DistNeo4j: Scaling Graph Databases through Dynamic Distributed Partitioning

by

Daniel Nicoara

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2014

Abstract

Social networks are large graphs which require multiple servers to store and manage them. Providing performant scalable systems that store these graphs through partitioning them into subgraphs is an important issue. In such systems each partition is hosted by a server to satisfy multiple objectives. These objectives include balancing server loads, reducing remote traversals (number of edges cut), and adapting the partitioning to changes in the structure of the graph in the face of changing workloads. To address these issues, a dynamic repartitioning algorithm is required to modify an existing partitioning to maintain good quality partitions. Such a repartitioner should not impose a significant overhead to the system. This thesis introduces a greedy repartitioner, which dynamically modifies a partitioning using a small amount of resources. In contrast to the existing repartitioning algorithms, the greedy repartitioner is performant (in terms of time and memory), making it suitable for implementing and using it in a real system. The greedy repartitioner is integrated into DistNeo4j, which is 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 data sets, this thesis shows that DistNeo4j leverages the greedy repartitioner to maintain high quality partitions and provides a 2 to 3 times performance improvement over the de-facto hash-based partitioning.

Acknowledgements

I would like to thank Shahin Kamali for the invaluable input on the theorems. I would also like to thank Gobaan Raveendran and Nika Haghtalab for their support.

Additionally, special thank you to Professor Khuzaima Daudjee for in-depth feedback and suggestions on the content of this document. Special thank you to my thesis readers, Professors Tamer Özsu and Wojciech Golab, who took the time to read and provide comments to improve the content.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Large scale, connected, graphs permeate our lives. Online social networks are used to connect millions of people through many relationships. RDF triples in the context of the semantic web form graphs that are used to represent many real-world applications. The scale of such large graphs, often in millions of vertices or more, means that it is often infeasible to store, query and manage them on a single server. Thus, there is a need to partition, or shard, such large graphs across multiple servers, allowing the load and concurrent processing to be distributed over multiple servers to provide good performance and to increase availability. The focus of this work is on partitioning of the graphs associated with social networks. These networks exhibit a high degree of correlation for accesses of certain groups of records, for example through frictionless sharing [24]. In such cases users are free to access and repost the activities of all users they are connected to without asking for permision. To achieve a good partitioning which improves the overall performance, the following objectives should be met:

- The partitioning needs to be *balanced*. Each vertex of the graph has a *weight* which indicates the popularity of the vertex (e.g., in terms of the frequency of queries to that vertex). In the context of social networks, a small number of users (e.g., celebrities, politicians) are extremely popular while a large number of users are relatively unpopular. This discrepancy reveals the importance of achieving a balanced partitioning in which all partitions have almost equal aggregate weight. Here, aggregate weight of a partitioning is the total weight of vertices in the partition and defines the load on the server which hosts the partition.
- The partitioning should minimize the number of *edges cut*. An edge-cut is defined by an edge which connects vertices in two different partitions. Each edge-cut involves queries that need to transition from a partition on one server to a partition on another server, shifting from local traversal to remote traversal, thereby incurring significant network latency.

Particularly for social networks, it is critical to minimize edgecuts since most operations are done on the user's data and his neighbors. Since these *1-hop traversal* operations are so prevalent in these networks, minimizing the edge-cut is analogous to keeping communities intact. This leads to highly local queries similar to those in SPAR [45] and minimizes the network load, allowing for better scalability by reducing network IO [5].

- 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. A good partitioning solution should dynamically adapt its partitioning to the changing state of the network. 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 upon an existing partitioning. This usually involves *migrating* some vertices from one partition to another (moving data from one server to another). Excessive migration incurs excess network latency and decreases performance. Hence, it is desirable to migrate as few vertices as possible.

- The repartitioning solution should perform well in terms of time and memory requirements. To achieve this, 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 using almost all existing approaches for the repartitioning problem.

There are several partitioning algorithms which result in relatively balanced solutions with small edge-cuts (e.g., [27, 48, 29, 26, 9]). Some of these algorithms also tend to minimize vertex migration [48]. 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. 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 implemented on real data management systems.

From a data management perspective, the current storage model used in online social networks (OSNs) is based on traditional database management systems (DBMSs). While relational databases can store this type of data, they are optimized to perform index based queries. Their storage model is often very complex since it is made to handle very complex data structures and queries. Hence the performance will generally suffer. Thus, the current trend is to migrate to more simplistic models such as key-value stores. Key-value stores trade off the flexibility of the relational model for the more simplistic API supporting only simple read and write operations. Key-value stores also take advantage of simple locking models (such as optimistic locking) to increase performance. However key-value stores assume no correlation between records, a property at the core of OSN access patterns. In addition, both DBMSs and key-value stores can not easily represent the traversal based operations used in OSNs.

In contrast, graph databases use a simplistic data representation model which is often very similar to key-value stores, however graph databases also provide a complex querying model based on traversals. These queries are processed fully on the server side before results are returned to the user. Traversals are recursive joins, however due to the way data is represented in graph databases, performing these queries is a highly optimized process.

The focus of this thesis is on the design and provision of a *practical* partitioned graph data management system that can support remote traversals while providing an effective method to *dynamically* repartition the graph using only local views. The dynamic distributed partitioning aims to co-locate vertices of the graph so as to satisfy the above requirements. The contributions of this thesis are: (i) a dynamic partitioning algorithm, referred to as *lightweight, greedy repartitioner*, that can identify which parts of graph data can benefit from co-location. The algorithm is designed to improve performance by applying a fast repartitioning algorithm which uses only a small amount of knowledge on the graph structure (ii) a system called DistNeo4j, which extends the existing Neo4j [1] open source graph database system to provide the functionality to move data on-the-fly to achieve data locality and reduce the cost of remote traversals for graph data. Experimental evaluation on real-world data shows that our techniques are effective in producing performance gains and work almost as well as the popular Metis partitioning algorithms [27, 29, 9] that have the advantage of performing static partitioning offline and relies on having a global view of the graph.

1.1 Graph Repartitioning Problem

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 [12] and changes in user popularity. These changes will create long lived skews on some partitions, which reduce the performance of the system. [33] shows that traffic patterns are generally well defined. Popular users generally have more relationships [33, 16]. 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 [12]. This process leads to a slowly, but constantly evolving graph structure.

[1] Neo4j is being used by a wide variety of customers, among them are Adobe and HP. A full list of customers can be found at http://www.neotechnology.com/customers/.

A variety of partitioning methods can be used to create an initial, *static*, partitioning of the graph resulting in highly localized read traffic patterns (i.e., a small number edge-cuts) and good load balance. This should be followed by a repartitioning strategy to maintain good partitioning which 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. Moreover, the newly created partitions might be far different from previous partitions which results in high migration cost. Hence, an incremental partitioning algorithm needs to adapt the existing partitions to changes in the graph structure.

It is desired to have a lightweight repartitioner which maintains only a small amount of metadata to perform repartitioning. Since such an algorithm refers to only metadata (which is significantly smaller than the actual data required for storing the graph), the repartitioning algorithm is not a system performance bottleneck. The metadata maintained in 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 metadata 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 the greedy repartitioner is introduced through an example.

Example: Consider the partitioning problem on the graph shown in Figure 1.1. 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 1.1a 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 web blogger 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 1.1b 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 maintaining the number of edge-cuts as small as possible).

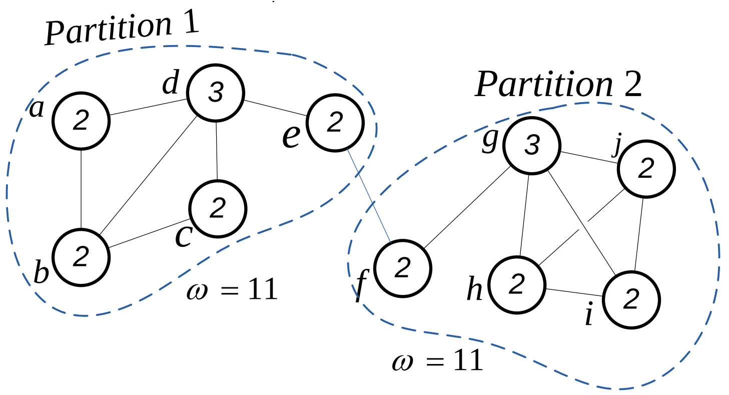
The metadata of the lightweight repartitioner available to each partition includes the weight of each of the two partitions, as well as the number of neighbors of each vertex v hosted in the partition, e.g., metadata available to partition 1 in Figure 1.1b implies that vertex *e* is connected to one vertex in each partition. Provided with this metadata, 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 greedy repartitioner makes use of its metadata which includes the number of neighbors of each vertex in each partition. For example, when the repartitioner starts from the state in Figure 1.1b, 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 1, 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 1.1c.

The above example is a simple case to illustrate how the greedy repartitioner works. Note that the only information maintained by the algorithm for each partition is the cumulative weight of all partitions as well as the number of neighbors of each hosted vertex in the other partition. Several issues are left out of the example, e.g., two highly connected clusters of vertices might repeatedly exchange their clusters to decrease edge-cut. This results in an *oscillation* which is considered in detail in Section 3.1.

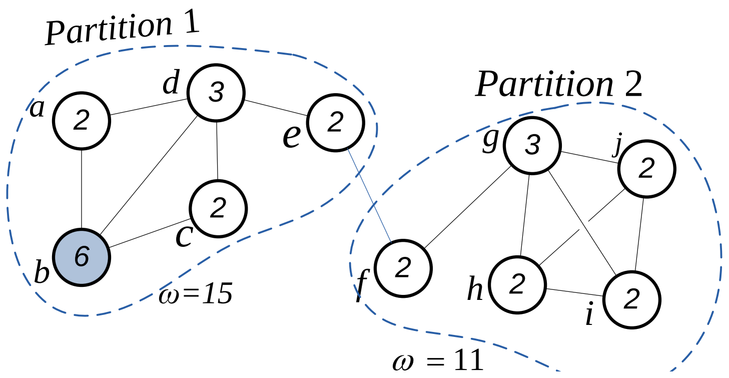
It should be mentioned that replicating graph data to deal with changes in the workload like that illustrated in the above example is also possible. However, replication in this context is complementary to partitioning and the system could be augmented for replication. Replication introduces a different set of problems that include consistent maintenance of replicas and controlling replication in the face of storage space constraints (to mention a few) and this is not the focus of this paper.

1.2 Thesis Outline

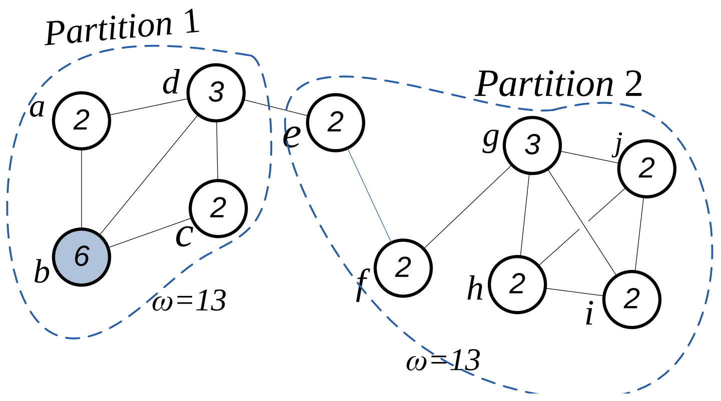
Chapter 2 will survey the existing databases that have been used with similar workloads as graph databases. The chapter will then survey existing partitioning and repartitioning algorithms, describing how they work. Chapter 3 presents the proposed lightweight greedy repartitioner. Chapter 4 details the changes made to Neo4j in order to allow distributed querying. Chapter 5 presents the experimental evaluation of DistNeo4j and the greedy repartitioning algorithm. Finally, Chapter 6 concludes the thesis.



(a) Balanced partitioned graph



(b) Skewed graph



(c) Repartitioned graph

Figure 1.1: Example graph evolution and effects of repartitioning as response to imbalances

Chapter 2

Related Work

2.1 Databases

2.1.1 Graph Databases

Previous work on graph databases focused on a centralized approach [25, 39, 2]. Some of the most popular graph databases[19] are Dex[39], Neo4j[2] and HyperGraphDB[25]. All of these systems support a similar traversal based querying model, however they focus on optimizing different parts of the system.

Dex is presented as a layered implementation where query processing is broken down in multiple stages. The initial querying stage scans the stores and filters results to form collections of sub-graphs. The next stage is a preparation and mining stage where the sub-graphs are further modified due to creation, removal operations or further filtering. The key insight is that graph manipulations are highly parallelizable since they generally require only local information to sub-graphs. The optimizations involved reducing the memory footprint, allowing queries to filter or identify valid records by using optimized data structures such as bitmaps. Their experimental evaluation focused on relatively large queries such as minimum collaboration distance between two individuals or between one individual and everyone else. Other queries looked at finding the context of keywords. Their experiments show good scalability and high optimization of the data-layout such that query performance is roughly the same in memory constrained system to the ones where data fully fits in memory.

Neo4j is designed around a network model where data is stored as nodes and different records are linked through relationships. The storage layer resembles key-value stores where nodes, relationships and properties are stored separately in their own store. This allows the system to perform quick traversals as record sizes are kept small and reads do not need to process unnecessary information. Neo4j is build to support transactions with ACID consistency properties. For increased performance the Neo4j high availability mode is currently recommended to scale read traffic[3]. [56] compares Neo4j and MySQL using different types of queries. They show that on structural queries (such as joins) and text searches Neo4j clearly performs better than MySQL.

HyperGraphDB is a graph database system built to represent hypergraph problems. The reasoning is that some problems, such as multi-input/output problems are easier to represent in hypergraph format allowing much simpler operations[25]. In addition, HyperGraphDB's format is highly desirable in learning algorithms or natural language processing. HyperGraphDB can also be considered an object store as every record is considered an atom and the user is able to perform complex type based queries on top of it. HyperGraphDB has been extended to provide a simple peer-to-peer distribution model in which different instances of HyperGrapbDB can communicate. However data management and distributed query processing is left to the user, meaning that the user needs to know which peer holds data to be able to perform any query. Unfortunately [25] focuses more on the flexibility of the system leaving out performance evaluation.

[35] presents Concerto, a distributed, in-memory graph database built on-top of Sinfonia (a distributed shared memory system). The focus of their work is on interactive event processing triggered by updates to the system. Concerto extends the concept of a view from DBMSs to graph databases. The use of views is relevant to systems such as highway monitoring where sensor events are processed on a regular basis. Given specific definitions of views the system is able to monitor and respond to changes in the graphs based on events. Another use is in load balancing as views can be used to monitor for system usage. Changes in the workload triggers a process responsible for migrating hotspots. They compare their system to MySQL, Neo4j and GemFire (an in-memory distributed store). Their results show significant performance gains during k-hop traversals and full graph queries (k-core).

In [19] a performance-centric evaluation of some graph databases is performed. They use different query types (insertions, single record queries and variations of traversals, such as finding neighbours to complex queries such as multi-hop queries or computing the *betweenness centrality* of the graph). The results show DEX is the fastest system overall showing the best scalability. Neo4j is second being faster than DEX only on a few query types, though it shows worse scalability than DEX. Finally, HyperGraphDB is placed last, showing good performance on the smallest dataset. Due to its poor scalability it is omitted from further experiments.

2.1.2 Related Systems

This section will present other systems that work in a similar way, or are meant to process similar queries but they cannot be classified as a graph database.

SPAR[45] is a middleware that runs on top of key-value stores or relational databases to provide on-the-fly partitioning and replication of data. SPAR's use case is *1-hop* traversals where users want or need access to their direct neighbours for querying purposes. Thus SPAR's partitioning algorithm guarantees data locality through replication. SPAR's partitioning and replication algorithm is presented as an optimization problem where the goal is to minimize the number of replicas while keeping all 1-hop traversals local. Their experimental evaluation shows that SPAR can triple the request rate over Cassandra with random, hash based partitioning. SPAR also reduces the network traffic overhead by a factor of 8. In comparison to random and Metis partitioning, their results show a lower number of required replicas to preserve 1-hop local semantics. Unfortunately OSNs exhibit traffic patterns that require multi-hop traversals as well. In addition, even for 1-hop traversals, the replication overhead they present in the paper may be unacceptable, especially since the replication overhead increases as the number of servers increases.

Titan[7] is another middleware which builds on top of key-value stores to provide a graph querying interface but uses only static hash-based, random partitioning scheme supported by the underlying key-value store. Titan also provides ACID and eventual consistency. [8, 1] show performance numbers while running with typical OSN queries, unfortunately there are no comparisons with different systems. [1] uses a domain based partitioner which allows communities to be co-located. However it is not always possible to know communities ahead of time and social graphs tend to be more flexible and change more often than the educational dataset used by Titan.

Pregel[38] is a distributed graph processing framework built around the graph structure. The computational model relies on the *bulk synchronous parallel* (BSP) [55] processing model to process information. Pregel algorithms implement a vertex interface where each vertex processes some piece of information and transmits intermediate results to its neighbours. The algorithm in executed in successive steps until all vertices finish processing. Pregel is ideal for large scale graph computations such as PageRank where performance of batch processing is important.

SEDGE[60] is a system built on top of Pregel which focuses on partition replication such that workloads can be diverted to one of the replicas that exhibits the best query locality. SEDGE uses multiple partitioning techniques to achieve the best diversity of partitionings such that multiple query types can be handled at the same time. The first type of partitioning is named *complementary partitioning* which focuses on finding multiple partition sets such that each set's partition boundries differ from other sets. Intuitively this allows queries that may land on the border of one set to be fully local to a partition in another set. The second type is partition replication which is used to handle highly localized queries to certain partitions by replicating them to idle machines. In order to handle cross-partition hotspots they employ a third type of partitioning algorithm, referred to as *dynamic partitioning* which relies on a coarsening stage to discover node clusters with high communication patterns. Their experiments compare simple partition replication with complementary partitioning and with complementary partitioning and dynamic partitioning combined. The results show 2 to 3 times improvement in processing time of queries. The results show that long lived queries benefit much more from these partitioning and replication schemas.

