Hermes: Dynamic Partitioning for Distributed Social Network Graph Databases

Daniel Nicoara
University of Waterloo
daniel.nicoara@gmail.com

Shahin Kamali
University of Waterloo
s3kamali@uwaterloo.ca

Khuzaima Daudjee
University of Waterloo
kdaudjee@uwaterloo.ca

Lei Chen
HKUST
leichen@cse.ust.hk

TECHNICAL REPORT CS-2015-1
David R. Cheriton School of Computer Science
University of Waterloo, Cananda, 2015

ABSTRACT

Social networks are large graphs that require multiple graph database servers to store and manage them. Each database server hosts a graph partition with the objectives of balancing server loads, reducing remote traversals (edge-cuts), and adapting the partitioning to changes in the structure of the graph in the face of changing workloads. To achieve these objectives, a dynamic repartitioning algorithm is required to modify an existing partitioning to maintain good quality partitions while not imposing a significant overhead to the system. In this paper, we introduce a *lightweight repartitioner*, which dynamically modifies a partitioning using a small amount of resources. In contrast to the existing repartitioning algorithms, our lightweight repartitioner is efficient, making it suitable for use in a real system. We integrated our lightweight repartitioner into Hermes, which we designed as an extension of the open source Neo4j graph database system, to support workloads over partitioned graph data distributed over multiple servers. Using real-world social network data, we show that Hermes leverages the lightweight repartitioner to maintain high quality partitions and provides a 2 to 3 times performance improvement over the de-facto standard random hash-based partitioning.

1. INTRODUCTION

Large scale graphs, in particular social networks, permeate our lives. The scale of these networks, often in millions of vertices or more, means that it is often infeasible to store, query and manage them on a single graph database server. Thus, there is a need to partition, or shard, the graph across multiple database servers, allowing the load and concurrent processing to be distributed over these servers to provide good performance and increase availability. Social networks exhibit a high degree of correlation for accesses of certain groups of records, for example through frictionless sharing [21]. Also, these networks have a heavy-tailed distribution for popularity of vertices. The focus of this work is on partitioning of the graphs associated with social networks, though the provided solutions can be used to partition any graph. To achieve a good partitioning which improves the overall performance, the following objectives need to be met:

- The partitioning should be *balanced*. Each vertex of the graph has a *weight* that indicates the popularity of the vertex (e.g., in terms of the frequency of queries to that vertex). In social networks, a small number of users (e.g., celebrities, politicians) are extremely popular while a large number of users are much less popular. This discrepancy reveals the importance of achieving a balanced partitioning in which all partitions have almost equal *aggregate weight* defined as the total weight of vertices in the partition.

- The partitioning should minimize the number of *edge-cuts*. An edge-cut is defined by an edge connecting vertices in two different partitions and involves queries that need to transition from a partition on one server to a partition on another server. This results in shifting local traversal to remote traversal, thereby incurring significant network latency. In social networks, it is critical to minimize edge-cuts since most operations are done on the node that represents a user and its immediate neighbors. Since these *1-hop traversal* operations are so prevalent in these networks, minimizing edge-cuts is analogous to keeping communities intact. This leads to highly local queries similar to those in SPAR [34] and minimizes the network load, allowing for better scalability by reducing network IO.

- The partitioning should be *incremental*. Social networks are *dynamic* in the sense that users and their relations are always changing, e.g., a new user might be added, two users might get connected, or an ordinary user might become popular. Although the changes in the social graph can be much slower when compared to the read traffic [9], a good partitioning solution should dynamically adapt its partitioning to these changes. Considering the size of the graph, it is infeasible to create a partitioning from scratch; hence, a repartitioning solution, a *repartitioner*, is needed to improve on an existing partitioning. This usually involves *migrating* some vertices from one partition to another.

- The repartitioning algorithm should perform well in terms of time and memory requirements. To achieve this efficiency, it is desirable to perform repartitioning locally by accessing a small amount of information about the structure of the graph. From a practical point of view, this requirement is critical and prevents us from applying existing approaches, for the repartitioning problem.

There are several partitioning algorithms which result in relatively balanced solutions with small edge-cuts (e.g., [24,

A preliminary version of this paper appeared in [32].

38, 25, 38, 39, 7]). Some of these algorithms also tend to minimize vertex migration [39]. However, these algorithms require an *almost* complete or global view of the graph in their repartitioning step which makes them infeasible for incremental, online partitioning. As we will show, even in the distributed versions of these algorithms where each local site maintains its own local graph and summary graphs of other partitions, the size of the maintained information is proportional to the size of the graph. Furthermore, these approaches perform a large number of look-ups on the structure of the graph in the repartitioning phase. This significantly drops the performance of the system. Because of these issues, these algorithms are not used in real data management systems.

The focus of this paper is on the design and provision of a *practical* partitioned social graph data management system that can support remote traversals while providing an effective method to *dynamically* repartition the graph using only local views. The distributed partitioning aims to co-locate vertices of the graph *on-the-fly* so as to satisfy the above requirements. The fundamental contribution of this paper is a dynamic partitioning algorithm, referred to as *lightweight repartitioner*, that can identify which parts of graph data can benefit from co-location. The algorithm aims to incrementally improve an existing partitioning by decreasing edge-cuts while maintaining almost balanced partitions. The main advantage of the algorithm is that it relies on only a small amount of knowledge on the graph structure referred to as *auxiliary data.* Since the auxiliary data is small and easy to update, our repartitioning algorithm is performant in terms of time and memory while maintaining high-quality partitionings in terms of edge-cut and load balance.

We built Hermes as an extension of the Neo4j[1] open source graph database system by incorporating into it our algorithm to provide the functionality to move data on-the-fly to achieve data locality and reduce the cost of remote traversals for graph data. Our experimental evaluation of Hermes using real-world social network graphs shows that our techniques are effective in producing performance gains and work almost as well as the popular Metis partitioning algorithms [24, 38, 7] that performs static offline partitioning by relying on a global view of the graph.

The rest of the paper is structured as follows. Section 2 describes the problem addressed in the paper and reviews classical approaches and their shortcomings. Section 3 introduces and analyzes the lightweight repartitioner. Section 4 presents an overview of the Hermes system. Section 5 presents performance evaluation of the system. Section 6 covers related work, and Section 7 concludes the paper.

2. PROBLEM DEFINITION

In this section we formally define the partitioning problem and review some of the related results. In what follows, the term 'graph' refers to an undirected graph with weights on vertices. We are particularly interested in *scale-free graphs* whose vertex degree sequence follows *power-law* (i.e., the degree of vertices in the sorted sequence exponentially decreases). This informally means that most vertices have small degree while a few vertices have relatively high degree. Our interest in scale-free networks is rooted in the fact that social networks are conjectured to be scale-free. Regardless, the problem definition and algorithms in this paper work independently of this property.

2.1 Graph Partitioning

In the classical (α, γ)-graph partitioning problem [26], the goal is to partition a given graph into α vertex-disjoint subgraphs. The weight of a partition is the total weight of vertices in that partition. In a *valid solution*, the weight of each partition is at most a factor $\gamma \geq 1$ away from the average weight of partitions. More precisely, for a partition P of a graph G, we need to have $\omega(P) \leq \gamma \times \sum_{v \in V(G)} \omega(v)/\alpha$. Here, $\omega(P)$ and $\omega(v)$ denote the weight of a partition P and vertex v, respectively. Parameter γ is called the *imbalance load factor* and defines how imbalanced the partitions are allowed to be. Practically, γ is in range $[1, 2]$. Here, $\gamma = 1$ implies that partitions are required to be completely balanced (all have the same aggregate weights), while $\gamma = 2$ allows the weight of one partition to be up to twice the average weight of all partitions.[2] The goal of the minimization problem is to achieve a valid solution in which the number of edges between components (the number of edge-cuts) is minimized.

The partitioning problem is NP-hard even for the simple case of (2,1)-partitioning problem which is also referred to as the *bisection problem* [19]. Moreover, there is no approximation algorithm with a constant approximation ratio unless P=NP [8]. This reveals the hard nature of the problem. To facilitate the analysis for $\gamma > 1$, the performance of the algorithm is compared to an optimal solution in which partitions have equal size (i.e., the optimal algorithm is more restricted). However, even in this relaxed setting, the best existing approximation algorithm achieves a ratio of $O(\log^2 n)$ in general [8] and $O(\log n)$ when $v \geq 2$ [17]. Interestingly, the problem remains NP-hard (and even Approximate-hard) for simple graph families like trees and grids [16]. To conclude, from a theoretical point of view, it is not possible to introduce algorithms which provide worst-case guarantees on the quality of solutions, and it makes more sense to study the typical behavior of algorithms. Consequently, the problem is mostly approached through heuristics which are aimed to improve the average-case performance. One of the basic heuristics for the problem is that of Kernighan and Lin [26] that is used as a subroutine in many other heuristics. This algorithm initially partitions the graph into two arbitrary partitions. Then, it iteratively finds the best pair of vertices (two vertices in the two partitions) so that interchanging the vertices between the partitions results in the greatest decrease in the number of edge-cuts. If such a pair exists, the two vertices are swapped and the algorithm continues by repeating the entire process. If no exchange can decrease the edge-cut, then the algorithm terminates. If more than two partitions are required, the algorithm is recursively called on the resulting partitions. Each iteration of the algorithm runs in $O(m \log m)$ where m is the number of edges in the graph; this can be improved to $O(m)$ using an appropriate data structure like that of Fiduccia and Mattheyses [18]. Regardless, the time complexity of these heuristics $\Omega(n^3)$ for graphs of size n which makes them un-

[1]Neo4j is being used by customers such as Adobe and HP . A full list of customers can be found in [3].

[2]Unbounded values for γ relate the problem to the min-cut problem which is not the focus of this paper.

suitable in practice.

