IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received November 1, 2019, accepted December 16, 2019, date of publication December 19, 2019,
date of current version December 31, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2960868

Query-Sensitive Graph Partitioner for Pattern Matching Applications

LI LU AND BEI HUA, (Member, IEEE)
School of Computer Science and Technology, University of Science and Technology of China, Hefei 230027, China
Corresponding author: Li Lu (luly9527@mail.ustc.edu.cn)

ABSTRACT Searching and mining in large graphs is critical to a variety of applications, at the core of which is the pattern matching activity. The scalable processing of large graphs requires careful distribution of graphs across clusters. Graph partitioning is the technique that divides a big graph into several non-overlapped subgraphs and assigns each subgraph to a compute node. Traditional workload agnostic partitioners aim to minimize the number of inter-partition edges using only graph topology, which, however, may not obtain the best solution if the workload exhibits skew. Some workload-aware partitioners choose to mine information from a specific workload and use it to minimize the number of inter-partition traversals during execution; however, their methods are not suitable for pattern matching applications. In this work, we propose a query-sensitive graph partitioner that aims to improve existing partitioning for a given pattern matching workload. The partitioner takes any initial partitioning as a starting point and iteratively adjusts it by exchanging chosen clusters across partitions, heuristically reducing the probability of inter-partition traversals. We determine a few implementation-irrelative factors that may increase the traversal probability of an edge and quantify them into a calculable indicator with information from query patterns and graph topology. Then, we propose an efficient algorithm to calculate the indicator and implement a graph repartitioner by combining the indicator with a greedy cluster-exchanging mechanism. Finally, we generate a large heterogeneous labeled graph with real-world data crawled from the Netease Music website and evaluate the partitioning quality of our repartitioner with a few meaningful query patterns of common topologies including line, loop and branching. Compared with a hash-based partitioning, our system can reduce the inter-partition traversals by at least 70%. Compared with the state-of-the-art graph partitioner *Metis*, our repartitioner can reduce the inter-partition traversals by at least 50%.

INDEX TERMS Graph computing, graph partitioning, pattern match, workload mining.

I. INTRODUCTION

Modern big data increasingly appear in the form of large heterogeneous labeled graphs. Examples include the World Wide Web, online social networks, and shopping histories and preferences. Pattern matching queries over labeled graphs are widely used in applications such as recommender systems, social analysis, and fraud detection, to name a few. Fig. 1(a) shows a fragment of a labeled graph that contains information from an online music website including tracks, artists, users, and relationships between them, and Fig. 1(b) shows a recommendation rule represented as a query graph. The recommendation activity becomes a subgraph search task, where some subgraph search methods are used to identify the items that should be recommended.

Currently, the size of a labeled graph easily exceeds the storage and computing capacity of a single commodity type machine. For example, the crawled Web graph is estimated to have more than 20 billion pages (vertices) with 160 billion hyperlinks (edges). A viable approach is to split the graph across a large cluster of machines and use a distributed algorithm to do the computation. *K-balanced graph partitioning* is a well-studied problem that divides a graph into k components of approximately the same size and minimizes the cut size, which is the number of edges connecting different components. However, this problem is known to be NP-hard, even if one relaxes the balanced constraint to approximately balanced [1]. There are many heuristics solving this problem, among which is the de facto standard

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

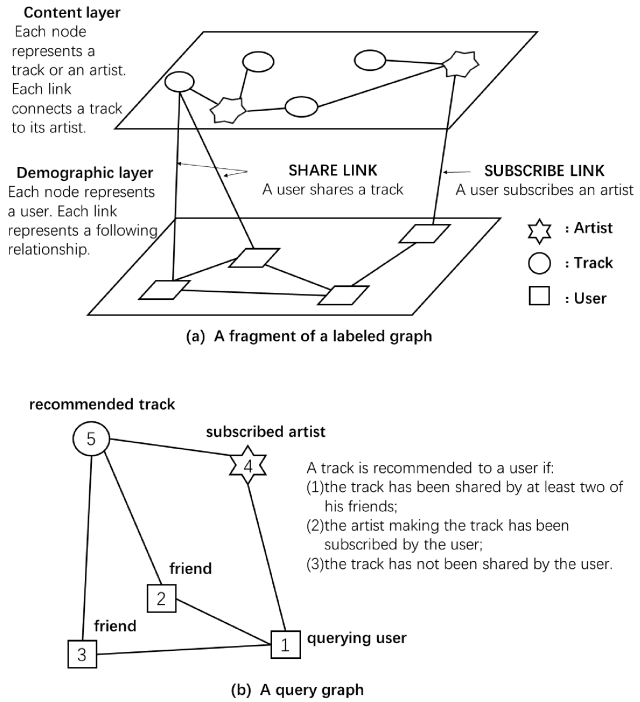


FIGURE 1. A graph-based recommendation system.

software package *Metis* [2]. Generally, these heuristics are quite effective; however, due to their high computing complexity, they are only suitable for offline operations, typically performed ahead of analytical workloads.