Surfer[15] is built on top of Pregel. Surfer's contribution relates optimizing initial data placement in a cloud based environment. Their motivation relates to the uneveness of the network topology in cloud systems. Due to the hierarchical model of cloud environments, servers located within the same rack will have much better network bandwidth than servers located in different racks, which in turn will have better bandwidth to those located in different parts of the datacenter or even different datacenters. Surfer proposes using a topology aware graph partitioner in combination with ParMetis to partition the graph efficiently. Since cloud operators do not release topology information to users, their algorithm first tries to map the network's topology with small data transfers, allowing them to create a weighted graph of the server network. Given a network graph and a data graph, the next step is to partition the data graph such that it respects the constraints in the network graph. As an optimization, they perform this operation as recursive bissection on the graphs, allowing them to fully parallelize each inner recursive step. The key insight is that in the initial partitioning levels, the quality of the partitioning is much higher than in the next recursive levels. In hierarchical datacenter networks, the top level links represent the most oversubscribed links. Thus, but optimizing the initial bissections, they optimize the traffic on the most constrained network links. Their evaluation shows a significant performance increase from using ParMetis without any topology information.

Mizan[32] is another system built on Pregel. Mizan's scope is to migrate data during execution to minimize load skew. Their motivation is that no single partitioning algorithm can reduce run time consistently over all types of datasets while running different types of algorithms. Mizan monitors the system for three key metrics: 1) outgoing messages, 2) incoming messages and 3) response time. Each metric represents a system constraint: network cost, disk paging and load. Based on these metrics, their migration planner will find the strongest cause of workload imbalance from these three and migrate vertices to reduce imbalance. Their experiments show that on variable workloads (hotspots tend to vary in the system throughout the run), Mizan is able to improve the run time up to two orders of magnitude. Though, in less variable algorithms, such as PageRank, where the base algorithm dominates the run time, the slight imbalances resulting from the partitioning algorithms mean Mizan has little room to improve run times.

GraphChi[34] is a centralized system used for bulk graph processing tasks such as PageRank. Their work focuses on optimizing the updating process such that disk I/O overhead is minimized. They take advantage of the compressed sparse row (CSR) and compressed sparse column (CSC) storage format for graphs such that disk accesses are sequential. In addition, their work proposes a parallel sliding windows (PSW) model in order to reduce the number of non-sequential disk writes. In this model, they load a *shard* (sub-graph) in memory, execute the update algorithm on the shard and perform all updates in memory. After processing the shard, it is written back to disk in a sequential manner. Their evaluation shows that the system's performance is comparable (per-server basis) with other existing distributed systems.

[41] is a middleware implemented on top of CouchDB key-value store that performs dynamic replication, similar to SPAR. While they have similar goals, this system does not try to keep data fully local to 1-hop traversals. Instead they only replicate frequently accessed records. In addition, they also use two different types of updates. For frequently updated records a push based update system is used where the master is responsible for updating all replicas. For infrequently updated records a pull based system is used where the replica queries the master for updates. They go further and analyze the access patterns over periods of time which allows them to switch between the two update modes based on time of day. Combined with a lazy replication model, this allows the system to keep replicas only for frequently accessed records.

Horton[47] is a query execution engine built for distributed in-memory graphs. Horton provides a graph processing interface to abstract distributed graph queries. The graph engine handles the intricacies of sending the query to the appropriate server and returning the results to the user. Unlike systems such as MapReduce[18] and Pregel[38] which optimize for large, batch processes, Horton focuses on ad-hoc, small queries and tries to optimize query latencies.

2.1.3 Summary

Current graph database systems focus on a cetralized approach and optimize for this case. HyperGrahpDB is the only one providing an interface for distributed data storage and querying, however it leaves data placement to the user. In addition due to its low performance compared to the other graph databases it is a poor candidate. SPAR focuses on 1-hop traversals only keeping data local for these queries by using replication. Titan uses key-value stores as backend, but uses the partitioning algorithms supplied by the underlying store. Pregel, Sedge, Surfer and Mizan are used for bulk processing, so they are not suitable for short-lived ad-hoc traversals. The CouchDB middleware is similar to SPAR, but relaxes the fully local constraint. By providing replicas only for the frequently accessed records it reduces the number of replicas needed. Horton is an in-memory store and focuses on the system and querying details leaving partitioning for future work.

DistNeo4j extends Neo4j to allow dynamic, distributed management of graph data in two ways. First it allows automatic data sharding across multiple servers and abstracts communication within the system such that queries spanning multiple servers are executed on multiple servers transparently to the user. Second, DisNeo4j dynamically partitions data such that data locality is maintained.

Table 2.1 summarizes the key features of each system presented in this chapter, allowing easy comparison between the system and with the proposed DistNeo4j system.

	Processing Model	**Persistence**	**Transactional Model**	**Distributed**	**Partitioning**
Dex	ad-hoc traversals	disk-based	-	no	-
Neo4j	ad-hoc traversals	disk-based	ACID	no	-
HyperGraphDB	ad-hoc traversals	disk-based	ACID	yes (limited)	none
Concerto	ad-hoc traversals	in-memory	ACID	yes	random, hash-based
SPAR	1-hop ad-hoc traversals	disk-based (depends on backend)	atomic get/set	yes	replication based guaranteeing 1-hop locality
Titan	ad-hoc traversals	disk-based (depends on backend)	ACID or atomic get/set	yes	random, hash-based
Pregel	bulk synchronous processing	disk-based	-	yes	random, hash-based
Sedge	bulk synchronous processing	disk-based	-	yes	replication based combined with greedy heuristics
Surfer	bulk synchronous processing	disk-based	-	yes	network topology based
Mizan	bulk synchronous processing	disk-based	-	yes	greedy partitioning
CouchDB Middleware	ad-hoc traversals	disk-based	atomic get/set	yes	replication based
GraphChi	bulk asynchronous	disk-based	atomic get/set	no	-
Horton	ad-hoc traversals	in-memory	atomic get/set	yes	random, hash-based
DistNeo4j	ad-hoc traversals	disk-based	ACID	yes	greedy partitioning

Table 2.1: Comparison of different graph systems

2.2 Graph Partitioning Survey

2.2.1 Graph Partitioning

In the classical (α, γ)-graph partitioning problem, 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 have:

$$\omega(P) \leq \gamma \times \sum_{v \in V(G)} \omega(v)/p$$

Here, $\omega(P)$ and $\omega(v)$ denote the weight of a partition P and vertex v, respectively. Parameter p denotes the number of partitions and 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. (Unbounded values for γ relate the problem to the min-cut problem which is not the focus of this thesis.) The goal of the minimization problem is to achieve a valid solution in which the number of edges between components (the number of edges cut, or *edge-cut*) is minimized. Thus, in this context, an optimal solution has the least number of edges cut and the weights are evenly distributed across all partitions (is a valid solution). Note, there may be other solutions with a lower number of edges cut but with skewed weight distribution.

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* [23]. Moreover, there is no approximation algorithm with a constant approximation ratio unless P=NP [11]. This reveals the very 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 [11] and $O(\log n)$ when $n \geq 2$ [21]. Interestingly, the problem remains NP-hard (and even Approximate-hard) for simple graph families like trees and grids [20]. 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.

To choose an adequate partitioning scheme, a survey of existing partitioning schemes is performed. Each algorithm is evaluated based on the following categories: partitioning quality, memory usage, network communication and degree of scalability. The evaluated algorithms can be divided in six classes: greedy, spectral, geometric, stochastic, multilevel and streaming.

Most partitioning algorithms measure *partitioning quality* by minimizing the number of edges crossing partition boundries (edge-cuts). As a secondary goal, some algorithms try to keep the aggregate partition weights similar. In general, vertices are assigned some weight based on the problem domain. The aggregate partition weight is the sum of vertex weights in the partition. Some algorithms will also allow optimizations of edge-weights or even minimizing the number of vertices migrated.

Greedy Algorithms

The two major algorithms in this category are *Kernighan-Lin (KL)*[31] and *Fiduccia-Mattheyses (FM)*[22].

KL is an iterative algorithm whose goal is to move sets of vertices between two partitions such that it maximizes the decrease in the number of edges cut. The algorithm finishes when no swaps can be performed that improve the edge-cut. In order to measure the decrease in edge-cut they define a gain function for each vertex as $g = E - I$ where E is the cost of edges connecting it to vertices from a different partition and I is the cost of edges connecting it to vertices from the same partition. In each iteration, the algorithm will try to find pairs of vertices such that swapping them maximizes $g_a + g_b - 2c_{a,b}$, where $c_{a,b}$ is the cost of the edges between vertex a and b. Note that in the simplest case where each edge has unit weight, then the cost is the sum of the number of edges.

Unfortunately, due to the greedy nature, it is possible to have sub-optimal matchings leading to a sub-optimal solution. Due to $c_{a,b}$ KL requires frequent communication between partitions in order to find the best pairs.

FM is inspired by KL and improves partitioning by iteratively selecting a vertex from the largest partition and moving it, rather than moving groups. One key factor is that by moving one vertex at a time the algorithm is guaranteed to select a vertex with the most impact on edge-cut.

From a storage perspective, the data required for intermediate state is O(n*p) where n is the number of vertices and p is the number of partitions. All the intermediate results can be incrementally updated as vertices and edges are added or removed. However, the serial nature of these algorithms makes them difficult to scale in terms of time complexity.

Spectral Algorithms

The major algorithm in this category is Spectral Clustering[37]. Spectral Clustering uses the similarity matrix of a dataset to cluster similar elements. In graph partitioning the similarity matrix is represented by the adjacency matrix, thus, *similar* vertices are part of the same communities and will cluster as they form a dense relationship graph. Given the similarity matrix S where S_{ij} represents the similarity between vertices i and j, and matrix D, a diagonal matrix where $D_{ii} = \sum_j S_{ij}$, the eigenvectors for the *k* larges eigenvalues of the matrix $P = D^{-1}S$ [51] can be used to partition the graph in *k* partitions using clustering algorithms such as *k-means clustering*. An alternative method to computing the eigenvectors is to compute them from the Laplacian matrix $L_{rw} = I - D^{-1/2}SD^{-1/2}$[52].

Spectral algorithms have high quality, however they do not scale beyond local parallelism due to the high data sharing requirements.

Power Iteration Clustering[36] is an alternative to spectral clustering that approximates the largest eigenvector using power iteration. Their results show linear scalability with graph sizes. The network communication is dependent on the cross-partition vertices. Since social networks have good clustering coefficients (the degree to which vertices cluster together), the expected communication overhead is low. However the quality of the resulting partitions are influenced by the initial vector chosen in power iteration. Their solution is to run PIC multiple times with random initial values, however this would negate the performance benefits of PIC.

Geometric Algorithms

Algorithms such as *Recursive Coordinate Bisection (RCB)*[43] take advantage of the coordinates of vertices to perform bisections. Such algorithms look at the coordinate distance between vertices and bisects the graph such that the distance within partitions is minimized. While geometric algorithms perform orders of magnitude faster[43] than other methods, the resulting partitions are often imbalanced. In addition, since the algorithm does not look at the edge-cut it often leads to partitions with a high edge cut.

Stochastic Algorithms

One representative algorithm is *Simulated Annealing(SA)*[43] which is an iterative method that searches the solution space. In the context of SA, a solution is one possible partitioning and a neighbouring solution is a partitioning with exactly the same grouping with the exception of one vertex which has been migrated to a different partition. In each iteration, the algorithm visits a neighbouring solution and selects it if the solution improves quality or randomly, by a user defined probability. This allows SA to move towards more optimal solutions, however it does allow some degree of flexibility such that it can leave locally optimum solutions. SA doesnt perform as well as other methods on sparse graphs[43] and its computation model is inherently sequential.

Multilevel Algorithms

Multilevel algorithms are based on a three stage process[49]. In the first stage the graph is repeatedly coarsened into smaller graphs. In the second stage the smallest graph is partitioned using a traditional partitioning method such as KL or FM. Finally, in the last stage the partitions are projected back to the initial graph.

In the coarsening phase vertices are matched and form a smaller graph. In the process of coarsening vertices, edges may be collapsed as well since they refer the same source and destination vertices. When edges are collapsed, the edge representing them will be assigned a weight equal to the sum of the weights of the represented edges.

In the uncoarsening stage, algorithms generally employ some refinement algorithms such that the edge-cut or the partitioning balance is improved.

The most representative algorithm in this category is *Metis*[29, 27]. In the coarsening phase [29, 27] propose different heuristics for matching vertices. The most trivial heuristic is *random matching*, where a vertex is randomly matched with a neighbour which has not been matched yet. A more complex heuristic proposed is *heavy edge matching* (HEM). HEM tries to match vertices that are connected by heavy weight edges. By matching vertices with heavy weights the algorithm maximizes the decrease in edge-cut. In the uncoarsening stage they propose a variation of the KL algorithm. Since the KL algorithm only assesses border vertices, they optimize the algorithm by computing the vertex weights and gains for border vertices only and updating the set of border vertices on demand as migrations happen.

[9] further improves the coarsening stage by allowing matching already matched vertices. The optimization improves the convergence on power law graphs where the least popular users tend to be left unpaired, resulting in larger number of iterations until they can be paired such that the graph size is significantly reduced.

While the original algorithm was sequential, it has since been parallelized [30, 50] and results show near linear scaling with the number of processors allocated.

More recently [57] proposes a multi-level label propagation (MLP) method to partitioning graphs. Label propagation (LP) is a heuristic used to detect communities in social networks. In

LP each vertex is initialy assigned a label. In each iteration each vertex will take the most popular label in its neighbourhood. The process terminates when labels no longer change. [57] adapts LP in the multi-level partitioning algorithm. In each coarsening stage, vertices with the same label are coarsened together. They evaluate their algorithm against Metis showing that MLP can produce high quality partitionings with significantly less memory resources and significantly lower run-time.

Streaming Algorithms

Streaming algorithms are greedy algorithms that partition data as it is read. Normally, these algorithms will only know of data they already processed and often, they only buffer a limited amount of it. The insight is that low cost partitioning may improve partition quality over random, hash based without having a computational overhead (or at least a minimal one). Compared to offline partitioners that require a global view of the data, streaming algorithms are one pass algorithms which limited knowledge of the graph.

[53] proposes such an algorithm and assesses the quality of multiple heuristic functions. They show that some heuristics can significantly improve quality over random partitioning. Their results also show that these heuristics are also consistent over multiple types of data sets. Compared to Metis, their results show that data set types are important. In the best case scenario their algorithm was within 10% of Metis. Unfortunately, on average, 25% of cross partition edges can still be optimized.

Fennel [54] proposes an improved heuristic algorithm that improves data locality, but at the cost of increased partition imbalance. Unlike the heuristics in [53], Fennel, does not guarantee partition balance.

2.2.2 Graph Repartitioning

I further looked at repartitioning algorithms to analyze their performance and quality. Most of the repartitioning algorithms are modifications of partitioning algorithms mentioned above. In addition most (KL, FM, SA) suffer from the same scalability problems mentioned in the previous section. Thus I will not mention them again.

Linear Programming Repartitioning

[44] presents a repartitioning algorithm based on linear programming. It tries to optimize the edge-cut by ”smoothing” the partition borders by applying linear constraints on the edge cut,

partition sizes and vertices moved. They compared it with *recursive spectral bisection* (special case of spectral clustering optimized using a multilevel approach) and found that a sequential implementation is two orders of magnitude faster. In addition, by parallelizing their implementation they gain up to 20 times the performance over the sequential.

One potential problem mentioned in the paper is the inability to solve the linear programming problem due to constraint values which would require user intervention.

ParMetis

[49] presents a modification of the Metis algorithm. They take advantage of the relatively partitioned state of the graph and completely parallelize the coarsening stage by collapsing only local vertices. In the un-coarsening stage they apply a diffusion scheme in order to balance vertex distribution and decrease edge-cut. Their results show that the quality of the partitions are similar to those generated by Metis while taking orders of magnitude less time.

Re-streaming Algorithms

[42] extends the work on streaming algorithms from [53, 54] to multi-iteration streaming. The basic idea is that the same data sets will often be streamed periodically, thus, keeping track of results from the previous streaming can improve quality. Their results show that this process can significantly improve quality after a few iterations. The partition quality was similar to Metis' quality while requiring less memory.

Unfortunately, their focus is on data sets that need to be fully loaded. A problem with this assumption is the amount of vertices that will have a new partition at the end of an iteration can be high. Additionally they need to process the full graph to be able to repartition it.

2.2.3 Summary

	Partition Quality	**Memory Usage**	**Communication**	**Migrations**	**Scalability**
Kernighan-Lin	medium-high	O(n*p)	high	low	low
Fiduccia-Matheyses	high	O(n*p)	low	low	low
Spectral Clustering	high	O(n+e)	low	low	low
Power Iteration Clustering	high-unstable	O(n+e)	low	low	high
Recursive Coordinate Bisection	low	O(n+e)	low	low	high
Simulated Annealing	high	O(n*p)	low	low	low
Metis	high	O(i*(n+e))	low	high	high
MLP	high	O(i*(n+e))	low	high	high
Streaming	medium	low (depends on settings)	low	high	high
Re-streaming	high	O(n)	low	high	high

Table 2.2: Summary comparison of partitioning algorithms. *n* represents the number of vertices, *e* represents the number of edges, *p* represents the number of partitions and *i* represents the number of iterations the algorithm is executed.

A summary of the partitioning comparison can be viewed in Table 2.2. Greedy algorithms do not scale well and they require large numbers of iterations to converge. Spectral and stochastic algorithms can not scale well. Geometric algorithms require additional information what is not available. Multi-level algorithms have good quality and scale well, however memory requirements are high. Streaming algorithms are one shot algorithms and have average gains. Linear programming repartitioners have good quality and performance, though they can get stuck or are unable to solve the problem. Re-streaming algorithms improve the quality of streaming algorithms, though they require full graph processing to finish any additional iterations and do not guarantee minimization of data migration.

