The Journal of Supercomputing (2021) 77:12336–12354
https://doi.org/10.1007/s11227-021-03769-6

Effective partitioning mechanisms for time-evolving graphs in the Flink system

Yi-Hsuan Lee[1] · **Sheng-Jia Jian**[1]

Accepted: 22 March 2021 / Published online: 6 April 2021

Abstract
Graphs are suitable data structures for expressing the relationship between different types of data. With a continuous increase in the graph size, using suitable methods to divide graphs and parallelize the processing load becomes crucial. Balanced graph partitioning has been extensively studied for static and streaming graphs. However, for a time-evolving graph (TEG), whose size and structure are periodically updated, related partitioning methods are lacking. A straightforward approach is to capture snapshots of a TEG and adopt the partitioning methods designed for static or streaming graphs. Although feasible partitioning quality can be expected, the time overhead is high due to frequent repartitioning. This paper proposes two TEG partitioning methods, namely seed and similarity, to decrease the partitioning time. According to the experimental results, on average, seed and similarity require 29–39% of the partitioning time required by snapshot. Moreover, the proposed methods maintain reasonable partitioning quality.

Keywords Graph partitioning · Time-evolving graph · Vertex-cut partitioning · Flink

1 Introduction

With extensive technological advances, including the Internet of things, powerful cloud computing platforms, and widespread smart devices, considerable data are generated from heterogeneous sources every day. With massive increases in the scale of data generation, the storage, processing, analysis, and visualization of big data have become crucial issues [1–3]. Different data structures can be selected to represent data with variety and potential relevance. A graph, which is mainly

✉ Yi-Hsuan Lee
yslee@mail.ntcu.edu.tw

[1] Department of Computer Science, National Taichung University of Education, Taichung, Taiwan, ROC

composed of vertices and edges, is an effective model for representing collections of relationships between entities. Different types of graphs, such as web graphs, social networks, road networks, and citation networks, can be easily plotted [3–5]. By observing how entities behave and interact with each other, these graphs can provide useful information for several applications [1–3].

Since the introduction of the concept of big data, graph processing, which refers to performing computations on large graphs to extract the desired knowledge, has been gradually receiving increased research attention [3–6]. Graph processing algorithms are usually associated with graph types. For example, PageRank is usually applied to a web graph; a single-source shortest path is commonly used in a road network; a connected component and link prediction are useful in a social network; and other scientific or biological graphs are often associated with the centrality, graph matching, and minimum spanning tree algorithms. Many graph processing engines, such as Google Pregel, Apache Giraph, Spark GraphX, Flink Gelly, and CMU PowerGraph, have been designed to process graph-related algorithms efficiently.

Depending on the timing and frequency required to collect and process data, graphs can be divided into three types: static graphs, streaming graphs, and time-evolving graphs (TEGs). A static graph is the most traditional type of graph. With complete data collected in advance, this graph can be directly constructed to represent all entities and their relationships. Moreover, because the structure of a static graph remains unchanged after construction, graph processing algorithms are applied only once. If the data source is continuously input as a stream, the incoming data generate a streaming graph. The authors of [7] defined two streaming models: edge and vertex streaming. Edge streaming assumes that the edges of a graph, which are indicated by source–destination vertex pairs, arrive in a stream. Vertex streaming assumes that the vertices of a graph successively arrive with existing sets of neighboring vertices. On its arrival, an incoming edge or vertex is immediately added to the graph. To accommodate the changing structure of streaming graphs, graph processing algorithms must be applied multiple times when necessary. A TEG is a new graph type that has received relatively low attention [6, 8]. Similar to a streaming graph, a TEG exhibits structural changes over time and its size is expected to increase gradually. However, instead of being modified frequently and irregularly similar to a streaming graph, a TEG is periodically updated and processed. Thus, a time period exists within which all the received update events are added to a TEG simultaneously. Moreover, graph processing algorithms can be applied periodically when a TEG is updated.

When a data graph becomes too large to be processed efficiently by using a single machine, balanced graph partitioning is performed to parallelize the processing loads for multiple machines [9]. Traditional graph partitioning methods are only suitable for static graphs with a limited size [10]. These methods usually achieve the best partitioning quality because the entire information regarding static graphs is known in advance. Streaming graph partitioning (SGP) is another type of graph partitioning method that has been extensively studied [7]. When an edge or a vertex is received, SGP methods must not only assign it to one of the partitions according to the partial graph structures but also meet the low-latency requirements. In a sense, due to the changing structure of a streaming graph, the streaming order

considerably affects the partitioning quality. Currently, partitioning methods for TEGs are lacking [6]. A simple and straightforward approach to partition TEGs is to adopt directly partitioning methods designed for a static or streaming graph. Thus, a snapshot is captured every time a TEG is updated, and this snapshot is treated as a new graph to reassign all edges and vertices. By using the aforementioned simple solution, suitable partitioning quality is expected to be achieved because the snapshot contains complete graph structure information for a certain time. However, the frequent repartitioning can be time-consuming and cause some unnecessary execution overhead.

This paper proposes two TEG partitioning methods, namely seed and similarity, to avoid frequent repartitioning. In the seed method, every time a TEG is periodically updated, seed vertices are extracted from existing partitions to obtain significant information. Each update event received during this period is assigned to one of the seed vertex sets and then merged with the corresponding previous partitions to generate new partitioning results. In the similarity method, update events are partitioned independently without the use of any previous information. These new partitions are then successively merged with the previous partitions, where two partitions with the highest similarity are merged first. Because seed vertices and update events are smaller than the entire snapshot, both the seed and similarity methods are expected to work efficiently. The current study implemented these two methods on the Flink system and evaluated them using real-world graphs. The experimental results prove that both the proposed methods are efficient and can achieve reasonable partitioning quality.

The main contributions of this paper are as follows:

TEGs and their corresponding vertex-cut partitioning problem are defined.

1. Two efficient TEG partitioning methods are used with SGP methods to avoid time-consuming repartitioning and maintain suitable partitioning quality.
2. An edge streaming model is proposed to input a real-world example incrementally for simulating the behavior of TEGs.
3. Performance comparisons are performed to demonstrate the efficiency and scalability of the proposed methods.

The rest of this paper is organized as follows. Section 2 presents a brief review of some related streaming graph and TEG partitioning methods. Section 3 provides detailed descriptions of the proposed methods. Section 4 presents the experimental results for evaluating the partitioning quality. Finally, Sect. 5 presents the conclusions of this study and recommendations for future research.

