadoop. In: Proceedings of the Hadoop Summit, Vol. 11 of 3, Santa Clara, pp. 5–9 (2011)
71. Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., Hellerstein, J.: GraphLab: A new framework for parallel machine learning. In: Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence, UAI'10, pp. 340–349. AUAI Press, Arlington (2010)
72. Salihoglu, S., Widom, J.: Gps: a graph processing system. In: Proceedings of the 25th International Conference on Scientific and Statistical Database Management, SSDBM, vol. 22, pp. 1–12. ACM, New York (2013)
73. Yan, D., Cheng, J., Lu, Y., Ng, W.: Blogel: a block-centric framework for distributed computation on real-world graphs. Proc. VLDB Endow. **7**(14), 1981–1992 (2014)

74. Xin, R.S., Gonzalez, J.E., Franklin, M.J., Stoica, I.: Graphx: a resilient distributed graph system on spark. In: First International Workshop on Graph Data Management Experiences and Systems, GRADES '13, pp. 1–6. ACM, New York (2013)
75. Junghanns, M., Petermann, A., Gómez, K., Rahm, E.: GRADOOP: Scalable Graph Data Management and Analytics with Hadoop. CoRR abs/1506.00548
76. Ghemawat, S., Gobioff, H., Leung, S.-T.: The Google file system. In: ACM SIGOPS Operating Systems Review, vol. 37, pp. 29–43. ACM, New York (2003)
77. Schank, T., Wagner, D.: Finding, counting and listing all triangles in large graphs, an experimental study. In: Nikoletseas, S.E. (ed.) Experimental and Efficient Algorithms. Lecture Notes in Computer Science, pp. 606–609. Springer, Berlin (2005)
78. Jain, N., Liao, G., Willke, T.L.: Graphbuilder: scalable graph ETL framework. In: First International Workshop on Graph Data Management Experiences and Systems, GRADES '13, pp. 1–6. ACM, New York (2013)
79. Chonbodeechalermroong, A., Hewett, R.: Towards visualizing big data with large-scale edge constraint graph drawing. Big Data Res. **10**, 21–32 (2017)