Overall the algorithms that performed best in each analyzed category were *PIC* and *Metis*. To my knowledge there is no published work that compares these two methods. I initially planned on using both methods and analyze the performance differences. However my initial tests on PIC showed that the partition quality can be severely impacted by the initial start parameters. I found that running PIC multiple times as suggested in [36] to be unacceptable as it decreases performance of the algorithm. Metis also suffers from decreased performance as graph sizes increase. In addition, due to the high memory overhead of the coarsening stage, using Metis or its variants is unfeasible.

Repartitioning algorithms show better run-time performance as they assume relatively well partitioned graphs. This key insight allows them to optimize several expensive operations or even discard them. Unfortunately, *Linear Programming Repartitioning* has limited experiments on very small and very sparse graphs. It is also possible that the linear optimization problem cannot be solved without loosening some constraints[44]. ParMetis improves run-time performance over Metis, however it still suffers from large memory constraints making it unfeasible.

Chapter 3

Proposed Greedy Repartitioner

The previous chapter presented the most significant partitioning and repartitioning algorithms in the literature. The survey shows that these algorithms are generally used to solve generic graph partitioning problems with little to no prior knowledge of graph structure or partitioning quality. In addition to the goals present in the partitioners, the surveyed repartitioning algorithms add additional assumptions, such as the number of changes to the graph partitioning is minimized. Using this assumption allows them to optimize certain stages of existing partitioning algorithms to decrease the time required to completion. Unfortunately, by re-using parts of partitioning algorithms, repartitioners require large amounts of in-memory state, making them unfeasible to execute in parallel in a database system.

This chapter presents a greedy, iterative algorithm used to repartition a graph. The key constraint imposed on the graph is: the graph should be well partitioned, such that when running the repartitioner, the graph would have only changed by small amounts. With respect to the algorithm, the constraints include: small memory footprint, small network overhead, and high parallelization in order to allow the algorithm to scale.

3.1 Algorithm Description

When new nodes join the network or the traffic patterns (weights) of nodes change, it is required to update the partitioning by migrating vertices between partitions. This helps to reduce the number of remote traversals. The greedy repartitioner uses an iterative process to rebalance vertex weights while decreasing edge-cut. To increase performance, instead of looking at the graph structure, the algorithm makes use of aggregate vertex weight information as its metadata.

Assuming there are α partitions, for each vertex v, the metadata includes α integers indicating the number of neighbors of v in each of the α partitions. This metadata 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 metadata is collected and updated based on execution of user requests e.g., when a new edge is added, the metadata of the partitioning(s) including the endpoints of the edge get updated (two integers are incremented). Hence, the cost involved in maintenance of metadata 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 metadata is not a bottleneck of the system. Each partition collects and stores aggregate vertex information relevant to only the local vertices. Moreover, the metadata includes the total weight of all partitions, i.e., in doing repartitioning, each server knows the total weight of all other partitions.

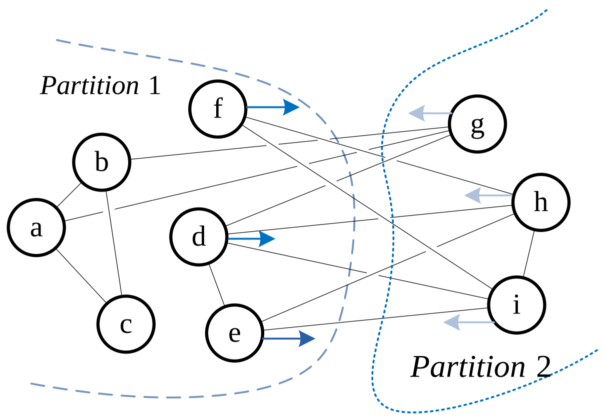
The repartitioning process has two *phases*. The first phase is an iterative process; in each iteration, each server runs the repartitioner algorithm using the metadata to indicates some vertices in its partition that should be migrated to other partitions. Before the next iteration, these vertices are *logically* moved to their target partitions. Logical movement of a vertex means that only the metadata associated with them is sent to the other partition. This process continues up to a point (iteration) in which no further vertices are indicated 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 described later) and metadata 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. Note that this process continues up until a point where no further vertex is selected for migration. 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 neighbors of v in P_t and P_s, respectively, i.e., $gain(v) = d_v(t) - d_v(s)$ (where $d_v(k)$ denotes the number of neighbours 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 some vertices as the *candidates* for migrating. Basically, the algorithm selects vertices which result in the maximum gain when moving from the partition. However, to avoid *oscillation* and ensure a valid packing in term of load balance, the algorithm has some rules in migrating vertices. First, it defines two *stages* in each iteration. In the first stage, the migration of vertices is only allowed 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 manner in two stages prevent the algorithm from oscillation. Oscillation happens when two group of vertices, shared in two different partitions, share a large number of edges (see Figure 3.1). 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 3.1b). In a one-way migration, however, the vertices in one group remain in their partitions while the other group joins them in that partition (Figure 3.1d).

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

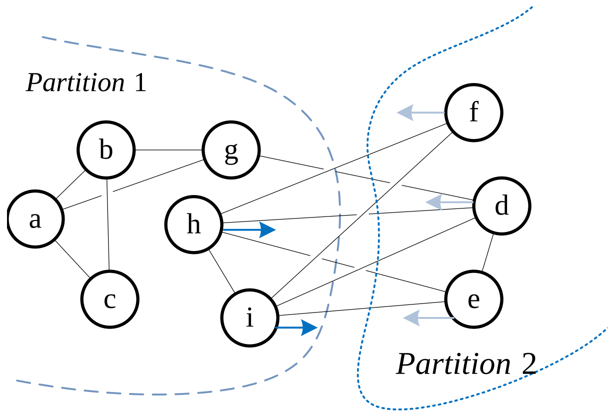
- P_s and P_t fulfill the above one-way migration rule.
- Moving v from P_s to P_t does not cause P_t to be *overloaded* nor P_s to be *underloaded*. Recall that the imbalance ratio of a partition is the ratio between the weight of the partition (the total weight of vertices it is hosting) and the average 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 the system 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_t to P_s. When a partition is overloaded, vertices are tagged as candidates for migrating to any other partitions as long as they do not cause an overload of the target partitions. When the partition is not overloaded, it is good to move only vertices which have positive weight, i.e., improve the edge-cut.

When a vertex v is a candidate for migration to more than one partitions, the partition with maximum gain is selected as the target partition of the vertex. This is illustrated in Algorithm 3.2. Note that detecting whether a vertex v is a candidate for migration and selecting its target partition is performed using only the repartitioning metadata. Precisely, for detecting underloaded and overloaded partitions (respectively Lines 2, 5 and 11) the algorithm uses the weight of the vertex and all partitions; these are included in the metadata. Similarly, for calculating the gain of moving v from partition P_s to partition P_t (Line 8), it uses the number of neighbors of v in any of the partitions, which is also included in the metadata.

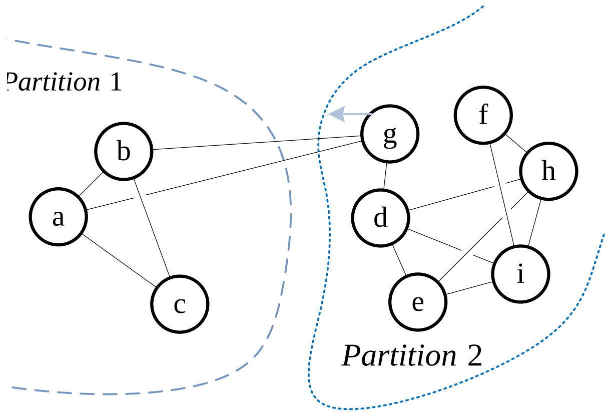
Recall that the repartitioning algorithm runs on each partition separately. Here, we describe



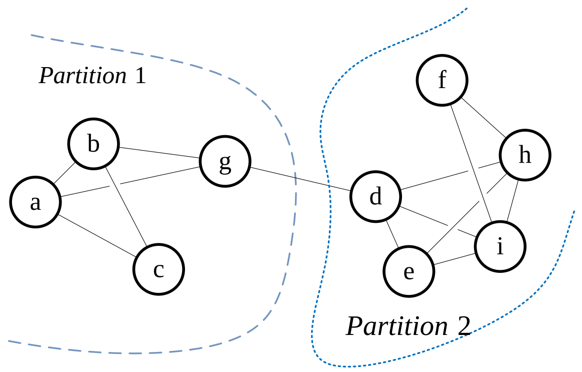
(a) Initial graph, before the first iteration.



(b) The resulted graph if vertices migrate at the same stage.



(c) The resulted graph after the first stage.



(d) The final graph after the second stage.

Figure 3.1: 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 rerpartioning of (a), the vertices d, e, f are migrated from partition 1 (lower ID) to partition 2 [dark arrows]. After this, as depicted in (c), the only vertex which requires to migrate in the second stage is vertex g which moves from partition 2 (higher ID) to migration 1 (d).

```
1: procedure GET_TARGET_PART(vertex $v$ currently hosted in partition $P_s$)
2:     if imbalance_factor($P_s - \{v\}$) $< 2 - \gamma$ then
3:         return $(null, 0)$
4:     end if
5:     $target = null$; $maxGain = 0$;
6:     if imbalance_factor($P_s$) $> \gamma$ then
7:         $maxGain = -\infty$
8:     end if
9:     for partition $P_t \in$ partitionSet do
10:        $gain \leftarrow d_v(t) - d_v(s)$
11:        if imbalance_factor($P_t \cup \{v\}$) $< \gamma$ and $gain > maxGain$ then
12:            $target \leftarrow P_t$; $maxGain = gain$
13:        end if
14:    end for
15:    return $(target, maxGain)$
16: end procedure
```

Figure 3.2: Choosing the migration target partition

how the algorithm works in each iteration. 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 metadata associated with the vertex to its target destination [and updating the metadata 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 cumulative weight of partitions by restricting the number of migrating vertices. Later, we discuss 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 3.3 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 a logical migration only, in the sense that only the metadata associated with vertices is migrated. After each phase of each iteration, the metadata associated with each migrated vertex

```
1: procedure REPARTITIONING_ITERATION(partition $P_s$, $k$)
2:     for stage $\in \{1, 2\}$ do
3:         $candidates \leftarrow \{\}$
4:         for Vertex $v \in$ VertexSet($P_s$) do
5:             GET_TARGET_PART(v)                          ▷ setting $target(v)$ and $gain(v)$
6:             if target(v) $\neq$ null and [ $(stage = 1$ and $target(v)$.ID $> P_s$.ID) or $(stage = 2$
   and $target(v)$.ID $< P_s$.ID)] then
7:                 candidates.add ($v$)
8:             end if
9:         end for
10:        top-k $\leftarrow k$ candidates with maximum gains
11:        for Vertex $v \in$ top-k do
12:            MIGRATE($v, P_S, target(v)$)
13:        end for
14:        $P_s$.update_metadata
15:    end for
16: end procedure
```

Figure 3.3: Repartitioning algorithm

v should be updated. This is because the neighbors of v might also be migrated; this implies that the degree of v in each partition, i.e., metadata associated with v, is changed. The algorithm continues moving vertices until an iteration in which there is no candidate vertex for migration, i.e., further movement of vertices does not improve edge-cut.

The following example illustrates how the greedy repartitioner works.

Example: We show two iterations of the repartitioning algorithm on the graph of Figure 3.4 in which there are $\alpha = 3$ partitions and the average weight of partitions is 10/3. Assume the value of γ is 1.3. Hence, the aggregate weight of a partition needs to be in range $[2.3, 4.3]$; otherwise the partitioning is overloaded or underloaded. Figure 3.4a 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 greedy 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 swap behavior previously discussed. At the end of the first stage of the first iteration, the state of the graph is as presented in Figure 3.4b. 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 3.4c). In addition, such migration would cause partition 1 to be underloaded (its load will be 2 which is less than 2.3). In the second iteration, vertex *c* is migrated to partition 2. The result of the first stage of iteration 2 is presented in Figure 3.4d. At this point, the graph reaches an optimal grouping, thus the second stage of the second iteration will not perform any migrations. In fact any further iteration would not migrate anything since the graph has an optimal partitioning.

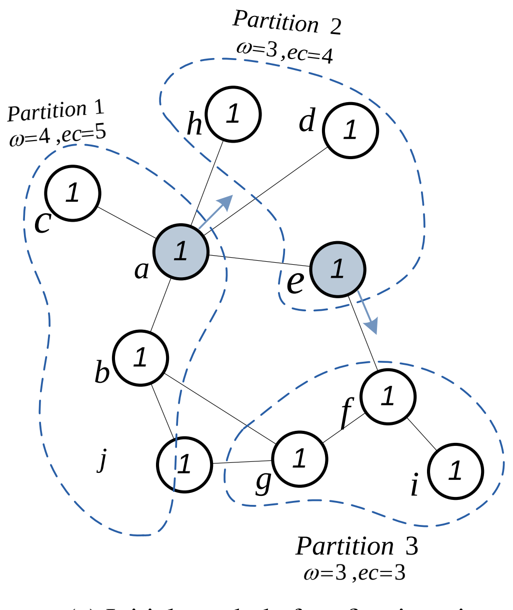
3.2 Physical Data Migration

Physical data migration is the final step of the repartitioner. Vertices and relationships that were marked for migration by the repartitioner will be 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).

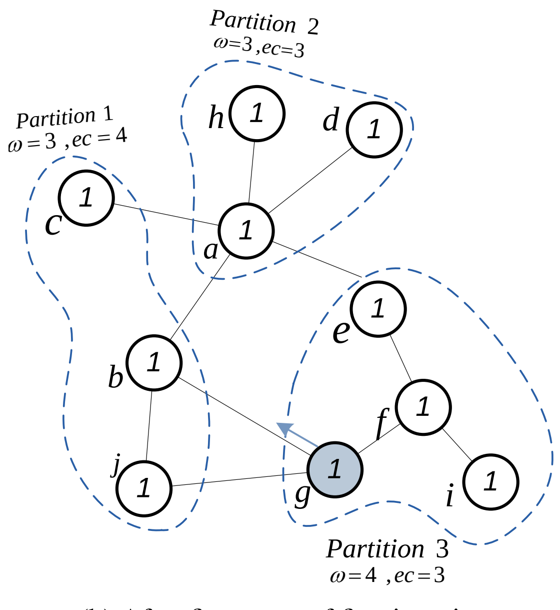
This two stage process can be compared to streaming. In the first stage, a list of all vertices selected for migration to a partition are received by the partition, which will request these vertices and add them to its own local database. Essentially, at the end of the first stage, all moved vertices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

Between the two stages there is a synchronization process between all partitions. This process takes place to make sure that partitions have completed the copy process before starting to remove 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 stage. The synchronization itself is not expensive, though partitions may need to wait until a straggler finishes copying.

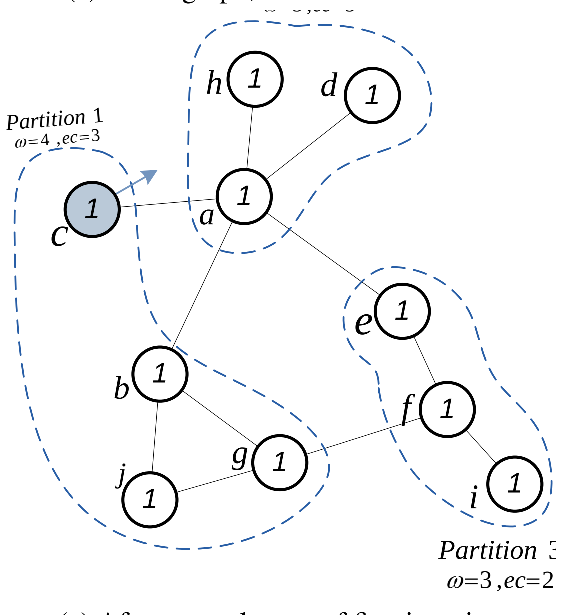
The remove stage takes advantage of replication to decrease the number of operations and their impact on the system. First, all marked vertices will enter an unavailable state. When a vertex enters this state, all queries referencing the vertex will be executed as if the vertex is not part of the local 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. One example is if two related vertices are moving from the local partition. In this case the relationship between the two can simply be deleted rather than first changing it to a reflect the migration of one end, then deleting it.



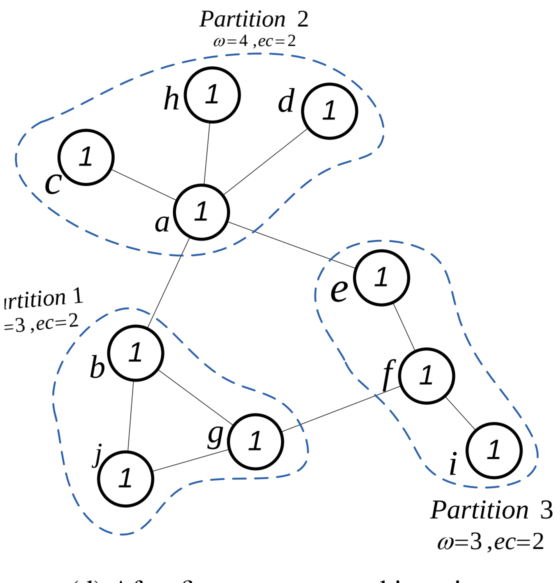
(a) Initial graph, before first iteration



(b) After first stage of first iteration



(c) After second stage of first iteration



(d) After first stage, second iteration

Figure 3.4: Example of 2 iterations of repartitioning. 2 metrics are attached to every partition: *w* representing the weight of the partition and *ec* representing the edge-cut.