Fast graph partitioning usually uses simple heuristics. The simplest and the default partitioning strategy adopted by many distributed graph processing platforms, such as *Pregel* [3], *PEGASUS* [4] and *GraphLab* [5], is hashing. The hashing method may produce balanced partitioning if a good pseudorandom hash function is chosen, but the choice may result in a suboptimal edge cut as it is entirely oblivious to the graph structure. Article [6] carries out a rigorous and empirical study of a set of natural heuristics and shows that they can generate better partitioning than the hashing method. *FENNEL* [7] overcomes the high computing complexity of the traditional *k-balanced graph partitioning* problem by relaxing the hard cardinality constraints. This method also provides a unifying framework that accommodates many of the previously proposed heuristics as special cases.

However, the above works do not take into account the workload running on the partitioning. In fact, workload may have a great impact on the partitioning quality. Taking Fig. 1 as an example, edges connecting hot tracks and their artists are accessed more frequently than edges connecting outdated tracks and their artists; in this case, minimizing the edge cut size is not the best solution. Therefore, workload-aware graph partitionering has been proposed. Early workload-aware partitioners are primarily applied to distributed relational databases, such as *Schism* [8] and *SWORD* [9]. Tuples are represented by nodes, and transactions are represented by edges that connect the tuples used within the transaction. A previous SQL trace is used as the representative workload, and the database is repartitioned to minimize the number of multisited transactions. In recent years, many workload-aware graph partitioners have been used to improve analytical workloads designed for the *bulk synchronous parallel* (*BSP*) computation model, such as *LogGP* [10]. A historical log of edges traversed in previous superstep is kept and used to predicate future traversals, and the graph is repartitioned for the next superstep. Different from the trace/log based graph partitioners that directly measure individual edge traversals, *TAPER* [11] uses query patterns and local graph structures to predict edge traversal probabilities in run time without the use of workload trace, thus avoiding the overfitting problem.

In a sense, *TAPER* is the most relevant work to this paper, i.e., improving an existing graph partitioning for a set of query patterns without use of historical trace or log. However, *TAPER* only considers path queries that can be expressed using a *regular path queries* (*RPQ*) formalism, not including complex topologies such as branching and loop, which are common in general query patterns, such as the query graph shown in Fig. 1. As general pattern matching is a NP-C problem with various implementations, and moreover, while executing, a pattern matching process can start at any vertex of the graph, search in any direction and may even return approximate results. Thus, it is impossible to use a deterministic method, like the method used by *TAPER*, to predict the edge traversal probability. We address this problem in this paper.

The specific contributions of this work are as follows:

(1) First, we determine a few implementation-irrelative factors that influence the *edge traversal probability* (*ETP*) in pattern matching and develop a heuristic formula to estimate it with information from query patterns and graph topology.

(2) Second, we implement a query-sensitive graph repartitioner that includes an efficient algorithm for calculating the *ETP*s and a greedy algorithm for iteratively improving the quality of partitioning.

(3) Third, we evaluate our graph repartitioner with a real-world dataset from the Netease Music website [12] and a few meaningful query patterns. Compared with the state-of-the-art graph partitioner *Metis*, our repartitioner can reduce the inter-partition traversals by at least 50%.

The rest of this paper is organized as follows. Section II discusses background material and related works. Section III defines the terms used in this paper. Section IV presents the formulae for computing *edge traversal probability* under pattern query workload, and Section V discusses the implementation of the graph repartitioner. Section VI presents the experimental results, and Section VII concludes this paper.

II. RELATED WORKS

The *K-balanced graph partitioning* problem has been studied since the 1970s and has many proposed solutions. We do not seek a new graph partitioning algorithm; rather, we seek to propose a query-sensitive method that may improve an

existing partitioning for a given set of query patterns. This section only focuses on the most relevant works.

Metis [2] is the de facto standard software package for offline graph partitioning, which is typically used as a ''one-off'' step rather than for repeatedly repartitioning a graph. To the best of our knowledge, there is no query-sensitive graph partitioner optimized for general query patterns; therefore, we use *Metis* as one of the comparison baselines.

Mature research on query-sensitive partitioning is currently confined to relational Database Management System(DBMS). *Schism* [8] captures a query workload over a period of time, models it as a graph, and partitions the graph using existing ''one-off'' partitioner to achieve a minimum edge cut. *SWORD* [9] builds it upon the ideas presented in *Schism*, but the query workload is modeled as a hypergraph to reduce the memory footprint and computing complexity. All these works focus on a relational data model where typical workloads consist of short query paths with only one or two hops; thus, they are not applicable to our work.

TAPER [11] aims to improve existing graph partitioning for a specific set of path queries. *TAPER* creates a new measure of partitioning quality referred to as *workload-aware stability*, which is the probability of the network flow staying within a partition when executing a given query workload. For ease of calculation, *TAPER* defines an operational metric called *extroversion of vertex* as an indicator of inter-partition traversals, which denotes the likelihood of a vertex being the source of inter-partition traversals. Conceptually, while given a labeled graph and a set of path queries expressed as regular expressions, *TAPER* first computes a set of transition matrices called *Visitor Matrix* by using a *multistep walk model* over the graph and uses the information in *Visitor Matrix* to calculate the extroversion of each vertex. After identifying a set of vertices with high extroversion in each partition, *TAPER* swaps them between partitions such that the total extroversion of the graph decreases. As *Visitor Matrix* is impractically large to compute, *TAPER* uses heuristics to reduce both space and computing complexities. However, *TAPER* does not consider complex topologies such as branching and loop, and moreover, the deterministic method used to compute the vertex extroversion is not applicable to those topologies. Therefore, *TAPER* cannot deal with general pattern matching on labeled graphs.