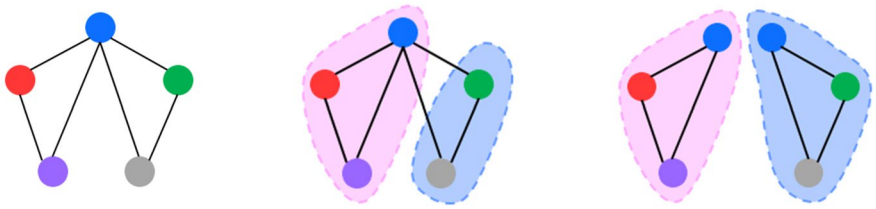

Fig. 1 Two types of partitioning approaches

2 Related work

2.1 Vertex-cut partitioning versus edge-cut partitioning

Graph partitioning is a traditional NP-complete problem in graph theory [9]. As indicated in Fig. 1, two types of partitioning approaches exist: the edge-cut and vertex-cut approaches. The edge-cut approach, which is also called vertex partitioning, involves placing vertices in disjoint sets with balanced sizes. The partitioning goal is to minimize the number of cutting edges because these edges can result in increased communication costs in a distributed computing system. The vertex-cut approach, which is also called edge partitioning, involves placing edges in disjoint sets with balanced sizes. Vertices are replicated to multiple partitions when necessary, and the number of replicated vertices is minimized to reduce the synchronization and storage overheads. Currently, the vertex-cut approach is widely adopted with the edge streaming model in graph processing engines, such as PowerGraph, GraphX, and GraphChi, because many studies have proven that the vertex-cut method is appropriate for skewed power-law graphs, which account for most real-world graphs.

2.2 Partitioning methods for static graphs

Most traditional partitioning methods for static graphs are edge-cut approaches. After loading a graph completely, vertices are divided into disjoint and balanced partitions. METIS [10] is a representative and widely used method for partitioning static graphs. Despite its extremely high partitioning overhead, METIS can achieve a low edge-cutting ratio and is usually set as a baseline for comparison with other partitioning methods. As the graph size increases, METIS uses the coarsening, partitioning, and uncoarsening phases to reduce the partitioning time.

2.3 Streaming graph partitioning

SGP, edge and vertex streaming models were first introduced in [7]. An edge or vertex is assigned to one of the partitions as soon as being received according to the partial graph structures available at that time. To match the rapidly incoming edges or

vertices of a streaming graph, SGP must usually meet the low-latency requirement. LDG [7] and Fennel [11] are two well-known and efficient edge-cut SGP methods that use simple heuristics to assign vertices. Both these methods can achieve balanced partitioning; however, their cutting-edge ratio is significantly affected by the order of the incoming vertices.

Greedy and high-degree replicated first (HDRF) are two vertex-cut approaches mainly aimed at achieving less replicated vertices with an average load balance. Greedy is a graph partitioning method implemented in PowerGraph [12]. Greedy assigns an incoming edge with source and destination vertices on the basis of one of the following cases. First, if both the source and destination vertices are new, the incoming edge is placed in the partition with the least edges. Second, if both the source and destination vertices are already in the same partition, the incoming edge is directly assigned to that partition. Third, if one vertex is new and the other vertex is already present in some partitions, the incoming edge is assigned to the smallest partition that contains the existing vertex. Finally, if both vertices exist in multiple partitions but without intersecting, the incoming edge is placed in the partition containing one vertex and the least number of edges. The last case causes vertex replication.

The main concept of HDRF, which is mostly used for the power-law graph, is to replicate vertices with a higher degree first [13]. When an incoming edge arrives, HDRF shares the same partitioning rules as greedy if one of the aforementioned first three cases is satisfied. For the last case, unlike greedy, which simply assigns the incoming edge according to the load balance, HDRF considers the degree of vertices in the entire graph. In HDRF, an objective function is used to evaluate the fitness of all incoming edge–partition pairs, and the partition with the maximum objective value is selected. Partial HDRF (PHDRF) is a vertex-cut SGP method modified from HDRF [14]. Because a vertex in the vertex-cut approach is replicated in multiple partitions, the vertex degree in different partitions should be counted separately. In PHDRF, an objective function considering the partial vertex degree is defined to select the appropriate partition for an incoming edge. By using PHDRF, the number of replicated vertices is further reduced and balanced partitioning is maintained.

2.4 Partitioning methods for TEGs

Only a few partitioning methods are suitable for TEGs. The authors of [15] defined TEGs and proposed an incremental edge-cut partitioning method for them. According to their definition, when a TEG is updated, all modifications to the graph structure can be represented as four modification events: add node, delete node, add edge, and delete edge. The incremental edge-cut partitioning method comprises partitioning strategies used in static and streaming graphs. Suppose that an initial graph has been well-partitioned using METIS. Then, all modification events are considered to join the partitions sequentially. A new edge or vertex can be placed in the partition, which results in minimal cutting edges and load balance. In the case of a delete event, all corresponding vertices or edges are directly removed. Boundary vertices

and edges are allowed to migrate between partitions; however, the migration costs are very high.

Although the partitioning methods proposed in [16, 17] are SGP methods, they can be roughly extended to partition TEGs due to their window-based design. The method presented in [16] is an edge-cut approach with an edge streaming model. Incoming edges are first stored in a buffer and reordered before being processed when the buffer is full. A condensed spanning tree (CST) is constructed to record continuously the graph structure and partitioning result. As the number of incoming edges increases, if the partition sizes become gradually unbalanced, vertices are repartitioned according to the graph structure information recorded in the CST as long as the migration cost is acceptable. The authors of [17] proposed a vertex-cut approach called adaptive window-based streaming edge partitioning (ADWISE). Similar to the method presented in [16], incoming edges are temporarily stored in a window in ADWISE. Some candidate edges are selected from the window according to a window traversal mechanism, which indicates that they have priority to be assigned to partitions first. Then, heuristic values are counted for all candidate edge–partition pairs, and the pair with the maximal heuristic value is assigned. Using a larger window would result in better partitioning quality but longer latency. ADWISE also provides a mechanism to adjust the window size to balance the partitioning quality and cost.

3 Methods

In this section, we first define TEGs. Then, a simple and straightforward method for TEG partitioning, which is called *snapshot*, is introduced. Subsequently, the two proposed methods, namely seed and similarity, are introduced in detail.