3.3 Algorithm Analysis

3.3.1 Memory and Time Analysis

Recall that the main advantage of the greedy repartitioner over multilevel algorithms is that it makes use of only repartitioning metadata to perform repartitioning. Metadata has a small size compared to the size of the graph. This is formalized in what follows.

Theorem 1. *The amortized size of metadata (number of integers) stored on each partition to perform repartitioning is $n + \alpha$ 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 metadata 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 metadata associated with vertices in each partition is n. Beside the number of neighbors in each partition for each vertex, the metadata includes aggregated weight of all partitions. The aggregate weight of each partition can be stored in a constant number of integers, equal to the number of partitions, α. □

As mentioned in Chapter 2, multi-level algorithms do repartitioning by looking at adjacency lists of vertices, which might be a large fraction of number of *edges*. In contrast, an implication of the above theorem is that the size of metadata used by greedy 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 [13] which implies that the number of edges is roughly 65 times the number of vertices. Hence, the memory requirement of greedy 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 5.4.

Theorem 2. *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 if 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 $O(\alpha n_s)$ time, thus 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 nodes 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 around 10 iterations, while there are millions of vertices in the graph data sets). To conclude, the time complexity of greedy repartitioner is linear to the number of hosted vertices; this makes the algorithm much faster than its multilevel counterparts.

3.3.2 Parallelism

The greedy 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. Each iteration of the algorithm (see Algorithm 3.3, lines 3-10) is executed in parallel on each server. Ater finding all the candidate vertices, the metadata information is moved between partitions (logical migration). In this stage, the repartitioner reads each vertex in the neighbor list to update the metadata (Line 10 of Algorithm 3.3). This process is more expensive as it interacts with the database engine to read the neighborhood list of hosted vertices. To improve the performance of the repartitioner, this stage has been further parallelized on each server (by sharding the set and using multiple threads to fetch the data), allowing the algorithm to finish much faster as timing results showed that this part of the algorithm is the second most expensive step (after physical data migration in the second phase). As such, when configuring the repartitioner, it is important to know the maximum throughput a partition can support and choose this parallelization such that it does not have a major performance impact on current user queries.

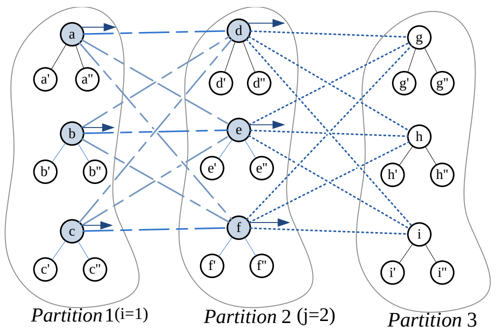
In the second phase of the repartitioning algorithm, physical data migration is performed. This is the most expensive stage because it will require writing the migrated records and deleting them from their old partition. As mentioned in Section 3.2, this stage has been decomposed into two sub-stages for simplicity and performance. Because information is only copied in the first sub-stage, it allows for maximum parallelization with little need to synchronize between servers. In fact, the servers do not need to know of each other's existence and rely on the database's local locking to make sure only one can modify a record. (For clarification, the only issue would be when two servers move vertices that share a relationship and both would try to add the relationship record on the target partition. In this case one server can simply skip the operation since it will be done by the other server.) The second sub-stage is even simpler and easier to parallelize since it only removes information.

3.3.3 Algorithm Convergence

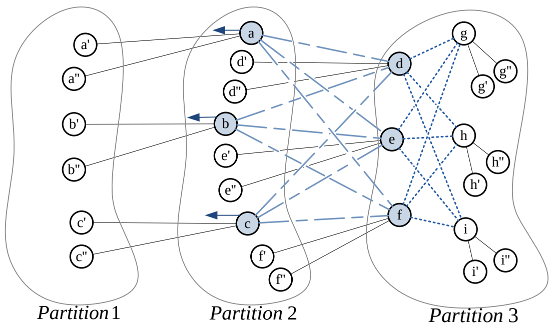
When the greedy repartitioner triggers, the algorithm starts by migrating vertices from overloaded partitions. Note that no vertex is a candidate for migration to an overloaded partition. Hence, after a bounded number of iterations, the partitioning becomes valid in terms of load balance. When there 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 proof for the convergence of the algorithm.

Theorem 3. *After a bounded number of iterations, the greedy repartitioner algorithm converges to a stable partitioning in which further migration of vertices (as done by the algorithm) does not result in better partitionings.*

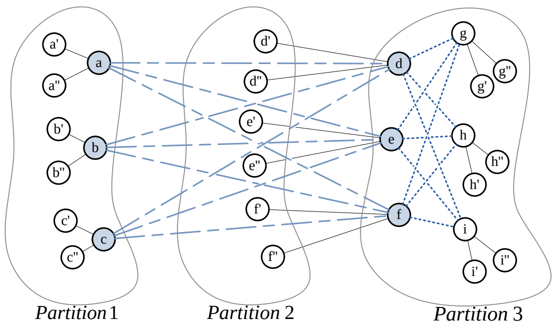
Proof. To prove the convergence, we show that the algorithm constantly decreases the number of edge-cuts. For each vertex v, let $d_{ex}(v)$ denote the number of external neighbors of v, i.e, number of neighbors of v in partitions other than that of v. With this definition, the number of edge-cuts in 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 $d_p^t(v)$ denote the number of neighbors of vertex v in partition p before stage t. Assume that vertex v is migrated from partition i to partition j at stage t (see Figure 3.5). 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 in the same sub-stage (Figure 3.5b). Let $x(v)$ denote the number of neighbors of v in 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 be external after the migration of v. If $d_j^t(v) - x(v) > d_i^t(v)$, the value of $d_{ex}(v)$ is decreased at the end of the stage and we are done. Otherwise, we say a *bad migration* occurred. In these cases, assuming k (the number of migrated vertices in an iteration - top-k) is sufficiently large, in the subsequent stage $t+1$, v



(a) Original graph



(b) After the first stage



(c) After the second stage

Figure 3.5: 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).

migrates back to partition i since there is a positive gain in such a migration (Figure 3.5c), 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 in each two 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 will be constantly decreasing. $\square$

The graph structure in social networks does not evolve quickly and its evolution is towards community formation. Hence, as the experiments confirm, after a small number of iterations, the greedy repartitioner converges to a stable partitioning(where no further migration is done by the algorithm). 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 achieving partitioning with almost optimal number of edge-cuts. However, as mentioned earlier, large values of k can degrade the balance factor of partitioning.To converge quickly, an algorithm should select the value of k so that the edge-cuts improve quickly while the load balance does not suffer. Finding the right 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 the degree of vertices). In practice, it was 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 the experiments, k is set as a small fraction of the number of vertices (typically k is slightly smaller than the imbalance permitted on a partition).

3.3.4 Alternate Optimizations

The repartitioner only keeps track of vertex weights, rather than also having edge weights, due to the nature of the graph operations and due to memory constraints. Given that all operations are in the form of graph traversals, the weight of an edge is the sum of the weights of each ending vertex. Thus edge weights do not need to be stored as they relate to the vertices they connect. In adition, by modifying the algorithm to look at edge weights while choosing the migration candidates would increase the execution time by two orders of magnitude due to the high number of relationships. In addition it adds an inter-partition communication cost, proportional to the number of relationships, during the candidate selection phase.

Chapter 4

Prototype Distributed Graph Database

4.1 Neo4j Description

Neo4j is a centralized graph database system which provides a disk-based, fully transactional persistence engine. The main querying interface to Neo4j is traversal based. Traversals use the graph structure and relationships between records to answer user queries. In terms of transactions, Neo4j is ACID compliant.

4.1.1 Storage

Physical Representation

Internally, Neo4j stores information in three main stores: node store, relationship store and property store. Neo4j splits data into multiple stores for performance and simplicity. By splitting data into three stores, it 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 to hold the properties.

The main advantage of the storage layout is in resource allocation. Typically, the number of relationships is orders of magnitude higher than the number of nodes. Similarly, the number of properties is orders of magnitude higher than the number of relationships. By separating the stores based on their type, Neo4j can better cache critical data and improve query times. To better understand how the storage layout improves performance, we look at a typical use pattern. The typical pattern involves basic traversals over the node and relationship stores. Users access the property store infrequently and, normally, only after they finish a traversal, which restricts their query space.

Taking into acount the patterns, the normal memory allocation tends to focus primarily on node and relationship store caching.

Logical Representation

In order to improve system performance, Neo4j relies on two caching strategies: 1) memory-mapped record cache and 2) object cache. In Neo4j each store type has its own cache in order to fine tune performance of each store independently.

The memory-mapped record cache is an in-memory record cache that keeps track of records in their raw format (physical representation). This is a coarse grained cache which keeps track of blocks of records. This means that when Neo4j needs to read from disk, it will read a block of records even if only one record is required from the block. This is done to improve disk performance. Since disk hard drives read records in block sizes, reading multiple records at the same time is equivalent to reading one record. Since Neo4j may need to read multiple records from the same block, this improves performance.

The second cache level is an object cache. Once Neo4j parses a raw record into an object, the object record is stored in this cache.

Neo4j record objects store only basic information about the record. For example, for a node record, the object would store its ID, the ID of the first relationship and the ID of the first property. When a traversal asks for the node's relationships, it receives an iterator. The iterator will then use the relationship ID to query for a relationship record. Relationship records are part of a doubly linked-list such that the iterator can query for the next relationship using the current relationship and the node ID. Conversions from IDs to objects is done lazily within the objects or within iterators such that any user query or traversal interacts with record objects.

4.1.2 Transactions

While Neo4j is ACID compliant, these properties are infrequently used. By default, all read operations use short lived locks (releasing the lock right after it finishes reading), while queries are generally not associated with a transaction. Due to highly intensive read workloads [33, 12], running queries without a transaction overhead allows the system to scale better with occasional stale query results.

Transactions are enforced when writes happen. Writes need to happen under a transaction since they often modify multiple records. However, based on traffic analysis in [33, 12], most write traffic is update based or addition of new relationships. These types of updates modify only 2-3 records in the worst case scenario, meaning that the transactions used in general are light weight.

4.2 From Neo4j to DistNeo4j

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 Neo4j features are preserved. Figure 4.1 shows the components of Neo4j with the components that were modified to enable distribution in light gray shading while the components in dark gray shading are newly added. Each of these components will be described next. In addition to understanding Neo4j, we would like to mention that extending it to support distribution as we do in this paper needs to overcome several design and development challenges, which are described in the sections that follow.

In DistNeo4j servers are connected in a peer-to-peer fashion similar to that presented in Figure 4.2. A client can connect to any one server and perform a query. Generally user queries are in the form of a traversal. To send a query the client would first request the starting point of the query (the vertex), then send the traversal query to the server hosting the initial vertex. While any server can handle any query in the system, if a server does not have the starting vertex for a query, 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. At the end of the traversal, the results will be returned to the user.

4.2.1 Initial Partitioning

To increase query locality and decrease query response times, the initial data placement process (i.e., which parts of the data graph are placed on which server) needs to be optimized. In distributed graph databases there are two key metrics that need to be optimized to increase performance: (i) minimize load distribution imbalance, and (ii) minimize graph edge-cut (edges

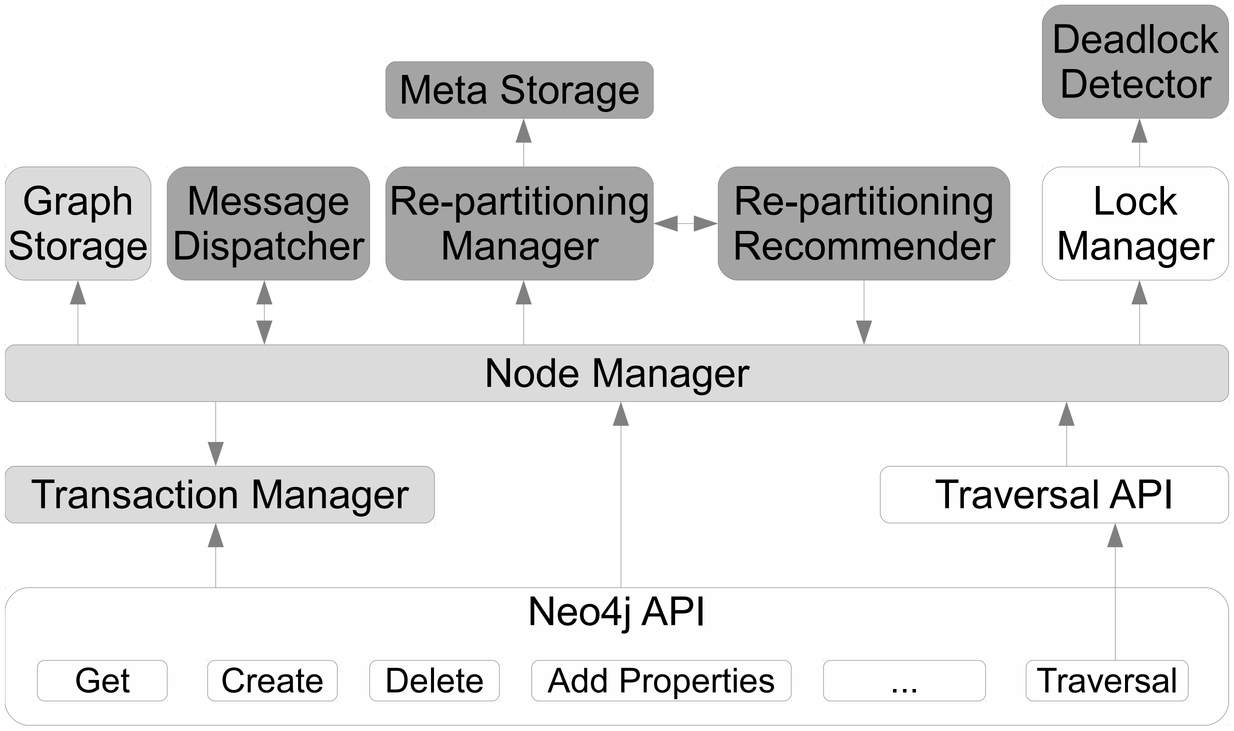


Figure 4.1: Neo4j system layers together with modified and new components designed make it run in a distributed environment.

crossing partitions). The first metric tries to avoid having a small number of servers become hot spots. The second goal tries to reduce the chance of a query having to access remote objects, thereby increasing query (data) locality.

It is important to understand why minimizing graph edge-cut is critical. In social networks most operations are done on the user's data and his neighbours. Since this 1-hop traversal operation is so prevalent in social networks (graph networks), minimizing the edge-cut is similar to keeping communities intact. This leads to highly local queries similar to those obtained in SPAR[45] and minimizes the network load, allowing for better horizontal scalability by reducing network IO in cases such as [5].

Existing graph partitioning tools such as Metis [29] minimize edge-cut. In addition, most of them can also optimize for secondary goals such as (workload) weight balance (or partition balance). I 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 DistNeo4j performs. For the initial vertex weights, a simple representative traffic trace is synthetically generated and used to initialize them

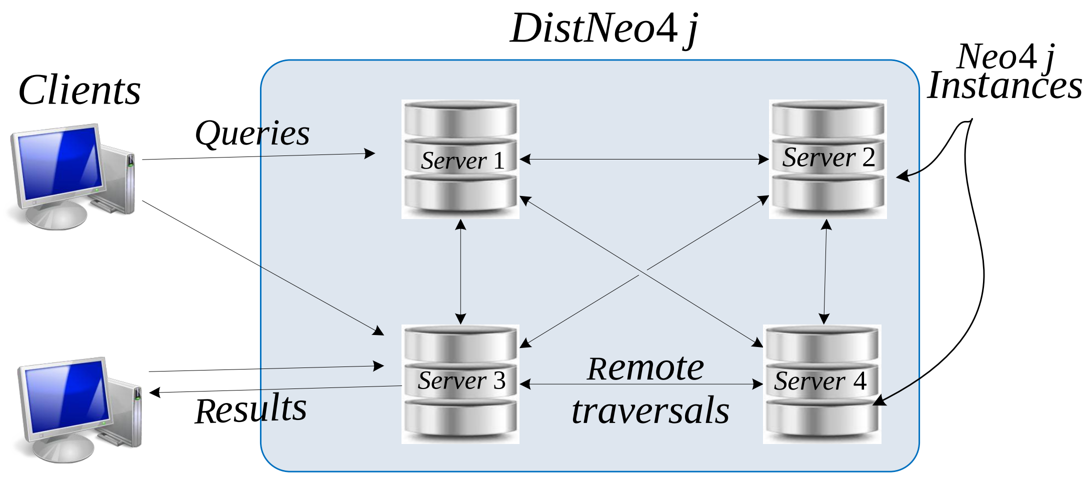


Figure 4.2: Overview of how DistNeo4j servers interact with clients

(see Chapter 5 for a detailed workload and experimental setup). The trace consists of 1-hop or 2-hop traversals where the starting vertex for each traversal is randomly selected from the set of all vertices. This type of trace gives a uniform trace over the graph as each vertex has an equal chance of being selected.

4.2.2 Online Repartitioning

The initial partitioning helps only as long as new data is not inserted or the traffic patterns remain the same. When these change in the workload, the set of frequently requested data also changes. This could lead to more remote traversals when these data are located across partition boundaries. Thus, migrating such data so that it is co-located on the same partition would reduce the number of remote traversals, significantly reducing traversal cost. Two modules were created for migrating data: a recommender module and a repartitioner.

Recommender

The repartitioning recommender is a monitoring process that is part of each DistNeo4j instance. Small skews in access patterns happen often which could lead to small imbalances. However these imbalances often have no overall impact on performance or the performance impact is short lived. As such, migrating data does not need to happen unless the system undergoes a permanent shift in workload. Algorithm 4.4 tries to asses the system state to determine if the repartitioner needs to start due to system changes. *StatsWindow* is a statistics tracking and aggregation process. *StatsWindow* collects performance counters as DistNeo4j answers queries and aggregates them in time-based rolling windows. A rolling window type of aggregation allows the system to keep both fine grained statistics of system performance and long term trends.