LOOM [13] proposes a streaming graph partitioner for the workload of pattern matching queries. The main idea is first capturing the most common motifs from the query graphs using a generalized trie data structure, identifying subgraphs that match these motifs in a graph-stream within a window over the workload, and finally placing these subgraphs wholly within beneficial partitions. However, *LOOM* is published as an ongoing work, neither detailed algorithm description nor any experimental evaluation is reported in [13], and no follow-up work has been reported so far.

Article [14] proposes a *greedy refinement* algorithm to adaptively reduce the edge cut size by exchanging vertices between partitions. Each partition randomly selects a cluster of boundary vertices, calculates the potential gains defined by an objective function about edge cut size and global workload balance if the cluster is swapped to different partitions, and chooses the best partition as the destination. The core idea of the algorithm is adopted by many works, such as *TAPER* and *Metis*, and is also adopted by this work.

III. DEFINITION

Labeled graph: a labeled graph is defined as $G = (V_G, E_G, L_G)$, where L_G is a label set, and each node $v \in V_G$ is attached with a set of labels. The label set of a node v in G is denoted by $L(v) \subseteq L_G$.

Query graph: using the same symbolic notation, a query graph (or a query pattern) over a labeled graph G is defined as $q = (V_q, E_q, L_q)$, where $L_q \subseteq L_G$.

Subgraph isomorphism: given two labeled graphs H and G, H is called subgraph isomorphic to G, denoted as $H \sim G$, if there exists an injective function $f: V_H \rightarrow V_G$, such that (1) $\forall u \in V_H$, $L(u) \subseteq L(f(u))$, and (2) $\forall (u, v) \in E_H, (f(u), f(v)) \in E_G$.

Pattern matching query: a pattern matching query is defined in terms of subgraph isomorphism. Given a labeled graph G and a query q, execution of q over G returns a set of subgraphs of G, each of which is subgraph isomorphic to q.

K-way partitioning: a k-way partitioning of G is defined as $P = \{P_i | i = 1, 2, \cdots, k\} = \{(V_i, E_i) | i = 1, 2, \cdots, k\}$, where $\bigcup V_i = V$, and $\bigcap V_i = \emptyset$. Each $P_i = (V_i, E_i)$ is called a partition, where E_i contains all the edges that have at least one vertex in V_i. In other words, if an edge connects P_i and P_j, then it is included in both P_i and P_j. Edges connecting two partitions are called inter-partition edges.

Edge cut: given a labeled graph G and a graph partitioning P, the edge cut $E_{cut}(G, P)$ is a set of of all the inter-partition edges.

Diameter of a graph: the diameter of a connected graph G, denoted as $diam(G)$, is the maximum length of all the shortest paths between pairs of vertices in graph G.

Distance between two edges: given two edges e and e' in a connected graph, the distance between e and e', denoted as $dist(e, e')$, is defined as the minimum number of hops between them.

IV. INDICATOR OF EDGE TRAVERSAL PROBABILITY

In this work, we address the problem of efficiently and incrementally improving patter matching query performance over a k-way partitioning of a large labeled graph. The problem can be informally described as follows: given a labeled graph G, a query workload Q over G, and an original k-way partitioning $P_k(G)$, compute a new partitioning $P'_k(G, Q)$ such that $P'_k(G, Q)$ is better than $P_k(G)$ for workload Q.

Traditional *k-balanced graph partitioning* aims to minimize the edge cut size while maintaining a restricted workload balance.Many heuristic approaches choose to relax the balance constraint to reduce computing complexity. This strategy is not only effective but also reasonable, as real-

world workloads cannot be simply measured as the number of vertices and edges. As the graph is not uniformly processed in many applications, some heuristics decide to minimize the probability of inter-partition traversals rather than the edge cut size. In this work, we also use this measure to evaluate partitioning quality.

To estimate the probability of an edge to be traversed (accessed), we need some knowledge of how pattern matching query works. As subgraph searching is a computationally intensive task, practical implementations use heuristics to avoid searching the whole graph directly. Many implementations adopt a filtering-matching method that divides the task into two stages. In the filtering stage, a few rules are employed to identify a set of candidate vertices for further verification, with each vertex in query graph q corresponding to a set of vertices in graph G. In the matching stage, a proper pattern matching strategy such as depth-first search is used to search for the subgraphs, starting from these candidate vertices. Many applications only require a subset of the results returned, such as the *top-k* matched subgraphs or the first k matched subgraphs. In some cases, even approximate matching results are allowed. Due to the uncertainty of filtering rules, pattern matching strategies, search termination conditions, etc., it is impossible to estimate the traversal probability of each edge definitely, and an implementation-irrelative indicator is needed to approximate the edge traversal probability.

To this end, we determine the factors that may increase the traversal probability of an edge, and we quantify them into a calculable indicator. In general, all pattern matching implementations start a search from an edge that matches one of the edges of a pattern (hereinafter called a matched edge) and then expand the search around it to check if there exists a complete match. Apparently, matched edges and their neighboring edges are more likely to be traversed in the matching stage. Specifically speaking, for a chosen edge e in the graph, its traversal probability increases in the following situations: (1) edge e is a matched edge, so it is highly likely to be chosen as the starting point of a search; (2) edge e is neighboring to a partially matched subgraph, so it is highly likely to be traversed when the pattern matching algorithm expands the search towards it.