Table 1 Notations used in this paper and their description

Notation	Description
$G^t=(V^t, E^t)$	Data graph at time step t, with vertex set V^t and edge set E^t
P^t	$P^t=(P_1^t, \ldots, P_k^t)$, $P_i^t \cap P_j^t=\phi$, $\cup_1^k P_k^t=E^t$
k	Number of partitions
V_i^t	Vertex set of partition P_i^t
$\lvert V_i^t\rvert$	Number of vertices in partition P_i^t
$\lvert E_i^t\rvert$	Number of edges in partition P_i^t
UE^t	Update event set received between time step $t-1$ and t
$Deg(v_i, P_j^t)$	Degree of vertex v_i in partition P_j^t
SP	$SP=(SP_1, \ldots, SP_k)$, set of seed vertices
$TG=(TV, TE)$	Temporal graph constructed from edges belonging to a given UE^t, with vertex set TV and edge set TE
TP	$TP=(TP_1, \ldots, TP_k)$, $TP_i \cap TP_j=\phi$, $\cup_1^k TP_k=TE$
TV_i	Vertex set of partition TP_i
M	$M=(m_1, \ldots, m_k)$, a vector to indicate how to pair P_i^{t-1} and TP_j

3.1 Definition of TEGs

Table 1 lists the notations used in this paper and their corresponding description. In this paper, the definition of TEGs is extended from that presented in [15]. Suppose that $G^t = (V^t, E^t)$ is a TEG named G at time step t, where V^t and E^t represent the vertex and edge sets of the graph G at time step t, respectively. An update event indicates a step to change the graph structure. We define four types of update events: add vertex, add edge, delete vertex, and delete edge. Let UE^t represent an update event set that arrives between the time steps $t-1$ and t. After applying all update events in UE^t to graph G^{t-1}, the graph G^t at time step t is constructed.

Partition $P^t = (P_1{}^t, \ldots, P_k{}^t)$ is the partitioning result obtained at time step t. In the vertex-cut approach, each P_i^t is essentially an edge set and all P_i^t are disjoint. Let $|V^t|$ and $|E^t|$ denote the numbers of vertices and edges in partition P_i^t, respectively. The notation $VC(P^t)$ represents the total number of vertices in P^t, and $LB(P^t)$ is the ratio between the sizes of the largest and smallest partitions in P^t. The partitioning goal of the vertex-cut approach is to minimize $VC(P^t)$ under the following condition: $LB(P^t) < 1 + \varepsilon$.

3.2 Straightforward method: snapshot

Any SGP method can be simply extended to TEGs by performing repartitioning. Hereinafter, we name this straightforward method snapshot because it directly obtains and partitions a snapshot of graph G^t at time t.

Figure 2 lists the pseudocode of the snapshot method. The initial graph G^0 can be partitioned using any static or SGP method to obtain partition P^0. When UE^t is received at time step t, the update events in UE^t are merged into G^{t-1} to construct a graph G^t, which is treated as a new graph and partitioned using any selected method to obtain P^t. The snapshot method is expected to provide suitable

Fig. 2 Pseudocode of the snapshot TEG partitioning method

```
Input: Initial data graph G^0, update events UE^t
Output: Partitioning result P^t

1.  Read initial data graph G^0
2.  P^0 ← Partition G^0 using any SGP method
3.  t ← 1
4.  do
5.      Receive UE^t
6.      G^t ← G^(t−1) ∪ UE^t
7.      P^t ← Partition G^t using any SGP method
8.      t ← t + 1
9.  until all UE^t are received
10. Return P^t
```

partitioning quality; however, this method is very time-consuming due to the periodic repartitioning of the entire graph G^t at each time step t.

3.3 Proposed method: seed

To reduce the partitioning cost, an obvious approach is to avoid repetitive repartitions. The authors of [15] proposed an incremental method to process modification events successively and assigned incoming vertices or edges by considering the structure information of the entire graph. Thus, the aforementioned method is equivalent to the conventional SGP method even though the vertices or edges are assigned in batches and not instantly. However, the frequent recording and extraction of the structure of an enlarged graph are highly expensive. In addition, the authors of [15] proposed an edge-cut approach, which is not the main approach used in current graph processing engines.

This paper proposes a vertex-cut TEG partitioning method named seed, which focuses on designing a strategy to process the update events in UE^t and merge them with P^{t-1}. Hence, the seed method must be used with an SGP method to assign

Input: Initial data graph G^0, update events UE^t
Output: Partitioning result P^t

1. Read initial data graph G^0
2. $P^0 \leftarrow$ Partition G^0 using any SGP method
3. $t \leftarrow 1$
4. **do**
5. Receive UE^t
6. $\forall\ P_i^{t-1} \in P^{t-1}$
7. $SP_i \leftarrow \phi$
8. Sort $v_j \in V_i^{t-1}$ according to $Deg(v_i, P_j^{t-1})$ in descending order
9. $n \leftarrow \lambda \times |V_i^{t-1}|$
10. **for** $j = 1$ to n
11. $SP_i \leftarrow SP_i \cup v_j$
12. $\forall\ e_{uv} \in UE^t$
13. Select SP_i using any SGP method
14. $SP_i \leftarrow SP_i \cup e_{uv}$
15. $P_i^t \leftarrow P_i^{t-1} \cup SP_i \quad 1 \le i \le k$
16. $G^t \leftarrow G^{t-1} \cup UE^t$
17. $t \leftarrow t + 1$
18. **until** all UE^t are received
19. **Return** P^t

Fig. 3 Pseudocode of the seed method

incoming edges or vertices. In this study, the greedy, HDRF, and PHDRF methods were selected for this purpose.

Figure 3 lists the pseudocode of the seed method. Similar to the snapshot method, in the seed method, first, the partition P^0 of the initial graph G^0 is obtained using any SGP method. Next, each time UE^t is received, the seed vertices $SP=(SP_1, \ldots, SP_k)$ are extracted from $P^{t-1}=(P_1^{t-1}, \ldots, P_k^{t-1})$ to obtain significant structure information regarding the partition P^{t-1}. Seed vertices are defined as vertices with the highest degree in P_i^{t-1}, and their number is adjustable. For simplicity, in this paper, all update events in UE^t are assumed to be edge additions, and the other types of update events are reserved for future research. Therefore, UE^t can be considered as an edge set, and each incoming edge is assigned to the most appropriate SP_i by using any SGP method. Because the size of SP is considerably smaller than that of P^{t-1}, the partitioning process of seed is expected to be more efficient than that of snapshot. Finally, each SP_i is united with the corresponding P_i^{t-1} to obtain partition P_i^t, and the edges in UE^t are inserted into G^{t-1} to construct a graph G^t. The aforementioned steps are conducted periodically until all UE^t are received and processed.

3.4 Proposed method: similarity

Although the two proposed partitioning methods use different partitioning strategies, they process the update events in UE^t by considering the structure of the recent graph G^{t-1}. The snapshot method directly merges UE^t into G^{t-1} and repartitions the new graph G^t, whereas seed assigns each incoming edge or vertex individually to an existing partition Pi^{t-1}. This paper proposes the similarity method to partition a TEG incrementally. Incoming edges or vertices are independently divided into k partitions and then added to an existing partition P^{t-1} in appropriate pairs. As is the case for the seed method, the similarity method must also be used with an SGP method, such as greedy, HDRF, or PHDRF.