It is important to note that social networks are highly connected graphs. In fact, networks such as Twitter and Orkut have very small diameters. As such, even 1-hop traversals can touch a high fraction of vertices. Minimizing the edge-cut helps keep traversals highly local, however not every possible vertex is optimized in terms of locality. Vertices in sub-optimal locations will cause large remote traversals skewing the statistics in the short term. A rolling window-based statistics aggregation system is ideal for identifying and correctly handling these types of occurrences. Generally, a large aggregate window may see no change in average latencies or throughput. However, even if changes are observed, the fine grained windows can be used to identify spikes in the system and account for them in the decision-making process.

Algorithm 4.4 is used at the end of each statistics window to determine whether repartitioning is required. In lines 2-4 the algorithm will check if the amount of new records added to the system are within a lower limit. Since new vertex additions are performed at random, due to lack of connectivity information, the graph engine needs to defer this process to a later time when more information is known.

The second condition checked in lines 5-7 relate to the previously discussed locality test. If the overall query rate is stable but the remote query rate is increasing then it may be necessary to repartition. However, the query rate can be described as increasing only if a certain percentage of windows are observed at the increased rate. For example, if the system keeps 5 statistics windows and requires at least 3 to be at an increased query rate, then when the new window rolls over and it detects an increase, it will classify the query rate as stable. Figure 4.3a shows an aggregate *StatsWindow* that contains 4 windows with stable query rates and the last window with increased rate. The classification is stable such that short lived fluctuations in the system do not trigger a repartitioning. However if the next 2 windows roll and both show an increase (note, the last 3 windows could have similar rates, but they considerably differ from the first 2), then the query rate is reported as increasing. This change is exemplified in Figure 4.3b showing increased stable rates in the last three windows.

The last condition checked, shown in lines 8-10, monitors latencies and query rates. This condition checks that increases in workload on the partition do not affect query latencies negatively. If both are increasing, then it is possible that the partition is overloaded. However, if only the query rate is increasing, then it may be that the load on the partition is low, so there is no reason to repartition. If only the latency is increasing, then it is most likely related to an increase

AggregateStatsWindow						
Expired		Active				
10 q/s	10 q/s	9 q/s	10 q/s	11 q/s	12 q/s	20 q/s

(a) Last window shows increased query rate.

AggregateStatsWindow						
Expired		Active				
9 q/s	10 q/s	11 q/s	12 q/s	20 q/s	21 q/s	19 q/s

(b) Last 3 windows show increased query rate.

Figure 4.3: Example of rolling StatsWindow with changing query rates.

```
procedure NEEDS_REPART(StatsWindow w)
    if w.vtxWrites() $>$ vtx_lim $\wedge$ w.relWrites() $>$ rel_lim then
        return $True$
    end if
    if w.queryRate() is stable $\wedge$ w.remoteQueryRate() is increasing then
        return $True$
    end if
    if w.queryLatency() is increasing $\wedge$ w.queryRate() is increasing then
        return $True$
    end if
    return $False$
end procedure
```

Figure 4.4: Repartitioning Recommender

in remote queries, which should be caught by the previous condition once it becomes critical.

Repartitioner

The greedy repartitioner previously described is used to re-balance the graph. In order to keep the repartitioner's impact to a minimum, a separate metadata store is used to keep track of vertex information used by the repartitioner. The metadata keeps track of the following pieces of information: vertex weight (representing the number of accesses to performed on the vertex) and the number of relationships each vertex has in each partition. This information is collected and updated in real-time based on user information. When the repartitioner is active, it will use the metadata to decide which vertices to migrate, thus decreasing the performance impact it has on the system.

Further, to increase the performance, the repartitioner is executed in a two stage process: (1) the repartitioner is run on the metadata and (2) the actual data is moved based on the first stage results. Instead of a single stage, the algorithm is split in two stages, because border vertices are likely to change partitions often within the first stage and metadata records are lightweight compared to the actual data records.

Dynamic Data Migration

Data migration is the final step of the repartitioner. Vertices and relationships that were marked for migration by the repartitioner will be moved to the new partitions using a two step process: 1) Copy marked vertices and relationships 2) Remove marked vertices and relationships from the old partitions.

The alternative migration process considered was to copy and remove each vertex and its associated relationships one at a time. However experimental results showed that this migration process is more expensive in terms of the throughput performance of the system, takes longer to finish and involves more complex operations to keep the graph consistent. Because each operation would require leaving the graph consistent, it also meant that parallelization of operations can be complex if related vertices moved in parallel.

In contrast, the two stage process can be compared to streaming. In the first stage, each partition receives a list of all the vertices migrated to it. It can then request these vertices and add them to its own local database. Essentially, at the end of the first step, all moved vertices will be replicated. Because of the insertion-only operations in this step, the complexity of the operations is lower as all operations in this step can be performed fully locally to each partition, meaning less network contention and locks held for shorter periods. Note however that each vertex addition is executed within a transaction, thus leaving the graph structure consistent.

Between the two steps there is a synchronization process between all partitions. This process takes place to make sure that partitions have completed the copy process before starting to remove 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 remove step takes advantage of the replicated status to decrease the number of operations and their impact on the system. First, all marked vertices will enter an unavailable state. This allows performing the transactional operations much faster as locks will not be held. In addition, since this operation is performed in a batch like process, it is easy to detect operations that can be collapsed together. One example is if two related vertices are moving from the local partition. In this case the relationship between the two can simply be deleted rather than first changing it to a ghost relationship (described in the next section) and then deleting it.

4.2.3 Storage

Data Representation

To shard data across multiple instances of Neo4j, changes were needed to allow local nodes and relationships to connect with remote ones. Neo4j uses a doubly-linked list record model when keeping track of relationships. As 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 is simply there to keep the graph structure valid. One advantage of this is complete locality in finding the adjacency list of a graph node. This operation is also important since traversal operations build on top of adjacency list. Figure 4.5 shows how relationships crossing the partition boundary appear in both partitions, however only one partition holds the actual relationship; the other holds the ghost relationship.

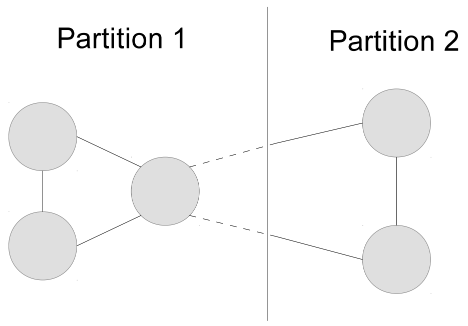


Figure 4.5: Sharded graph - The dotted lines represent the ghost relationships while the continuous lines represent the actual relationships.

The storage layer returns raw node, relationship or property records requested by the *NodeManager* on behalf of the user (query) back to the *NodeManager*. The *NodeManager*, handles all primitive graph operation requests (getNode(nodeId), getRelationship(relId), getProperty(propId)) and takes care of data caching and locking. Neo4j uses *Proxy* wrappers over all (*Node* and *Relationship*) primitives. To support remote primitive query operations, the *NodeManager* was extended to query remote servers if records are not found locally. To provide transparency to algorithms running above the storage layer, the *Proxy* wrappers were also extended to wrap remote objects. These wrappers simply forward requests to the remote partition using an inter-partition communication module presented in a later section. Thus, any high level algorithm does not need to be modified as operations are performed on only *Proxy* objects.

B+Tree Indexed Store

The Neo4j storage model revolves on a fixed size record model. Each record is assigned a unique identifier (ID) which will then be used to link different records in a graph structure. Neo4j uses a sequential ID generator, this model allows for simple indexing scheme based on *ID* offsets. Thus reading or writing a record is done by seeking at an offset *ID * record size* in the store file. This indexing model is ideal in a centralized system as any lookup requires only constant time operations and has no storage overhead.

In a distributed system, *ID* generation can no longer be sequential within each partition as they are used to uniquely identify a record in the distributed system. Thus a different storage model is necessary. The chosen solution was to use a B+Tree index as it provides good performance, is scalable, fits Neo4j's fixed size record model and B+Trees have been well studies in previous work, and the default choice for most database systems.

Alternative designs have been assessed, however each has its own limitations. The following list briefly describes the designs considered:

1. **Range based generator.** Each server would have some ID range allocated to itself. This ID range could be fixed size or servers could periodically synchronize and allocate ID ranges. The initial problem is each partition would have large chunks of records unallocated since they belong to a different partition, leading to large storage files. Worse, as data migrates, allocated records will be sparse throughout the storage which will ultimately impact caching and disk seek performance.

2. **Local offset embedded in global ID.** In general this system involves only one ID generator. Generally some high bits of the ID will represent the partition ID while the lower bits will represent the local ID. This method is ideal in static graph databases. However data migration will require changing the id. Since the record may be referenced by all the partitions, then this method would involve a global synchronization step to update all

references. Worse, active queries need to be updated as they potentially cached outdated information.

3. **Global ID generator with decoupled indexed store.** IDs are global entities with no relationship to the local offset. This means keeping a separate index to map record IDs to record offsets. This solution involves more lookups to read a record, however, in practice there is only a constant number of extra lookups. This solution improves the storage of data as all records will be highly clustered at the cost of extra lookups and higher latencies.

The last design is the chosen solution and was implemented as mentioned above.

4.2.4 Inter-Partition Communication

Communication between partitions is done in a peer-to-peer fashion. When a DistNeo4j server (peer) starts up, it initializes a local discovery service that listens for other servers (peers). When a server is found it creates a connection to that server. This connection is then used to transfer messages. Messages between servers is sent in a command/response design pattern. To minimize changes to Neo4j, some of the basic classes (such as *Node* and *Relationship*) used in queries have been extended to support remote queries. Each operation they perform translates to a command sent to a remote server. The remote server handles the request and, optionally, sends back the requested data, if any. This reduces the amount of changes to the underlying systems, which in turn makes remote queries transparent to the querying system.

4.2.5 Deadlock Detection

Since requests can span multiple servers, distributed deadlock detection is required to break such multi-server deadlocks. To detect deadlocks, Neo4j performs loop detection on the resource graph before attempting to acquire a resource. If a loop is detected the acquisition fails and the current transaction fails. This approach is well suited for centralized systems but the amount of state, cooperation and synchronization between different servers would be very high, making it impractical to implement graph-based deadlock detection in a distributed system.

For deadlocks that span multiple servers in DistNeo4j, we use a simple timeout-based detection scheme [14]. There are several advantages to using timeout-based deadlock detection over other, more complex methods. First, it is decentralized and requires no synchronization between different servers, making it highly scalable. The memory and computational overhead is minimal since it requires keeping track of timeouts only for locks. The timeout-based deadlock detection scheme is triggered only when a deadlock is detected whereas graph-based systems proactively check for deadlocks. The proportion of writes in typical graph workloads is low, making it highly unlikely for deadlocks to occur.

4.2.6 Transaction Manager

In a distributed system, transactions can span multiple servers, thus the system needed to be modified to support distributed transactions. To provide the same ACID properties as Neo4j, the two-phase commit (2PC) [14] protocol is used. For simplicity and load balance, the coordinator is the server starting the transaction.

Generally, 2PC is very expensive and only used when consistency is required. Thus, it is important to assess the impact of 2PC on DistNeo4j. As previously mentioned in Section 4.1.2, transactions are infrequent in Neo4j and used only when writes happen. As such, 2PC will impact only writes (small subset of queries). Further, because writes are very local in nature, the effect of 2PC would have little impact on performance. It is also important to mention that 2PC would be used only for a subset of write transactions. Since 2PC is used to synchronize transactions over multiple servers, only writes that require access to multiple partitions would be affected. The only write operation that requires such changes is the relationship addition operation where the two nodes it connects are on different partitions. Thus, it is expected that the effects of 2PC will have minimal impact on read traffic and only affect a subset of writes. Since one of the main goals of the partitioner is to minimize the number of links between partitions, the number of transactions requiring 2PC will decrease even further.

In DistNeo4j, distributed transactions are implemented on top of local transactions. Any transaction that needs to be executed on multiple partitions are tracked by a special transaction manager. The initial partition where the transaction starts is considered the *master* for the transaction. Its transaction ID and partition ID are associated with the remote request. The remote partition creates a local transaction ID and associates the pair {master transaction ID, partition ID} with the local transaction. All inter-partition communication that happens in the transaction will contain the above mentioned pair which is used to identify the transaction globally. Locally, the distributed transaction manager will convert between global ID and local ID. The local ID is used to take advantage of the existing infrastructure. When a distributed transaction commits, the 2PC algorithm is executed and data is committed.

4.2.7 Recovery

Neo4j uses a logical log for recovery purposes. Write operations are initially written to a log file, then committed. If the system crashes, Neo4j can simply replay the log to fix any inconsistencies.

Similar to Neo4j, DistNeo4j uses a the logical log to recover from crashes. DistNeo4j extends logging in order to ensure distributed transactions are recovered correctly. Since DistNeo4j builds distributed transactions on top of local transactions, the only changes required to the log is keeping track of which transactions are distributed. To achieve this goal, the global transaction ID previously mentioned is written to the log. Upon recovery, if the transaction is distributed, recovery is handled by the distributed transaction manager which assumes abort as the default behavior. I use abort as the default behavior since it simplifies recovery. In this case any peer can be brought back to a consistent state without having to communicate with any peer.

Chapter 5

Evaluation

5.1 Experimental Setup

All experiments were executed on a cluster with 16 servers. Each server has the following hardware configuration: 2 AMD Opteron 252, 8 GB RAM and 160GB SATA HDD. The servers are connected using 1Gb ethernet. In each experiment one DistNeo4j instance runs on its own server.

5.2 Datasets

The Orkut social graph [40, 6] is a collection of users and relationships between them. It contains over three million users and 223.5 million relationships. This dataset has symmetric links, meaning that connected users know each other and information flows both ways on the links.

DBLP collaboration network [59, 6] is a co-authorship network for research publications. It contains 300 thousand authors and 1 million edges. This dataset is also fully symmetric. One interesting aspect of the DBLP dataset is despite the low relationship count, its clustering coefficient is relatively high. In fact, as seen in Table 5.1, the clustering coefficient is much higher than Orkut or Twitter, strongly signifying the presence of highly clustered communities.

Twitter social network [61] is a social graph of twitter users. The graph contains 11.3 million users and 85 million directed relationships. In contrast with the previous 2 datasets, only 22% of relationships in Twitter are symmetric.

Table 5.1 shows a detailed description of some structural statistics for the three datasets. It is notable that both social networks (Twitter and Orkut) share remarkably similar parameters.

One of the most interesting metrics is the average path length, which in both datasets is very low (4). The small path length signifies high connectivity between users in the datasets. DBLP does differ in path length, but the large difference is attributed to the strong community. The last metric shown in Table 5.1 is the power law coefficient. All three datasets show power-law distributions similar to those expected in scale-free networks and is very similar to values found in other social networks [46, 58]. However, as mentioned in [40, 33], social network graphs tend to deviate from the power law distribution when looking at the most popular users. One reason that is cited is the inherent limits in the systems which initially had upper limits on the number of relationships a user can have.

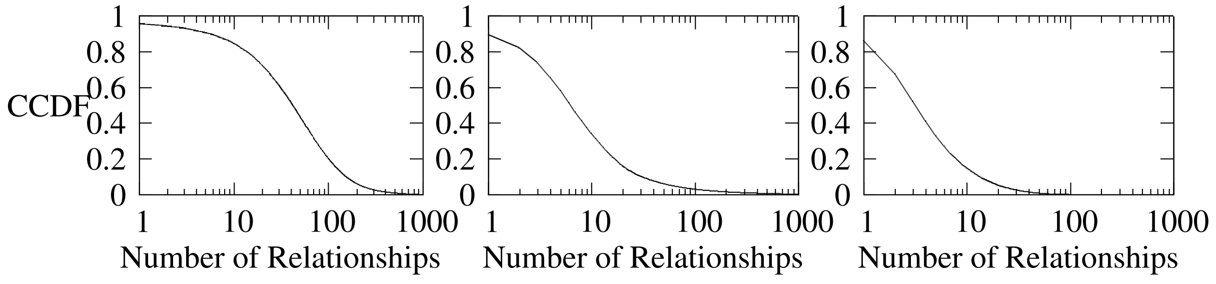
In Figure 5.1 the CCDF of the relationship distribution of the datasets is present. All datasets exhibit a similar power law distribution. Most users have a relatively low number of relationships and only a small fraction are very popular. For example, the Orkut dataset shows that only 50% of the users have more than 50 friendships, while only 20% have more than 100. Similar results can be seen in Twitter, though users tend to follow slightly more users. DBLP is the only dataset where the CCDF abruptly falls. In DBLP a majority of the users have worked with only a few other authors. In fact 50% of authors have only collaborated with 3 or less other authors and 80% with 8 or less authors.

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

Table 5.1: Summary description of datasets

5.3 Experiment Description

The experimental evaluation is based on previous work in the graph database field[17] and based on real-world workloads [33, 12, 45, 41] used to describe social network queries. Social networks queries focus mostly on looking up friends' profiles and updates or recommendations (friend or ads recommendations)[12, 33]. These query times map onto 1-hop and 2-hop traversals, thus the focus of the experimental evaluation will be on performance of these query types.



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

Figure 5.1: CCDF distribution of number of relationships over data sets

In addition, [17] proposes three benchmark experiments for graph databases: 1) data loading (or pure write experiments), 2) 2-hop traversals and 3) 3-hop traversals. Out of these three experiments (1) does not make sense from a system's perspective since data loading is often done offline and is generally performed without running the actual database engine. It is normally performed by an optimized version without transactions or locks. This is true for DistNeo4j as well since I employ a specialized loader. Instead of a pure write workload, the experimental evaluation will employ a mixed read/write workload where the percentage of writes is varied to test the overhead of write queries on the system. (3) is not performed since, as shown in 5.1, the graph diameter is just over 4 hops. This would mean that most 3-hop traversals would read the full graph or a high percentage of the vertices in the graph. Since the scope of the system is optimizing small, localized queries, this experiment is not suitable for showing improvements in query throughput and latency since the same amount of network traffic would be required in querying all vertices on all partitions.