However, subgraph matching is a high overhead operation and we do not want to introduce such a heavy weight operation into graph partitioning, so we use simple heuristics to estimate the traversal probability brought by e's neighboring edges. The main idea is that within a circular area with edge e as the center and $(diam(q) - 1)$ as a radius, any matched edge $e'(e \neq e')$ adds a value to the traversal probability of edge e; the longer the distance between them, the smaller the value is.

For the convenience of representation, we think of an edge as a specific subgraph; thus, "edge e matches an edge of pattern q" is represented as "$e \sim q$". According to the above analysis, we define the *edge traversal probability* (*ETP*) of edge e under query q as follows:

$$ETP(e, q) = I(e \sim q) + \sum_{\substack{e' \neq e \\ dist(e', e) < diam(q)}} \frac{I(e' \sim q)}{dist(e', e)}$$

$$I(e \sim q) = \begin{cases} 1, & \text{if } e \sim q \\ 0, & \text{otherwise.} \end{cases} \quad (1)$$

If the workload consists of several query graphs, denoted as $Q = \{q_i | i = 1, 2, \cdots, k\}$, then edge traversal probability of edge e under workload Q is defined as:

$$ETP(e) = \sum_{q \in Q} ETP(e, q) \quad (2)$$

With these definitions, the *inter-partition traversal probability* (*IPT*) of a partitioning P over graph G is defined as:

$$IPT(G, Q, P) = \sum_{e \in E_{cut}(G,P)} ETP(e) \quad (3)$$

Now our query-sensitive graph repartitioning problem can be formally described as follows: given a labeled graph G, a query workload Q, and an original graph partitioning P, compute a new partitioning P' such that $IPT(G, Q, P') < IPT(G, Q, P)$.

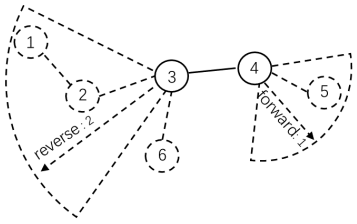
V. QUERY-SENSITIVE GRAPH REPARTITIONER

The implementation of a query-sensitive graph repartitioner consists of the following three parts: (1) calculating the *edge traversal probability* for each edge in the graph; (2) computing a cluster of vertices and accompanying edges in each partition that is to be exchanged to another partition, hereinafter called an *exchanging cluster*; and (3) sending all the exchanging clusters to their destinations to form a new partitioning.

A. CALCULATING EDGE TRAVERSAL PROBABILITY

A simple realization of formula (1) is using a pull mode. Starting from an edge e and expanding outwards, for every edge e' that is subgraph isomorphic to a query q and is located within the radius of $diam(q)$, adding to $ETP(e)$ a probability that is in inverse proportion to the distance between e and e'. However, pull mode incurs high computing overhead as the formula calculation must be performed on every edge. To reduce computing complexity, we adopt a push mode. Starting from an edge e that is subgraph isomorphic to a query q, for every edge e' that is subgraph isomorphic to q and is located within the radius of $diam(q)$, a probability that is in inverse proportion to the distance between e and e' is added to $ETP(e')$.

Another problem with the simple realization of formula (1) is that it is inefficient to delimit the search area starting from each edge with the same maximum radius. Taking the query graph in Fig. 2 as an example, when edge connect vertex 3 and vertex 4(hereinafter denoted as e_{34}) is chosen as the search starting point, only one-hop probe is needed



e_{ij}	$r_{fs}(e_{ij})$	$r_{rs}(e_{ij})$
e_{12}	3	0
e_{23}	2	1
e_{34}	1	2
e_{45}	0	3
e_{36}	0	2

FIGURE 2. Example of query decomposition.

along the direction from vertex 3 to vertex 4, rather than the the diameter of the pattern(4). To this end, for edge e_{ij} we define the search direction from vertex i to vertex j as forward and the opposite direction as reverse. To reduce unnecessary searches, for each edge of a query, we compute the forward and reverse search radii. The forward and reverse search radii of e_{34} are shown in Fig. 2.

Prior to the *ETP* computation, each query is decomposed into individual edges accompanying their forward and reverse search radius. Fig. 2 shows the decomposition of a query, where $r_{fs}(e_{ij})$ is the forward search radius and $r_{rs}(e_{ij})$ is the reverse search radius.

Algorithm 1 describes the process of *ETP* calculation of all edges in a graph G under query workload Q. $ETP(E)$ is a set of *ETP* values of all edges in E. Each matched edge e first increases its *ETP* value by one (line 7) and then does a breadth-first search along the forward direction (line 8-line 10) and the reverse direction (line 11-line 13), increasing the *ETP* value of each edge that it traverses. As each edge needs a spread process in the worst case, take the total number of edges as the scale of problem, the time complexity of Algorithm 1 is O(n).