Figure 4 lists the pseudocode of similarity. The initial graph G^0 is partitioned into P^0 by using the selected SGP method. When UE^t is received at time step t, a temporary graph $TG=(TV, TE)$ is constructed according to the incoming edges and vertices. Then, TG is used in any SGP method to obtain $TP=(TP_1, \ldots, TP_k)$ independently without considering the structure information obtained from the recent graph G^{t-1}. Subsequently, each temporary partition TP_i is merged with a suitable P_i^{t-1}. In Eq. (1), *SimilarRatio* is defined as the ratio of the number of intersecting vertices between two partitions:

$$SimilarRatio = \frac{\left|TV_i \cap V_j^{t-1}\right|}{\left|V_j^{t-1}\right|} \quad 1 \le i, j \le k \tag{1}$$

If the partition pair (TP_i, P_j^{t-1}) has a larger *SimilarRatio* than other partition pairs do, it should be merged first because its elements are more similar than those of other pairs. To reduce the partitioning time, not all $k \times k$ pairs are validated in the similarity method. However, all temporary partitions TP_i are validated sequentially

Input: Initial data graph G^0, update events UE^t
Output: Partitioning result P^t

1. Read initial data graph G^0
2. $P^0 \leftarrow$ Partition G^0 using any SGP method
3. $t \leftarrow 1$
4. **do**
5. Receive UE^t
6. $TG \leftarrow UE^t$ // Construct a temporary graph using edges in UE^t
7. $TP \leftarrow$ Partition TG using any SGP method
8. $\forall\, m_i \in M \quad m_i \leftarrow -1$
9. **for** $i = 1$ to k
10. $Maxrate \leftarrow 0$
11. **for** $j = 1$ to k
12. **if** $m_j == -1$
13. $SimilarRatio(i, j) \leftarrow |TV_i \cap V_j^{t-1}| \,/\, |V_j^{t-1}|$
14. **if** $SimilarRatio(i, j) > Maxrate$
15. $pair \leftarrow j$
16. $m_{pair} \leftarrow i$
17. $P_i^t \leftarrow P_i^{t-1} \cup TP_{mi} \quad 1 \le i \le k$
18. $G^t \leftarrow G^{t-1} \cup TG$
19. $t \leftarrow t + 1$
20. **until** all UE^t are received
21. **Return** P^t

Fig. 4 Pseudocode of the similarity method

to obtain the most similar P_j^{t-1} that is not yet matched; thus, a total of $k \times (k-1)/2$ pairs are validated. Finally, partition P^t is obtained by pairing TP and P^{t-1}, and the temporary graph TG is merged with G^{t-1} to construct a graph G^t. The aforementioned steps are repeated periodically until all UE^t are received and processed.

4 Experimental results

4.1 Evaluation platform

In this study, Apache Flink was selected as the evaluation platform. Apace Flink is a distributed streaming-processing framework that enables the execution of batch and streaming jobs with the DataSet API and DataStream API, respectively. The authors of [18] widely used SGP methods on Flink. We deployed Flink v1.8.0 with Java8 on a machine with Intel® Core™ i3-8100 CPU @ 3.6 GHz, 32 GB of RAM, and Linux OS. Three TEG partitioning methods, namely snapshot, seed,

Table 2 Characteristics of the selected datasets

Dataset	Vertices	Edges	Type	ACC[a]	Vertices in largest SCC[b]	Edges in largest SCC
wiki-Vote	7115	103,689	Social	0.1409	1300 (0.183)	39,456 (0.381)
soc-Slashdot0902	82,168	948,464	Social	0.0603	71,307 (0.868)	912,381 (0.962)
web-Google	875,713	5,105,039	Web	0.5143	434,818 (0.497)	3,419,124 (0.670)
web-Stanford	281,903	2,312,497	Web	0.5976	150,532 (0.534)	1,576,314 (0.682)
Amazon0302	262,111	1,234,877	Product	0.4198	241,761 (0.922)	1,131,217 (0.916)
com-DBLP	317,080	1,049,866	Community	0.6324	317,080 (1.000)	1,049,866 (1.000)

[a]Average clustering coefficient

[b]Strongest connected component

and similarity, were implemented with a vertex-cut SGP method (greedy, HDRF, or PHDRF) on Flink and evaluated using real-world graphs. Table 2 lists the basic characteristics of the wiki-Vote, web-Google, web-Stanford, Amazon0302, soc-Slashdot0902, and com-DBLP datasets selected from Stanford Network Analysis Platform (SNAP) [19] for evaluation.

4.2 Evaluation metrics

The following three evaluation metrics are used in this paper: partitioning time, replication factor, and load balance. Partitioning time is the execution time required for performing the partitioning method. Replication factor, which is defined in Eq. (2), indicates the average number of replicas for all vertices. Equation (3) defines the load balance as the ratio between the sizes of partitions with the maximum and least number of edges.

$$Replication\ \ factor = \frac{\sum_1^k \left|V_i^t\right|}{|V^t|} \tag{2}$$

$$Load\ balance = \frac{Max\left|E_i^t\right|}{Min\left|E_i^t\right|} \quad 1 \le i \le k \tag{3}$$

To simulate the incoming model of a TEG, a data graph is assumed to be input in five batches. For a given dataset, 40% of its edges are randomly selected and input in the first batch to construct the initial graph G^0. The remaining edges of the dataset are incrementally added in the second to fifth batches with the update event sets UE^1–UE^4, where each UE^t contains 15% of the incoming edges. In seed, the parameter λ is defaulted to 0.1, which means that 10% vertices with highest degree in partition P_i^{t-1} are extracted as the seed vertices SP_i.

