Data Science and Engineering (2021) 6:163–179
https://doi.org/10.1007/s41019-021-00156-2

A Workload-Adaptive Streaming Partitioner for Distributed Graph Stores

Ali Davoudian[1] · Liu Chen[2] · Hongwei Tu[2] · Mengchi Liu[3]

Received: 27 October 2020 / Revised: 4 February 2021 / Accepted: 25 March 2021 / Published online: 15 April 2021
© The Author(s) 2021

Abstract

Streaming graph partitioning methods have recently gained attention due to their ability to scale to very large graphs with limited resources. However, many such methods do not consider workload and graph characteristics. This may degrade the performance of queries by increasing inter-node communication and computational load imbalance. Moreover, existing workload-aware methods cannot consistently provide good performance as they do not consider dynamic workloads that keep emerging in graph applications. We address these issues by proposing a novel workload-adaptive streaming partitioner named WASP, that aims to achieve low-latency and high-throughput online graph queries. As each workload typically contains frequent query patterns, WASP exploits the existing workload to capture active vertices and edges which are frequently visited and traversed, respectively. This information is used to heuristically improve the quality of partitions either by avoiding the concentration of active vertices in a few partitions proportional to their visit frequencies or by reducing the probability of the cut of active edges proportional to their traversal frequencies. In order to assess the impact of WASP on a graph store and to show how easily the approach can be plugged on top of the system, we exploit it in a distributed graph-based RDF store. Our experiments over three synthetic and real-world graph datasets and the corresponding static and dynamic query workloads show that WASP achieves a better query performance against state-of-the-art graph partitioners, especially in dynamic query workloads.

Keywords Graph partitioning · Streaming · Workload-adaptive · Topology-aware · Dynamic workloads

1 Introduction

Modern real-world graphs such as social networks and web graphs are typically big, constantly changing and simultaneously queried by many clients. Hence, it is no longer feasible for a single database server to provide computing resources for managing such graphs and still be capable to provide quality services to their client applications [13]. A traditional solution is resorting to the vertical scaling of servers and full replication, which is costly or even unattainable. This leads to the cost-effective design of distributed graph stores[1] that rely on the horizontal partitioning or sharding and parallel processing of graph data over large clusters of cheap commodity servers.

Existing graph partitioning strategies are mostly designed for static graphs. When they are used for dynamic graphs, whose vertices and edges are continuously changing (e.g., the semantic Web and social networks), it requires the heavyweight repartitioning of graphs after a batch of changes, which may take hours in big graphs [41, 48, 54]. For this reason, some graph stores such as OrientDB,[2] Titan[3] and Microsoft Trinity [43] use naive partitioning methods such as the random hash-based partitioning,[4] whereby each vertex along with its incident edges are assigned to a server

✉ Mengchi Liu
liumengchi@scnu.edu.cn

[1] School of Computer Science, Carleton University, Ottawa, Canada

[2] School of Computer Science, Wuhan University, Wuhan, China

[3] Guangzhou Key Laboratory of Big Data and Intelligent Education, School of Computer Science, South China Normal University, Guangzhou, China

[1] According to [8] contemporary database systems are referred to as “data stores” where more flexible data models are used and DBMS functionalities may not be fully provided.

[2] http://orientdb.com/.

[3] https://thinkaurelius.github.io/titan/.

[4] It is a de-facto standard in many data stores due to creating balanced partitions and its decentralized nature.

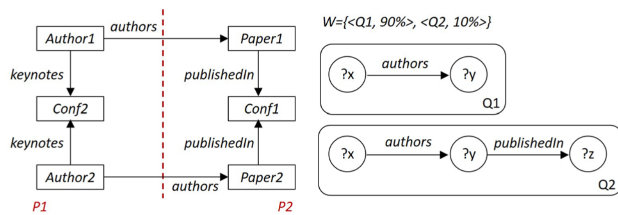


Fig. 1 A balanced min-cut example

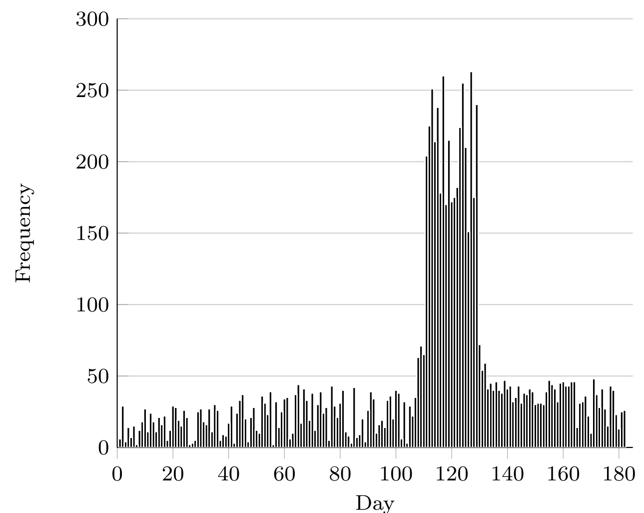


Fig. 2 The arrival rate of the top four query patterns in the BM query log

in the hash bucket [11]. However, these methods can lead to costly inter-partition traversals which greatly impact the performance of queries. Thus, some graph stores such as Neo4j [53], HypergraphDB [24] and DEX [31] avoid the partitioning of graph datasets.

The quality of partitions in real-world graphs may degrade due to continuous changes in query workloads and graph topologies. An adaptive graph partitioner can deal with this problem by exploiting streaming approaches [35] along with the incremental adaptation of the obtained partitions to the above changes. This usually incurs online monitoring of changes in the existing query workload and graph topology followed by the probable movement of some vertices between partitions. This adaptation may impose an overhead on the system and may reduce the efficiency and throughput of online querying. Therefore, adaptive strategies should be "lightweight" in terms of time and memory requirements. So far, there have been several research efforts on workload-driven partitioning strategies to achieve online low-latency graph querying [12, 5, 13–17, 21, 33, 37, 56]. However, there are still several significant shortcomings.

1. The existing strategies are mostly workload-agnostic (e.g., [10, 22]), as they presume the same probability of traversing edges or visiting vertices, which does not always hold with different query workloads. In other words, they do not consider frequent query patterns and locality of access to graph elements, which may degrade system performance. For example, Fig. 1 depicts a simple property graph and query workload. After running a vertex-centric partitioner, we get an optimal partitioning {P1, P2}. However, it is not optimal for the workload. Each query may require an expensive inter-partition traversal because of the cutting edge "?x authors ?y".
2. The existing strategies are mostly graph topology-agnostic, as they do not differentiate between high-degree and low-degree vertices which may result in load imbalance. In a hybrid-cut model, the vertex-centric partitioning is exploited for low-degree vertices while incident edges of high-degree vertices are partitioned via the edge-centric partitioning. There are only two graph analytics engines [9, 29] that exploit the hybrid-cut model. However, no workload-driven partitioning strategy has yet exploited this model.
3. Many existing workload-aware strategies are unable to adapt to dynamic workloads where the frequency of query patterns fluctuate. For example, WARP [21], Partout [17] and the Peng et al. method [37] exploit a given query log to extract frequent query patterns whereby the associated triple patterns are partitioned to increase access locality. However, such strategies suffer from two drawbacks: (1) Over time, the popularity of frequent query patterns may change. Accordingly, we have done some research on the British Museum (BM) query log [40]. It spans from April 12, 2014 to October 16, 2014 and records over 1.2 million query requests. The daily arrival rate of the top four frequent query patterns is demonstrated by Fig. 2. We have added the frequencies of the four patterns together as they follow a similar arrival trend. As we can see, despite being frequent (over 150 times a day) during a short period from August 4, 2014, to August 22, 2014, these patterns are infrequent most of the time. Therefore, a partitioning plan based on frequent query patterns with temporary spike results in inefficient queries most of the time. (2) Over time, the existing frequent query patterns may be outdated. In other words, queries in the future can rarely be reflected by query logs in the past.

To address the above issues, we present a workload-adaptive streaming graph partitioner, named WASP, that is also topology aware. Being workload-adaptive, WASP incrementally adjusts partitions (initially obtained by the random hashing of graph vertices) regarding the frequently

traversed active edges of the existing query workload and the frequently explored endpoint active vertices of the edges. In this sense, our partitioner distributes active vertices across partitions proportional to their frequency of visits. This leads to balance in the existing computational load, which in turn increases the throughput. On the other hand, it reduces the probability of the cut of active edges proportional to their frequency of traversals. As such, active vertices belonging to the same query are likely to be collocated into the same partition, which in turn decreases the query response time. WASP monitors active edges and vertices by tracking their activity weights where larger activity weights mean more activities. In order to adapt to the existing workload and diminish the effect of old active edges and vertices, WASP exploits a set of active edge logs, each of which is stored in a computing node. By inserting new active edges (of the existing workload) to an edge log, old active edges (of previous workloads) are gradually removed from the log. This removal turns the old active edges and their associated old active endpoint vertices to the passive ones, which are no longer considered in adjusting graph partitions. Being topology-aware, WASP utilizes a hybrid-cut model whereby the exploration of high-degree vertices is distributed across multiple nodes, which in turn increases the throughput. Our contributions are summarized as follows.

- We propose a dedicated cost model to manage vertex reassignment according to frequent query patterns and graph topology. The model decides which vertex should be moved and where to move in order to maximize the reassignment gain. Simultaneously, the load-balance on each computing node is preserved.
- We propose incremental lightweight metadata management, where data structures are mainly weight counters. As such, time-consuming computations such as calculating the degree of interest of active vertices to be hosted on different nodes are replaced by a continuous update of weights. We exploit Redis [7] for the quick in-memory storage and access of various weights in each node in the form of key-value pairs.
- We carry out an extensive evaluation using both real-world and synthetic graph datasets. Results show that our method is much faster than state-of-the-art graph stores, regarding dynamic workloads and increases the parallelism for accessing high-degree vertices of the graph.