Algorithm 1 ETP_Calculation

Input: graph $G = (V, E)$, workload Q
Output: $ETP(E)$
1: **for** each e in E **do**
2: $ETP(e) \leftarrow 0$
3: **end for**
4: **for** each e in E **do**
5: **for** each q in Q **do**
6: **if** $e \sim q$ **then**
7: $ETP(e) \leftarrow ETP(e) + 1$
8: **for** each forward e' with $dist(e', e) \leq r_{fs}(e)$ **do**
9: $ETP(e') \leftarrow ETP(e') + \frac{1}{dist(e',e)}$
10: **end for**
11: **for** each reverse e' with $dist(e', e) \leq r_{rs}(e)$ **do**
12: $ETP(e') \leftarrow ETP(e') + \frac{1}{dis(e',e)}$
13: **end for**
14: **end if**
15: **end for**
16: **end for**
17: **return** $ETP(E)$

B. COMPUTING EXCHANGING CLUSTERS

When $ETP(E)$ is calculated, each partition needs to determine an exchanging cluster such that the inter-partition traversal probability P_{IPT} decreases when the exchanging cluster is sent to its destination partition.

An exchanging cluster that is extracted from partition P_i and bound for partition P_j is denoted as C_{ij}. $C_{ij} = (V_C, E_C)$ is a specific partition taken out from $P_i = (V_i, E_i)$ with $V_i \subset V_c$, $E_i \subset E_c$, and E_c consisting of all the edges associated with at least one vertex in V_c. In cluster C_{ij}, edges connecting C_{ij} and the outer world are called boundary edges, and edges only connecting vertices in V_C are called inner edges.

The variation of P_{IPT} caused by exchange of C_{ij} can be computed from the *ETP* values of all the boundary edges of C_{ij}, as inner edges do not contribute to *IPT*. For a boundary edge $e \in C_{ij}$, we consider the following three cases: (1) if e connects C_{ij} and the rest of P_i, P_{IPT} increases by $ETP(e)$ as e would transform from an inner edge to an inter-partition edge; (2) if e connects C_{ij} and P_j, P_{IPT} decreases by $ETP(e)$ as e would transform from an inter-partition edge to an inner edge; and (3) if e connects C_{ij} and a partition other than P_i and P_j, P_{IPT} does not change as e would still be an inter-partition edge. Thus, the variation of P_{IPT} caused by exchange of C_{ij}, denoted as $\Delta P_{IPT}(C_{ij})$, can be expressed as:

$$\Delta P_{IPT}(C_{ij}) = \sum_{e \in C(P_i, P_j)} [I(e) * ETP(e)]$$

$$I(e) = \begin{cases} 1, & \text{if } e \text{ connects } C(P_i, P_j) \text{ and } P_i \\ -1, & \text{if } e \text{ connects } C(P_i, P_j) \text{ and } P_j \\ 0, & \text{otherwise} \end{cases} \quad (4)$$

We use a simple greedy algorithm to compute the exchanging cluster in each partition, which is shown in Algorithm 2. For a given partition P_i, the algorithm returns an exchanging cluster C_{exh} and its destination partition ID *PID*. Initially, an inter-partition edge e_0 with the highest *ETP* value is chosen as the starting point for constructing the exchanging cluster, and the other side partition that e_0 connects is chosen as the destination partition (line 2-line 4). Then, in a *WHILE* loop (line 5-line 24), the algorithm tries to add more vertices (one at a time) and associated edges in the cluster, assesses the contribution of each intermediate cluster to the reduction of *IPT* and records the best cluster so far.

In each loop, each boundary edge of C is checked to see if there is any benefit to turn it into an inner edge. For a given boundary edge e of C, this is done by adding the remote vertex v_e of e and all the edges connecting e at v_e (i.e., the future boundary edges) into a testing cluster N_e (line 8) and computing the reduction of P_{IPT} brought by merging N_e and C (line 9). If N_e can bring the lowest ΔP_{IPT} so far, it is recorded as N_{exh} (line 10-line 12). After all the boundary edges have been checked, N_{exh} is merged with C to form a new cluster (line 20). If the newly formed cluster brings the lowest ΔP_{IPT} so far, it is recorded as C_{exh}, (line 21-line 23).

The *WHILE* loop terminates when C reaches the maximum cluster size (line 5) or all the boundary edges have

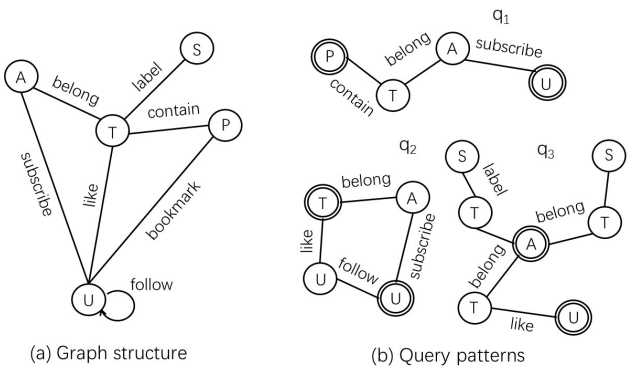
Algorithm 2 Exh_Cluster_Calculation for P_i

Input: graph $G = (V, E)$, $ETP(E)$, Partition P, max cluster size L
Output: C_{exh},PID