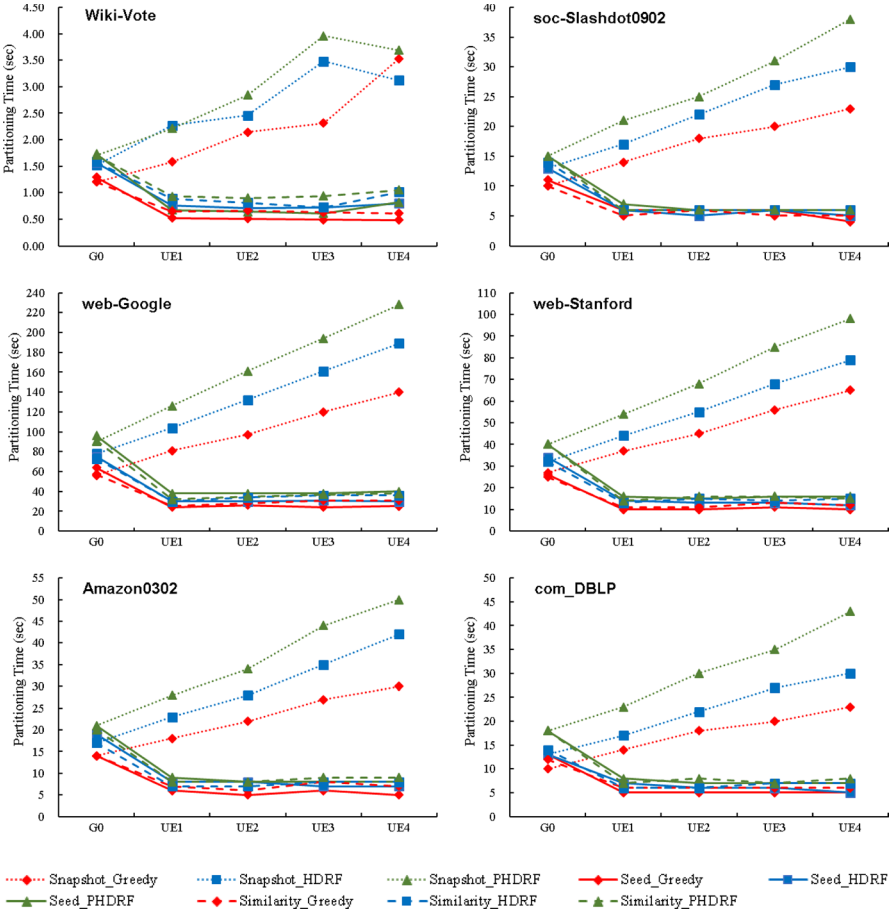

Fig. 5 Partitioning time of individual batches ($k = 32$)

4.3 Evaluation results

Figure 5 displays the partitioning times required by different methods to process individual batches when $k = y32$. Nine methods, including three TEG partitioning methods (i.e., snapshot, seed, and similarity) and three SGP methods (i.e., greedy, HDRF, and PHDRF), were used in the experiments. As displayed in Fig. 5, the time required to partition the initial graph G^0 is completely determined by the SGP method used. Greedy was more efficient than HDRF and PHDRF were for all datasets because HDRF and PHDRF require additional time to consider the vertex degree. As the update events UE^1–UE^4 were successively received, the partitioning time required for the snapshot method increased, whereas those required for seed and similarity remained stable. The main reason for this result is that snapshot repeatedly reassigns all edges of the enlarging graph G^t in each

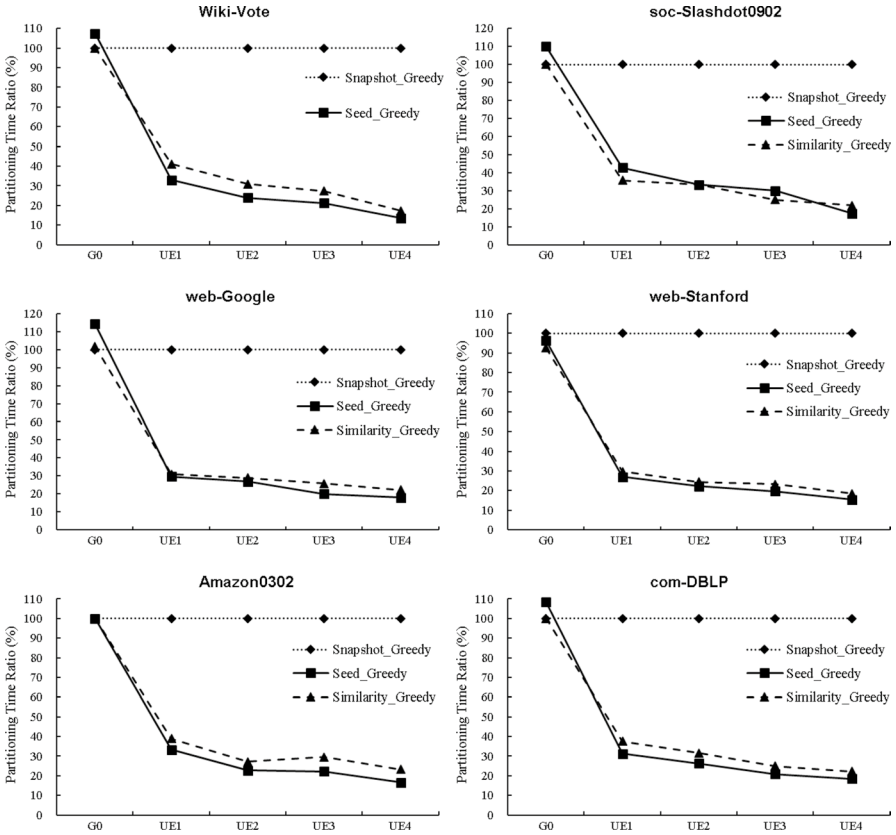

Fig. 6 Partitioning time ratios for individual batches ($k = 32$)

batch, which is expected to be a time-consuming process that is dependent on the graph size. Seed and similarity only focus on how to assign new edges received in a batch and merge them with the recent partitions. In all experiments, UE^1–UE^4 contained 15% of the edges of the entire dataset. Therefore, the partitioning times required by seed and similarity to process UE^1–UE^4 were not only close but also lower than those required by seed and similarity to process the initial graph G^0.

Figure 6 presents the time ratios of *Seed_Greedy* and *Similarity_Greedy* in each batch when the partitioning time of *Snapshot_Greedy* was set to 100%. For processing UE^1 in different datasets, *Seed_Greedy* and *Similarity_Greedy* spent approximately 27–43% and 30–41% of the partitioning time of *Snapshot_Greedy*, respectively. For processing UE^4, the partitioning times spent by *Seed_Greedy* and *Similarity_Greedy* were only 14–19% and 17–23%, respectively, of that spent by *Snapshot_Greedy*. The aforementioned results are expected because *Snapshot_Greedy* repartitions the entire enlarging graph G^1–G^4, whereas *Seed_Greedy* and *Similarity_Greedy* only process UE^1–UE^4 with the same size.