The rest of this paper is organized as follows. Sections 2 and 3 introduce the background and related work for the proposed algorithms, respectively. Our WASP framework architecture is described in detail in Sect. 4, which mainly includes the details of our workload-adaptive and topology-aware graph partitioning strategy. Section 5 reports the evaluation results. Finally, we conclude the paper in Sect. 6.

2 Background

Balanced k-way graph partitioning divides the graph into k disjoint and balanced partitions and minimizes the cut size. This is a well-known NP-hard problem, where computational load balance, in order to maximize parallelism, and data access locality, in order to minimize inter-node communication, are two conflicting issues [18]. This results in lots of heuristic partitioning methods for graph datasets, which can be classified as two orthogonal categories: vertex-centric/edge-centric and offline/online.

Vertex-centric partitioners assign each source vertex along with its incident edges into the same partition, which in turn increases locality. However, the corresponding destination vertices may be assigned to different partitions which results in cutting their in-between edges or edge-cuts. These partitioners aim at performing a balanced distribution of vertices across nodes, as well as minimizing the number of edge-cuts. Although vertex-centric partitioners promote locality, they may severely impact the computational load balance for power-law graphs.[5] In other words, by grouping all edges of high-degree vertices together, a subset of nodes are overloaded. On the other hand, edge-centric methods tend to assign the edges incident to a particular vertex into different partitions. However, the endpoint vertices of an edge are replicated in the same node as the edge places. These partitioners aim at performing a balanced distribution of edges across nodes, as well as minimizing the number of replicas. Although edge-centric partitioners alleviate the computational load imbalance of high-degree vertices, they often incur higher communication and synchronization cost through the poor locality [19].

Offline or non-streaming partitioners, such as METIS [25], require accessing to the whole graph dataset in order to perform preprocessing prior to partitioning. However, they scale poorly against very large graphs due to their heavy usage of memory and high computational cost, which in turn impacts the performance of online (non-analytical) query processing. This partitioning approach has been later improved through parallelization techniques, such as ParMETIS [26]. These parallel strategies yet suffer from the need for a global view of the graph that reduces their scalability.

Since scaling offline approaches for large graphs are difficult, online or streaming approaches have been introduced which continually update the partitioning as new changes are streamed into the system. More precisely, they partition the incoming vertices (for vertex-centric partitioning) or edges (for edge-centric partitioning) one at a time based on the local knowledge of the input graph such as the current

[5] A small fraction of vertices have extremely high degrees in proportion to others [14].

properties of streamed elements and the information of previously partitioned ones. These streaming approaches are one-pass since after assigning a vertex or edge to a partition, no reassignment is performed. Due to the online nature of these approaches, lightweight heuristics, such as Fennel [49], are used to decide where to assign incoming elements. However, as graph elements are assigned once, new streamed elements of a graph may deteriorate its previous partitioning. Hence, there are several extensions [34, 50] of the streaming approach, where graphs are partitioned in several passes or iterations. But the quality of partitions is still dependent on the ordering of streamed elements as there may not be enough local knowledge of the input graph.

3 Related Work

In recent years, several online partitioning strategies have been proposed for supporting low-latency query execution of large-scale dynamic graphs. They aim at increasing the performance of either offline graph analytics as in [27, 30, 39, 42, 51, 52, 55, 12, 13], or online graph queries as in [10, 13–17, 21, 22, 32, 33, 37, 38, 56] whose workload-driven ones are more relevant to our work in this study. As we only review a subset of graph partitioning methods, the interested readers are referred to the recent surveys on graph partitioning [6, 20, 35].

Hermes [33] is a workload-driven partitioning method, where each vertex knows the number of its neighbors in each partition, the weight of each partition and the aggregate weight of partitions. Nodes are balanced based on the weight of their hosted vertices, where the weight of a vertex indicates the frequency of queries toward it. Vertex reassignment is triggered when the weights of vertices change. The gain of reassigning a vertex from its source to a target partition is how many more neighbors it has in the target than the source partition. Peng et al. [36, 37] propose a workload-driven partitioning method that mines frequent query patterns from a representative query workload. Then it puts matches of the same frequent pattern into the same fragment to improve the workload throughout.

WARP [21] is a workload-driven replication method, whereby RDF triples are initially partitioned using METIS, regarding their subjects. It then uses a representative query workload to replicate frequently accessed triples across the cluster using the n-hop guarantee method [23]. Given a user query, WARP determines its center vertex and radius. If the query is within the n-hop guarantee, WARP sends the query to all servers, which evaluate the query in parallel. Otherwise, the query is decomposed into subqueries for which a distributed query evaluation plan is created. Subqueries are evaluated in parallel by all servers, and the results are sent to the master which combines them. Partout [17] is also workload-driven by extracting frequent query patterns from a representative query workload and using them to partition the data into fragments.

Table 1 Workload-driven partitioning methods supporting online graph queries

Method	Cut model	Initial partitioning	Workload-adaptive	Topology-aware
Hermes [33]	Vertex-centric	Simple hashing	✤	✗
Peng et al. [36]	Vertex-centric	Existing workload	✤	✗
Partout [17]	Vertex-centric	Existing workload	✤	✗
WARP [33]	Vertex-centric	METIS	✗	✗
Loom [15]	Vertex-centric	Existing workload	✤	✗
Taper [16]	Vertex-centric	METIS	✤	✗
WASP	Hybrid-cut	Simple hashing	✓	✓

Full support (✓), Limited support (✤), No support (✗)

Loom [15] is a streaming partitioning strategy that assumes a given query workload of graph patterns and their relative frequencies. During the workload, it discovers common patterns of edge traversals. It then compares the sub-graph pattern matching queries against these common patterns and attempts to reduce inter-partition traversals of frequently traversed sub-graphs by allocating each match to a single partition. Taper [16] takes any given initial partitioning as a starting point, and iteratively enhances it by estimating traversal probabilities for a given path queries workload. These are then used to swap chosen vertices across partitions, and reducing the probability of inter-partition traversals.

Table 1 summarizes the state-of-the-art workload-driven partitioning strategies for supporting online graph queries. Loom, Partout, WARP and the Peng et al. method are based on knowing a priori query workload. Also, Taper assumes a given frequency of patterns in the existing path query workload. By exploiting this prior knowledge, the parts of the dataset that are targeted together by future queries can be highlighted. However, not only it might be practically difficult to have such knowledge in advance, but such strategies do not (properly) adapt to changes. This results in degrading the quality of partitions by evolving the workload while there is no repartitioning. Hermes takes into account a uniform frequency of edge traversals despite the non-uniform edge weights of real-world graphs. On the other hand, WARP extensively exploits the replication of graph vertices for improving their locality of access. However, maintaining replicas means additional metadata management, which in turn increases the system overhead.

Replicas also become useless by changing the workload, which in turn increases the storage overhead. On the contrary, WASP can take any given initial partitioning and assumes nothing about the workload upfront. Moreover, it avoids additional storage overheads by using no replication. WASP also exploits the hyper-cut model to alleviate the load imbalance of high-degree vertices and improve parallelism.

4 WASP Framework

In this section, we describe the design of WASP in more detail.

4.1 Data and Query Model

In this paper, data are represented by the property graph model, as it has gained wide acceptance and is used in many graph database systems such as Neo4j and Titan. It is defined as follows.

Definition 1 A property graph is a tuple $G = (V_G, E_G, L_G, \lambda_G, \sigma_G)$, where V_G is a finite set of vertices; $E_G \subseteq V_G \times V_G$ is a finite set of directed edges; L_G is a finite set of labels; $\lambda_G : V_G \cup E_G \rightarrow L_G$ is a total function that maps a (vertex or edge) identifier to a label; and $\sigma_G : (V_G \cup E_G) \times \mathrm{Pro}_G \rightarrow \mathrm{Val}_G$ is a partial function that maps a (vertex or edge) identifier and a property (or attribute) to a value, assuming that Pro_G and Val_G are a final set of properties and a set of values, respectively.

Intuitively, G is a directed, labeled and attributed multigraph, where each vertex represents an entity and has a label or type as well as a (possibly empty) set of properties associated with this entity, and each edge represents a binary relationship between entities and has a label and some properties as well.

Online graph queries can be classified into two major types, namely path queries and pattern matching queries [11]. Hence, we assume a query workload $W = \{<Q_1,f_1>, <Q_2,f_2>, \ldots, <Q_n,f_n>\}$ as a set of either path queries or pattern matching queries [2, 3] and their frequencies, processed by an exploration-based query processor [44, 57]. A pattern matching query Q follows the same structure as the property graph, but instead of allowing its vertices V_Q, edges E_Q, labels L_Q and property values Val_Q to contain only constants, it permits variables as well. A path query, which determines the existence of a path connecting two vertices of a property graph, can be considered as a subset of pattern matching queries.

4.2 Workload Characteristics

As query workloads are usually dynamic (via changing the frequency of associated queries), the quality of graph partitions may degrade over time. Hence, WASP encodes workload characteristics into vertex and edge weights, according to the following definitions.

Definition 2 Given a property graph G, for each directional edge $<u, v> \in E_G$, where u and v are in V_G and hosted on nodes N and M, respectively, there is a traversal weight $\omega(<u, v>)$ denoting the amount of data passed by traversing the edge.

In more detail, during the processing of a query Q, a traversal from the source vertex u to the target vertex v sends both Q and u from N to M where Q's processing continues, followed by receiving back the exploration result required by u. This weight starts from a default minimum value of 0 indicating that its corresponding passive edge has not yet been traversed. By traversing an edge during the existing query workload, the edge weight is gradually increased along with decreasing the probability of cutting the edge.

Definition 3 Given a property graph G, for each vertex $v \in V_G$, where v is hosted on node N, there is an activity weight $\omega(v)$ that is equal to the total weight of v's incident edges.