To improve the time complexity, a class of *multi-level* algorithms were introduced. In each *level* of these algorithms, the input graph is *coarsened* to a representative graph of smaller size; when the representative graph is small enough, a partitioning algorithm like that of Kernighan-Lin [26] is applied to it, and the resulting partitions are mapped back (uncoarsened) to the original graph. Many algorithms fit in this general framework of multi-level algorithms; a widely used example is the family of Metis algorithms [25, 38, 7]. The multi-level algorithms are *global* in the sense that they need to know the whole structure of the graph in the coarsening phase, and the coarsened graph in each stage should be stored for the uncoarsening stage. This problem is partially solved by introducing distributed versions of these algorithms in which the partitioning algorithm is performed in parallel for each partition [4]. In these algorithms, in addition to the local information (structure of the partition), for each vertex, the list of the adjacent vertices in other partitions is required in the coarsening phase. In what follows, we show that in the worst case, acquiring this amount of data is close to having a global knowledge of graph.

THEOREM 1. *Consider the (α, γ)-graph partitioning problem where $\gamma < 2$. There are instances of the problem for which the number of edge-cuts in any valid solution is asymptotically equal to the number of edges in the input graph.*

PROOF. Consider an input graph with n vertices out of which α vertices are designated as 'core' vertices. Assume any cover vertex is connected to all $n - \alpha$ non-core vertices and no other core vertex. Further, assume the weight of the core vertices is $\omega = \gamma/(2-\gamma) \times n/\alpha + 1$ while the weight of other vertices is 1. So, the average weight of a partition is $\omega + n/\alpha - 1$. We claim that in a valid solution, each partition should include exactly one hob. Consider otherwise when there is a partition with two hobs. The weight of such a partition will be at least $2\omega + 1$. For the imbalance factor of the partitioning the following holds:

$$\text{imbalance factor} \geq \frac{2\omega + 1}{\omega + n/\alpha - 1} > \frac{2\gamma/(2-\gamma) \times n/\alpha}{\gamma/(2-\gamma) \times n/\alpha + n/\alpha} = \frac{2\gamma/(2-\gamma)}{2/(2-\gamma)} = \gamma$$

Consequently, the imbalance factor is more than γ and the partitioning with two hobs is invalid. So, in any valid partitioning, there is exactly one hob in each partition. Let $x_1, \ldots, x_\alpha$ denote the number of ordinary vertices in the α partitions; so $x_1 + \ldots + x_\alpha = n - \alpha$. The number of edge-cuts for partitions will be $(n-\alpha-x_1), \ldots, (n-\alpha-x_\alpha)$, and the total number of edge-cuts is $\alpha(n-\alpha) - (n-\alpha) = (\alpha-1)(n-\alpha)$. Note that the total number of edges in the graph is $\alpha(n-\alpha)$. Consequently, assuming that there is a constant number of partitions, the number of edge-cuts is asymptotically equal to the number of edges in any valid solution. □

The above theorem implies that the average amount of data required in the coarsening phase of multi-level algorithms can be a constant fraction of all edges. The graphs used in the proof of the above theorem belong to the family of power-law graphs which are often used to model social networks. Consequently, even the distributed versions of multi-level algorithms in the worst case require *almost*

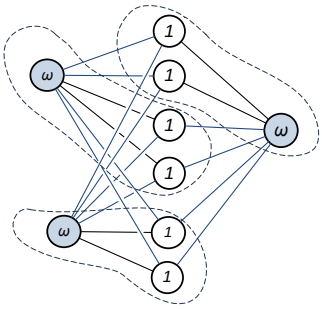


Figure 1: An instance of the (k, v) partitioning problem for which the number of edge-cuts in any valid solution is a constant fraction of all edges. The blue vertices denote core vertices; here, we have $k = 3$ and the weight of the hobs are $w = v/(2-v) \times n/k + 1$.

global information on the structure of the graph (particularly when used for partitioning social networks). This reveals the importance of providing practical partitioning algorithms which need only a small amount of knowledge about the structure of the graph that can be easily maintained in memory. The lightweight repartitioner introduced in this paper has this property, i.e., it maintains only a small amount of data, referred to as auxiliary data, to perform repartitioning. This enables us to implement it as a practical solution for the partitioning problem which can be used to enhance existing graph database systems.

2.2 Repartitioning

Social graphs are evolving structures that change over time due to user interaction. The most common changes are the addition of relationships between different users [9] and changes in user popularity. These changes will create long lived skews on some partitions, which reduce the performance of the system. In [27], it is shown that traffic patterns are generally well defined. Popular users generally have more relationships [27, 14]. Celebrities, companies or bloggers will generate and consume more information. As such users come online and form relationships their popularity will generate different traffic patterns. However, the change in the social graph can be much slower when compared to the read traffic [9]. This process leads to a slowly, but constantly evolving graph structure.

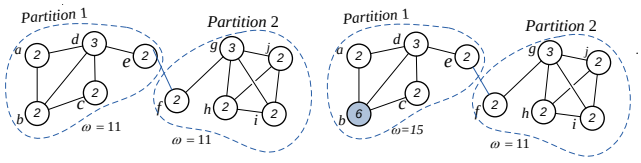
A variety of partitioning methods can be used to create an initial, *static*, partitioning.This should be followed by a *repartitioning* strategy to maintain good partitioning that can adapt to changes in the graph. One solution is to periodically run an algorithm on the whole graph to get new partitions. However, running an algorithm to get new partitions from scratch is costly in terms of time and space. Hence, an incremental partitioning algorithm needs to adapt the existing partitions to changes in the graph structure.

It is desirable to have a lightweight repartitioner that maintains only a small amount of auxiliary data to perform repartitioning. Since such algorithm refers only to this auxiliary data, which is significantly smaller than the actual data required for storing the graph, the repartitioning algorithm is not a system performance bottleneck. The auxiliary data maintained at each machine (partition) consists of the

list of accumulated weight of vertices in each partition, as well as the *number* of neighbors of each hosted vertex in each partition. In particular, if a partition hosts n_1 vertices, it maintains $\alpha + n_1\alpha$ numbers (integers) as auxiliary data which is easy to maintain and update. Note that maintaining the number of neighbors is far cheaper that maintaining the *list* of neighbors in other partitions. In what follows, the main ideas behind our lightweight repartitioner are introduced through an example.

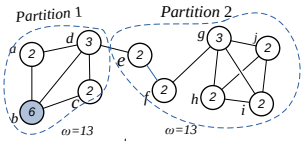
Example: Consider the partitioning problem on the graph shown in Figure 2. Assume there are $\alpha = 2$ partitions in the system and the imbalance factor is $\gamma = 1.1$, i.e., in a valid solution, the aggregate weight of a partition is at most 1.1 times more than the average weight of partitions. Assume the numbers on vertices denote their weight. During normal operation in social networks, users will request different pieces of information. The most common operations are 1-hop traversals (see what your friends are up to) or single get requests (check a popular user such as a celebrity or news source). In this sense, the weight of a vertex is the number of read requests to that vertex. Figure 2a shows a partitioning of the graph into two partitions, where there is only one edge-cut and the partitions are well balanced, i.e., the weight of both partitions is equal to the average weight, i.e., 11. Assuming user b is a popular weblogger who posts a post, the request traffic for vertex b will increase as its neighbors poll for updates, leading to an imbalance in load on the first partition. Figure 2b shows the state of the graph after user b becomes popular and skews the aggregate weight to 15 on partition 1. Such skews will degrade performance by increasing the response time of queries and lowering query throughput on a skewed partition. Here, the ratio between aggregate weight of partition 1 (i.e., 15) and the average weight of partitions (i.e., 13) is more than γ. This means that the repartitioning needs to be triggered to rebalance the load across partitions (while keeping the number of edge-cuts as small as possible).

The auxiliary data of the lightweight repartitioner available to each partition includes the weight of each of the two partitions, as well as the number of neighbors of each vertex v hosted in the partition, e.g., auxiliary data available to partition 1 in Figure 2b implies that vertex e is connected to one vertex in each partition. Provided with this auxiliary data, a partition can determine whether load imbalances exist and the extent of the imbalance in the system (to compare it with γ). If there is a load imbalance, a repartitioner needs to indicate where to migrate data to restore load balance. Migration is an iterative process which will identify vertices that when moved will balance loads (aggregate weights) while keeping the number of edge-cuts as small as possible. In doing so, our lightweight repartitioner makes use of its auxiliary data which includes the number of neighbors of each vertex in each partition. For example, when the repartitioner starts from the state in Figure 2b, on partition 1, vertices a through d are poor candidates for migration because their neighbors are in the same partition. Vertex e, however, has a split access pattern between partitions 1 and 2. Since vertex e has the fewest neighbors in partition one, it will be migrated to partition 2. On partition 2, the same process is performed in parallel; however, vertex f will not be migrated since partition 1 has a higher aggregate weight. Once vertex e is migrated, the load (aggregate weights) becomes balanced, thus any remaining iterations will not result in any migrations. The resulting graph is depicted in Figure 2c.



(a) Balanced partitioned graph

(b) Skewed graph



(c) Repartitioned graph

Figure 2: Graph evolution and effects of repartitioning in response to imbalances.

The above example is a simple case to illustrate how the lightweight repartitioner works. Several issues are left out of the example, e.g., two highly connected clusters of vertices may repeatedly exchange their clusters to decrease edge-cut. This results in an *oscillation* which is discussed in detail in Section 3.

3. PARTITIONING ALGORITHM

Unlike Neo4j which is centralized, Hermes can apply hash-based or Metis algorithm to partition a graph and distribute the partitions to multiple servers. Thus, the system starts with an initial partitioning and incrementally applies the lightweight repartitioner to maintain partitioning with good performance in the dynamic environment. In this section, we introduce the lightweight repartitioner algorithm behind Hermes. Embedding the initial partitioning algorithm and the lightweight repartitioner into Neo4j required modification of Neo4j components.