1: $best\_\Delta P_{IPT} \leftarrow +\infty$
2: find an inter-partition edge e_0 in P_i with the highest ETP
3: $C \leftarrow (\{v_{e_0}\}, \{e_0\})$ // v_{e_0} is a vertex of e_0 and $v_{e_0} \in V_i$
4: $PID \leftarrow$ ID of the partition on the other side of e_0
5: **while** $|C| < L$ **do**
6: $curr\_\Delta P_{IPT} \leftarrow +\infty$, $FLAG \leftarrow 0$
7: **for** each boundary edge e in E_c **do**
8: $N_e \leftarrow (\{v_e\}, \{e' | e'$ connects e at $v_e\})$ // v_e is the remote vertex of e
9: $temp\_\Delta P_{IPT} \leftarrow \Delta P_{IPT}(C \cup N_e)$
10: **if** $temp\_\Delta P_{IPT} < curr\_\Delta P_{IPT}$ **then**
11: $curr\_\Delta P_{IPT} \leftarrow temp\_\Delta P_{IPT}$, $N_{exh} \leftarrow N_e$
12: **end if**
13: **if** $ETP(e) > 0$ **then**
14: $FLAG \leftarrow 1$
15: **end if**
16: **end for**
17: **if** $FLAG = 0$ **then**
18: **break**
19: **end if**
20: $C \leftarrow C \cup N_{exh}$
21: **if** $curr\_\Delta P_{IPT} < best\_\Delta P_{IPT}$ **then**
22: $best\_\Delta P_{IPT} \leftarrow curr\_\Delta P_{IPT}$, $C_{exh} \leftarrow C$
23: **end if**
24: **end while**
25: **return** C_{exh}, PID

$ETP = 0$ (indicated by $FLAG = 0$, line 17-line 19), no matter which meets first. From the goal of maximally reducing the IPT value, the algorithm should continuously expand the exchanging cluster so long as the enlargement can further reduce the IPT, and the algorithm only ceases when all the boundary edges of C have a zero ETP value. However, to prevent the cluster from growing too large in some extreme cases, we limit the maximum cluster size to L. According to our experiments, we find that the maximum cluster size is less than 200 even if we do not limit its size artificially. In the experiments in Section VI, we set L to 1000.

The greedy mechanism of Algorithm 2 has an expanding search range by adding a neighbor edge set N to the exchanging cluster C continually and choosing the best result C ever found. This expanding process ends when algorithm find all the boundary edges of cluster C have a zero ETP value. To avoid extreme case of cluster growing too big, we set a maximum size N. As all the parameters in Algorithm 2 are constants or have a upper limit, the time complexity of Algorithm 2 is theoretically O(1). In the experiment in Section 6 with N set to 1000, we find all the maximum sizes of C are all less than 200.



(a) Graph structure (b) Query patterns

FIGURE 3. Graph structure and query patterns used in the experiment.

C. EXCHANGING CLUSTERS

After an exchanging cluster is computed, it is sent to the destination partition and merged with that partition. As each partition calculates an exchanging cluster independently, it is possible that a single inter-partition edge is included in two exchanging clusters and causes confusion. To avoid this situation, we borrow the method used by *Metis*. All the partitions are ordered by their IDs, and clusters are transmitted either in ascending order or in descending order. The cluster exchange process is completed in a series of iterations, and each iteration is divided into two rounds. In the first round, clusters must be transmitted in descending order, i.e., from partitions with higher IDs to partitions with lower IDs; in the second round, clusters must be transmitted in ascending order.

VI. EVALUATION

A. EXPERIMENTAL SETUP AND DATASET

We use a real-world dataset from the Netease Cloud Music website [12] to test our graph repartitioner. The Netease Cloud Music is a famous freemium music streaming service developed and owned by the NetEase Incorporation in China. As of April 2017, the platform has 300 million users and a music database consisting of over 10 million songs. On the platform, a user can follow other users, subscribe to songs of specific musicians, make comments or do other activities. We write a web crawler to collect data from the website and organize the data into a labeled graph with the structure commonly used in recommendation systems. This highly heterogeneous graph contains 12,312,091 vertices and 39,445,732 edges, and the total size is about 24.1GB.

Fig. 3(a) shows the graph structure used in the experiment, and some information such as comments and user profiles are omitted for simplicity. There are five types of vertices in the graph: artist (A), track (T), playlist (P), style (S) and user (U); there are seven relationships between vertices. Vertex attribute values that are needed for matching recommendation rules, such as the influence value of a user and the play count of a playlist, are also collected and stored.

To the best of our knowledge, there is no well-accepted benchmark for evaluating pattern matching applications. Therefore, we design three basic patterns that are meaningful in a real music application and can be extended or combined

to form complex patterns. The topologies of the three patterns are line, loop and branching, which are shown in Fig. 3(b).

Query pattern q_1 is used to recommend a hot playlist to a user, which is interpreted as follows: if a playlist (P) is hot (i.e., having a higher number of play times) and contains a few tracks (T) that belong to an artist (A) who is subscribed by a user (U), then the playlist is recommended to the user.

Query pattern q_2 is used to recommend a missed track to a user, which is interpreted as follows: if a track (T) is liked by a VIP (U) (i.e., a person with a high influence value) who is followed by a user (U), and the track (T) also belongs to an artist (A) who is subscribed by the user (U) but track (T) is not yet heard by this user (U), then the track is recommended to the user.

Query pattern q_3 is used to recommend a versatile artist to a user, which is interpreted as follows: if an artist (A) has released some well-known tracks (T) (i.e., tracks with high play counts) with at least two different styles (S) (selected by the website as a weekly recommendation), then the artist (A) is recommended to a user (U) according to the track (T) the user has ever heard.