In more detail, as any traversal toward or from v requires the exploration of its neighborhood, $\omega(v)$ denotes the computing load imposed on N by accessing the corresponding local indices (see Sect. 4.4). Accordingly, by visiting an active vertex during the existing query workload, its activity weight becomes greater than 0. As an illustration, Fig. 3a shows the vertex/edge weights of a sample property graph. In this figure, the thicker the edges graphically indicate the ones with more frequent traversal.

4.3 Vertex Reassignment

The Fennel streaming heuristic [49] is used for the online one-pass partitioning of large-scale graphs, whereby a newly added vertex is assigned (only once) to an existing partition with the highest number of its neighbours; while at the same time, a large partition should be penalized to prevent it from becoming too large with respect to the number of its hosted vertices. This heuristic is presented in Eq. 1, where v refers to a vertex to be assigned, $N(v)$ refers to the set of v's neighbors, V_i indicates the set of vertices hosted on the ith node, n refers to the number of nodes, and α and γ are parameters.

$$\underset{1 \le i \le n}{\operatorname{argmax}} \left\{ |N(v) \cap V_i| - \alpha \frac{\gamma}{2} (|V_i|)^{\gamma-1} \right\} \quad (1)$$

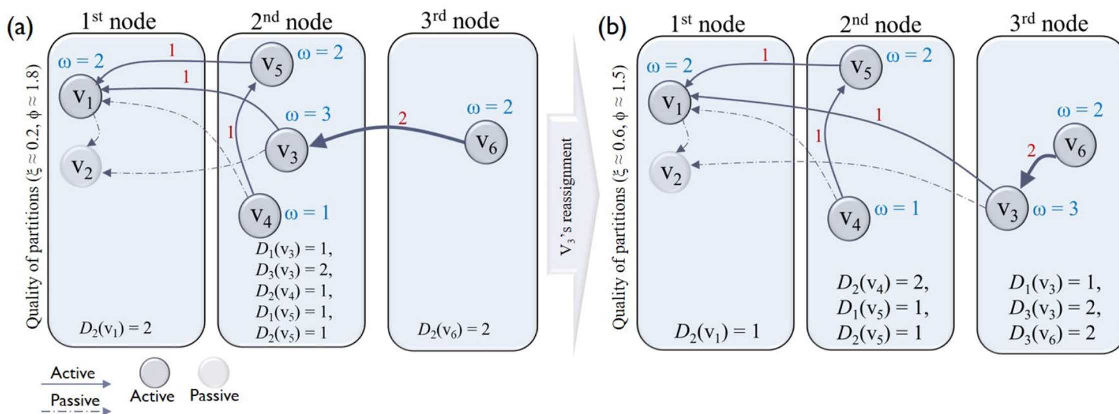


Fig. 3 **a** Existing partitions before reassigning vertex v_3; **b** the improved quality of partitions after reassigning v_3 from the 2nd to the 3rd node

Such a one-pass streaming heuristic is similar to the dynamic partitioning of graphs whose newly streamed vertices/edges are incrementally added to the existing partitions [45, 49]. However, one-pass partitioning falls short in four areas to be workload-adaptive: (1) an assigned vertex is never reassigned, (2) the removal of vertices/edges is not considered, (3) a uniform frequency of edge traversals is considered, and (4) partitions may not be balanced based on the aggregate activity weight of their hosted vertices. These drawbacks motivate us to use a workload-adaptive selective reassignment that continuously revisits active vertices and reassigns them when appropriate. This necessitates maintaining some workload-based metadata in the main memory of computing nodes. This amount of information is not comparable to the huge amount of the given graph dataset. In more detail, according to the existing workload, assume $v \in S_i$, where S_i is the set of active vertices hosted on the ith node. The following metadata need to be maintained in the main memory of the ith node:

- v's degree of interest to be hosted on each node, where each degree is initialized from a default minimum value zero. In more detail, v's degree of interest toward the jth node ($j \in [1..n]$) is called $D_j(v)$, which indicates the total weight of v's incident edges to/from vertices hosted on the jth node. During a query traversal, $D_j(v)$ is incremented by sending a request from v to a vertex on the jth node or vice versa. Accordingly, $\omega(v)$ can be simply calculated by summing up v's degrees of interest toward all nodes. Note that degrees with a default value of zero are not stored in the memory. They are not also shown in Fig. 3.
- The activity weight of the ith node, that is called $\omega(S_i)$. It indicates the aggregate activity weight of all active vertices hosted on the node.
- An edge log hosted on the ith node, that is called ℓ_i. The log stores all active edges and corresponding weights, that are incident to the active vertices hosted on the ith node. In more detail, during the existing workload, the most recently traversed edge incident to an active vertex hosted on the ith node is inserted on top of ℓ_i. As a result, an edge e that is not traversed any more (i.e., belonging to the previous workload) is gradually shifted to the bottom of the log and finally moved out of it. This turns e into a passive edge and its weight is set to 0, which in turn changes the degrees of locality of e's endpoint vertices. More precisely, assuming e's weight is w and its endpoint vertices are u hosted on the jth node, and v hosted on the ith node, then w is subtracted from both $D_i(u)$ and $D_j(v)$. w is also subtracted from $\omega(S_i)$ and $\omega(S_j)$.

  Each edge log has a configurable size Δ, which is uniformly set for all nodes as their logs contain almost the same number of active edges. This is because of alleviating the load skew at query time through the edge-centric partitioning of high-degree vertices (see Sect. 4.3.3). When Δ is too small, each log stores a subset of active edges traversed in the existing workload. This means moving out some frequently traversed edges belonging to the workload and mistakenly making them passive. On the other hand, when Δ is too large, each log stores active edges that were traversed during the previous workloads, not the existing one. In both cases, the quality of partitioning may be impacted due to imprecise vertex reassignments. The choice and impact of Δ will be discussed in Sect. 5.4.

As aforementioned, our framework uses the simple hash partitioning scheme for the initial partitioning of the vertices across the nodes for two reasons. First, there is no complicated logic involved in assigning new vertices to partitions.

Second, for a given vertex, we can simply look up its original hosting node. Hence, the initial node hosting a newly arrived vertex can be simply found through the hashing of the vertex *ID*. However, reassigning the vertex necessitates using a lookup table to find its new hosting node. This table can be implemented in a distributed manner through a set of lookup variables. More precisely, each reassigned vertex v has a lookup variable stored as metadata in the v's initial node. Note that, as long as v is hosted on its initial node, there is no need for this lookup variable.

Our selective reassignment heuristic is presented in Eq. 2. By comparing to Eq. 1, we set $\gamma = 2$ and $\alpha = \frac{1}{|S_i| \times n}$.

$$\begin{aligned} &\underset{1\le i\le n}{\operatorname{argmax}} \left\{ D_i(v) - \frac{\omega(S_i)}{|S_i| \times n} \right\} \\ \omega(S_i) &= \sum_{1\le j\le |S_i|} \omega(v_j), \text{ where } v_j \in S_i \\ \omega(v_j) &= \sum_{1\le i\le n} D_i(v_j) \end{aligned} \tag{2}$$

This reassignment takes a vertex v as input, computes a score for every node in the cluster, and a node with the highest score is determined as the potential target node to host v. To ensure a balanced partitioning with respect to the activity weight of nodes, there is a penalty function $\frac{\omega(S_i)}{|S_i| \times n}$, where by increasing the activity weight of each node, its score decreases.

In order to prevent the source node (assume it is the sth node) to be underloaded, the condition: $\{\omega(S_s) - \omega(v)\} \geq (2 - \Phi) \times \Omega$ is checked before performing the selective reassignment. Here, Ω indicates the average aggregate activity weight of all nodes. In addition, parameter $\Phi \in [1, 2]$, which is called maximum load imbalance, indicates how imbalanced a partition can be. For example, $\Phi = 1$ indicates that all nodes are required to have the same aggregate of activity weights. If the source node is underloaded, the vertex v is not considered to be moved. On the other hand, the condition: $\{\omega(S_t) + \omega(v)\} \leq \Phi \times \Omega$ is checked on the potential target node (assume it is the tth node) in order to prevent overloading it. If the node becomes overloaded, the condition is checked on the next highest score node; otherwise, the tth node is selected as the target node.

Assume the quality of partitions is determined by a pair of quality factors: (1) the probability of intra-partition traversals (ξ) and (2) the load imbalance factor (ϕ) indicating how imbalanced existing partitions are. These factors are defined as follows:

$$\begin{aligned} \xi &= \frac{\text{the aggregate weight of edges that are not cut}}{\text{the weight of all edges}} \\ \phi &= \frac{\max_{1\le i\le n} \omega(S_i)}{\Omega} \end{aligned} \tag{3}$$

The selective reassignment improves ξ as the higher the weight of an edge, the lower the probability to cut it. In other words, by increasing the weight of an edge, the interest of its endpoints to be collocated on the same node is increased. Higher the value of ξ means less the probability of inter-partition traversals during the existing query workload.

Figure 3 illustrates the selective reassignment. Suppose there are three nodes, $\Phi = 1.6$, and vertex v_3 is selected for reassignment. As Fig. 3a shows, $\omega(S_1) = 2$, $\omega(S_2) = 6$ and $\omega(S_3) = 2$ and $\Omega \approx 3.33$. With respect to Eq. 2, we have the following calculations: $\text{Score}_1 \approx 0.33$, $\text{Score}_2 \approx -0.66$ and $\text{Score}_3 \approx 1.33$. As Fig. 3b shows, after moving v_3, the 2nd node (as the source node) is not underloaded ($\omega(S_2) \geq 1.32$), and no one of the potential target nodes is overloaded (each one has an aggregate activity weight less than or equal to 5.33). As a result, v_3 is moved to the third node having the highest score. The quality of partitions after moving v_3 is improved as ξ has increased from 0.2 to 0.6, and ϕ has decreased from 1.8 to 1.5 in Fig. 3a, b, respectively.