To increase query locality and decrease query response times, the initial partitioning needs to be optimized in terms of having almost balanced distributions (valid solutions) with small number of edge-cuts. We use Metis to obtain the initial data partitioning, which is a static, offline, process that is orthogonal to the dynamic, on-the-fly, partitioning that Hermes performs. For the initial vertex weights, a simple representative traffic (query workload) trace is generated and used to initialize them. The trace consists of 1-hop read query traversals where the starting vertex for each traversal is randomly selected from the set of all vertices.

3.1 Lightweight Repartitioner

When new nodes join the network or the traffic patterns (weights) of nodes change, the lightweight repartitioner is triggered to rebalance vertex weights while decreasing edge-cut through an iterative process. The algorithm makes use of aggregate vertex weight information as its auxiliary data. Assuming there are α partitions, for each vertex v, the auxil-

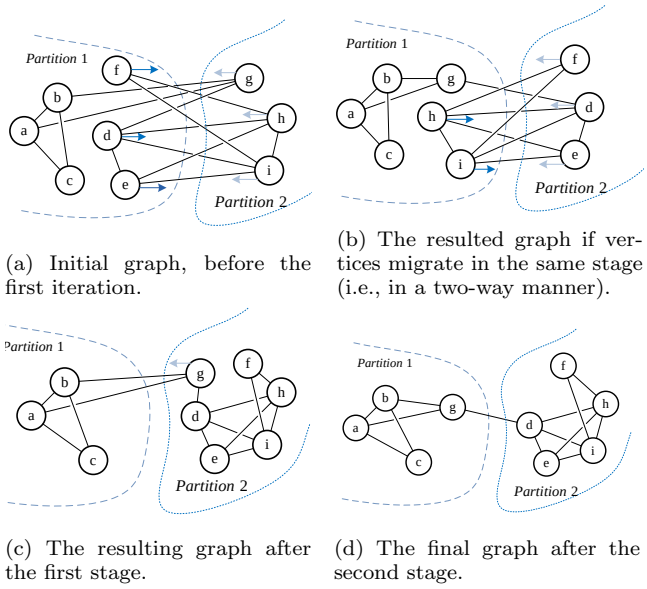
iary data includes α integers indicating the number of neighbors of v in each of the α partitions. This auxiliary data is insignificant compared to the *physical data* associated with the vertex which include adjacency list and other information referred to as *properties* of the vertex (e.g., pictures posted by a user in a social network). The repartitioning auxiliary data is collected and updated based on execution of user requests, e.g., when a new edge is added, the auxiliary data of the partitioning(s) including the endpoints of the edge get updated (two integers are incremented). Hence, the cost involved in maintenance of auxiliary data is proportional to the rate of changes in the graph. As mentioned earlier, social networks change quite slowly (when compared to the read traffic); hence, the maintenance of auxiliary data is not a system bottleneck. Each partition collects and stores aggregate vertex information relevant to only the local vertices. Moreover, the auxiliary data includes the total weight of all partitions, i.e., in doing repartitioning, each server knows the total weight of all other partitions.

The repartitioning process has two *phases*. In each iteration of the first phase, each server runs the repartitioner algorithm using the auxiliary data to indicate some vertices in its partition that should be migrated to other partitions. Before the next iteration, these vertices are *logically* moved to their target partitions. Logical movement of a vertex means that only the auxiliary data associated with the vertex is sent to the other partition. This process continues up to a point (iteration) in which no further vertices are chosen for migration. At this point the second phase is performed in which the physical data is moved based on the result of first phase. The algorithm is split into two phases because border vertices are likely to change partitions more than once (this will be discussed later) and auxiliary data records are lightweight compared to the physical data records, allowing the algorithm to finish faster. In what follows, we describe how vertices are selected for migration in an iteration of the repartitioner.

Consider a partition P_s (source partition) is running the repartitioner algorithm. Let v be a vertex in partition P_s. The *gain* of moving v from P_s to another partition P_t (target partition) is defined as the difference between the number of neighbors of v in P_t and P_s, respectively, i.e., $gain(v) = d_v(t) - d_v(s)$ ($d_v(k)$ denotes the number of neighbors of v in partition k). Intuitively, the gain represents the decrease of the number of edge-cuts when migrating v from P_s to P_t (assuming that no other vertex migrates). Note that the gain can be negative, meaning that it is better, in terms of edge-cuts, to keep v in P_s rather than moving it to P_t. In each iteration and on each partition, the repartitioner selects for migration *candidate* vertices that will give the maximum gain when moved from the partition. However, to avoid *oscillation* and ensure a valid packing in term of load balance, the algorithm enforces a set of rules in migrating vertices. First, it defines two *stages* in each iteration. In the first stage, the migration of vertices is allowed only from partitions with lower ID to higher ID, while the second stage allows the migration only in the opposite direction, i.e., from partitions with higher ID to those with lower ID. Here, partition ID defines a fixed ordering of partitions (and can be replaced by any other fixed ordering). Migrating vertices in one-direction in two stages prevent the algorithm from oscillation. Oscillation happens when there is a large number of edges between two group of vertices hosted in two different partitions (see Figure 3). If the algorithm allows two-way migration of vertices, the vertices in each group migrate to the partition of the other group, while the edge-cut does not improve (Figure 3b). In one-way migration, however, the vertices in one group remain in their partitions while the other group joins them in that partition (Figure 3d).

In addition to preventing oscillation, the repartitioner algorithm minimizes load imbalance as follows. A vertex v on a partition P_s is a candidate for migration to partition P_t if the following conditions hold:

- P_s and P_t fulfill the above one-way migration rule.
- Moving v from P_s to P_t does not cause P_t to be *overloaded* nor P_s to be *underloaded*. Recall from Section 2.1 that the imbalance ratio of a partition is the ratio between the weight of the partition (the total weight of vertices it is hosting) and the average weight of all the partitions. A partition is overloaded if its imbalance load is more than γ and underloaded if its weight is less than $2 - \gamma$ times the average partition weight. Here, γ is the maximum allowed imbalance factor ($1 < \gamma < 2$); the default value of γ in Hermes is set to be 1.1, i.e., a partition's load is required to be in range $(0.9, 1.1)$ of the average partition weight. This is so that imbalances do not get too high before repartitioning triggers.
- Either P_s is overloaded *OR* there is a positive *gain* in moving v from P_s to P_t. When a partition is overloaded, it



(a) Initial graph, before the first iteration.

(b) The resulted graph if vertices migrate in the same stage (i.e., in a two-way manner).

(c) The resulting graph after the first stage.

(d) The final graph after the second stage.

Figure 3: An unsupervised repartitioning might result in oscillation. Consider the partitioning depicted in (a). The repartitioner on partition 1 detects that migrating d, e, f to partition 2 improves edge-cut; similarly, the repartitioner on partition 2 tends to migrate g, h, i to partition 1. When the vertices move accordingly, as depicted in (b), the edge-cut does not improve and the repartitioner needs to move d, e, f and h, i again. To resolve this issue, in the first stage of repartitioning of (a), the vertices d, e, f are migrated from partition 1 (lower ID) to partition 2 (higher ID). After this, as depicted in (c), the only vertex to migrate in the second stage is vertex g which moves from partition 2 (higher ID) to migration 1 (d).

Algorithm 1 Choosing target partition for migration

```
1: procedure GET_TARGET_PART(vertex v currently hosted in partition P_s, the current stage of the iteration.)
2:     if imbalance_factor(P_s − {v}) < 2 − γ then
3:         return (null, 0)
4:     target = null; maxGain = 0;
5:     if imbalance_factor(P_s) > γ then
6:         maxGain = −∞
7:     for partition P_t ∈ partitionSet do
8:         if (stage = 1 and P_t.ID > P_s.ID) or
9:            (stage = 2 and P_t.ID < P_s.ID) then
10:            gain ← Gain(v, P_s, P_t)
11:            if imbalance_factor(P_t ∪ {v}) < γ and
12:               gain > maxGain then
13:                target ← P_t; maxGain = gain
14:    return (target, maxGain)
```

Algorithm 2 Lightweight Repartitioner

```
1: procedure REPARTITIONING_ITERATION(partition P_s)
2:     for stage ∈ {1, 2} do
3:         candidates ← {}
4:         for Vertex v ∈ VertexSet(P_s) do
5:             target(v) ← GET_TARGET_PART(v,stage)
6:                         ▷ setting target(v) and gain(v)
7:             if target(v) ≠ null then
8:                 candidates.add (v)
9:         top-k ← k candidates with maximum gains
10:        for Vertex v ∈ top-k do
11:            MIGRATE(v, P_S, target(v))
12:        P_s.update_auxiliary data
```

is good to consider all vertices as candidates for migration to any other partition as long as they do not cause an overload on the target partition. When the partition is not overloaded, it is good to move only vertices which have positive weight so as to improve the edge-cut.

When a vertex v is a candidate for migration to more than one partition, the partition with maximum gain is selected as the target partition of the vertex. This is illustrated in Algorithm 1. Note that detecting whether a vertex v is a candidate for migration and selecting its target partition is performed using only the auxiliary data. Precisely, for detecting underloaded and overloaded partitions (Lines 2, 5 and 11), the algorithm uses the weight of the vertex and the accumulated weights of all partitions; these are included in the auxiliary data. Similarly, for calculating the gain of moving v from partition P_s to partition P_t (Line 10), it uses the number of neighbors of v in any of the partitions, which is also included in the auxiliary data.