Further, the experimental evaluation also looks at scale-up and scale-out results in order to confirm that DistNeo4j can handle increasingly more concurrent client queries and increasing number of servers.

5.4 Repartitioner Experiments

For performance comparison and improvements, the repartitioner is compared with two different partitioning algorithms. For an upper bound the offline Metis[29] partitioner is used. Metis is a popular partitioner that generates high quality partitions. It is also flexible enough to allow custom weights to be specified and used as secondary goals. Note that ParMetis, can be used, however the memory requirements would not be better than Metis'. For the same reason ParMetis can not be used to replace the repartitioner.

I also compare 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.

It is important to understand why Metis cannot be used as a repartitioner. First, Metis is an offline static partitioning algorithm meaning either allocating additional resources to partition and re-load the graph every time the partitioner is executed, or taking the system offline to perform data loading on the production servers. If the server would be taken offline, it would take 2 hours to load the Orkut or Twitter graphs. This period of time is unacceptable for production systems. Alternatively, if DistNeo4j is augmented to take as input Metis-partitioned graphs, there would be 2 performance issues: 1) the number of vertices and edges migrated would be much higher and 2) the resource overhead for running Metis would be much higher than that for running the repartitioner. The first issue will be presented in the next section in detail, while the second issue arises from the information Metis and the repartitioner store. Metis' memory requirements scale with the number of relationships and coarsening stages while the repartitioner scales with the number of vertices and partitions. Since the number of relationships dominates by orders of magnitude, Metis will require a lot more resources. As an example, Metis requires around 23GB of memory to partition the Orkut dataset, while the repartitioner only requires 2GB.

5.4.1 One-hop Performance

The following experiments start with a partitioning based on 1-hop traversals, 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 improvements. This shift in workload is caused by a skewed traffic trace where 10% of users on one partition are randomly selected as starting points for traversals twice as many times as before, creating a hot-spot on a partition. This workload skew is used for the full duration of the experiments that follow.

The 10% skew is chosen such that the skew impact on throughput is noticeable. In addition, the skew was not chosen higher since it would be unrealistic as the repartitioner would have triggered earlier and the skew would have never gotten to that level.

As mentioned before, minimizing the edge-cut is a good metric when analyzing query locality and the potential impact of locality on performance. Figure 5.2 presents the relative edge-cut based on the execution of the repartitioner and Metis on the skewed data. While some differences are noticeable in all datasets, it is important to note that the difference in edge-cut is less than 3%

in all datasets, showing that the quality of the repartitioned data is close to the quality resulting from a leading partitioner.

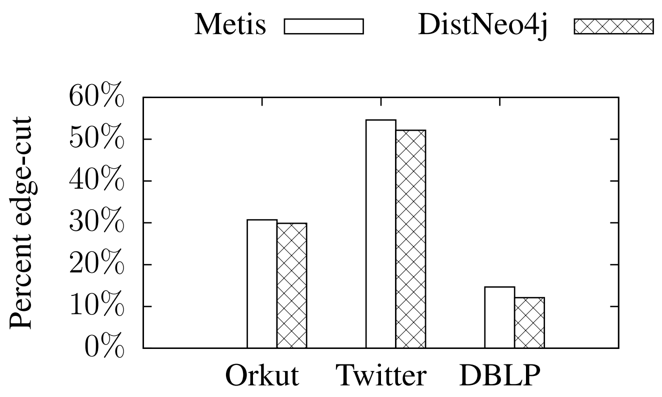


Figure 5.2: Compares the edge-cut resulting from repartitioning with running Metis. Results are presented as a percentage of edges cut from the total number of edges in each specific dataset.

While the above results show that the quality of the resulting partitions are fairly high, the repartitioner's performance is also affected by the amount of information that it needs to migrate. As such, the partitions resulting from running the repartitioner and Metis are compared with the initial partitioning used. Figure 5.3 shows the number of vertices migrated due to the skew based on the two partitioning algorithms. The results show that the proposed greedy repartitioner requires the migration of significantly fewer vertices. For example, for the Orkut dataset, Metis migrated 30% of the vertices to a different partition while the greedy repartitioner only migrated 5%. Thus the repartitioner did not have to migrate 25% of vertices.

Further analysis was done to see how many relationships would be affected by migration. Figure 5.4 shows that the greedy repartitioner requires significantly less changes to relationships compared to Metis. For example, for the Orkut dataset, Metis migrated 60% of relationships while the repartitioner migrated only 7%. Thus, by using the repartitioner, 53% of relationships did not have to be migrated.

Overall, both the number of vertices and relationships migrated is important as they directly relate to the performance of the system. A lower number of migrated vertices and relationships means fewer reads or writes to the database and smaller network footprint. These savings can be translated to increased throughput for user queries.

Figure 5.5 presents the throughput over time of a 16 partition setup using the Orkut dataset. In these experiments 32 client workers submit 1-hop traversal requests using the skewed traffic

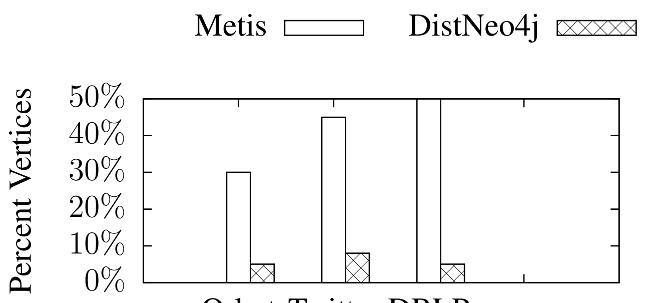


Figure 5.3: Compares the number of vertices migrated when using the repartitioner versus Metis.

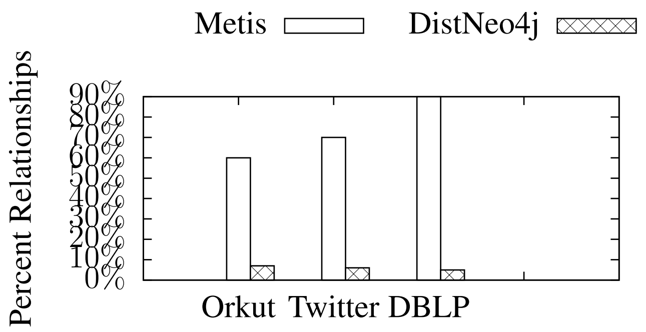


Figure 5.4: Compares the number of relationships changed or migrated as a result of repartitioning versus running Metis.

pattern previously mentioned. The first experiment, marked as *Metis*, shows the system's performance if Metis partitioned the data using the skewed trace. This experiment is considered ideal from a performance perspective since Metis partitioning produces high quality partitions. The second experiment, *Repartitioner* shows the performance overhead of the repartitioner and the gains over time. This experiment starts with the an initial partitioning based on a trace with no skew. The initial partitioning was done using Metis offline. The results shown in Figure 5.5 for *Repartitioner* are running a skewed trace. The initial drop in performance observed is due to the repartitioner triggering and computing a new optimal partitioning. Since this operation requires access to the relations of a subset of vertices, it has a noticeable impact on performance. Once a new partitioning is computed the performance of the system increases and comes close to *Metis*'s performance. The final experiment is *Random*, which uses random hash-based partitioning, and performs poorly compared to our repartitioning algorithm.

To better understand the performance implications of the skew and repartitioner in the above

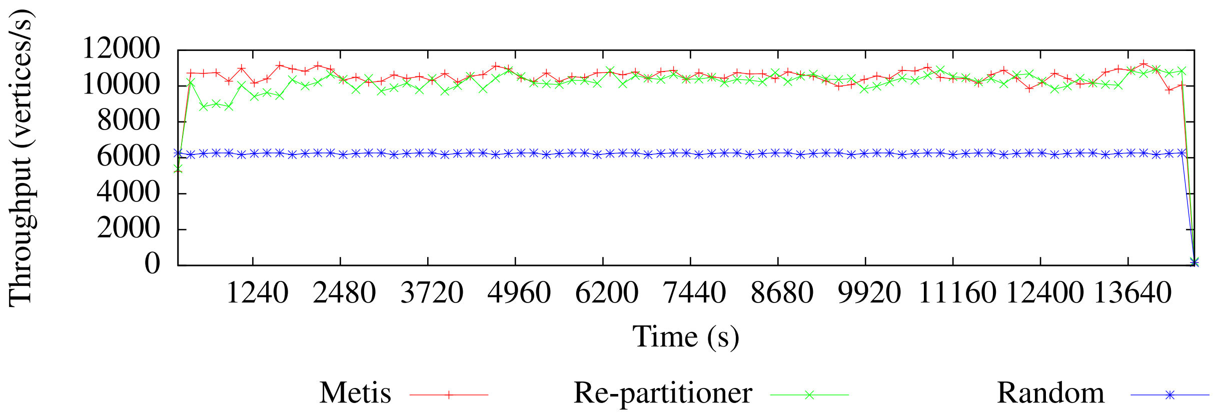
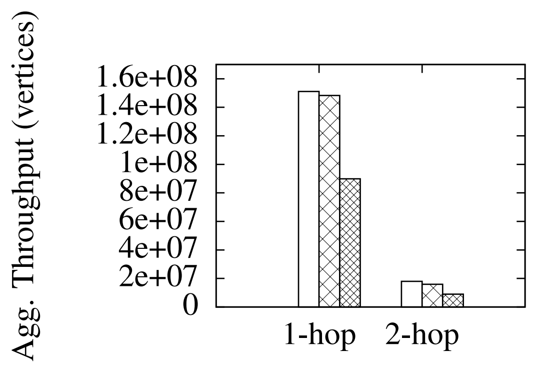


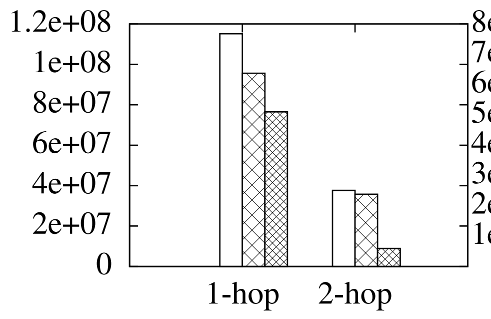
Figure 5.5: Throughput performance over time for Orkut dataset

experiments, Figure 5.6a presents the total throughput of each experiment. Results show that by introducing the skew and triggering the repartitioner a 6% drop in throughput is observed between *Metis* and *Repartitioner*. Further, comparing *Metis* and *Repartitioner* to *Random*, the performance drop is much more significant, at 1.7 times lower than *Metis* and 1.64 times lower than *Repartitioner*.

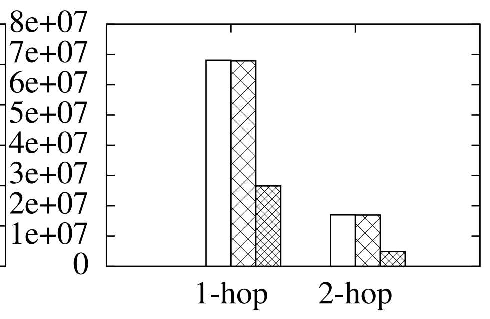
While the above aggregated results include the overhead of the repartitioner, Figure 5.6d presents the aggregate throughput for the last hour of the experiments to show the throughput gains after repartitioning finishes. These results show a 1% difference in performance compared to *Metis*, showing that the repartitioner does a good job at redistributing load.



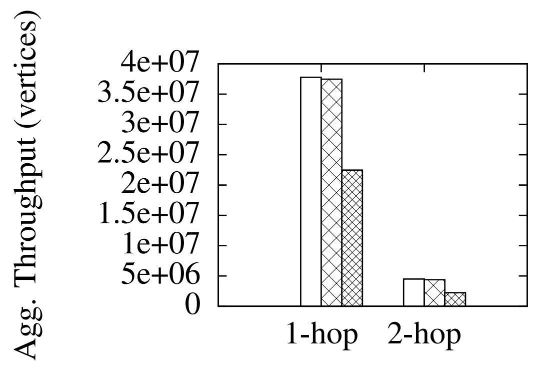
(a) Orkut



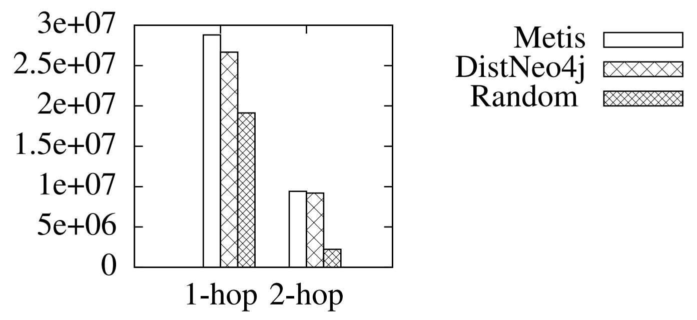
(b) Twitter



(c) DBLP



(d) Last 1 hour Orkut



(e) Last 1 hour Twitter

Figure 5.6: Aggregate throughput results for datasets. The first three, 5.6a, 5.6b and 5.6c, show throughput aggregate over the entire experiment, while 5.6d and 5.6e show the aggregate over the last hour of the experiment

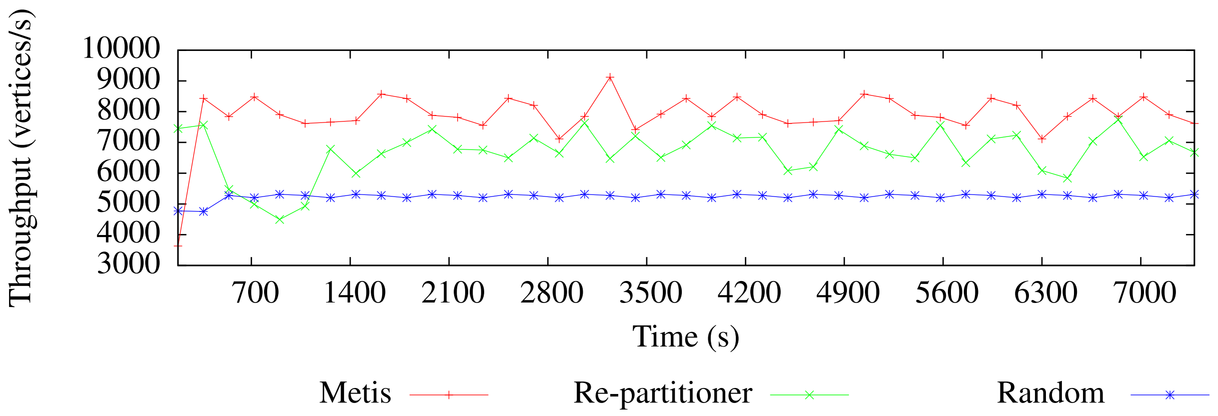


Figure 5.7: Throughput performance over time for Twitter dataset

Similar experiments were executed on the Twitter dataset. Figure 5.7 shows a similar drop in performance as the repartitioner triggers and starts data migration. As the run progresses, the migrated data starts improving the performance of the system. Figure 5.6b shows the aggregated throughput while running with the Twitter dataset. The results show very similar performance between the repartitioner and Metis. To better understand the performance gains from running the repartitioner, Figure 5.6e shows the aggregated throughput over the last hour of the experiment, showing a 8% improvement in performance. Lastly, DistNeo4j partitioning performs 1.5 times better than the randomly partitioned run. Note, the reason the Orkut runs performed better than the Twitter runs against random partitioning is due to the lower relative edge-cut in the Orkut dataset (the edge cut in the Orkut dataset is around 30% while in Twitter is around 55%).

Finally, in the DBLP experiments, the relatively small changes required by the repartitioner meant almost no performance degradation due to the repartitioner. Figure 5.8 shows that performance of the repartitioner is very close to that of Metis. In fact, based on results from Figure 5.6c, the performance difference is minimal. Interestingly, the DBLP dataset is the only dataset where performance differences are not noticeable. The similar performance is attributed to the highly clustered and well partitioned dataset. Given an edge-cut of 15%, the high query locality means that partition skews have little effect on performance as it does not shift workloads towards partition borders.

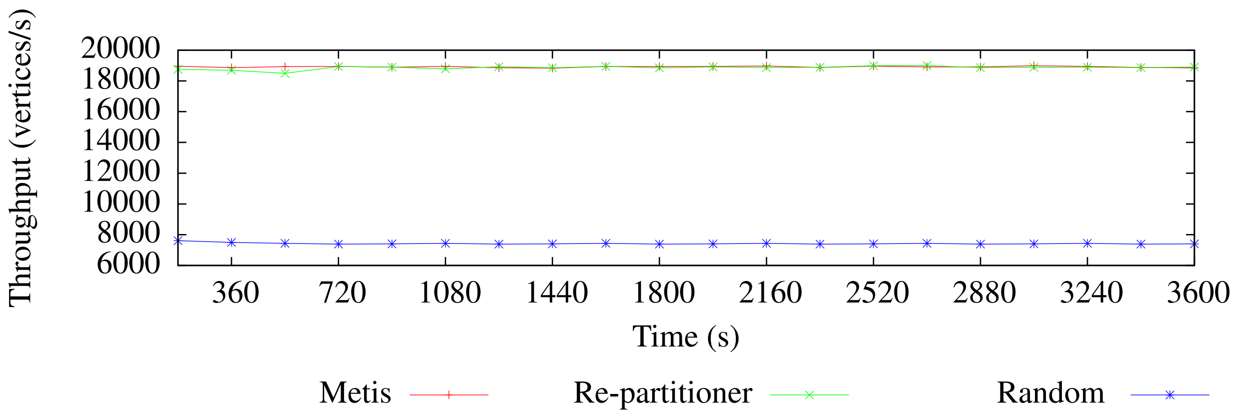


Figure 5.8: Throughput performance over time for DBLP dataset

5.4.2 2-hop Traversal Performance

The previous section focused on performance of 1-hop traversals since 1-hop is the most popular type of traversals in social networks. In order to fully test the performance and quality of the repartitioner, this section will focus on 2-hop traversals.