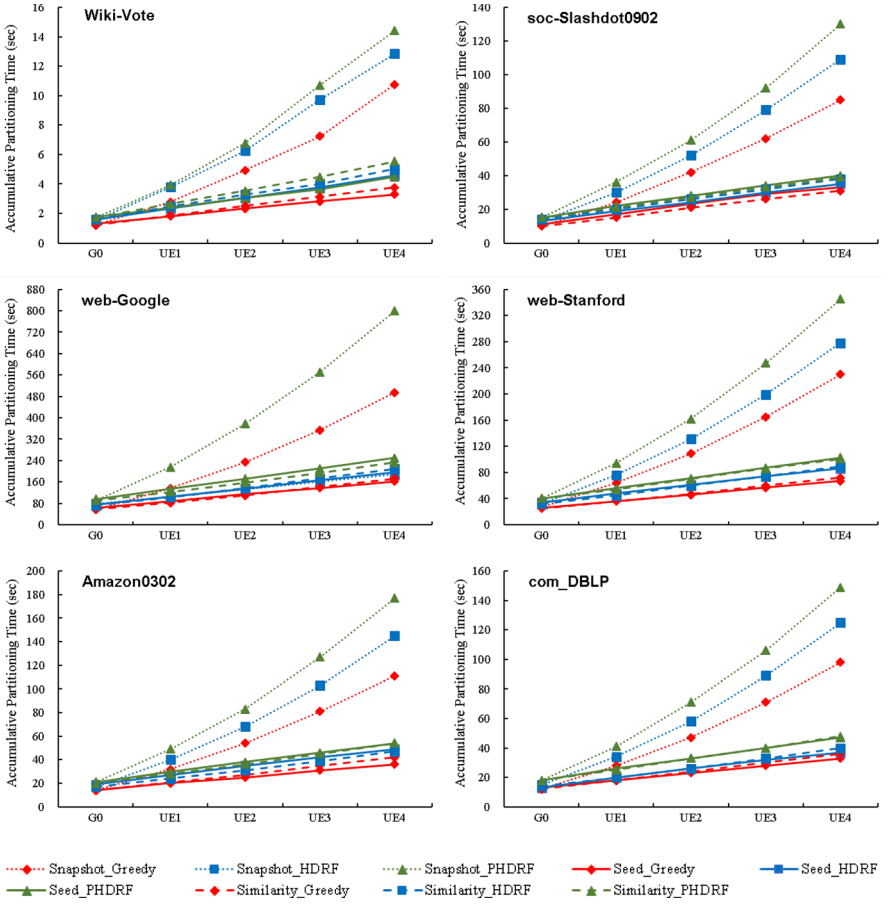

Fig. 7 Accumulative partitioning times of individual batches ($k = 32$)

Similar time ratios to those displayed in Fig. 6 were obtained when replacing greedy with HDRF and PHDRF.

Figure 7 displays the accumulative partitioning time of the nine methods for different datasets. Every time a TEG was updated, the partitioning time accumulated by snapshot was short. The seed and similarity methods exhibited a flat time accumulation trend, and the partitioning time difference between the applied SGP methods was relatively small. In the experiments, a dataset was assumed to be input with five batches. After processing an entire dataset, the accumulative partitioning time spent using seed or similarity was approximately 29–39% of that spent using snapshot. According to the experimental results presented in Figs. 5, 6 and 7, the partitioning time is considerably lower when using both the proposed methods than when using the snapshot method because repartitioning is avoided when using both the proposed methods. Greedy was more efficient than HDRF and PHDRF were;

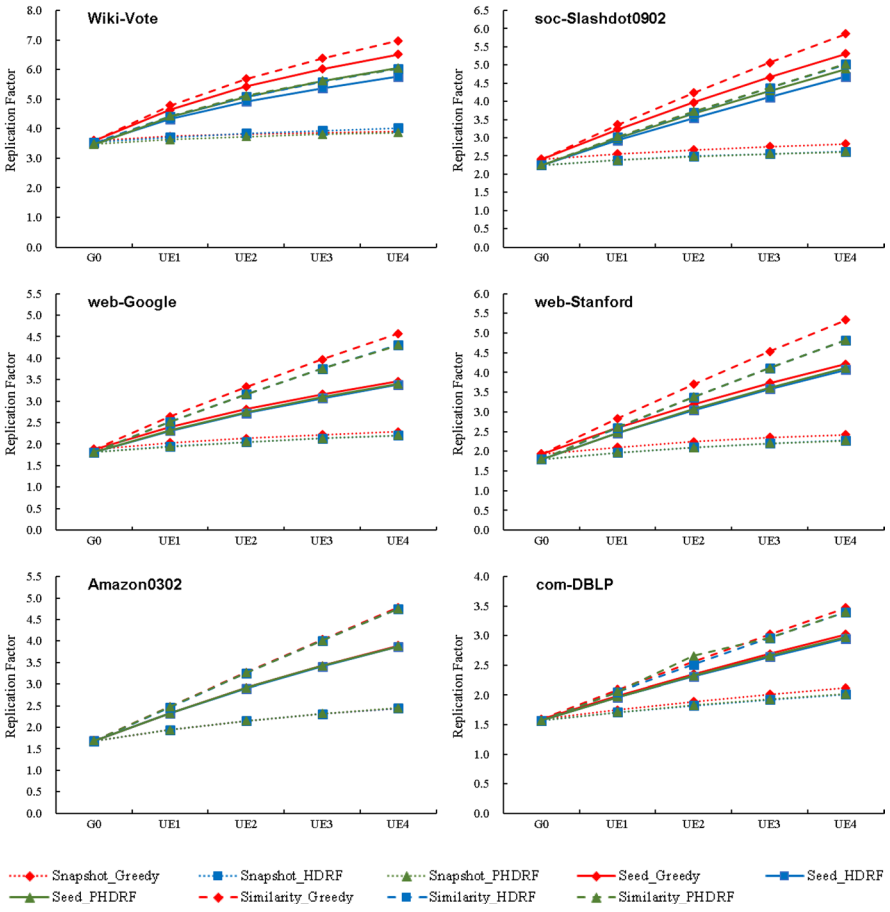

Fig. 8 Replication factor of individual batches ($k = 32$)

however, the partitioning time difference was only evident when these methods were applied with snapshot.