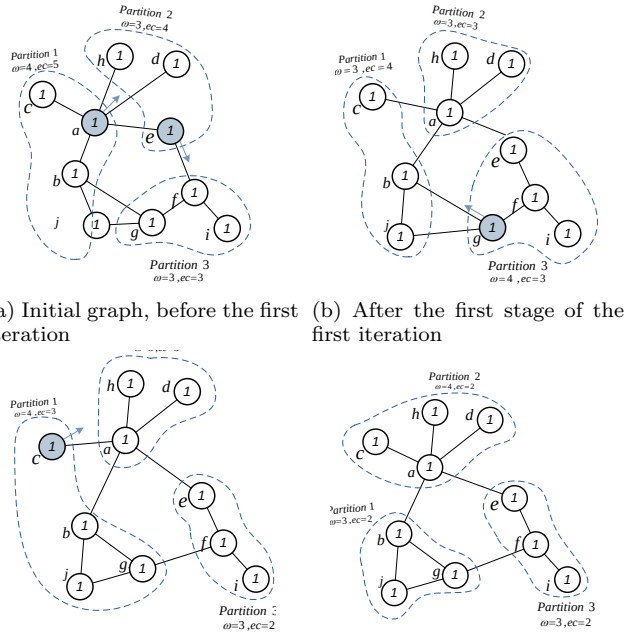
Recall that the repartitioning algorithm runs on each partition independently, a property that supports scalability. For each partition P_s, after selecting the candidate vertices for migration and their target partitions, the algorithm selects k candidate vertices which have the highest gains among all vertices and proceeds by (logically) migrating these *top-k* vertices to their target partitions. Here, migrating a vertex means sending (and updating) the auxiliary data associated with the vertex to its target destination and updating the auxiliary data associated with partition weights accordingly. The algorithm restricts the number of migrated vertices in each iteration (to k) to avoid imbalanced partitionings. Note that when selecting the target partition for a migrating vertex, the algorithm does not know the target partition of other vertices; hence, there is a chance that a large number of vertices migrate to the same partition to improve edge-cut. Selecting only k vertices enables the algorithm to control the accumulative weight of partitions by restricting the number of migrating vertices. We discuss later how the value of k is selected. In general, taking k as a small, fixed fraction of n (size of the graph) gives satisfactory results.

Algorithm 2 shows the details of one iteration of the repartitioner algorithm performed on a partition P_s. The algorithm detects the candidate vertices (Lines 4-8), selects the *top-k* candidates (Line 9), and moves them to their respective target partitions. Note that the migration in Line 11 is logical in the sense that only the auxiliary data associated with vertices is migrated. After each phase of each iteration, the auxiliary data associated with each migrated vertex v is updated. This is because the neighbors of v may also be migrated, which would mean that the degree of v in each partition, i.e., auxiliary data associated with v, has changed. The algorithm continues moving vertices until there is no candidate vertex for migration, i.e., further movement of vertices does not improve edge-cut.

Example: To demonstrate the workings of the lightweight repartitioner, we show two iterations of the repartitioning algorithm on the graph of Figure 4 in which there are $\alpha = 3$ partitions and the average weight of partitions is $10/3$. Assume the value of γ is $1.\bar{3}$. Hence, the aggregate weight of a partition needs to be in range $[2.\bar{2}, 4.\bar{4}]$; otherwise the partitioning is overloaded or underloaded. Figure 4a shows the initial state of the graph. The partitions are sub-optimal as 6 of the 11 edges shown are edge-cuts. Consider the first stage of the first iteration of the lightweight repartitioner. Since the first stage restricts vertex migrations from lower ID partitions to higher ID only, vertices *a* and *e* are the migration candidates since they are the only ones that can improve edge-cut. Note that if the algorithm was performed in one stage, vertices *h* and *d* would be migrated to partition 1 causing the oscillating behavior discussed previously. At the end of the first stage of the first iteration, the state of the graph is as presented in Figure 4b. In the second stage, the algorithm migrates only vertex *g*. While vertex *c* could be migrated to improve edge-cut, the migration direction does not allow this (Figure 4c). In addition, such migration would cause partition 1 to be underloaded (its load will be 2 which is less than $2.\bar{2}$). In the second iteration, vertex *c* is migrated to partition 2. The result of the first stage of iteration 2 is presented in Figure 4d. At this point, the graph reaches an optimal grouping, thus the second stage of the second iteration will not perform any migrations. In fact further iterations would not migrate anything since the graph has an optimal partitioning.

3.2 Physical Data Migration

Physical data migration is the final step of the repartitioner. Vertices and relationships that were marked for mi-



(a) Initial graph, before the first iteration

(b) After the first stage of the first iteration

(c) After the second stage of the first iteration

(d) After the first stage of the second iteration

Figure 4: Two iterations of the lightweight repartitioner. Two metrics are attached to every partition: ω representing the weight of the partition and *ec* representing the edge-cut.

gration by the repartitioner are moved to the target partitions using a two step process: (1) Copy marked vertices and relationships (copy step) (2) Remove marked vertices and relationships from the host partitions (remove step).

In the first step, a list of all vertices selected for migration to a partition are received by that partition, which will request these vertices and add them to its own local database. At the end of the first step, all moved vertices are replicated. Because of the insertion-only operations, the complexity of the operations is lower as all operations can be performed locally in each partition, meaning less network contention and locks held for shorter periods.

Between the two steps there is a synchronization process between all partitions to ensure that partitions have completed the copy process before removing marked vertices from their original partitions. This is required since partitions may request removal of data that is still used by other partitions in the copy step. The synchronization itself is not expensive as no locks or system resources are held, though partitions may need to wait until an occasional straggler finishes copying. The remove step takes advantage of replication to decrease the number of operations and their impact on the system. In this step, all marked vertices will enter an unavailable state in which all queries referencing the vertex will be executed as if the vertex is not part of the local vertex set. This allows performing the transactional operations much faster as locks on unavailable vertices cannot be acquired by any standard queries. In addition, since this operation is performed in a batch like process, it is easy to detect operations that can be collapsed together.

3.3 Lightweight Repartitioner Analysis

3.3.1 Memory and Time Analysis

Recall that the main advantage of the lightweight repartitioner over multilevel algorithms is that it makes use of only auxiliary data to perform repartitioning. Auxiliary data has a small size compared to the size of the graph. This is formalized in the following two theorems:

THEOREM 2. *The amortized size of auxiliary data stored on each partition to perform repartitioning is $n + \Theta(\alpha)$ on average. Here, n denotes the number of vertices in the input graph and α is the number of partitions.*

PROOF. Assume there are α partitions. The auxiliary data for a vertex v includes the number of neighbors of v in each of these partitions. In total, there will be α integers for each vertex. The amortized number of vertices in each partition is n/α; hence, the amortized size of auxiliary data associated with vertices in each partition is n. Beside the number of neighbors in each partition for each vertex, the auxiliary data includes aggregated weight of all partitions. The aggregate weight of each partition can be stored in a constant number of integers which sums to $\Theta(\alpha)$ for all partitions. □

As mentioned in Section 2, multi-level algorithms do repartitioning by looking at adjacency lists of vertices, which might be a large fraction of the number of *edges*. In contrast, an implication of the above theorem is that the size of auxiliary data used by lightweight repartitioner is roughly equal to the number of *vertices*. In social networks, the number of edges is significantly more than the number of vertices, e.g., the average friend count in Facebook is around 130 [11] which implies that the number of edges is roughly 65 times the number of vertices. Hence, the memory requirement of the lightweight repartitioner is far less than that of multilevel algorithms and can be easily maintained in memory without any impact on performance of the system. This is experimentally verified in Section 5.3.

THEOREM 3. *Each iteration of the repartitioning algorithm takes $O(\alpha n_s)$ time to complete. Here, α denotes the number of partitions and n_s is the number of vertices in the partition which runs the repartitioning algorithm.*

PROOF. Let P_s denote the partition on which the repartitioning algorithm runs. Assume there are α partitions in the system. In each iteration, detecting whether a vertex v is a candidate for migration can be done in $\Theta(\alpha)$ time. Namely, for each target partition P_t, the algorithm determines, in constant time, whether moving v causes an overload in P_t and if not, what is the gain in moving v from P_s to P_t; this is done by comparing the number of neighbors of v in both partitions. Comparing partition IDs of P_s and P_t also takes constant time. In total, the candidate vertices can be selected in time $O(\alpha n_s)$. Selecting the *top-k* candidates also takes linear time through a selection algorithm followed by a linear scan of the candidates. Consequently, the running time of each run of the algorithm is $O(\alpha n_s)$. □

In practice, the number of partitions is constant when compared to the number of vertices in the graph (α is a constant). Hence, the above theorem implies that each iteration of the algorithm runs in linear time. Moreover, the

algorithm converges to a stable partitioning after a small number of iterations relative to the number of nodes, e.g., in our experiments, it converges after less than 50 iterations, while there are millions of vertices in the graph data sets. To conclude, the time complexity of the lightweight repartitioner is linear to the number of hosted vertices; this makes the algorithm much faster than its multilevel counterparts.

The lightweight repartitioner is designed for scalability and with little overhead to the database engine. The simplicity of the algorithm supports parallelization of operations and maximizes scalability. In the first phase, each iteration is performed in parallel on each server. The auxiliary data information is fully local to each server, thus lines 4 through 9 of Algorithm 2 are executed independently on each server. In the second phase of the repartitioning algorithm, physical data migration is performed. As mentioned in Section 2.2, this part has been decomposed into two steps for simplicity and performance. Because information is only copied in the first step (in which vertices are replicated), it allows for maximum parallelization with little need to synchronize between servers.