Similar to the previous experiments, Figures 5.9, 5.10 and 5.11 show the throughput over time of the skewed workload on the Orkut, Twitter and DBLP datasets. Similar to previous experiments the repartitioner's positive impact can be observed as data is migrated. Further, to analyze the overall performance impact, Figures 5.6a, 5.6b and 5.6c show the aggregate number of vertices traversed throughout the experiments, while Figures 5.6d and 5.6e show the aggregate number of vertices traversed in the last 1 hour of the experiment. These results show a 2 times, 4 times and 3 times improvement over Random partitioning and comparable results to Metis.

One of the striking differences in the performance graphs presented in 1-hop and 2-hop traversals is the decrease in performance and increase in variance in the 2-hop case. To analyze why these differences occur, Table 5.2 presents the locality of queries when using Metis and Random partitioning. These results show the large impact of partitioning and show that increasing the number of hops has a significant negative impact on query locality.

Further, I looked at how the number of hops impacts the ratio between the number of unique vertices queried by the database and the number of vertices queried. The results show that doing 2-hop traversals, about 55% of vertices are duplicates of already queried vertices. The high percentage of duplicates is due to the highly clustered nature of the graphs. Because related users share relationships with other users, 2-hop traversals will visit users multiple times. Since

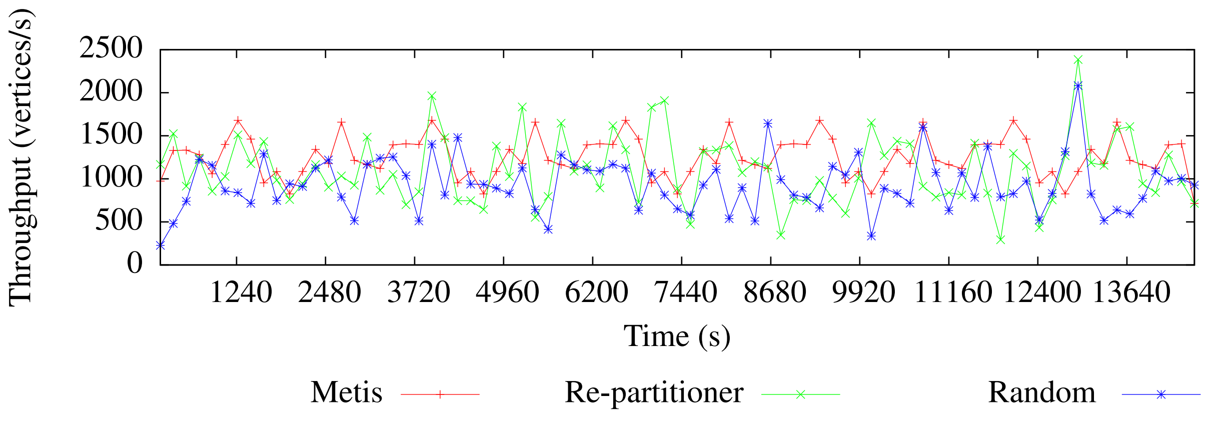


Figure 5.9: Throughput performance over time for Orkut dataset using 2-hop traversals

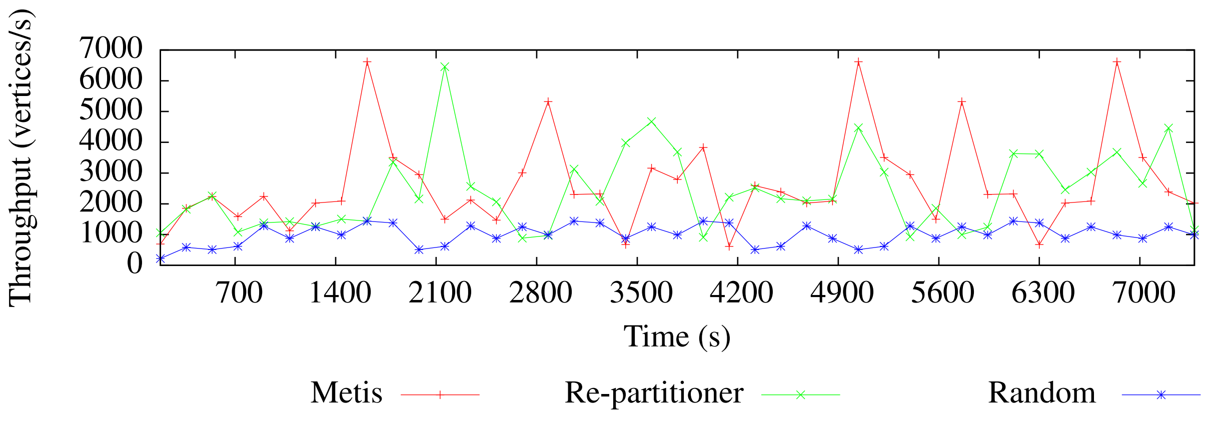


Figure 5.10: Throughput performance over time for Twitter dataset using 2-hop traversals

	1-hop	**2-hop**
Metis	70%	28%
Random	7.5%	6%

Table 5.2: Query locality

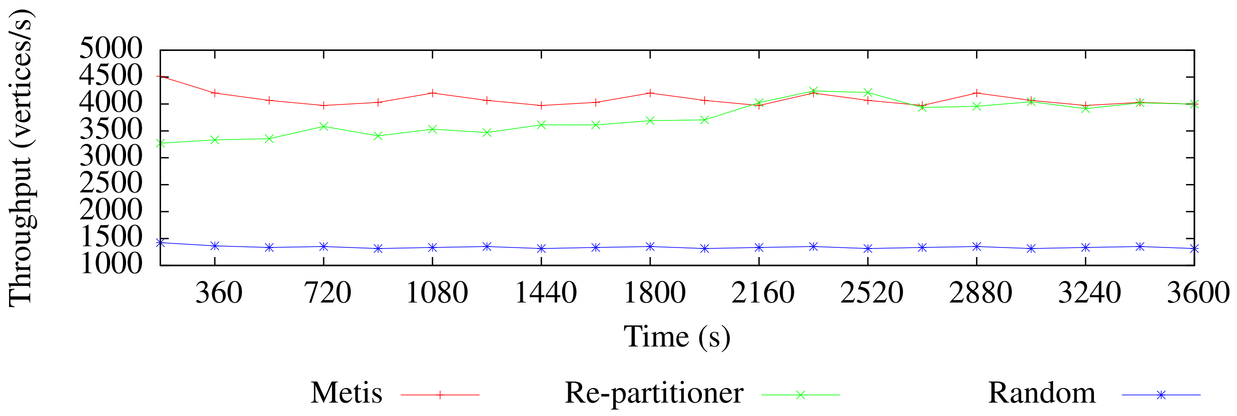


Figure 5.11: Throughput performance over time for DBLP dataset using 2-hop traversals

	Orkut	**Twitter**	**DBLP**
1-hop	76.63	13.57	7.64
2-hop	15591.23	140558.58	88.2

Table 5.3: Comparison of average traversal size using 1-hop and 2-hop traversals

the graph database optimizes the query response and only sends unique copies of each record, the clients show a decreased throughput. The decrease in query locality and increase in the percentage of duplicate queried records strongly correlate with the performance drop observed in the experiments.

Another reason for the spiking effect is caused by the server-side traversal processing model. In this model, the traversal is processed on the server side and results are sent to the client only when the full traversal completes execution. Thus varying traversal sizes will have an impact on how fast traversal results are returned to the user. Since the graphs for throughput performance over time are measured from a client side perspective, increases in traversal size will mean that over short time spans, throughput will vary. Table 5.3 compares the average traversal sizes of each dataset. Noticeably, the largest increases happen in the Orkut and Twitter datasets, with an over 10000 times increase in the Twitter dataset. The large difference between the three datasets is caused by the clustering characteristics of the datasets. While DBLP exhibits high clustering (thus users tend to be connected within only one community), Orkut and Twitter have less well defined communities, thus 2-hop traversals will have a higher probability of visiting multiple communities.

5.5 Variation of Repartitioner Parameters

This section will present experiments which vary different configuration parameters for the repartitioner to show how performance of the repartitioner is affected and how overall system performance is affected.

In Chapter 3, the greedy repartitioning algorithm used two key configuration parameters to restrict the number of vertices migrated and to terminate the repartitioning algorithm. These two parameters are: maximum number of vertices migrated in an iteration (the k in top-k) and the number of iterations. The following experiments will vary these parameters to evaluate their impact on system performance and partition quality.

The following experiments were executed using the same setup as the 1-hop experiments. The experiments measured the throughput while repartitioning and the partition balance.

5.5.1 Varying Number of Migrated Vertices per Iteration

For this set of experiments the repartitioner configuration parameters were set to 10 iterations and the *top-k* parameter was varied using the following values: 500, 1000 and 5000.

Figure 5.12 presents the above three experiments as *Auto 1x*, *Auto 2x* and *Auto 10x*. There are two types of runs for each experiment. The run marked as *During* shows the throughput rate while the repartitioner is active throughout the run. The run marked as *After* shows the performance after the *During* experiment finished and the repartitioner is turned off. This run is used to show the performance gain from repartitioning. In addition to these three experiments, two other experiments were executed for a baseline comparison. Experiment *No Repart* shows the performance of the system without any repartitioning. *Metis* shows the performance when a good partitioning is obtained while partitioning offline after the 10% vertices are inserted. *No Repart* is used for a lower bound comparison for repartitioning overhead, while *Metis* is used for an upper bound comparison.

Figure 5.12 shows that during repartitioning the throughput overhead of the repartitioner, compared to *No Repart*, is 6%, 4% and 3.7% respectively. After the repartitioner finishes, the experiments show a 7%, 3% and 6.5% improvement over *No Repart*. Compared to Metis, *Auto 1x* matches Metis' performance, while the other two experiments are at 4% and 0.5% difference from Metis.

There is one interesting behavior in the above experiments which comes as counter-intuitive: performance of the system suffers the least as the *top-k* parameter is increased. To better understand why this behavior happens, Figure 5.13 presents the throughput over time while the

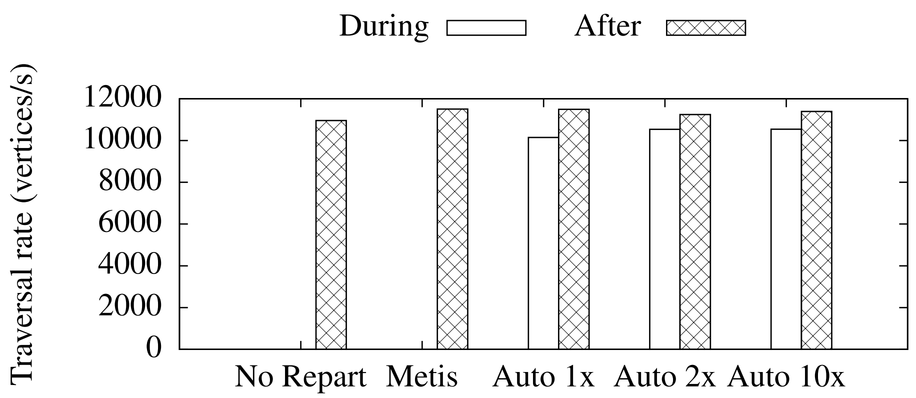


Figure 5.12: Performance comparison while varying the *top-k* parameter

repartitioner is active. The larger spikes visible in Figure 5.13 are due to different repartitioning runs being triggered. This is less obvious in *Auto 1x* as the initial repartitioning stage is relatively short, so the performance impact of this stage could be similar to that of vertex migration. *Auto 10x* shows a clearer difference between the first stage of the repartitioner and the migration stage. The large dip in performance is due to the repartitioner's first stage. Since it requires graph information for the border vertices, it queries the graph introducing some overhead. Interestingly, the migration stage does not show performance degradation. It shows performance gain relatively early. This positive performance gain is attributed to how vertex migration is performed. Vertices are migrated in order of locality. Vertices with poor locality are deemed more important and migrated earlier. This leads to fast performance improvements, even though the migration process has not completed.

Another interesting performance difference noticeable in Figure 5.12 is the very high, aggregate performance gain for *Auto 10x*. This is caused by the fact that the one execution of the repartitioner in this experiment is enough to optimize the partitioning. But one interesting difference is the much longer time required to finish repartitioning in *Auto 10x*. This is due to two factors: migration of larger number of vertices leads to increasingly localized traffic between a few partitions (in this experiment between 2 partitions) and the number of vertices migrated in a run. The first factor is due to a more aggressive migration strategy which leads to faster convergence combined with high data locality. The high data locality leads to vertices having remote relationships in a few partitions. When partition imbalances happen, migration will naturally happen only between these partitions. Combined with a more aggressive migration strategy, these cliques are easier to discover and only migrate the needed vertices. With a less aggressive strategy, the repartitioner may need to perform unneeded migrations to satisfy balancing constraints. In contrast, *Auto 1x* and *Auto 2x* show migrations between all partitions which increases

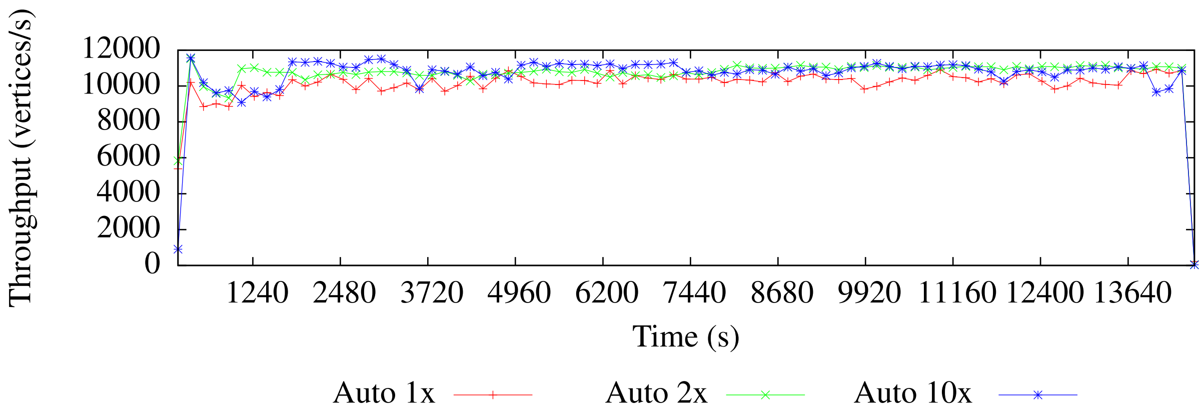


Figure 5.13: Performance over time while varying the *top-k* parameter

the parallelism of repartitioning operations, however it also leads to an increase in unneeded migrations, which, in turn, increases the time to convergence.

Figure 5.14 shows the skew in load over the partitions. The load skew measures the difference between the load on the most loaded partition with the average load on partitions (6.25% for 16 servers). The load is presented in percentage since the per partition load has been normalized to the aggregate load of the system. The load in each partition is measured as the total number of basic operations on the specific partition. The results in Figure 5.14 show an initial 8% skew in load on the *No Repart* data on a partition. Running the repartitioner shows an improvement in load balance of 3% for *Auto 1x* and 3.5% for *Auto 2x*, while *Auto 10x* shows a decrease in load balance of 8% (thus the partition is experiencing 16% more load). The constant decrease in load balance is due to the higher number of migrated vertices in each iteration. Increasing the *top-k* parameter will generally lead to larger migrations and higher accepted skew levels, thus causing larger skews.

In summary, when picking the *k* parameter one needs to take into consideration the acceptable imbalance between partitions. At the same time, larger values tend to perform better since they help the algorithm improve the edge-cut faster. In practice choosing *k* using the following formula gives good results: $k = N * imbalance\_factor * \lambda / p$ (where N is the number of vertices, p is the number of partitions and λ is a scaling factor that is based on the graph properties, such as power law coefficient and clustering coefficient). In general, λ should be higher if the graph is highly clustered. This allows the algorithm to migrate clusters (communities) easier and faster.

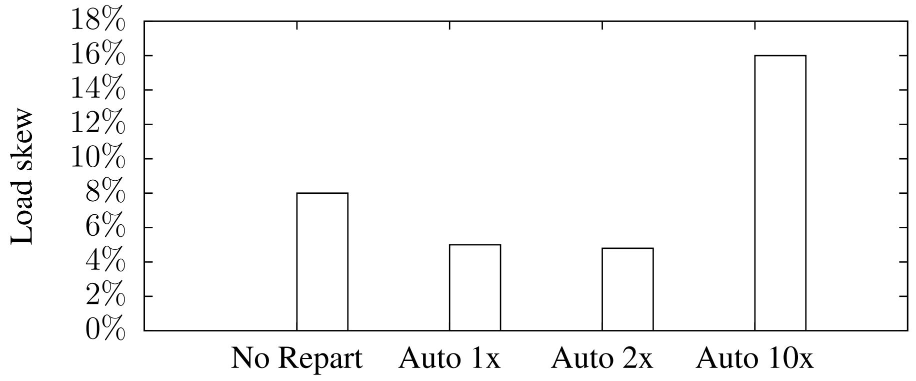


Figure 5.14: Partition balance while varying the *top-k* parameter

5.5.2 Varying Number of Iterations

In these experiments the *top-k* parameter was set to 500 and the *iterations* were varied using the following values: 10, 20 and 40.

Figure 5.15 presents the three experiments previously mentioned as *10 iter*, *20 iter* and *40 iter*. In addition, the *No Repart* and *Metis* experiments were provided for comparison. The last two experiments are the same experiments as presented in Subsection 5.5.1. The results in Figure 5.15 are interesting since they show about the same repartitioner impact on throughput performance and similar gains in performance once the repartitioner finishes, with one exception as *10 iter* shows better performance improvement. Compared to *No Repart*, there is a 6% decrease in performance while repartitioning and a 3% decrease after the repartitioner finishes for *20 iter* and *40 iter*, while *10 iter* has a 7% increase in throughput post repartitioning.