4.3.1 Vertex Reassignment Data Maintenance

Reassigning a vertex v incurs moving its topological data from the ith node as the source to the jth node as the target. Such data include v's relationships to other vertices along with v's properties. It also incurs maintaining the associated metadata as the following: (1) all active edges (and their weights) that are incident to v are removed from ℓ_i and inserted to ℓ_j, (2) v's degrees of interest toward all nodes are moved to the target node, (3) for each active edge e that is incident to u and v, u's degrees of interest toward the source and target nodes are changed. More precisely, $\omega(e)$ is subtracted from $D_i(u)$ and then added to $D_j(u)$, (4) the aggregate activity weights of the source and target nodes are changed. More precisely, w is subtracted from $\omega(S_i)$ and added to $\omega(S_j)$, and (5) v's lookup variable in its initial node is updated to refer to the target node.

During the reassignment, the source node's query processor does not answer any request on v, in order to prevent their access to inconsistent data. As such, a request queue is maintained in the query processor, whereby all requests on v are queued. By completing the reassignment, the v's lookup variable refers to the target node. Therefore, requests on v are released from the queue and redirected to the target node.

4.3.2 Vertex Reassignment Timing

The timing of reassigning vertices is critical to make a balance between the quality of partitions and the above overhead of data/metadata maintenance. Recall that by changing the activity weight of a vertex, its degree of interest toward different nodes may change. This, in turn, triggers checking the possibility of reassigning the vertex. On the other hand,

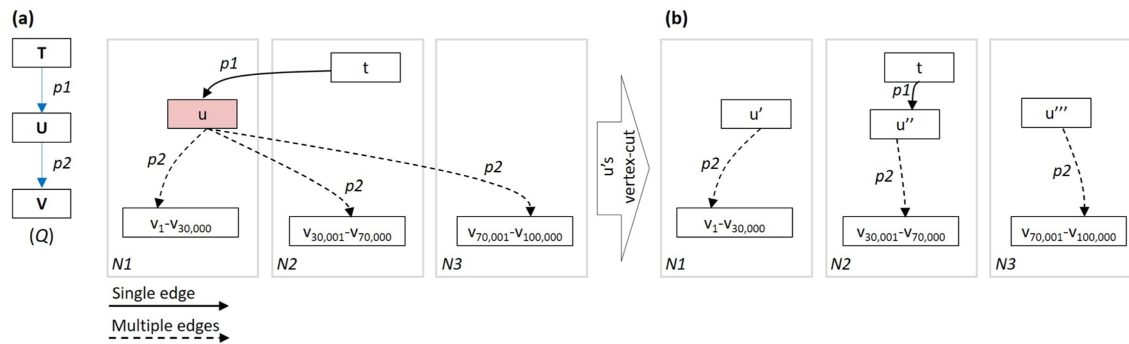


Fig. 4 **a** A high-degree active vertex u regarding Q as a query pattern belonging to the existing workload; **b** splitting up the edges of u across the nodes

by increasing the activity weight of a vertex, the influence of a new exploration in its degree of interest toward different nodes is negligible. Therefore, after a reassignment, we will check the possibility of another reassignment only after a similar amount of new explorations. In more detail, assuming a reassignment threshold k, vertex reassignments are triggered after $\{k, 2 \times k, 4 \times k, \ldots, 2^i \times k, \ldots\}$ explorations. This significantly reduces the number of reassignments for a vertex. For example, if k is equal to 10, for a vertex whose activity weight is 10,240, the maximum number of reassignments is only 10. Currently, we use a hardwired reassignment threshold. The choice and impact of the reassignment threshold will be discussed in Sect. 5.3.

4.3.3 High-Degree Vertices

High-degree active vertices may significantly reduce the efficiency and throughput of online querying. More precisely, their large neighbourhood may incur a significant processing overhead on their hosting nodes, as well as a large amount of network traffic on their incident edges. As an illustration, Fig. 4a depicts a high-degree active vertex u, hosted on node *N1*, whose neighbors are scattered across *N1* to *N3*. Also, assume there is a graph traversal through the pattern matching query Q, and vertices t, u and v_1 to $v_{100,000}$ in the input graph are instances of vertices T, U and V in the query. Accordingly, a traversal from the source vertex u to the target vertices v_1 to $v_{100,000}$ needs gathering and processing a huge amount of results that are sent back to *N1*. In addition, a traversal from the source vertex t, hosted on node *N2*, to the target vertex u incurs sending back the huge amount of gathered results from *N1* to *N2*.

WASP alleviates these issues by specifying and splitting high-degree vertices. Accordingly, for each vertex u when u's degree (of incoming and outgoing edges) exceeds a configurable splitting threshold, it is considered as a high-degree vertex which in turn results in splitting up u's edges, whereby vertex u is collocated with its neighbors. In more detail, an outgoing edge $u \rightarrow v$ is collocated with its target vertex v, and an incoming edge $u \leftarrow v$ is collocated with its source vertex v. Accordingly, as u's neighbors are randomly distributed through hashing, its edges will be evenly distributed. This uniformly divides the query processing overhead (which was already on u's hosting node) between all nodes that host its splits. As Fig. 4b shows, vertex u is split into three vertices u', u'' and u‴ hosted on *N1*, *N2* and *N3*, respectively. As such, traversing from t as the source vertex to u's splits as the target neighbors results in dividing the traffic load (which was already between two nodes *N1* and *N2*) between all nodes. The choice and impact of the splitting threshold will be discussed in Sect. 5.5.

4.4 Verifying WASP on an RDF Store

The popularity of property graphs is due to their flexibility to express other structures. Accordingly, by not using attributes in property graphs, Resource Description Framework (RDF)[6] or knowledge graphs are generated. Intuitively, an RDF graph consists of triples of the form ⟨*subject, predicate, object*⟩ which can be interpreted as either two entities (*subject* and *object*) connected via a labeled relationship (*predicate*) or an entity (*subject*) associated via an attribute name (*predicate*) to its corresponding value (*object*). In a graph-based RDF store the dataset is stored as a graph, where RDF triples are modeled as vertices and edges [4]. An RDF graph is explored by walking the graph in specific orders according to the edges of a given SPARQL graph pattern matching query[7] [44, 57].

In order to assess the impact of WASP on a graph store and to show how easily the approach can be plugged on top

[6] http://www.w3.org/TR/rdf-primer/.

[7] http://www.w3.org/TR/rdf-sparql-query/.

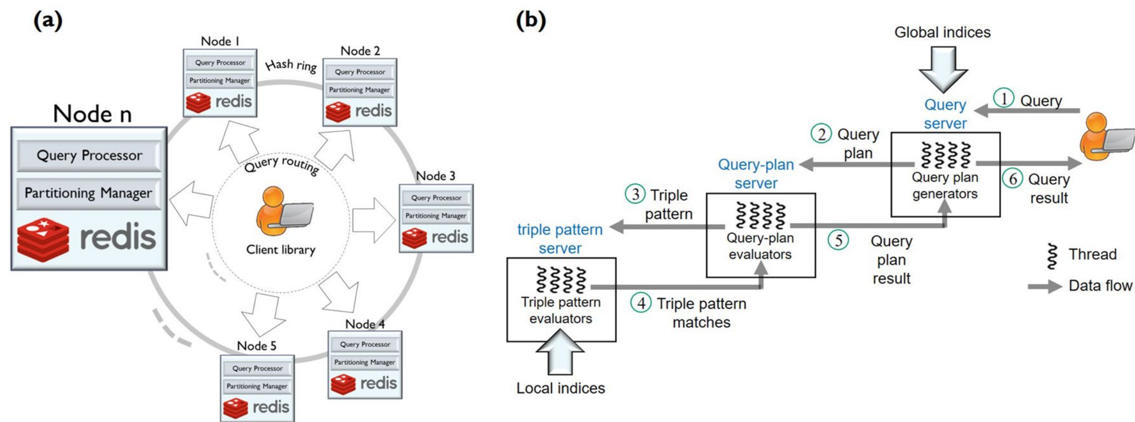


Fig. 5 **a** System architecture; **b** graph-based SPARQL query processing steps

of the store, it is exploited on a graph-based RDF store.[8] WASP is deployed on a cluster whose nodes are connected in a peer-to-peer fashion similar to the one presented in Fig. 5a. The key components of the system are as follows.

- A client library that contains a query routing module in order to balance the load among computing nodes. As such, a client can connect to any node and perform a query.
- A distributed memory storage that is made up of a batch of independent single-node and in-memory Redis servers[9] as the back-end stores. Redis provides a variety of data structures such as hashes, lists, sets and ordered sets, as well as various operations for handling them. Accordingly, each node stores the associated data/metadata in the following local/global indices for an input graph G (see Definition 1).

  - Global index of vertices is stored as a Distributed Hash Table (DHT) of Key-Value (KV) pairs, where for each $v \in V_G$, there is a KV pair consisting of the corresponding vertex identifier or *vID* as the key and the corresponding metadata as the value. Note that the name of v is initially hashed to a *vID* which, in turn, is hashed to a computing node. In addition, the metadata of v include its name and hosting node. Sending and storing *vIDs* instead of long names may result in saving network bandwidth and memory consumption. In addition, storing hosting nodes allows for changing the current hosting node of a vertex due to G's repartitioning.
  - Global index of predicates is stored as a DHT of KV pairs, where for each $p \in L_G$, there is a KV pair consisting of the corresponding predicate identifier or *pID* as the key and the corresponding metadata as the value. Note that the name of p is initially hashed to a *pID*. In addition, the metadata of p include its name and a pair of lists <*sbjNodeList*, *objNodeList*>, where *sbjNodeList* (or *objNodeList*) includes node(s) hosting some *subjects* (or *objects*) incident to an edge labelled with p. This index is globally used to determine those nodes where a triple pattern with a bound *predicate* should be submitted for evaluation.
  - Local index of vertices of each node is stored as a hash table of KV pairs, where each pair belongs to a vertex $v \in V_G$ that is hosted on the node. This pair consists of a combination of the corresponding *vID* and an OUT (=1) or IN (= 0) direction as the composite key, and a list of all unique *pIDs* of outgoing or incoming edges incident to v. This index is locally used for evaluating those triple patterns whose only *subjects* (or *objects*) are bound.
  - Local index of predicates of each node is stored as a hash table of KV pairs, where each pair belongs to a predicate $p \in L_G$. This pair consists of a combination of the corresponding *pID* and an OUT or IN direction as the composite key, and a list of all vertices on the node that are the source or the target of an edge labeled with p, respectively. This index is locally used for evaluating those triple patterns whose only *predicates* are bound.
  - Local index of vertices-predicates of each node is stored as a hash table of KV pairs, where each pair belongs to a vertex $v \in V_G$ that is hosted on the node. This pair consists of a combination of the corresponding *vID*, a *pID* where $p \in L_G$ and an OUT or

[8] The source code is publicly available for download at https://github.com/alidavoudian/WASP-Graph-Partitioner/.