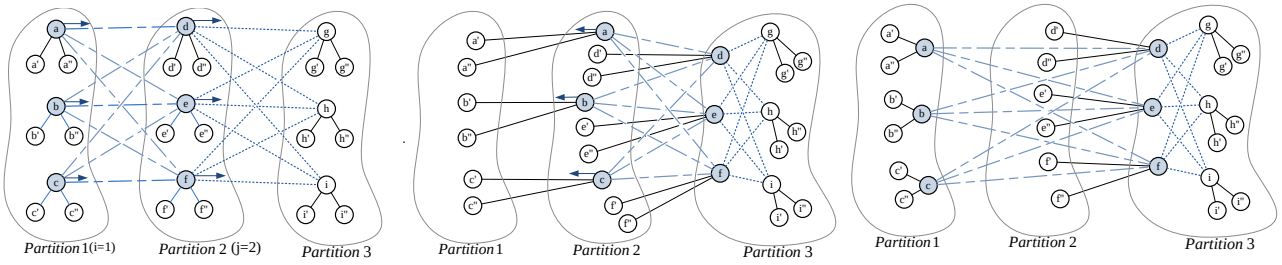
3.3.2 Algorithm Convergence

When the lightweight repartitioner triggers, the algorithm starts by migrating vertices from overloaded partitions. Note that no vertex is a candidate for migration to an overloaded partition. Hence, after a bounded number of iterations, the partitioning becomes valid in term of load balance. When there is no overloaded partition, the algorithm moves a vertex only if there is a positive gain in moving it from the source to the target partition. This is the main idea behind the following proof for the convergence of the algorithm.

THEOREM 4. *After a bounded number of iterations, the lightweight repartitioner algorithm converges to a stable partitioning in which further migration of vertices (as done by the algorithm) does not result in better partitionings.*

PROOF. We show that the algorithm constantly decreases the number of edge-cuts. For each vertex v, let $d_{ex}(v)$ denote the number of external neighbors of v, i.e., number of neighbors of v in partitions other than that of v. With this definition, the number of edge-cuts in a partition is $\chi/2$ where $\chi = \sum_{v=1}^{n} d_{ex}(v)$. Recall that the algorithm works in stages so that if in a stage migration of vertices is allowed from one partition to another, in the subsequent stage the migration is allowed in the opposite direction. We show that the value of χ decreases in every two subsequent stages; more precisely, we show that when a vertex v migrates in a stage t, the value of $d_{ex}(v)$ either decreases at the end of the stage t or at the end of the subsequent stage $t+1$ (compared to when v does not migrate). Let $d_k^t(v)$ denote the number of neighbors of vertex v in partition k before stage t. Assume that vertex v is migrated from partition i to partition j at stage t (see Figure 5). This implies that the number of neighbors of v in partition j is more than partition i. Hence, when v moves to partition j, the value of $d_{ex}(v)$ is expected to decrease. However, in a worst-case scenario, some neighbors of v in partition j also move to other partitions at the same stage (Figure 5b). Let $x(v)$ denote the number of neighbors of v in the target partition j which migrate at stage t; hence, at the end of the stage, the value of $d_{ex}(v)$ decreases by at least $d_j^t(v) - x(v)$ units. Moreover, $d_{ex}(v)$ is increased by at most $d_i^t(v)$; this is because the previous internal neighbors (those which remain at partition i) will become external after the migration of v. If $d_j^t(v) - x(v) > d_i^t(v)$, the value of $d_{ex}(v)$ decreases at the end of the stage and we are done. Otherwise, we say a *bad migration* occurred. In these cases, assuming k is sufficiently large, in the subsequent stage $t+1$, v migrates back to partition i since there is a positive gain in such a migration (Figure 5c), and this results in a decrease of $d_i^{t+2}(v)$ and an increase of at most $d_j^t(v) - x(v)$ in $d_{ex}(v)$. Consequently, the net increase in d_{ex} after two stages is $(d_i^t(v) - (d_j^t(v) - x(v))) + (d_j^t(v) - x(v) - d_i^{t+2}(v)) = d_i^t(v) - d_i^{t+2}(v)$. Note that if v does not move at all, d_{ex} increases $d_i^t(v) - d_i^{t+2}(v)$ units after two stages. Hence, in the worst case, the net decrease in $d_{ex}(v)$ is at least 0 for all migrated vertices (compared to when they do not move). Indeed, we show that there are vertices for which the decrease in d_{ex} is strictly more than 0 after two consecutive stages. Assuming there are α partitions, these are the vertices which migrate to partition α [in stages where vertices move from lower ID to higher ID partitions] or partition 1 [in stages where vertices move from higher ID to lower ID partitions]. In these cases, no vertex can move from the target partition to another partition; so the actual decrease in $d_{ex}(v)$ is the same as the calculated gain when moving the vertex and is more than 0. To summarize, for all vertices, the value of $d_{ex}(v)$ does not increase after every two stages, and for some vertices, it decreases. For smaller values of k, after a bad migration, vertex v might not return from partition j to its initial partitioning i in the subsequent stage (since there might be more gain in moving other vertices); however, since there is a positive gain in moving v back to partition i, in subsequent stages, the algorithm moves v from partition j to another partition (i or another partition which results in more gain). The only exception is when many neighbors of v move to partition j so that there is no positive gain in moving v. In both cases, the value of $d_{ex}(v)$ decreases with the same argument as above. To conclude, as the algorithm runs, the accumulated values of $d_{ex}(v)$ (i.e., χ), and consequently the number of edge-cuts, constantly decrease. □

The graph structure in social networks does not evolve quickly and its evolution is towards community formation. Hence, as our experiments confirm, after a small number of iterations, the lightweight repartitioner converges to a stable partitioning. The speed of convergence depends on the value of k (the number of migrated vertices from a partition in each iteration). Larger values of k result in faster improvement on the number of edge-cuts and subsequently achieve partitioning with almost an optimal number of edge-cuts. However, as mentioned earlier, large values of k can degrade the balance factor of partitioning. Finding the right of value of k requires considering a few parameters which include the number of partitions, the structure of the graph (e.g., the average size of the clusters formed by vertices), and the nature of changing workload (whether the changes are mostly on the weight or on the degree of vertices). In practice, we observed that a sub-optimal value of k does not degrade convergence rate by more than a few iterations; consequently the algorithm does not require fine tuning for finding the best value of k. In our experiments, we set k as a small fraction of the number of vertices.



(a) Original graph (b) After the first stage (c) After the second stage

Figure 5: The number of edge-cuts might increase in the first stage (in the worst case), but it decreases after the second stage. In this example, the number of edge-cuts is initially 18 (a); this increases to 21 after the first stage (b), and decreases to 15 at the end of the second stage (c).

4. HERMES SYSTEM OVERVIEW

In this section, we provide an overview of Hermes, which we designed as an extension of Neo4j Version 1.7.3 to handle distribution of graph data and dynamic repartitioning. Neo4j is an open source centralized graph database system which provides a disk-based, transactional persistence engine (ACID compliant). The main querying interface to Neo4j is traversal based. Traversals use the graph structure and relationships between records to answer user queries.

To enable distribution, changes to several components of Neo4j were required as well as addition of new functionality. The modifications and extensions were done such that existing Neo4j features are preserved. Figure 6 shows the components of Hermes with the components of Neo4j that were modified to enable distribution in light blue shading while the components in dark blue shading are newly added. Detailed descriptions of the remaining changes are omitted as they pose technical challenges which were overcome using existing techniques. For example, as the centralized loop detection algorithm used by Neo4j for deadlock detection does not scale well, it was replaced using a timeout-based detection scheme as described in [12].

Internally, Neo4j stores information in three main stores: node store, relationship store and property store. Splitting data into three stores allows Neo4j to keep only basic information on nodes and relationships in the first two stores. Further, this allows Neo4j to have fixed size node and relationship records. Neo4j combines this feature with a monotonically increasing ID generator such that a) record offsets are computed in O(1) time using their ID and b) contiguous ID allocation allows records to be as tightly packed as possible. The property store allows for dynamic length records. To store the offsets, Neo4j uses a two layer architecture where a fixed size record store is used to store the offsets and a dynamic size record store is used to hold the properties. To shard data across multiple instances of Hermes, changes were made to allow local nodes and relationships to connect with remote ones. Hermes uses a doubly-linked list record model when keeping track of relationships. Such a node in the graph needs to know only the first relationship in the list since the rest can be retrieved by following the links from the first. Due to tight coupling between relationship records, referencing a remote node means that each partition would need to hold a copy of the relationship. Since replicating and maintaining all information related to a relationship would incur significant overhead, the relationship in one partition has a ghost flag attached to it to connect it with its remote counterpart. Relationships tagged by the ghost flag do not hold any information related to the properties of the relationship but are maintained to there to keep the graph structure valid. One advantage of this is the complete locality in finding the adjacency list of a graph node. This is important since traversal operations build on top of adjacency list.

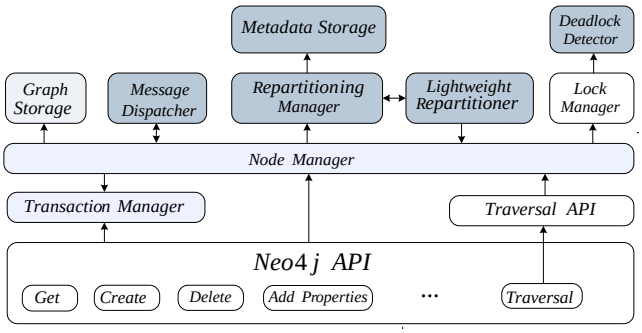


Figure 6: Hermes system layers together with modified and new components designed to make it run in a distributed environment.

The storage was also modified to use a tree-based indexing scheme (B+Tree) rather than an offset-based indexing scheme since record IDs can no longer be allocated in small increments. In addition, data migration would make offset based indexing impossible as records would need to be both compacted and still keep an offset based on their ID.

In Hermes, servers are connected in a peer-to-peer fashion similar to the one presented in Figure 7. A client can connect to any server and perform a query. Generally, user queries are in the form of a traversal. To submit a query the client would first lookup the vertex for the starting point of the query, then send the traversal query to the server hosting the initial vertex. The query is forwarded to the server containing the vertex such that data locality is maximized. On the server side, the traversal query will be processed by traversing the vertex's relationships. If the information is not local to the server, remote traversals are executed using the links between servers. When the traversal completes, the query results will be returned to the client.