Figure 8 presents the replication factors of different methods in individual batches when k was assumed to be 32. The replication factor refers to the average number of replicas a vertex has after partitioning [Eq. (2)]. Because a higher replication factor directly causes higher synchronization and storage overhead, almost all vertex-cut approaches are designed to reduce the replication factor. Because the snapshot method repartitions graph G^t, which contains all the received edges at time step t, this method is expected to achieve balanced partitioning results with a suitable replication factor. However, when designing seed and similarity, the primary goal is to reduce the partitioning time. Therefore, the incoming edges are assigned only according to partial graph structure information, which might result in suboptimal assignment and a high replication factor. As displayed in Fig. 8, for the arriving update events, the replication factors of seed and similarity were higher than

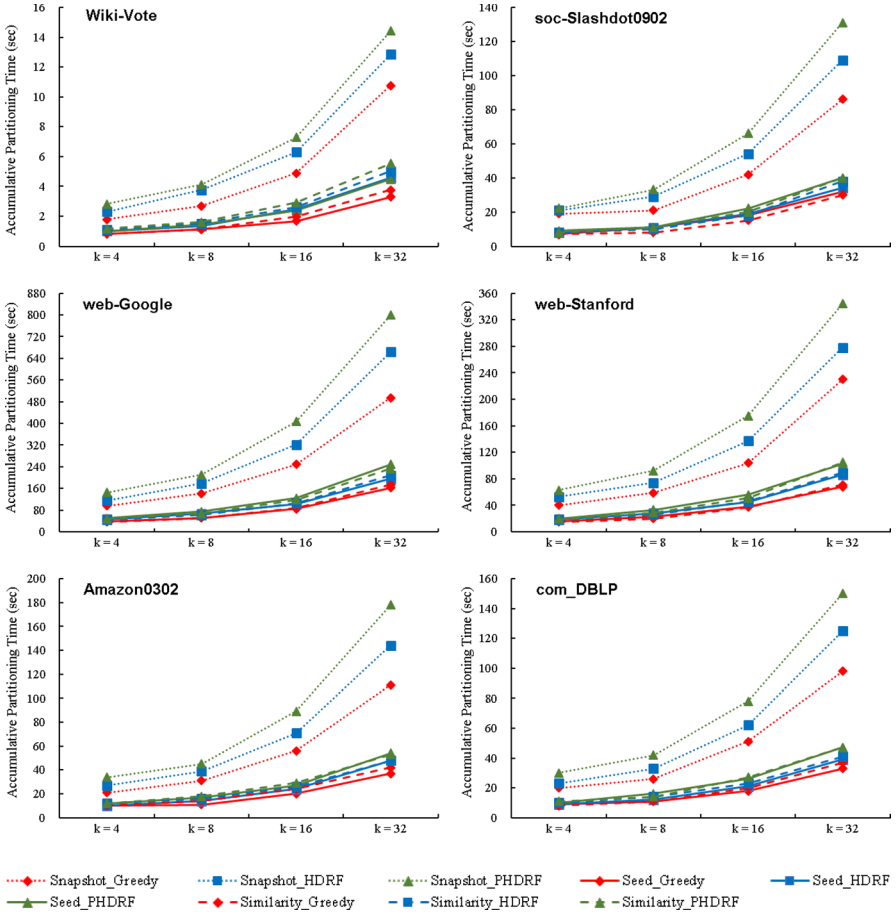

Fig. 9 Accumulative partitioning time for different numbers of partitions

that of snapshot. HDRF and PHDRF usually achieved lower replication factors than greedy. In summary, both the proposed TEG partitioning methods, namely seed and similarity, can significantly reduce the partitioning time, as expected. Although the resulting replication factors are higher than those obtained using the straightforward snapshot method, they are still within a reasonable range and acceptable.

Figures 9 and 10 depict the scalability of the nine adopted methods. Figure 9 illustrates the accumulative partitioning time of the nine methods for different datasets when assuming $k=4$, 8, 16, or 32. In all cases, for the same reason as that presented in Fig. 5, the partitioning times of seed and similarity were lower than that of snapshot. As the number of partitions increased, the nine methods required a longer time to partition TEGs. The time growth trends of seed and similarity were relatively flat, which indicates that these methods are scalable. In our experiments, for $k=4$, seed and similarity required approximately 32–48% and 30–48%, respectively, of the time required by snapshot for processing an entire dataset. For $k=32$, seed

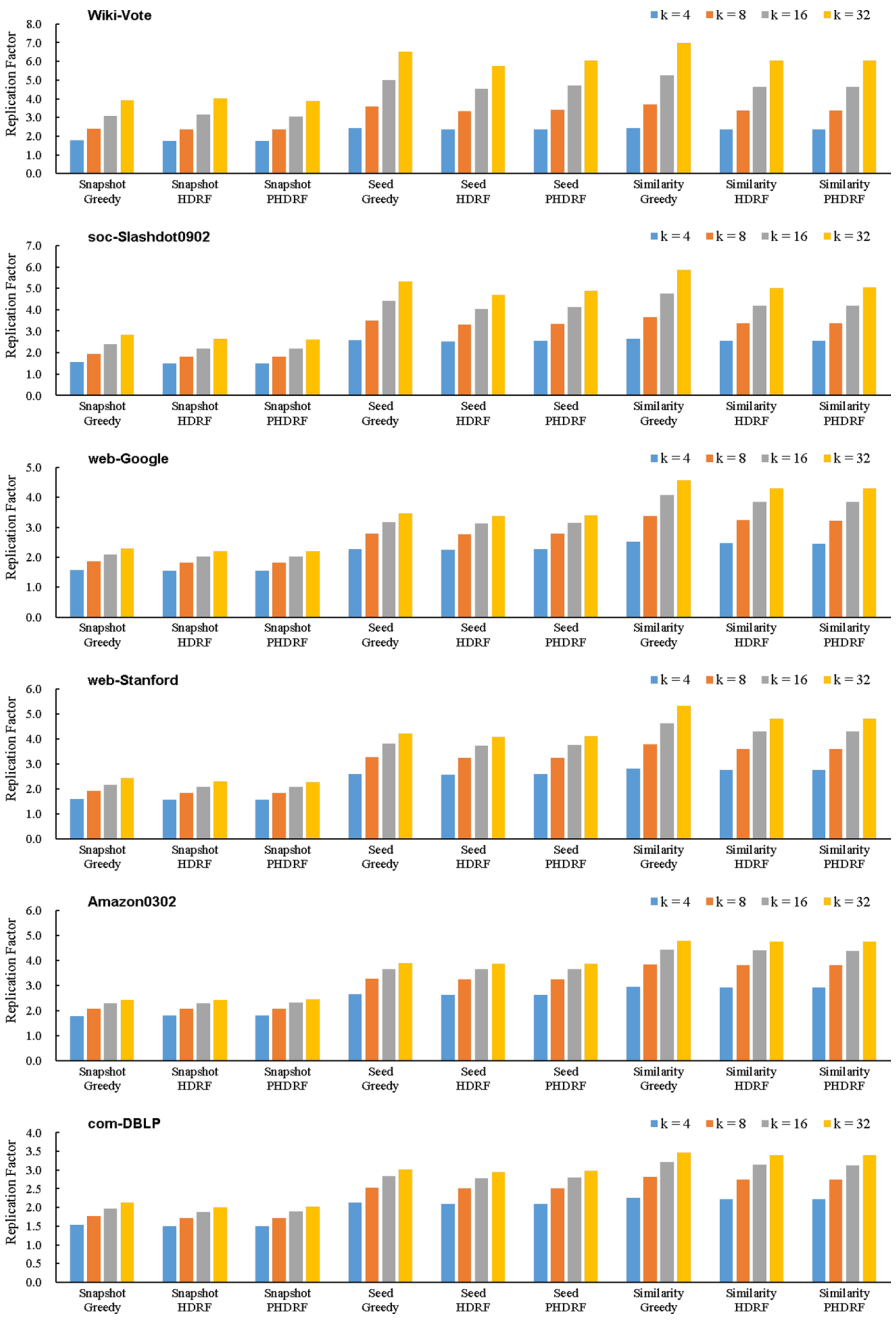

Fig. 10 Replication factor for different numbers of partitions

and similarity required approximately 30–37% and 29–39%, respectively, of the time required by snapshot for processing an entire dataset. These values are expected to decrease further if k is increased.