[9] http://redis.io/.

IN direction as the composite key, and the list of corresponding neighbor vertices as the value. This is used for evaluating those triple patterns whose only *subjects* (or *objects*) and *predicates* are bound.

- Local index of activity weights of each node is stored as a hash table of KV pairs, where each pair belongs to an active vertex $v \in V_G$ that is hosted on the node. This pair consists of a combination of the corresponding vertex identifier *vID* as the key and the corresponding activity weight ($\omega(v)$) as the value.
- Local index of degrees of interest of each node is stored as a hash table of KV pairs, where each pair belongs to an active vertex $v \in V_G$ that is hosted on the node. This pair consists of a combination of the corresponding *vID* and a node identifier *nID* as the composite key, and v's degree of interest to be hosted on that node as the value.

- A graph-based SPARQL processor that consists of three running processes: (1) *query plan generator* that heuristically calculates a query plan for each received query, where a query plan is an ordered sequence of triple patterns. It then uses the aforementioned global indices to determine proper nodes where the plan is sent. Finally, after receiving the results of all sent plans, they are combined and sent back to the corresponding user; (2) *query plan evaluator* that receives a query plan and sequentially sends the corresponding triple patterns to proper nodes determined via the aforementioned global indices. Finally, after receiving the matches of all sent triple patterns, they are merged and sent back to the corresponding query plan generator; (3) *triple pattern evaluator* that uses the aforementioned local indices, determines the matches of a received triple pattern, and finally sends back the matches to the query plan evaluator. Figure 5b depicts the query processing steps.
- WASP framework that is made up of several independent partitioning managers which are integrated with their peer query processors. Partitioners are in charge of watching the existing query workload during the graph exploration along with making autonomous decisions for relocating graph vertices hosted on their corresponding nodes.

Note that since Redis servers are single threaded, by having each server manage partitions of both graph dataset and metadata, the concurrency between the processing of data and maintenance of metadata is decreased which in turn impacts the efficiency of query processing. As a result, we exploited two Redis servers on each node for the separate management of partitioned data and metadata.

4.5 Memory and Time Complexities

The amount of metadata used by WASP has a small size compared to the huge amount of a given graph dataset. In more detail, by storing the weights of at most Δ active edges on a node, the metadata of at most $2 \times \Delta$ active vertices are stored in the node as each edge represents two endpoint vertices. On the other hand, for each active vertex hosted on the node, there are at most n degrees of interest toward all nodes. Therefore, there are at most $n \times 2 \times \Delta$ degrees of interest on each node. In addition, for each active vertex u hosted on a node, there is an activity weight $\omega(u)$. Hence, there are at most $2 \times \Delta$ activity weights on each node. Finally, each node stores the total activity weight of its hosted active vertices requiring one long integer. As a result, the maximum amount of metadata used by WASP is $2\Delta n^2 + (3\Delta + 1)n$, where n is the number of computing nodes, and Δ is the size of the edge log. An implication of the above complexity is that the size of metadata scales with Δ. However, according to our experiment in Sect. 5.4, Δ is far less than the number of edges of a given graph dataset. On the other hand, the performance of a partitioner is mainly affected by the amount of communication required by the partitioning algorithm. Accordingly, the time complexity of our partitioner relies on the number of times the selective reassignment and its examination are called. This requires a good balance between the edge-cut ratio and the number of vertex reassignments that is experimented in Sect. 5.3.

5 Experimental Evaluation

In this section, we evaluate WASP in extensive experiments to thoroughly test its adaptivity and performance, regarding different static and dynamic query workloads.

Hardware/software setup We have implemented WASP in C/C++. In more detail, MPICH-3.1.4 library and ZeroMQ[10] sockets are used for communication across nodes. In addition, Hiredis[11] that is an official C client library is used for connecting to Redis KV stores. WASP is deployed on a cluster with 10 homogeneous nodes connected in a peer-to-peer fashion, where each node has 48GB of RAM, 16 quad-core CPUs of 2.4GHz, 160GB SATA HDD, and runs Debian Linux 6.0.6.

Datasets and query workloads We have conducted our experiments using three synthetic and real datasets (see Table 2) as well as six query workloads over the datasets. WatDiv [1] and LUBM [47] are two synthetic property graph benchmarks whereby we create two datasets, each

[10] http://zeromq.org.

[11] https://github.com/redis/hiredis.

Table 2 Dataset statistics in millions (M)

Dataset	#Vertices	#Edges	Avg./Max. degree
`WatDiv-1B` [1]	97M	1092M	24/190K
`LUBM-10240` [47]	332M	1367M	17/4.3K
`DBpedia` [28]	14M	1096M	35/630K

of which contains over 1 billion edges. We use their query template generators to generate 20 and 14 basic templates or patterns, respectively. We then instantiate 1K queries for each template. Accordingly, we create two workloads for WatDiv, namely *WatDiv-SW* and *WatDiv-DW*, each of which contains a total of 20K queries. The former simulates a static workload by shuffling the queries of all patterns with no fluctuating frequency. The latter simulates a dynamic workload by the consecutive execution of queries of the same pattern. Similarly, we create *LUBM-SW* and *LUBM-DW* workloads for LUBM, each of which contains a total of 14K queries.

As both WatDiv and LUBM are synthetic, we also use DBpedia [28] as a real property graph dataset extracted from Wikipedia by crowdsourcing. We extract 44 query templates from the DBpedia query log [40], which spans from April 12, 2014, to October 16, 2014, recording over 1.7 million query requests.[12] We used this log as it is also used by the other state-of-the-art graph partitioners, which in turn results in a fair comparison. We remove the queries with parse error or runtime error, and there remain 1.2 million queries in the end. As before, we create two workloads, namely *DBpedia-SW* and *DBpedia-DW*, each of which contains a total of 44K queries. Note that most of the queries in the DBpedia query log consist of only one or a few triple patterns, so the templates of DBpedia are much simpler than the other datasets.

Competitors Regarding Table 1, we compare WASP against Hermes [33] that is the only strategy with no prior knowledge of query workloads. We also compare against the Peng et al. method [36] as a representative of strategies that are based on knowing a priori query workload. It has already shown to have a faster query response time against WARP [21] and Partout [17].

5.1 Partition Quality

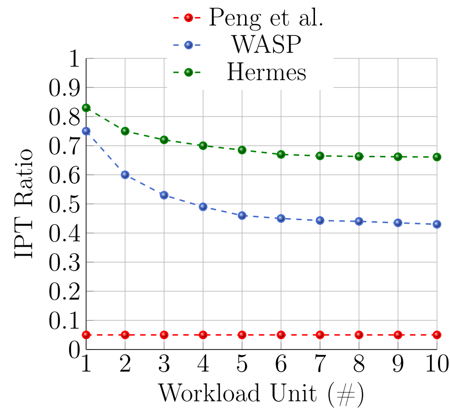
We first study the ratio of inter-partition traversals (IPT) achieved by the three partitioners, regarding the six static and dynamic workloads. In Fig. 6a, the *X*-axis shows the *WatDiv-SW* query stream in 10 units, each of which contains 2K queries. The Y-axis indicates the IPT ratio achieved by executing the queries of each unit. The diagram shows a gradual decrease of IPT ratio in both Hermes and WASP as they gradually improve data access locality via their vertex reassignment strategies. However, since Hermes does not consider the weight of active edges, the corresponding decreasing rate is lower than WASP. On the other hand, despite WASP and Hermes that initially partition the graph dataset via the simple hashing strategy, the Peng et al. method [36] partitions the whole graph dataset assuming a priori knowledge of the *WatDiv-SW* workload. Therefore, the diagram shows an almost steady IPT ratio less than 0.1, as the Peng et al. method has already assigned the matches of each frequent pattern to the same partition.

On the contrary, regarding the *WatDiv-DW* query stream as shown in Fig. 6b, the IPT ratio in the Peng et al. method is very high (more than 0.8). In more detail, it partitions the graph dataset based on the frequent query patterns in the first workload unit, assuming a priori knowledge of the unit. However, the existing frequent query patterns change in subsequent units which results in a severe increase in IPT ratio as the Peng et al. method is not workload adaptive. On the other hand, WASP and Hermes adapt themselves to the *WatDiv-DW* query stream. During the workload, the IPT ratio in WASP fluctuates approximately between 0.3 to 0.4. However, Hermes acts in an upper bound of ratios between 0.6 to 0.7, as it takes into account a uniform frequency of edge traversals despite the non-uniform ones in real-world scenarios. Similar reasoning can be used to justify the results obtained for LUBM and DBpedia datasets. As Fig. 6c shows, after executing the *LUBM-SW* query stream, WASP achieves a better IPT ratio (almost 0.27) compared to the one in *WatDiv-SW* (almost 0.43 in Fig. 6a). This is due to the existence of more complex queries with long paths in LUBM than WatDiv. In more detail, more complex queries cause more edge traversals, which results in the identification of hot edges and collocating their endpoint vertices by WASP. On the other hand, as Fig. 6e shows, the behavior of WASP and Hermes are closer to each other compared to Fig. 6a, c. This is because of simple short path queries in DBpedia.