The query-sensitive graph repartitioner is used to improve an existing graph partitioning for a given workload. To this end, we choose two widely used practical graph partitioners, hashing and *Metis* as our comparison references. The number of inter-partition traversals (*IPT*) in run time is used as the measure of partitioning quality.

All the experiments adopt the following process: (1) running hashing or *Metis* on the data graph to generate an initial partitioning; (2) running our graph repartitioner on the initial partitioning to produce an enhanced partitioning; (3) running the pattern matching application on the initial partitioning and counting the average *IPT* number; and (4) running the pattern matching application on the enhanced partitioning and counting the average *IPT* number.

All experiments are run on a machine with an Intel i7 CPU Intel 6700K at 3.4 GHz with 64 GB RAM. The *Pregel* framework is used for graph processing.

B. CONVERGENCE RATE AND OPTIMIZATION EFFECT

Cluster exchanging is an iterative process that progressively improves the partitioning quality at the price of computing overhead. This experiment evaluates the convergence rate and optimization effect of our repartitioner on an existing partitioning generated by hashing or *Metis*. The simple hashing method assigns vertexes to different compute nodes based on the main key value of each vertex, and the *Metis* software tool calculates partitioning result according to user input. Eight-way graph partitioning is used in this experiment, each partition is about 3GB.

For each given graph partitioning, we complete a set of experiments that differ in the number of iterations the repartitioner runs on the initial partitioning. We change the number of iterations from 0 (corresponding to no repartitioning) to 200 (on hashing generated partitioning) or 100 (on *Metis* generated partitioning) with an interval of 20. Therefore, for each given graph partitioning, the repartitioner generates a set of enhanced partitioning schemes. On each enhanced partitioning, we run a pattern matching application for 10 minutes and count the total number of inter-partition traversals (the *IPT* value). For each enhanced partitioning, the experiment is repeated 10 times, and the *IPT* values are averaged.

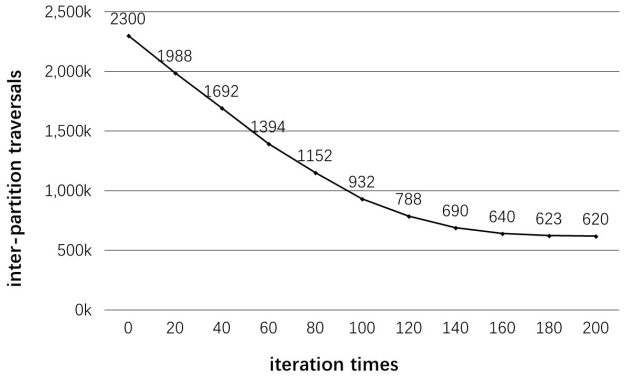


FIGURE 4. Convergence on the hash-produced partitioning.

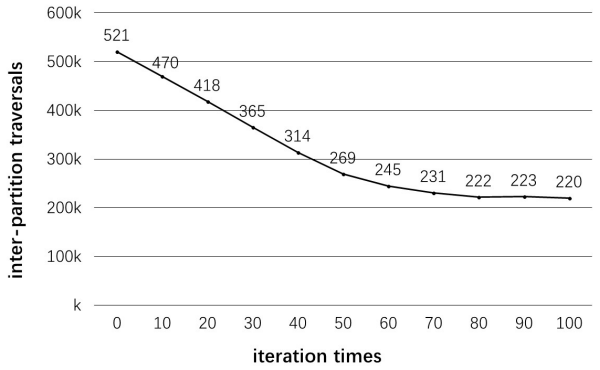


FIGURE 5. Convergence on the Metis-produced partitioning.

Fig. 4 shows the experimental data when the initial partitioning is generated by the hashing method. The horizontal axis displays the number of iterations (zero corresponds to the initial graph partitioning), and the vertical axis shows the average *IPT* value. The figure shows that the *IPT* value decreases linearly in the first 100 iterations, slows gradually, and finally converges after about 160 iterations. Compared with the hashing method, our repartitioner reduces the IPT value by approximately 75% when the algorithm converges, and nearly 80% improvement is achieved within the first 120 iterations.

Fig. 5 shows the experimental data when the initial partitioning is generated by *Metis*. This figure shows that the IPT value decreases linearly in the first 50 iterations, slows, and finally converges after approximately 80 iterations. Compared with Metis, our repartitioner reduces the IPT value by approximately 58% when the algorithm converges, and nearly 60% of the improvement is achieved within the first 60 iterations. Comparing the data in Fig. 4 and Fig. 5, we find that our repartitioner converges faster on *Metis*-generated partitioning. This result occurs because *Metis* generates much better graph partitioning, which leaves less space for optimization.

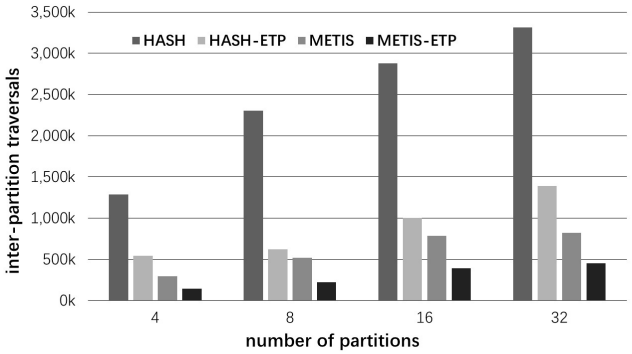


FIGURE 6. *IPT* counts of different number of partitions.