The replication factors obtained for different methods when assuming $k=4$, 8, 16, or 32 are displayed in Fig. 10. When a data graph is partitioned using a vertex-cut approach, its vertices are separated into k partitions with necessary replication. A high-degree vertex usually has replicas in all partitions, which is beneficial for obtaining balanced partitioning results. Hence, the overall replication factor is expected to increase as the value of k increases because a high-degree vertex is replicated an increasing number of times. As presented in Fig. 10, for all the methods, the number of replication factors increased with that of the partitions. When used with HDRF or PHDRF, the snapshot method consistently achieved the lowest replications factors irrespective of the k value; however, the results obtained using other methods were still within an acceptable range.

Balanced partitioning is a fundamental requirement that must be satisfied by all graph partitioning methods. Because snapshot, seed, and similarity apply the heuristics defined in greedy, HDRF, and PHDRF to assign incoming edges to partitions, balanced partitioning is achieved with snapshot, seed, and similarity. In the experiments, the nine adopted methods achieved load balance values of less than 1.004 in all the evaluation cases. This result indicates that both seed and similarity are balanced TEG partitioning methods.

5 Conclusions

This paper proposes two vertex-cut TEG partitioning methods, namely seed and similarity. The primary goal of both these methods is to avoid repeatedly repartitioning snapshots when a TEG is periodically updated, which can effectively reduce the partitioning time. The two proposed methods were implemented with the straightforward snapshot method on the Flink system and evaluated using real-world data graphs. The experimental results indicate that both seed and similarity are more balanced and time-efficient than snapshot is. The replication factors of the proposed methods were higher than those of snapshot but still maintained reasonable partitioning quality.

Acknowledgements This study was sponsored by the Ministry of Science and Technology, Taiwan, R.O.C., under contract numbers MOST 106-2221-E-142-005 and MOST 107-2221-E-142-006.

References

1. Oussous A, Benjelloun FZ, AitLahcen A, Belfkih S (2018) Big data technologies: a survey. J King Saud UnivComputInfScinces 30(4):431–448
2. Khan N, Yaqoob I, Hashem IA, Inayat Z, Ali WK, Alam M, Shiraz M, Gani A (2014) Big data: Survey, technologies, opportunities, and challenges. Sci World J. https://doi.org/10.1155/2014/712826

3. Zheng Y, Capra L, Wolfson O, Yang H (2014) Urban computing: concepts, methodologies, and applications. ACM Trans IntellSystTechnol. https://doi.org/10.1145/2629592
4. Elshawi R, Batarfi O, Fayoumi A, Bamawi A, Sakr S (2015) Big graph processing systems: state-of-the-art and open challenges. In: Proceedings of the 1st International Conference on Big Data Computing Service and Applications. https://doi.org/10.1109/BigDataService. 2015.11
5. Lim SH, Lee S, Ganesh G, Brown TC, Sukumar SR (2015) Graph processing platforms at scale: practices and experiences". Proc ISPASS. https://doi.org/10.1109/ISPASS.2015.7095783
6. Sharma S, Chou J (2020) A survey of computation techniques on time evolving graphs. Int J Big Data Intell 7(1):1–14. https://doi.org/10.1504/IJBDI.2020.106151
7. Stanton I, Kliot G (2012) Streaming graph partitioning for large distributed graphs. In: Proceedings of KDD'12. https://doi.org/10.1145/2339530.2339722
8. Iyer AP, Li LE, Das T, Stoica I (2016) Time-evolving graph processing at scale. In: Proceedings of GRADES'16. https://doi.org/10.1145/2960414.2960419
9. Buluc A, Meyerhenke H, Safro I, Sanders, P, Schulz C (2015) Recent advances in graph partitioning. arXiv:1311.3144v3[cs.DS].
10. Karypis G, Kumar V (1999) A fast and high quality multilevel scheme for partitioning irregular graphs. J Sci Comput 20(1):359–392
11. Tsourakakis C, Gkantsidis C, Radunovic B, Vojnovic M (2014) Fennel: streaming graph partitioning for massive scale graphs. In: Proceedings of WSDM'14. https://doi.org/10.1145/2556195.2556213
12. Gonzalez JE, Low Y, Gu H, Bickson D, Guestrin C (2012) PowerGraph: distributed graph-parallel computation on natural graphs. In: Proceedings of OSDI'12, pp 17–30
13. Petroni F, Querzoni L, Daudjee K, Kamali S, Iacoboni G (2015) HDRF: stream-based partitioning for power-law graphs. In: Proceedings of CIKM'15. https://doi.org/10.1145/2806416.2806424
14. Jiang JY, Lee YH, Lai KC (2018) A streaming graph partitioning approach on distributed systems. In: Proceedings of 8th International Conference on Engineering and Applied Science, pp 229–237.
15. Abdolrashidi A, Ramaswamy L (2015) Incremental partitioning of large time-evolving graphs. Proc CIC. https://doi.org/10.1109/CIC.2015.37
16. Filippidou I, Kotidis Y (2015) Online and on-demand partitioning of streaming graphs. ProcIntConf Big Data. https://doi.org/10.1109/BigData.2015.7363735
17. Mayer C, Mayer R, Tariq MA, Geppert H, Laich L, Rieger L, Rothermel K (2018) ADWISE: adaptive window-based streaming edge partitioning for high-speed graph processing. In: Proceedings of ICDCS. https://doi.org/10.1109/ICDCS.2018.00072
18. Abbas A, Kalavri V, Carbone P, Vlassov V (2018) Streaming graph partitioning: an experimental study. In: Proceedings of the VLDB endowment. https://doi.org/10.14778/3236187.3236208
19. Leskovec J, Krevl A (2014) SNAP datasets: Stanford large network dataset collection, Jun. 2014, https://snap.stanford.edu/data. Accessed 18 Mar 2021