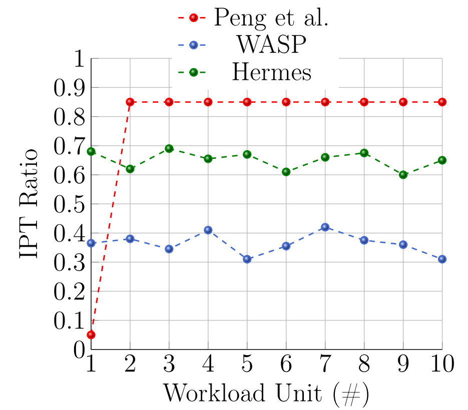
Note that the improvement of the IPT ratio does not necessarily result in the improvement of query performance unless there is a balanced load distribution [35]. Therefore, we study the "computational" load balance achieved by the three partitioners. In this respect, although Hermes and the Peng et al. method exploit the existing workload to achieve a balanced load distribution, they perform poorly in WatDiv and DBpedia as all edges of very high-degree vertices are grouped together. This causes a subset of machines to be computationally overloaded by active vertices with a very high degree. On the other hand, WASP splits such vertices (see Sect. 4.3.3). Accordingly, Fig. 7 shows the load imbalance factor (ϕ) achieved by the partitioning strategies on the three static workloads. As shown, WASP achieves a better load imbalance than the others in all the workloads.

[12] To extract query templates, we randomly select one-day queries from the log and transform them into a parameterized form [46].

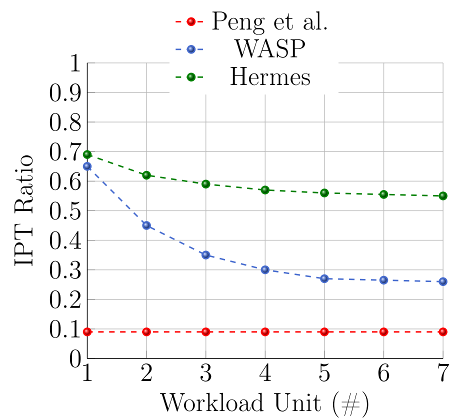
Fig. 6 IPT ratio regarding the six workloads



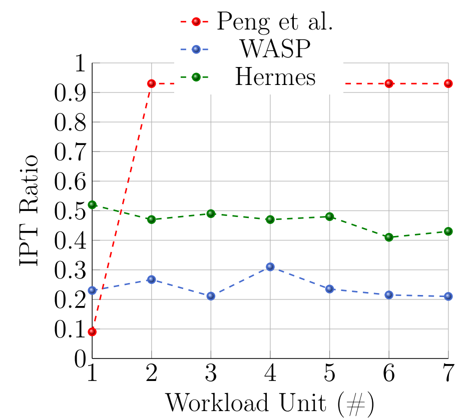
(a) WatDiv-SW (20K)



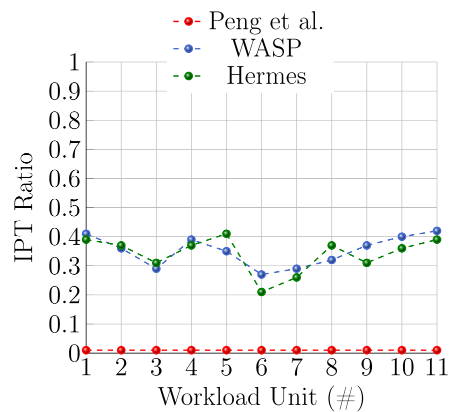
(b) WatDiv-DW (20K)



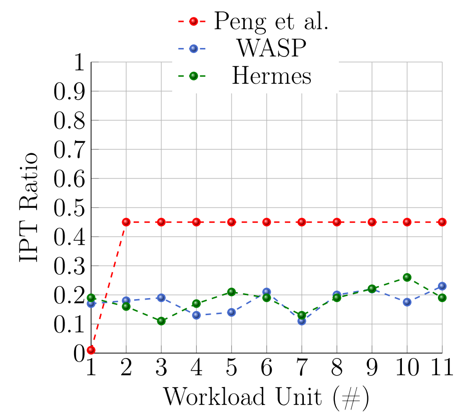
(c) LUBM-SW (14K)



(d) LUBM-DW (14K)



(e) DBpedia-SW (44K)



(f) DBpedia-DW (44K)

5.2 Query Performance

Table 3 shows the execution time of the six aforementioned static and dynamic query workloads, regarding the three partitioning strategies. Unlike the Peng et al. method which initially mines frequent query patterns from the static workloads, WASP and Hermes use initial hash partitioning that mounts communication on complex queries. This results in better performance of the Peng et al. method compared to Hermes and WASP, regarding WatDiv and LUBM static workloads. However, WASP achieves a lower execution time than Peng et al. regarding DBpedia-SW. This is because of

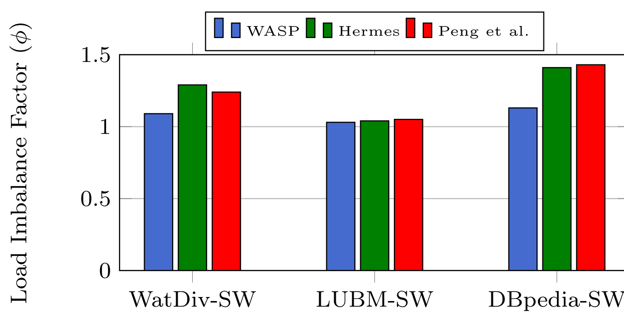


Fig. 7 Load imbalance factor regarding the three static workloads

either the low impact of IPT ratio on query performance due to the simple short path queries in DBpedia or the high impact of load imbalance factor on query performance regardless of the complexity of queries. On the other hand, unlike WASP and Hermes, the Peng et al. method does not adapt to changes in frequent query patterns. This incurs high communication costs for the dynamic workloads and results in achieving higher workload execution times in the Peng et al. method compared to the other strategies.

5.3 Reassignment Threshold Sensitivity Analysis

The reassignment threshold controls the triggering of the vertex reassignment process. Consequently, it influences the edge-cut ratio and the number of vertex reassignments in the system. In this experiment, we conduct an empirical sensitivity analysis to select the reassignment threshold value regarding the aforementioned static workloads. We execute each workload regarding a wide range of possible reassignment threshold values from 1 to 30 as load execution times get almost steady regarding the threshold value 30. The edge-cut ratio, the number of vertex reassignments and the resulting workload execution times are shown in Fig. 8a–c, respectively. As Fig. 8a, b show, there is a contrast between edge-cut ratio and the number of vertex reassignments. This necessitates to make a balance between these factors in order to decrease load execution times.We observe that WatDiv is very sensitive to slight changes in the reassignment threshold because of the complexity of its queries. As the reassignment threshold increases, the reassignment of active vertices is delayed causing more queries to be executed with communication. On the other hand, by decreasing the reassignment threshold, more vertices are reassigned causing more overhead of data/metadata maintenance. As it can be seen, DBpedia is not as sensitive to this range of reassignment thresholds because most of its queries are simple short path queries that are processed with low or no communication. As the both workloads show, either increasing or decreasing the reassignment threshold may impact the efficiency and throughput of query processing. In all our experiments, we use a reassignment threshold of 10; this results in a good balance between the edge-cut ratio and the number of vertex reassignments. We plan to study the autotuning of this parameter in the future.

5.4 Edge Log Size Sensitivity Analysis

Edge log size or Δ affects the lifespan of active edges and the quality of partitions. This incurs choosing a proper log size value. By increasing Δ, both recent and old active edges are considered in partitioning. This decreases the throughput due to an improper load balance. More precisely, the weights of endpoint vertices associated with old active edges are still considered in calculating their hosting nodes' weights. On the other hand, by decreasing Δ, only a subset of recent active edges are taken into account in partitioning. This decreases the efficiency of queries via increasing edge-cut ratio as the endpoint vertices of some recent active edges are not collocated. Hence, decreasing or increasing Δ can impact the workload execution time. In this experiment, we conduct an empirical sensitivity analysis to select a proper edge log size based on the workload execution time, regarding *WatDiv-SW* and *DBpedia-SW* workloads. As typical query workloads access only a small fraction of the whole graph, we execute each workload regarding a range of log sizes from 100K to 1000K as workload execution times get almost steady regarding the lower and upper bounds. Workload execution times are shown in Fig. 8d. In all subsequent experiments, we use an edge log size of 700K; this results in a high throughput and query efficiency. We plan to study the autotuning of reassignment threshold and edge log size in the future.