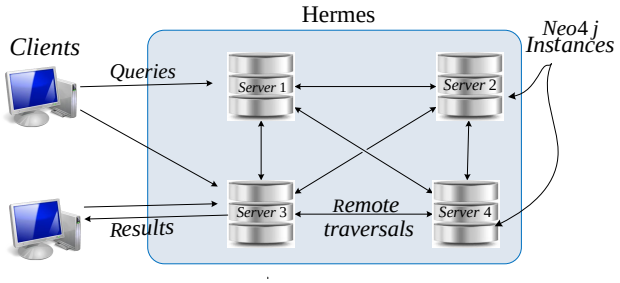


Figure 7: Overview of how Hermes servers interact with clients and with each other.

5. PERFORMANCE EVALUATION

In this section, we present the evaluation of the lightweight repartitioner implemented into Hermes.

5.1 Experimental Setup

All experiments were executed on a cluster with 16 server machines. Each server has the following hardware configuration: 2 AMD Opteron 252 (2 cores), 8 GB RAM and 160GB SATA HDD. The servers are connected using 1Gb ethernet. In each experiment, one Hermes instance runs on its own server.

The experiments are focused on typical social network traffic patterns, which based on previous work [9, 27], are 1-hop traversals and single record queries. We also consider 2-hop queries which are used for analytical queries such as ads and recommendations. Given the small diameters of social graphs (Table 1), queries with more than 2-hops are more typical of batch processing frameworks rather than social graphs where querying most or all of the graph data is required. The submission of traversal queries was described in Section 4.

5.2 Datasets

Three real-world datasets, namely Orkut, DBLP, and Twitter, are used to evaluate the performance of the lightweight repartitioner. We consider average path length, clustering coefficient, and power law coefficient of these graphs to characterize the datasets (Table 1). Average path length is the average length of the shortest path between all pairs of vertices. Low path lengths mean that vertices within the network are well connected. The clustering coefficient (a value between 0 and 1) measures how tightly clustered vertices are in the graph. A high coefficient means strong (or well connected) communities exist within the network. Finally, power law coefficient shows how the number of relationships increases as user popularity increases. The Orkut social graph [30, 5] is a collection of users and relationships between them. This dataset has symmetric links, meaning that connected users know each other and information flows both ways on the links. DBLP collaboration network [45, 5] is a co-authorship network for research publications which is also fully symmetric. Twitter social network [47] is a social graph of twitter users. In contrast with the previous 2 datasets, only 22% of relationships in Twitter are symmetric.

	Twitter	Orkut	DBLP
Number of nodes	11.3 million	3 million	317 thousand
Number of edges	85.3 million	223.5 million	1 million
Number of symmetric links	22.1%	100%	100%
Average path length	4.12	4.25	9.2
Clustering coefficient	unpublished	0.167	0.6324
Power law coefficient	2.276	1.18	3.64

Table 1: Summary description of datasets

Table 1 shows a detailed description of some structural statistics for the three datasets. It is notable that Twitter and Orkut share remarkably similar parameters, in particular, the average path length of both datasets is low (≈ 4). On the other hand, DBLP has a larger path length due to the strongly connected communities in its graph; as Table 1 indicates, despite the low relationship count, the clustering coefficient in DBLP is higher than Orkut and Twitter, strongly signifying the presence of highly clustered communities. All three datasets show power-law distributions similar to parameters found in other social networks [36, 44].

5.3 Experimental Results

The lightweight repartitioner is compared with two different partitioning algorithms. For an upper bound, we use a member of Metis family of repartitioners that is specifically designed for partitioning graphs whose degree distribution follows a power-law curve [7]. These graphs include social networks which are the focus of this paper.

Several previous partitioning approaches (e.g.[33, 35]) are compared against Metis as it is considered the "gold standard" for the quality of partitionings. It is also flexible enough to allow custom weights to be specified and used as a secondary goal for partitioning. We also compare the lightweight repartitioner against random hash-based partitioning, which is a de-facto standard in many data stores due to its decentralized nature and good load balance properties. Note that Metis is an offline, static partitioning algorithm that requires a very large amount of memory for execution. This means that either additional resources need to be allocated to partition and reload the graph every time the partitioner is executed, or the system has to be taken offline to load data on the servers. When the servers were taken offline, it took 2 hours to load each of the Orkut and Twitter graphs separately. This long period of time is unacceptable for production systems. Alternatively, if Hermes is augmented to run Metis on graphs, the resource overhead for running Metis would be much higher than the lightweight repartitioner. Metis' memory requirements scale with the number of relationships and coarsening stages, while the lightweight repartitioner scales with the number of vertices and partitions. Since the number of relationships dominates by orders of magnitude, Metis will require significantly more resources. For example, we found that Metis requires around 23GB and 17GB of memory to partition the Orkut and Twitter datasets, respectively; however, the lightweight repartitioner only requires 2GB and 3GB for these datasets. While Metis has been extended to support distributed computation (ParMetis [4]), the memory requirements for each server would still be higher than the lightweight repartitioner.

5.3.1 *Lightweight Repartitioner Performance*

Our experiments are derived from using real world workloads [27, 9] and are similar to the ones in related papers [34, 31]. We first study 1-hop traversals on partitions with

a randomly selected starting vertex. At the start of the experiments, the workload shifts such that the repartitioner is triggered, showing the performance impact of the repartitioner and the associated improvements. This shift in workload is caused by a skewed traffic trace where the users on one partition are randomly selected as starting points for traversals twice as many times as before, creating multiple hotspots on a partition. This workload skew is applied for the full duration of the experiments that follow.

Figure 8 presents the percentage of edge-cuts among all edges for both lightweight repartitioner and Metis on the skewed data. As the figure shows, the difference in edge-cut is too small (1% or less) to be significant, and we expect that this very small difference could shift in the other direction depending on factors such as query patterns and number of partitions. However, Figure 8 demonstrates that the lightweight repartitioner generates partitionings that are almost as good as those of Metis.

A repartitioner's performance is affected by the amount of data that it needs to migrate. Metis is expected to perform more migrations due to its coarsening stage. Recall that, in the coarsening stage, clusters of vertices are grouped as a single vertex to form a smaller graph that can be easily partitioned. However, the coarsening stage also looses information. Consequently, some vertices might be migrated unnecessarily due to their association with vertices that need to be migrated. To quantify the impact of migration on performance, the partitions resulting from the lightweight repartitioner and Metis are compared with the initial partitioning. Figure 9a shows the number of vertices migrated due to the skew based on the two partitioning algorithms. The results show a much lower count for the lightweight repartitioner. Figure 9b shows that the lightweight repartitioner requires, on average, significantly fewer changes to relationships compared to Metis. This difference is more extensive in the case of DBLP. The lightweight repartitioner is able to rebalance workload by moving 2% of the vertices and about 5% of the relationships, while Metis migrates an order of magnitude more data.

Overall, both the numbers of vertices and relationships migrated are important as they directly relate to the performance of the system. We note that, however, the relationship count has a higher impact on performance as this number will generally be much higher and relationship records are larger, and thus more expensive to migrate.

Figure 10 presents the aggregate throughput performance (i.e., the number of visited vertices) of 16 machines (partitions) using the three datasets. In these experiments, 32

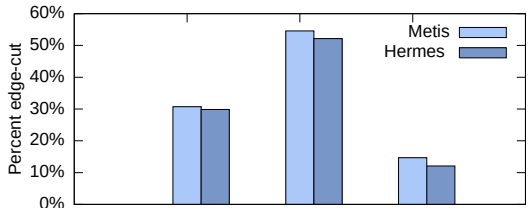
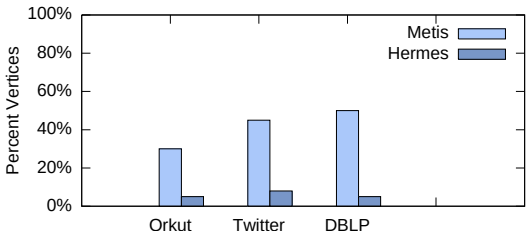
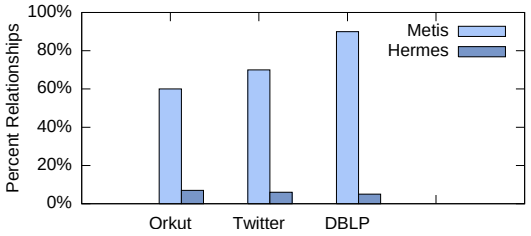


Figure 8: The number of edge-cuts in partitionings of the lightweight repartitioner (as a component of Hermes) versus Metis. Results are presented as a percentage of edge-cuts among the total number of edges.



(a) Migrated vertices.