C. OPTIMIZATION EFFECTS WITH DIFFERENT VALUES OF K'S

This experiment tests the influence of partition count on the optimization effect of our repartitioner. The number of partitions (i.e., parameter k) is set to 4, 8, 16, and 32, and the size of each partition is about 6GB, 3GB, 1.5GB and 0.75GB. For each k, the *IPT* values of four partitioning schemes are counted as follows: hashing-generated partitioning (denoted as HASH), enhanced graph partitioning over hashing-generated partitioning (denoted as HASH-ETP), *Metis*-generated partitioning (denoted as METIS), and enhanced graph partitioning over *Metis*-generated partitioning (denoted as METIS-ETP). The *IPT* value of an enhanced graph partitioning is obtained when the algorithm converges. For each k, the experiment is repeated 10 times, and the *IPT* values are averaged.

Fig. 6 shows the experimental data, where the horizontal axis displays the value of k, and the vertical axis shows the average *IPT* value. For all the four partitioning schemes, the *IPT* values increase with k, a natural result due to the increased edge cut size. For all the four k values, our repartitioner reduces the IPT value remarkably, and the reduction on hashing partitioning is more prominent.

It is worth noting that the partitioning quality of hashing + repartitioning is worse than Metis + repartitioning and is even worse than *Metis*. This result occurs because the greedy heuristics can only do local optimization; thus, the effect heavily depends on the initial partitioning.

VII. CONCLUSION

In this paper, we propose a query-sensitive graph repartitioner that aims to improve an existing graph partitioning method for a given pattern matching workload. First, we propose a simple heuristics to estimate the traversal probability of each edge by combining the topological information from graph and patterns and then design an efficient algorithm to calculate them. Second, we propose a simple greedy algorithm to compute the exchanging cluster of each partition so that the global inter-partition traversal probability would decrease if these clusters are exchanged between partitions. These two steps are combined and executed iteratively to improve the partitioning quality progressively. Third, we generate a large heterogeneous labeled graph with the data crawled from the Netease Cloud Music website and evaluate the partitioning quality of our repartitioner. Compared with the state-of-the-art graph partitioner *Metis*, our repartitioner can reduce the number of inter-partition traversals by at least 50%. To the best of our knowledge, this is the first graph partitioner optimized for general pattern matching applications.

REFERENCES

[1] K. Andreev and H. Racke, "Balanced graph partitioning," *Theory Comput. Syst.*, vol. 39, no. 6, pp. 929–939, 2006.
[2] G. Karypis and K. Lab. *Metis Official Website*. Accessed: Jun. 2019. [Online]. Available: http://glaros.dtc.umn.edu/gkhome/views/metis
[3] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 135–146.
[4] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Sci. Program.*, vol. 13, no. 3, pp. 219–237, 2005.
[5] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "GraphLab: A new framework for parallel machine learning," Jun. 2010, *arXiv:1006.4990*. [Online]. Available: https://arxiv.org/abs/1006.4990
[6] I. Stanton and G. Kliot, "Streaming graph partitioning for large distributed graphs," in *Proc. 18th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2012, pp. 1222–1230.
[7] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, "Fennel: Streaming graph partitioning for massive scale graphs," in *Proc. 7th ACM Int. Conf. Web Search Data Mining*, 2014, pp. 333–342.
[8] C. Curino, E. Jones, Y. Zhang, and S. Madden, "Schism: A workload-driven approach to database replication and partitioning," *Proc. VLDB Endowment*, vol. 3, nos. 1–2, pp. 48–57, 2010.
[9] A. Quamar, K. A. Kumar, and A. Deshpande, "Sword: Scalable workload-aware data placement for transactional workloads," in *Proc. 16th Int. Conf. Extending Database Technol.*, 2013, pp. 430–441.
[10] N. Xu, L. Chen, and B. Cui, "LogGP: A log-based dynamic graph partitioning method," *Proc. VLDB Endowment*, vol. 7, no. 14, pp. 1917–1928, 2014.
[11] H. Firth and P. Missier, "Taper: Query-aware, partition-enhancement for large, heterogenous graphs," *Distrib. Parallel Databases*, vol. 35, no. 2, pp. 85–115, 2017.
[12] N Incoporation. *Netease Music*. Accessed: Mar. 2019. [Online]. Available: https://music.163.com/
[13] H. Firth and P. Missier, "Workload-aware streaming graph partitioning," in *Proc. EDBT/ICDT Workshops*, 2016.
[14] G. Karypis and V. Kumar, "Multilevelk-way partitioning scheme for irregular graphs," *J. Parallel Distrib. Comput.*, vol. 48, no. 1, pp. 96–129, 1998.

LI LU was born in 1989. He received the B.S. degree in computer science from the University of Science and Technology of China, in 2011, where he is currently pursuing the Ph.D. degree in computer science. His research interests are big data processing and graph computing.

BEI HUA was born in 1966. She received the B.S. degree in electronic engineering from the University of Science and Technology of China, in 1990, the M.S. degree in electronic engineering from Peking University, in 1993, and the Ph.D. degree in computer science from the University of Science and Technology of China, in 2005. She is currently a Full Professor and the Ph.D. Tutor with the University of Science and Technology. Her research interests include high-performance computing, edge computing, and virtualization.