5.5 Splitting Threshold Sensitivity Analysis

The splitting threshold controls the cutting up of high-degree vertices. However, despite alleviating the computational load imbalance, splitting the edges leads to the replication of high-degree vertices. This in turn introduces extra network

Table 3 Workload runtime (s)

Method	WatDiv-SW	LUBM-SW	DBpedia-SW	WatDiv-DW	LUBM-DW	DBpedia-DW
Peng et al.	924	830	435	1332	1396	515
Hermes	1210	1153	481	1244	1136	487
WASP	987	1089	377	916	843	347

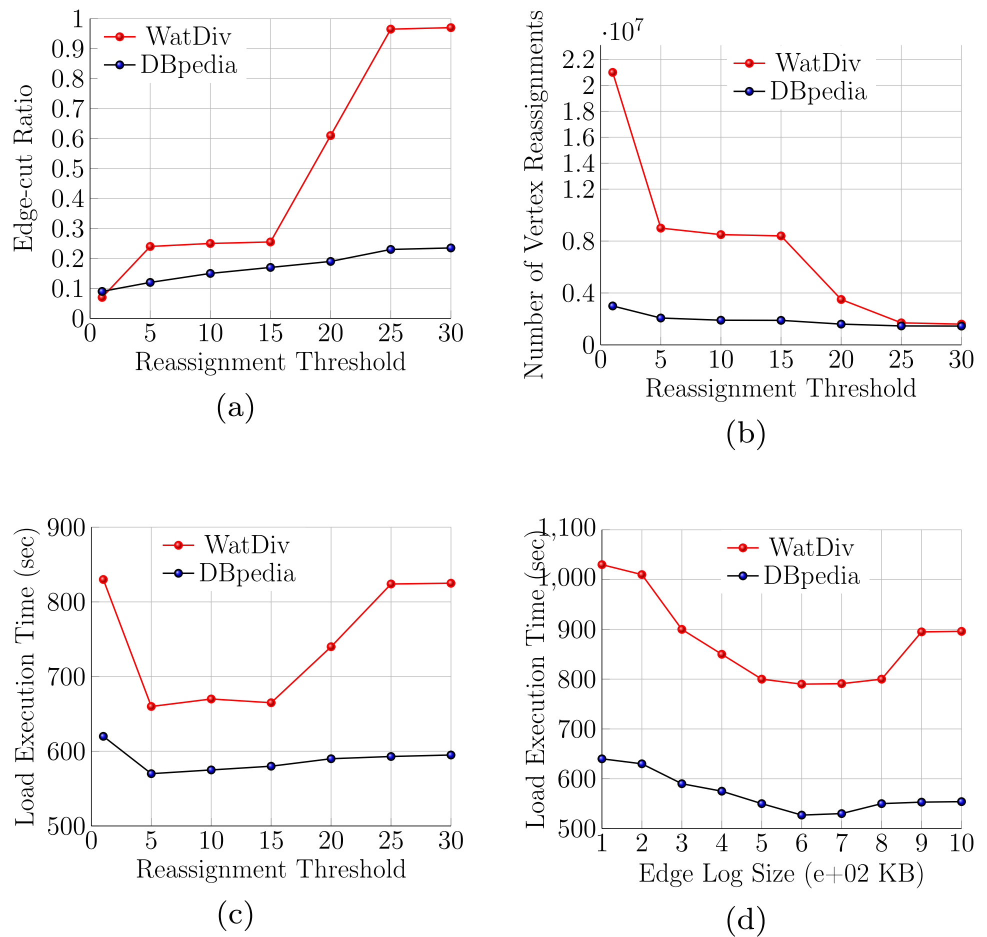


Fig. 8 Sensitivity analysis of reassignment threshold and edge log size

overhead to retrieve data from remote nodes. Therefore, it is important to find the best splitting threshold to make a trade-off between the computational load balance and the average number of replicas. In this experiment, we conduct an empirical sensitivity analysis to select the splitting threshold value regarding the WatDiv and DBpedia datasets in a cluster of 10 nodes.

The result (Fig. 9) shows that by increasing the splitting threshold, fewer vertices get replicated causing a sharp decrease in the average number of replicas. However, in both datasets, the average number of replicas is almost insensitive to the splitting thresholds larger than 100 that is selected as the splitting threshold in our experiments.

5.6 Workload Evolution

In this experiment, we use the aforementioned dynamic workloads to investigate the impact of changing frequent query patterns on our partitioning strategy. Regarding the *WatDiv-DW* workload, Fig. 10a shows the cumulative time as the execution progresses using our method with and without the adaptivity feature. After every sequence of 1K query executions, the pattern of queries changes. Without adaptivity, the cumulative time increases sharply. On the other hand, our method adapts to the dynamic workload with little overhead causing the cumulative time to drop significantly by almost 6 times. Once our method starts adapting, most of future queries are solved with less communication. The same behavior is observed for the *DBpedia-DW* workload (see Fig. 10b).

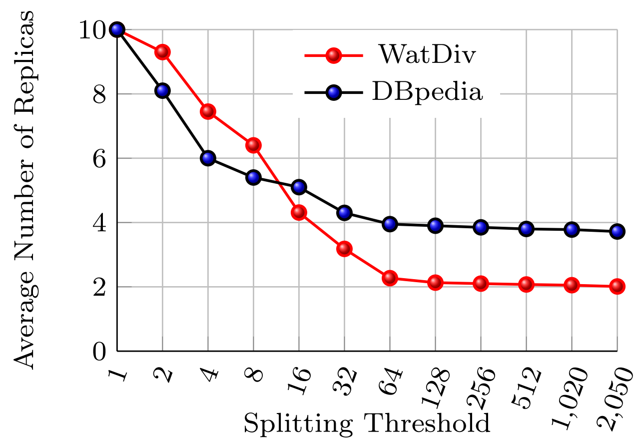


Fig. 9 Average number of replicas under different splitting thresholds

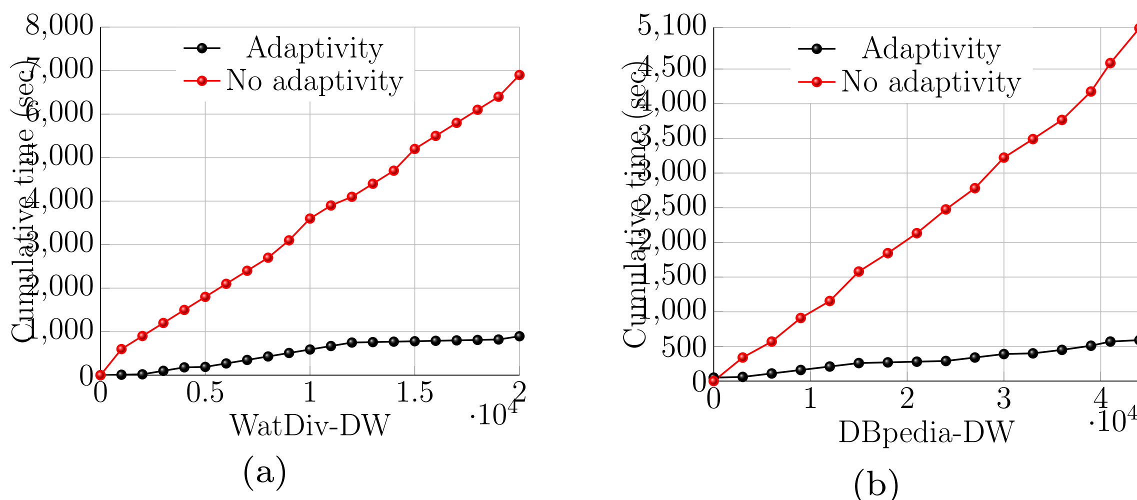


Fig. 10 WASP adaptivity to dynamic workloads

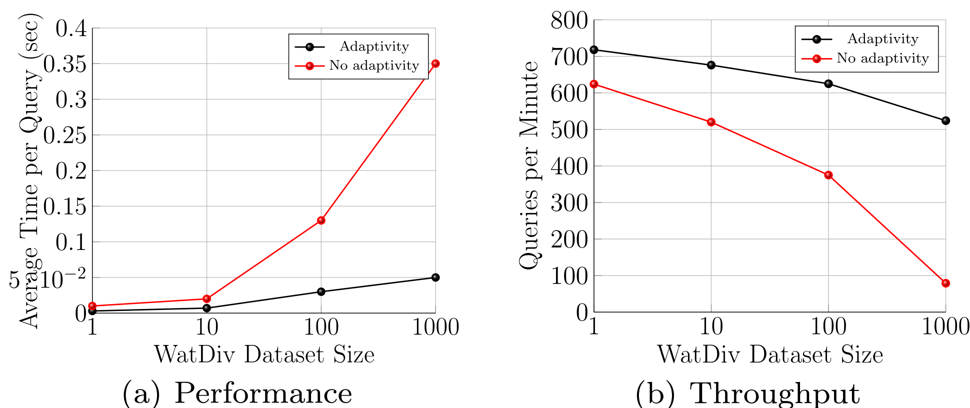


Fig. 11 WASP scalability regarding the WatDiv dataset

5.7 Data Scalability

In this experiment, we investigate the impact of dataset size on our fragmentation strategies. We use the WatDiv benchmark data generator to generate four datasets: WatDiv-1M, WatDiv-10M, WatDiv-100M, WatDiv-1B, varying from 1 million to 1 billion triples, respectively. Regarding the *WatDiv-SW* workload, Fig. 11a, b show the performance and throughput of our partitioning method with and without the adaptivity feature by increasing the dataset size. Generally, as the size of RDF datasets gets larger, the average response time per query increases and the number of queries answered in 1 min decreases accordingly. However, the rates of increase and decrease in the method with adaptivity are gradual compared to the method without adaptivity, and we can say that the query performance and throughput are scalable with RDF graph sizes.

6 Conclusion and Future Work

We have presented WASP, a novel workload-adaptive and topology-aware streaming partitioner, in order to achieve low-latency and high-throughput online queries in distributed graph stores. As each query workload typically contains popular or similar queries, WASP captures active vertices and edges that are frequently visited and traversed in the existing query workload. Using this information, the quality of partitions is improved by avoiding the concentration of active vertices in a few partitions proportional to their visit frequencies, or by reducing the probability of the cut of active edges proportional to their traversal frequencies. Our experiments show that WASP significantly reduces the number of inter-partition traversals during a query workload. It is able to handle evolving query workloads and graph topology while maintaining the quality of partitions over time. In the future, we plan to increase the performance of queries in static workloads through a workload-driven replication,

where the replication scheme of a vertex (i.e., how many replicas of the vertex are created and to which partitions these replicas are allocated) dynamically changes based on the read/write frequencies of the vertex.

Funding This work was partly supported by the National Natural Science Foundation of China (No.61672389) and Guangzhou Key Laboratory of Big Data and Intelligent Education (No.2015010009).