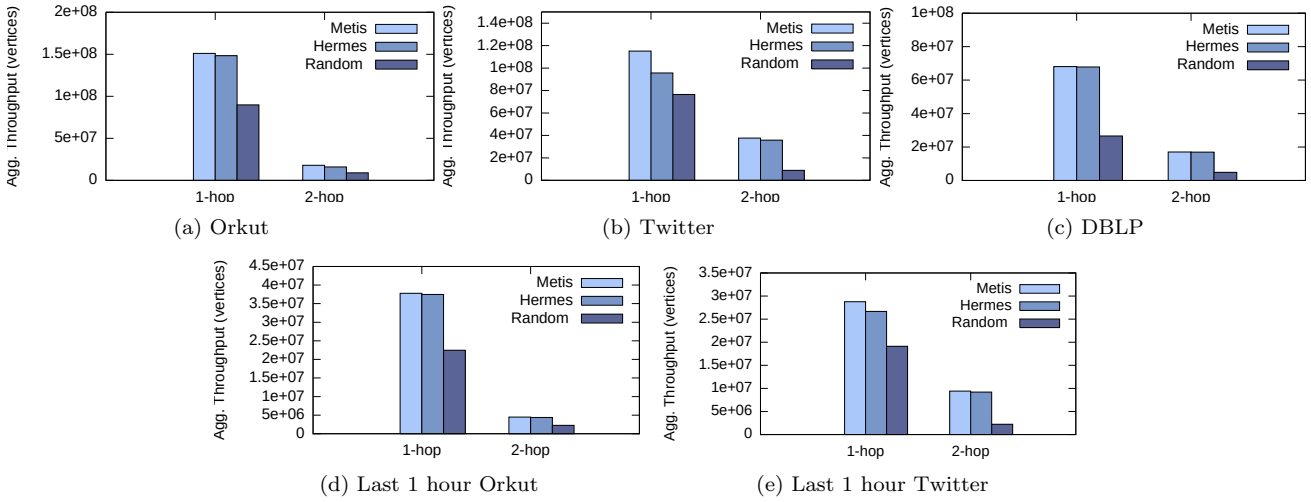


(b) Changed or migrated relationships.

Figure 9: The number of vertices (a) and relationships (b) changed or migrated as a result of the lightweight repartitioner (Hermes) versus running Metis.

clients concurrently submit 1-hop traversal requests using the previously described skew. Before the experiments start, Metis is applied to form an initial partitioning which has a trace with no skew so as to remove partitioning bias by starting out with a good partitioning. Once the experiment starts, the mentioned skew is applied; this skew triggers the repartitioning algorithm, whose performance is compared with running Metis*after* the skew. For Orkut, results show that by introducing the skew and triggering the lightweight repartitioner, a 1.7 times improvement in performance can be obtained over random partitioning while Metis shows a 6% improvement over the lightweight repartitioner. Figure 10b shows the aggregated throughput while running the Twitter dataset. The results show very similar performance for the lightweight repartitioner and Metis. Finally, Figure 10c shows the results related to the DBLP experiments that indicate there is no performance degradation due to the lightweight repartitioner, which benefits from the relatively small changes required by the algorithm. In fact, based on results from Figure 10c, the performance difference is not significant. Interestingly, the DBLP dataset is the only dataset for which the performance differences are not noticeable due to the highly clustered and well partitioned dataset. Given an edge-cut of l5%, the high query locality means that partition skews have little effect on performance as they do not shift workloads towards partition borders.

Figure 10 also includes the aggregate throughput for the last hour of the experiments to show the throughput gains after the lightweight repartitioner finishes. As the figure indicates, the lightweight repartitioner and Metis perform almost similarly, while the random partitioning performs roughly 1.7 times worse than the two. The advantage of the lightweight repartitioner and Metis over the random partitioning is more evident for Orkut; this is because of the relatively low edge-cut number in the Orkut dataset (the



(a) Orkut (b) Twitter (c) DBLP

(d) Last 1 hour Orkut (e) Last 1 hour Twitter

Figure 10: Aggregate throughput for the three datasets. Figures 10a, 10b and 10c, show aggregate throughput over the entire experiment, while 10d and 10e show the aggregate over the last hour of the experiment

edge-cut in Orkut is around 30% while it is around 55% in Twitter).

5.3.2 2-hop Performance

The previous section focused on 1-hop traversals. In this section, we conduct 2-hop experiments since they are representative operations used for recommendations, e.g., friend, events or ad recommendations in social networks. Figure 10 shows the aggregated performance of the system running with the three data-sets. The 2-hop experimental results are similar to 1-hop except for the decrease in performance. To analyze why this decrease occurs in the 2-hop case, we observe that the ratio between the number of vertices in the query response versus the number of vertices processed is 1 for both Metis and Random partitioners in case of 1-hop traversal queries, while these ratios degrade to 0.39 and 0.28, respectively, for 2-hop traversal queries. The reason 2-hop traversals return less vertices than what it processes is because some vertices are visited multiple times during a traversal while the query response contains only one copy of the queried vertices. Since social networks exhibit high clustering, a high fraction of processed vertices are accessed multiple times within the same traversal.

5.3.3 Mixed Read/Write Experiments

The following experiments test how the system handles mixed traffic workload and evolving social network graphs. The experiments insert data through random write traffic. The lightweight repartitioner in Hermes is then run to improve the quality of partitioning after records are inserted. Results of these experiments are shown in Figure 11 and indicate little performance degradation with increasing write traffic. A 10% write mix (with 90% reads) show a 3% decrease in performance, while 20% writes (80% reads) and 30% writes (70% reads) show 5% and 7% decreases in throughput (vertices per second) performance. The small performance impact of writes is attributed to how B+Trees store information and the monotonically increasing ID generator in Hermes. Since each new record will get the next, highest ID, insertions in the B+Tree always happen in the last page in a sequential manner. This translates to sequential writes to disk and the B+Tree requires caching only the last page to perform insertions. To verify that the quality of the graph is high after the insertions finish, we ran 100% read traffic and compared the throughput with the results for Metis. Results showed that Hermes was able to keep partition quality and system performance within 2% of Metis, which demonstrates the effectiveness and efficiency of Hermes's lightweight repartitioner.

5.3.4 Sensitivity of Repartitioner Parameters

Recall from Section 3 that in each iteration of the algorithm, the lightweight repartitioner moves at most k vertices from each partition. Here, we examine how the value of k affects the outcome of the algorithm. We run the lightweight repartitioner with three different values of k (500, 1000, and 2000). The first observation is that the load-balance factor slightly degrades from 1.05 for $k = 500$ to 1.16 for $k = 2000$. This is because, as mentioned earlier, larger values of k result in simultaneous migration of many vertices to partitions which have recently become popular (due to hosting popular vertices). Consequently, we excluded values of k large than 2000 from the experiment as they result in imbalance factor more than the maximum allowed value of $\gamma = 1.1$ (the default value of γ in the system). Next, we verified how

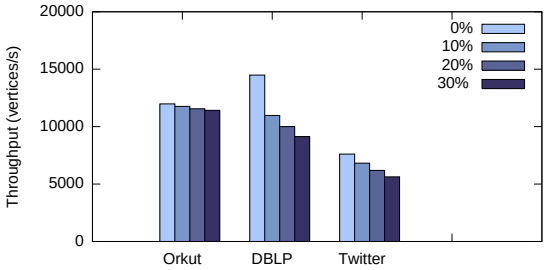


Figure 11: Throughput while varying the write rate.

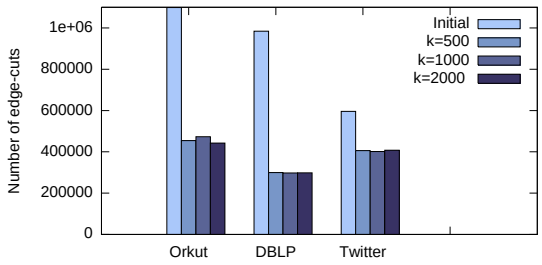


Figure 12: The number of edge-cuts for different values of k. The numbers are scaled by 10^{-2} for Orkut and Twitter datasets.

many iterations are required for the algorithms to converge. Table 2 shows the number of required iterations for different values of k. As expected, larger values of k result in slightly faster convergence since they move more vertices per iteration. Finally, we considered the quality of partitioning when different values of k are used. As Figure 12 shows, the number of edge-cuts in the final partitioning is almost the same for different values of k, indicating that the quality of partitioning does not depend on this value. To summarize, larger values of k result in faster convergence while increasing the load imbalance; at the same time, the value of k does not have a significant effect on the number of edge-cuts.

6. RELATED WORK

Previous work on graph databases focused on a centralized approach [23, 29, 1]. Some of these systems, e.g., Neo4j, have a high availability mode but this provides limited scalability [2]. HypergraphDB [23] provides a message passing API between server instances, however there is no partition management system and no support for distribution-aware querying. In addition, the focus of each of these systems is different. For example HyperGraphDB focuses more on the flexibility of its storage system, allowing users to store different types of objects. Dex and Neo4j focus more on system performance, with storage models closely resembling key-value stores. This leads to varying performance differences [15] and different utility for clients. None of these graph databases support data partitioning or distributed graph querying.

SPAR [34] and Titan [6] are middleware that run on top of key-value stores or relational databases to provide on-the-fly partitioning and replication of data. However, SPAR is restricted to keeping only one-hop neighbours local while Hermes can support general remote traversals. Titan uses only static hash-based, random partitioning scheme supported by the underlying key-value store. This is in contrast to the dynamic repartitioning that the lightweight repartitioner in Hermes uses.

	Twitter	**Orkut**	**DBLP**
k = **500**	30	30	40
k = **1000**	17	17	13
k = **2000**	10	10	11

Table 2: The number of iterations after which the lightweight repartitioner converges.

Unlike our system, SEDGE [46] focuses on partition replication in which a coarsening stage aggregates nodes which are then matched with nodes in another partition. SEDGE is not designed to handle dynamic workload changes that Hermes is designed for. An approach that performs dynamic replication is described in [31] but it does not involve a system that does on-the-fly partitioning. The system proposed in [31] is similar to SPAR in that it tries to optimize locality of vertices and replicas to minimize network costs, however, they generalize the problem, allowing some remote traversals based on the access frequency. They also propose a lazy replication model to minimize remote updates for infrequently accessed data. Horton [37] is a query execution engine built for distributed in-memory graphs. However it only provides a user abstraction, leaving partitioning to existing offline algorithms.

A streaming algorithm using simple heuristics has been proposed in [40] but focuses on improving initial data placement unlike the dynamic repartitioning in Hermes. In [41] an improved heuristic for better partitioning quality is proposed; however, the partition imbalance in the resulting solutions can be significantly impacted. While this approach extends the concept by saving state and allowing future data loads to reuse state from previous runs, the algorithm needs to parse the full dataset again, which can lead to expensive operations and large migrations.

Several Pregel-like [28] systems, e.g., [13, 22], have been proposed. These systems are quite different from Hermes in that they address only in-memory batch processing of graph analytics queries rather than the persistent management of graph data that Hermes is designed to support. In [43], a weighted multi-level partitioning algorithm is proposed which is based on label propagation (community detection technique). The edge-cut decrease in this approach can be small while communities might be detected improperly because of the weighted approach. Their multi-level algorithm does not guarantee communities are preserved over multiple calls of the algorithm and can lead to large migrations similar to Metis.

Some graph partitioning algorithms are experimentally compared in [10]. Among these, DiDiC [20] is the only distributed algorithm that tends to minimize the number of edge-cuts, but the resulting partitions of this approach may not be well-balanced [35, 10].

Ja-Be-Ja [35] embeds a distributed algorithm for balanced partitioning without global knowledge. In this algorithm, the initial partition of each node is selected uniformly at random; this ensures a balanced partitioning. In order to decrease the number of edge-cuts, vertices are swapped between partitions. This will ensure maintaining a balanced partitioning if vertices have fixed, uniform weights; however, this is usually not the case for social networks.

An adaptive algorithm for repartitioning large-scale graphs has the objective of minimizing the number of edge-cuts with respect to certain capacities for partitions [42]. The resulting partitionings might not be balanced if the capacity constraints are maintained. Moreover, it is assumed that vertices have fixed and uniform weights, which is usually not the case for social networks. Additionally, their work is targeted for graph analytics rather than the persistent management of graph data that Hermes is designed to support.

7. CONCLUSION

We presented an online, iterative, lightweight repartitioner designed to increase query locality, thereby decreasing network load and maintaining load balance across partitions. The lightweight repartitioner effectively adapts a partitioning to varying query workloads and the continuous evolution of the graph structure using only a small amount of auxiliary data. We implemented our lightweight repartitioner into Hermes, which we built to extend the open source Neo4j database system to support the partitioning of social network data across multiple database servers. The experimental evaluation of the algorithm on real-world datasets shows that the repartitioner is able to handle changes in query workloads while maintaining good performance. The overhead of the repartitioner is minimal, producing sustained performance comparable to that of static, offline, partitioning using Metis. Our evaluation shows sizable performance gains over random, hash-based partitioning, which is widely used for database partitioning.

8. REFERENCES

[1] Neo4j. http://www.neo4j.org/.

[2] Neo4j - chapter 26. high availability. http://docs.neo4j.org/chunked/stable/ha.html.

[3] Neotechnology. http://www.neotechnology.com/customers/.

[4] Parmetis. http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview.

[5] Stanford large network dataset collection. http://snap.stanford.edu/data.

[6] Titan. http://thinkaurelius.github.com/titan/.

[7] Abou-Rjeili, A., and Karypis, G. Multilevel algorithms for partitioning power-law graphs. In *proc. IPDPS* (2006), pp. 124–124.

[8] Andreev, K., and Räcke, H. Balanced graph partitioning. *Theo. Comput. Syst. 39*, 6 (2006), 929–939.

[9] Armstrong, T. G., Ponnekanti, V., Borthakur, D., and Callaghan, M. Linkbench: a database benchmark based on the facebook social graph. In *proc. SIGMOD* (2013), pp. 1185–1196.

[10] Averbuch, A., and Neumann, M. Partitioning graph databases. Master's thesis, KTH School of Computer Science and Communication, 2013.

[11] Backstrom, L. Anatomy of facebook. https://www.facebook.com/notes/facebook-data-team/anatomy-of-facebook/10150388519243859.

[12] Bernstein, P. A., and Newcomer, E. *Principles of Transaction Processing*, 2nd ed. 2009.

[13] Chen, R., Yang, M., Weng, X., Choi, B., He, B., and Li, X. Improving large graph processing on partitioned graphs in the cloud. In *proc. SoCC* (2012), pp. 1–13.

[14] Cheng, A., and Evans, M. An in-depth look inside the twitter world. http://www.sysomos.com/insidetwitter/.

[15] Dominguez-Sal, D., Urbón-Bayes, P., Giménez-Vañó, A., Gómez-Villamor, S., Martínez-Bazán, N., and Larriba-Pey, J. L. Survey of graph database performance on the hpc scalable graph analysis benchmark. In *Proc. WAIM* (2010), pp. 37–48.

[16] Emil, F. A. Fast balanced partitioning is hard even on grids and trees. In *proc. MFCS* (2012), pp. 372–382.

[17] Even, G., Naor, J., Rao, S., and Schieber, B. Fast approximate graph partitioning algorithms. *SIAM J. Comput. 28*, 6 (1999), 2187–2214.

[18] Fiduccia, C. M., and Mattheyses, R. M. A linear-time heuristic for improving network partitions. In *proc. DAC* (1982), pp. 175–181.

[19] Garey, M. R., and Johnson, D. S. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. 1990.

[20] Gehweiler, J., and Meyerhenke, H. A distributed diffusive heuristic for clustering a virtual P2P supercomputer. In *proc. IPDPS* (2010), pp. 1–8.

[21] Golbeck, J. *Analyzing the Social Web*. Elsevier Inc., 2013.

[22] Han, M., Daudjee, K., Ammar, K., Özsu, M. T., Wang, X., and Jin, T. An experimental comparison of pregel-like graph processing systems. *PVLDB 7*, 12 (2014), 1047–1058.

[23] Iordanov, B. Hypergraphdb: A generalized graph database. In *proc. WAIM Workshops* (2010), pp. 25–36.

[24] Karypis, G., and Kumar, V. Metis - unstructured graph partitioning and sparse matrix ordering system (vers. 2). Tech. rep., Univ. of Minnesota, 1995.

[25] Karypis, G., and Kumar, V. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput. 20*, 1 (1998), 359–392.

[26] Kernighan, B. W., and Lin, S. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell system technical journal 49*, 1 (1970), 291–307.

[27] Kwak, H., Lee, C., Park, H., and Moon, S. What is Twitter, a social network or a news media? In *Proc. WWW* (2010), pp. 591–600.

[28] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. Pregel: a system for large-scale graph processing. In *proc. SIGMOD* (2010).

[29] Martínez-Bazan, N., Muntés-Mulero, V., S. Gómez-Villamor, S., Nin, J., Sánchez-Martínez, M., and Larriba-Pey, J. Dex: high-performance exploration on large graphs for information retrieval. In *proc. CIKM* (2007).

[30] Mislove, A., Marcon, M., Gummadi, K., Druschel, P., and Bhattacharjee, B. Measurement and Analysis of Online Social Networks. In *proc. IMC* (2007).

[31] Mondal, J., and Deshpande, A. Managing large dynamic graphs efficiently. In *proc. SIGMOD* (2012), pp. 145–156.

[32] Nicoara, D. Distneo4j: Scaling graph databases through dynamic distributed partitioning. Master's thesis, David R. School of Computer Science, University of Waterloo, Canada, 2014.

[33] Nishimura, J., and Ugander, J. Restreaming graph partitioning: simple versatile algorithms for advanced balancing. In *proc. KDD* (2013), pp. 1106–1114.

[34] Pujol, J., Erramilli, V., Siganos, G., Yang, X.,

Laoutaris, N., Chhabra, P., and Rodriguez, P. The little engine(s) that could: scaling online social networks. *SIGCOMM CCR 40*, 4 (2010), 375–386.

[35] Rahimian, F., Payberah, A. H., Girdzijauskas, S., Jelasity, M., and Haridi, S. JA-BE-JA: A distributed algorithm for balanced graph partitioning. In *proc. SASO* (2013), pp. 51–60.

[36] Sala, A., Cao, L., Wilson, C., Zablit, R., Zheng, H., and Zhao, B. Y. Measurement-calibrated graph models for social network experiments. In *proc. WWW* (2010), pp. 861–870.

[37] Sarwat, M., Elnikety, S., He, Y., and Kliot, G. Horton: Online query execution engine for large distributed graphs. In *proc. ICDE* (2012).

[38] Schloegel, K., Karypis, G., and Kumar, V. Multilevel diffusion schemes for repartitioning of adaptive meshed. *TR 97-013, U. Minnesota* (1997).

[39] Schloegel, K., Karypis, G., and Kumar, V. Multilevel diffusion schemes for repartitioning of adaptive meshes. *J. Parallel Distrib. Comput. 47*, 2 (1997), 109–124.

[40] Stanton, I., and Kliot, G. Streaming graph partitioning for large distributed graphs. In *proc. KDD* (2012), pp. 1222–1230.

[41] Tsourakakis, C. E., Gkantsidis, C., Radunovic, B., and Vojnovic, M. Fennel: Streaming graph partitioning for massive scale graphs. In *proc. WSDM* (2014).

[42] Vaquero, L. M., Cuadrado, F., Logothetis, D., and Martella, C. Adaptive partitioning for large-scale dynamic graphs. In *proc. ICDCS* (2014), pp. 144–153.

[43] Wang, L., Xiao, Y., Shao, B., and Wang, H. How to partition a billion-node graph. In *proc. ICDE* (2014).

[44] Wilson, C., Boe, B., Sala, A., Puttaswamy, K. P., and Zhao, B. Y. User interactions in social networks and their implications. In *proc. EuroSys* (2009), pp. 205–218.

[45] Yang, J., and Leskovec, J. Defining and evaluating network communities based on ground-truth. *CoRR abs/1205.6233* (2012).

[46] Yang, S., Yan, X., Zong, B., and Khan, A. Towards effective partition management for large graphs. In *proc. SIGMOD* (2012), pp. 517–528.

[47] Zafarani, R., and Liu, H. Social computing data repository at ASU, 2009.