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Aluç G, Hartig O, Özsu MT, Daudjee K (2014) Diversified stress testing of RDF data management systems. In: International semantic web conference. Springer, pp 197–212
2. Angles R (2012) A comparison of current graph database models. In: 28th international conference on data engineering workshops (ICDEW). IEEE, pp 171–177
3. Angles R, Arenas M, Barcelo P, Hogan A, Reutter J, Vrgoc D (2016) Foundations of modern graph query languages. arXiv preprint arXiv:1610.06264
4. Angles R, Gutierrez C (2018) An introduction to graph data management. In: Graph data management. Springer, pp 1–32
5. Bok K, Kim J, Yoo J (2019) Dynamic partitioning supporting load balancing for distributed RDF graph stores. Symmetry 11(7):926
6. Buluç A, Meyerhenke H, Safro I, Sanders P, Schulz C (2016) Recent advances in graph partitioning. In: Kliemann L, Sanders P (eds) Algorithm engineering. Springer, pp 117–158
7. Carlson JL (2013) Redis in action. Manning Publications Co
8. Cattell R (2011) Scalable SQL and NoSQL data stores. ACM SIGMOD Rec 39(4):12–27
9. Chen R, Shi J, Chen Y, Zang B, Guan H, Chen H (2019) Powerlyra: differentiated graph computation and partitioning on skewed graphs. ACM Trans Parallel Comput 5(3):1–39
10. Dai D, Zhang W, Chen Y (2017) IOGP: an incremental online graph partitioning algorithm for distributed graph databases. In: Proceedings of the 26th international symposium
11. Davoudian A, Chen L, Liu M (2018) A survey on NoSQL stores. ACM Comput Surv 51(2):1–43
12. Davoudian A (2019) Helios: an adaptive and query workload-driven partitioning framework for distributed graph stores.In: Proceedings of the ACM SIGMOD international conference on management of data
13. Davoudian A, Liu M (2020) Big Data Systems: A Software Engineering Perspective. ACM Comput Surv 53(5):1–39
14. Faloutsos M, Faloutsos P, Faloutsos C (1999) On power-law relationships of the internet topology. In: Proceedings of the ACM SIGCOMM special interest group on data communication, vol 29
15. Firth H, Missier P (2016) Workload-aware streaming graph partitioning. In: EDBT/ICDT workshops
16. Firth H, Missier P (2017) TAPER: query-aware, partition-enhancement for large, heterogenous graphs. Proc Distrib Parallel Databases 35(2):85–115
17. Galárraga L, Hose K, Schenkel R (2014) Partout: a distributed engine for efficient RDF processing. In: Proceedings of the 23rd international conference on World Wide Web. ACM, pp 267–268
18. Garey MR, Johnson DS, Stockmeyer L (1974) Some simplified NP-complete problems. In: Proceedings of the 6th annual ACM symposium on theory of computing
19. Gonzalez JE, Low Y, Gu H, Bickson D, Guestrin C (2012) Power-Graph: Distributed graph-parallel computation on natural graphs. In: Proceedings of the 10th USENIX OSDI conference on operating systems design and implementation
20. Heidari S, Simmhan Y, Calheiros RN, Buyya R (2018) Scalable graph processing frameworks: a taxonomy and open challenges. ACM Comput Surv 51(3):1–53
21. Hose K, Schenkel R (2013) WARP: workload-aware replication and partitioning for RDF. In: 29th international conference on data engineering workshops (ICDEW). IEEE, pp 1–6
22. Huang J, Abadi DJ (2016) Leopard: lightweight edge-oriented partitioning and replication for dynamic graphs. Proc VLDB Endow 9(7):540–551
23. Huang J, Abadi DJ, Ren K (2011) Scalable SPARQL querying of large RDF graphs. Proc VLDB Endow 4(11):1123–1134
24. Iordanov B (2010) HyperGraphDB: a generalized graph database. In: Proceedings of the Springer international conference on web-age information management
25. Karypis G, Kumar V (1998) A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J Sci Comput 20(1):359–392
26. Karypis G, Kumar V (1999) Parallel multilevel series k-way partitioning scheme for irregular graphs. SIAM Rev 41(2):278–300
27. Khayyat Z, Awara K, Alonazi A, Jamjoom H, Williams D, Kalnis P (2013) Mizan: a system for dynamic load balancing in large-scale graph processing. In: Proceedings of the 8th ACM European conference on computer systems
28. Lehmann J, Isele R, Jakob M, Jentzsch A, Kontokostas D, Mendes PN, Hellmann S, Morsey M, Van Kleef P, Auer S et al (2015) DBpedia—a large-scale, multilingual knowledge base extracted from Wikipedia. Semant Web 6(2):167–195
29. Li D, Zhang Y, Wang J, Tan KL (2019) Topox: topology refactorization for efficient graph partitioning and processing. Proc VLDB Endow 12(8):891–905
30. Martella C, Logothetis D, Loukas A, Siganos G (2017) Spinner: scalable graph partitioning in the cloud. In: Proceedings of the 33rd IEEE ICDE international conference on data engineering
31. Martinez-Bazan N, Gomez-Villamor S, Escale-Claveras F (2011) DEX: a high-performance graph database management system. In: Proceedings of the 27th IEEE ICDEW international conference on data engineering workshops
32. Mondal J, Deshpande A (2012) Managing large dynamic graphs efficiently. In: Proceedings of the ACM SIGMOD international conference on management of data
33. Nicoara D, Kamali S, Daudjee K, Chen L (2015) Hermes: dynamic partitioning for distributed social network graph databases. In: Proceedings of the 18th EDBT international conference on extending database technology
34. Nishimura J, Ugander J (2013) Restreaming graph partitioning: simple versatile algorithms for advanced balancing. In: Proceedings of the 19th ACM SIGKDD international conference on knowledge discovery and data mining
35. Pacaci A, Özsu MT (2019) Experimental analysis of streaming algorithms for graph partitioning. In: Boncz PA, Manegold S, Ailamaki A, Deshpande A, Kraska T (eds) Proceedings of the 2019 international conference on management of data, SIGMOD

conference 2019, Amsterdam, The Netherlands, June 30–July 5, 2019. ACM, pp 1375–1392
36. Peng P, Zou L, Chen L, Zhao D (2016) Query workload-based RDF graph fragmentation and allocation. EDBT
37. Peng P, Zou L, Chen L, Zhao D (2019) Adaptive distributed RDF graph fragmentation and allocation based on query workload. IEEE Trans Knowl Data Eng 31(4):670–685
38. Pujol JM, Erramilli V, Siganos G, Yang X, Laoutaris N, Chhabra P, Rodriguez P (2010) The little engine (s) that could: scaling online social networks. Proc ACM SIGCOMM Comput Commun Rev 40(4):375–386
39. Rahimian F, Payberah AH, Girdzijauskas S, Jelasity M, Haridi S (2015) A distributed algorithm for large-scale graph partitioning. Proc ACM TAAS Trans Auton Adapt Syst 10(2):1–24
40. Saleem M, Ali MI, Hogan A, Mehmood Q, Ngomo ACN (2015) LSQ: the linked SPARQL queries dataset. In: International semantic web conference. Springer, pp 261–269
41. Schloegel K, Karypis G, Kumar V (2000) Graph partitioning for high performance scientific simulations. Army High Performance Computing Research Center
42. Shang Z, Yu JX (2013) Catch the wind: graph workload balancing on cloud. In: Proceedings of the 29th IEEE ICDE international conference on data engineering
43. Shao B, Wang H, Li Y (2013) Trinity: a distributed graph engine on a memory cloud. In: Proceedings of the ACM SIGMOD international conference on management of data
44. Shi J, Yao Y, Chen R, Chen H, Li F (2016) Fast and concurrent RDF queries with RDMA-based distributed graph exploration. OSDI 16:317–332
45. Stanton I, Kliot G (2012) Streaming graph partitioning for large distributed graphs. In: Proceedings of the 18th ACM SIGKDD international conference on knowledge discovery and data mining
46. Stegemann T, Ziegler J (2017) Pattern-based analysis of SPARQL queries from the LSQ dataset. In: International semantic web conference (posters, demos and industry tracks), pp 1–4
47. SWAT-Projects: The Lehigh University Benchmark (LUBM). http://swat.cse.lehigh.edu/projects/lubm/
48. Tian Y, Balmin A, Corsten SA, Tatikonda S, McPherson J (2013) From think like a vertex to think like a graph. Proc VLDB Endow 7(3):193–204
49. Tsourakakis C, Gkantsidis C, Radunovic B, Vojnovic M (2014) Fennel: streaming graph partitioning for massive scale graphs. In: Proceedings of the 7th ACM international conference on Web search and data mining
50. Ugander J, Backstrom L (2013) Balanced label propagation for partitioning massive graphs. In: Proceedings of the 6th ACM international conference on Web search and data mining
51. Vaquero L, Cuadrado F, Logothetis D, Martella C (2013) xDGP: a dynamic graph processing system with adaptive partitioning. arXiv preprint arXiv:1309.1049
52. Vaquero LM, Cuadrado F, Logothetis D, Martella C (2014) Adaptive partitioning for large-scale dynamic graphs. In: Proceedings of the 34th IEEE ICDCS international conference on distributed computing systems
53. Webber J (2012) A programmatic introduction to Neo4j. In: Proceedings of the 3rd ACM conference on systems, programming, and applications: software for humanity
54. Wu Z, Karimi HR, Dang C (2019) An approximation algorithm for graph partitioning via deterministic annealing neural network. Neural Netw 117:191–200
55. Xu N, Chen L, Cui B (2014) LogGP: a log-based dynamic graph partitioning method. Proc VLDB Endow 7(14):1917–1928
56. Yang S, Yan X, Zong B, Khan A (2012) Towards effective partition management for large graphs. In: Proceedings of the ACM SIGMOD international conference on management of data
57. Zeng K, Yang J, Wang H, Shao B, Wang Z (2013) A distributed graph engine for web scale RDF data. Proc VLDB Endow 6:265–276
58. Zheng A, Labrinidis A, Chrysanthis PK(2016) Planar: parallel lightweight architecture-aware adaptive graph repartitioning. In: Proceedings of the 32nd IEEE ICDE international conference on data engineering
59. Zheng A, Labrinidis A, Faloutsos C (2017) Skew-resistant graph partitioning. In: Proceedings of the 33rd IEEE ICDE international conference on data engineering