Figure 5.16 presents the performance of the three experiments over time. Interestingly, the repartitioning runs are much harder to visualize as the impact of the repartitioner is much smaller, especially while increasing the number of iterations. Looking closer, the larger dips in performance represent the start of each repartitioning run.

Figure 5.17 looks at the load balance over the partitions. Most notably increasing the number of iterations does not influence load balance significantly. In fact, all experiments exhibit an increase in load balance. *10 iter* show the highest increase in balance, at 5%, while *20 iter* shows 4.5% increase and 4.2% for *40 iter*.

Interestingly, varying the number of iterations does not show the performance increase that was shown in experiments varying *top-k*. Analyzing the migration patterns and the variation of

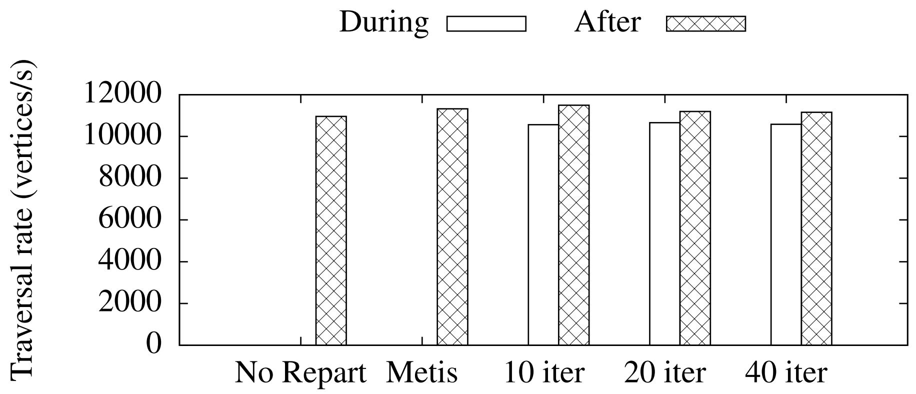


Figure 5.15: Performance comparison while varying the *iterations* parameter

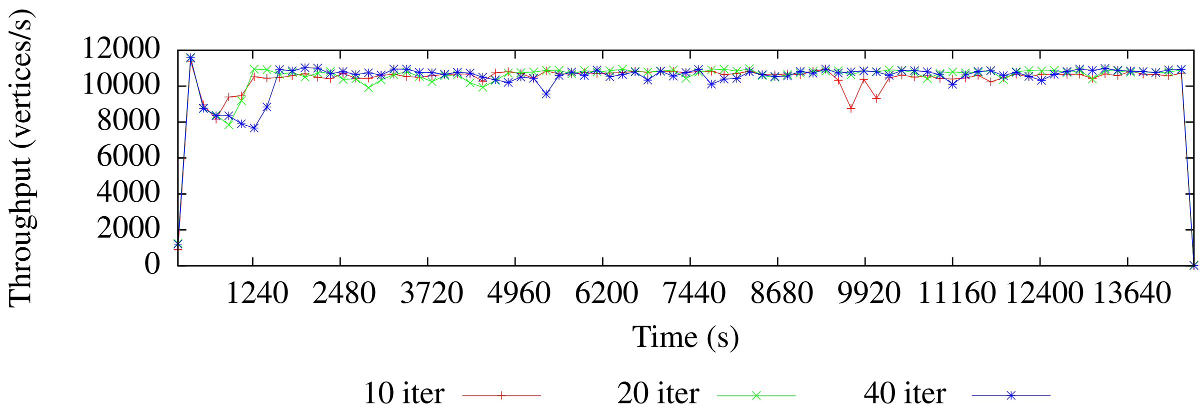


Figure 5.16: Performance over time while varying the *iterations* parameter

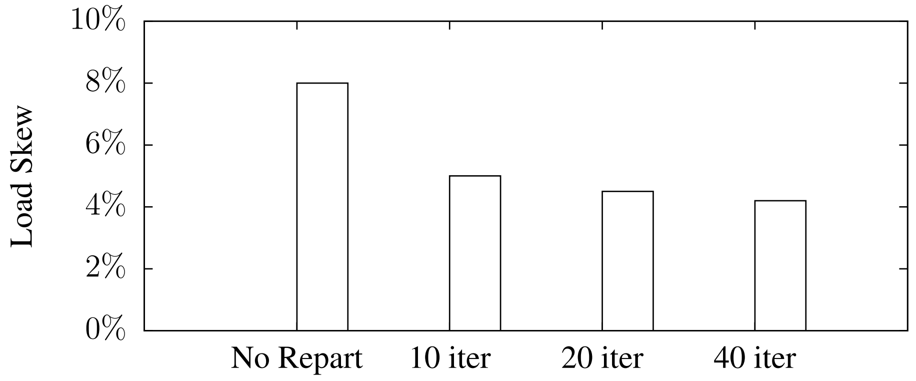


Figure 5.17: Partition balance while varying the *iterations* parameter

edge-cut for this set of experiments show that edge-cut converges much slower, while the majority of migrations will flow towards a lightly loaded partition, thus improving partition balance.

In summary, the number of iterations does not have an impact on system throughput since the algorithm itself has very little start-up cost. Thus, it is better to choose a small value as this allows the system to finish execution faster. While this means that the algorithm will be executed more often, it also means that it can react to changes in traffic patterns faster.

5.5.3 Repartitioning from Scratch

In addition to varying the parameters while repartitioning online, I also ran experiments where the repartitioning starts from a randomly partitioned dataset. These experiments validate that the repartitioner can handle arbitrary initial partitionings. Table 5.4 shows the number of iterations the repartitioner required to reach a *stable* state while varying *k*. Results show a gradual improvement in number of iterations required to reach stable state. Aside from the convergence rate, I have also measured the load skew. The skew at *k=500* was at 5%, decaying to 16% for *k=2000*. Values of *k* larger than 2000 were not considered as the imbalance rate would have been higher than 10% and unacceptable in a distributed system.

Further in Figure 5.18 shows the relative improvement in edge-cut from the randomly partitioned datasets. The results show similar partitioning quality regardless of the value of *k*.

	Orkut	**Twitter**	**DBLP**
k = **500**	2000	3000	500
k = **1000**	1000	1700	200
k = **2000**	700	900	150

Table 5.4: Number of iterations required to achieve a stable state where the edge-cut improvements are below 0.01%

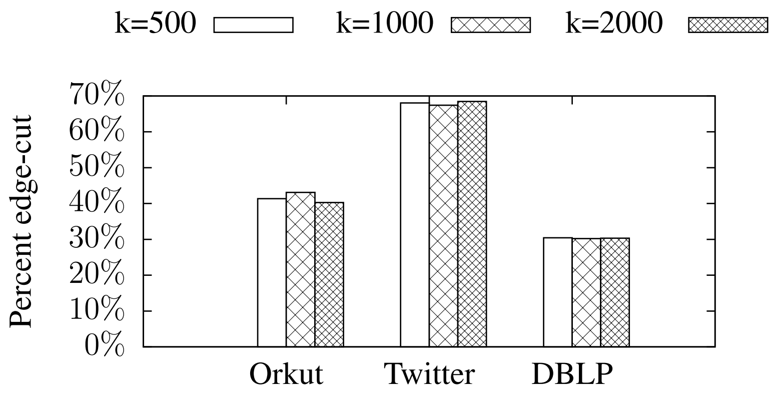


Figure 5.18: Ratio of edge-cut improvement when varying k

5.5.4 Summary

By increasing the *top-k* parameter the repartitioner is allowed to migrate larger numbers of vertices with poor locality while breaking load balance. Though, increasing *top-k* has a higher performance overhead while the repartitioner executes its first stage, performance improvements are much more obvious once vertex migration starts. In contrast, increasing the *iterations* has lower performance impact on the repartitioner overhead. In fact, *iterations* is more focused on long term partition load balance with incremental improvement in edge-cut gains.

Note, however, that it is important to understand that these two parameters are closely related and varying one can have major impact on how fast the algorithm converges, while the other focuses more on fine grained refinement. In fact, upon multiple rounds of repartitioning, as the partition quality increases (edge-cut decreases), higher *top-k* values will have no effect, or even negative effects as it decreases partition load balance. At the same time, higher *iterations* count will have a higher impact when the partition quality is close to optimal as it forces the repartitioner to focus more on load balance.

5.6 DistNeo4j Experiments

5.6.1 Scale Up

This section shows how well the system handles an increasing number of workers (or clients) continuously sending requests. In this scenario eight servers are used. While the initial partitioning is not critical in this experiment, data has been statistically partitioned using Metis since it provides a good partitioning that allows us to show system performance and scalability. Figure 5.19 shows the performance as the number of threads varies from 4 to 64. Results show almost perfect scaling up to 16 workers. It further shows good scaling up to 32 workers, after which performance starts degrading due to contention within the system.

5.6.2 Scale Out

This section aims at showing performance increases as the number of servers increase. The following experiments varies the number of servers between 4 and 16 showing performance improvements from increasing the number of servers. However, it is important to note that the performance increase is not expected to be linear as increasing the number of servers will also increase the number of edges cut, thus decreasing query locality. Tables 5.5, 5.6 and 5.7

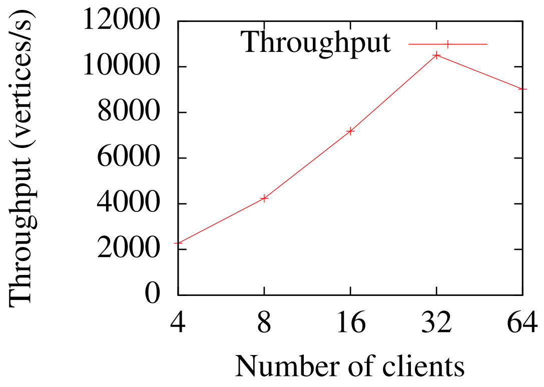


Figure 5.19: Performance of a 16 partition system while increasing # of clients

	Metis	**Random**
4 parts	19923340 (17%)	87887680 (75%)
8 parts	27003290 (23%)	102530431 (87%)
16 parts	35998540 (31%)	109861685 (93%)

Table 5.5: Edge cut for Metis partitioning and hash based random on Orkut. Percentage of edges cut is shown in parenthesis

show the edge-cut for the datasets as the number of partitions is varied. It also compares the Metis partitioning scheme with the Random one. The values show that the edge-cut in a well partitioned dataset is much lower than randomly partitioning data. It is also important to notice the slower rate at which the edge-cut increases while varying the number of partitions. This is important since increasing the number of partitions will have a smaller effect on the edge-cut of well partitioned data sets when compared to random partitioning.

Figures 5.20, 5.21 and 5.22 show how performance increases when adding new servers. The rate is close to doubling at each step, showing that the system will scale with additional resources

	Metis	**Random**
4 parts	20272228 (32%)	47666812 (75%)
8 parts	27937607 (44%)	55293502 (87%)
16 parts	34692526 (55%)	59585939 (93%)

Table 5.6: Edge cut for Metis partitioning and hash based random on Twitter. Percentage of edges cut is shown in parenthesis

	Metis	**Random**
4 parts	94222 (9%)	787599 (75%)
8 parts	124341 (11%)	918699 (87%)
16 parts	153930 (15%)	984157 (93%)

Table 5.7: Edge cut for Metis partitioning and hash based random on DBLP. Percentage of edges cut is shown in parenthesis

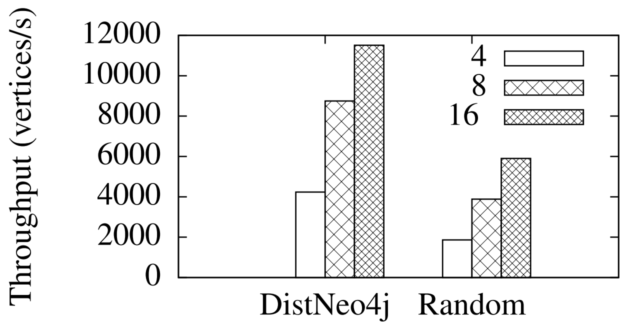


Figure 5.20: Shows the throughput rate while increasing the number of partitions for the Orkut dataset

when running the partitioned dataset. However, if the dataset is partitioned using random partitioning there is almost no performance gain when increasing the number of partitions. This is due to increasing network workload (due to edge-cut) counteracting the positive effects of additional system resources.

5.6.3 Read/Write Experiments

These experiments focus on showing the performance overhead from running a mixed read and write workload. These experiments focus on analyzing the overhead from writes.

Figure 5.23 shows four experiments where the percentage of write traffic is varied. Results show a sub-linear linear decrease in read traffic throughput while the percentage of write traffic is increased. The small performance impact of writes is attributed to how B+Trees store information and the monotonically increasing ID generator. Since each new record will get the next, highest, ID, it means that insertions in the B+Tree will always happen in the last page, in a sequential manner. This means that the B+Tree will perform sequential writes to disk and will only require caching the last page to perform these insertions.

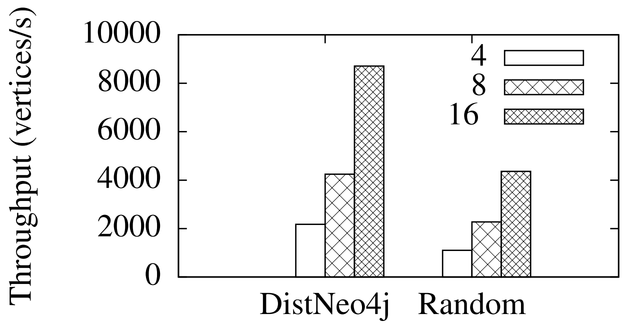


Figure 5.21: Shows the throughput rate while increasing the number of partitions for the Twitter dataset

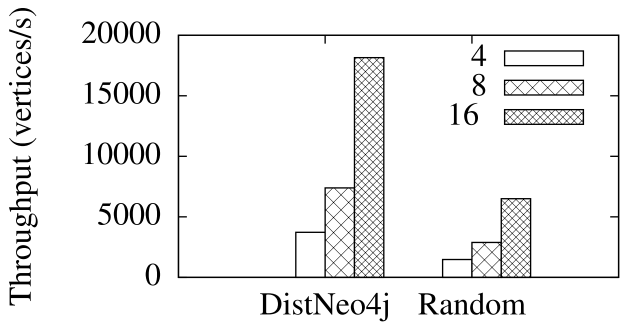


Figure 5.22: Shows the throughput rate while increasing the number of partitions for the DBLP dataset

In addition, due to the highly local read traffic patterns it is likely that the writes do not impact read traffic based on the transactional overhead of write traffic.

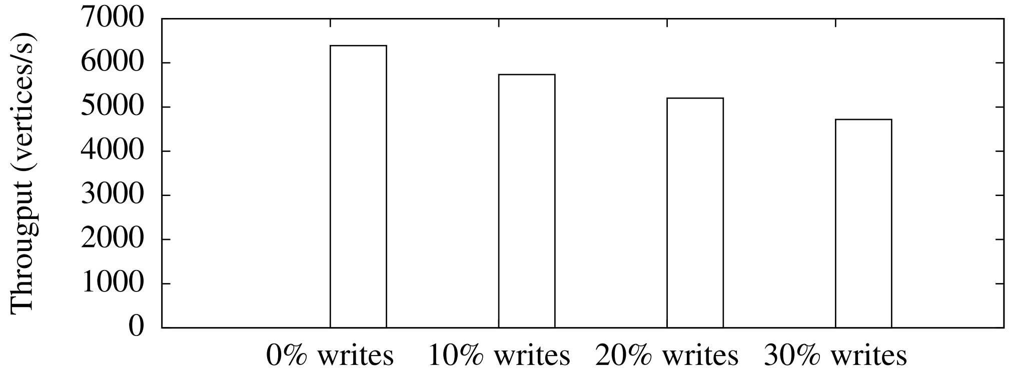


Figure 5.23: Aggregate performance while varying the write rate

Chapter 6

Conclusion

This thesis presents a lightweight, greedy repartitioning algorithm which is used to re-balance data distribution in a distributed graph database. As graph databases are continuously evolving systems, the initial data partitioning degrades over time. In order to counteract this process, the repartitioning algorithm is used to improve the quality as the system is evolving. Since the process of migrating data to improve locality has to be performed in parallel to user queries, the repartitioning algorithm needs to use few resources (low memory and CPU overhead) and keep the migrated vertices to a minimum (low network and disk overhead). Previous partitioning algorithms focused on the qualitative features of resulting partitioning (low edge-cut, high partition balance), leading to high number of migrations and high memory requirements. Most repartitioning algorithms are optimizations of existing partitioning algorithms, meaning that they still have the same flaws in resource allocation. In contrast, the proposed lightweight, greedy algorithm is designed with these constraints in mind. In addition, the greedy repartitioner takes advantage of the high quality partitioning and focuses on smaller improvements based on recent changes to the graph structure and to the query patterns. The constrained use case allows it to be simple, yet produce high quality results.

In order to validate the greedy repartitioner, a distributed graph database, DistNeo4j, is designed by augmenting the centralized Neo4j database. The greedy algorithm is implemented in DistNeo4j and evaluated using different query patterns used in previous work and in real world systems. The results show that the greedy repartitioner is able to maintain a high quality partitioning even when skews are added to the system or as new users or relationships are formed. In addition, the results show a much higher (2-3 times) throughput rate than the widely used hash based partitioning scheme.

References

[1] Education the planet with pearson. http://thinkaurelius.com/2013/05/13/educating-the-planet-with-pearson/.

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

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

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

[5] "scaling memcached at facebook". "https://www.facebook.com/note.php?note_id=39391378919".

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

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

[8] Titan: A highly scalable, distributed graph database, July 2012.

[9] Amine Abou-Rjeili and George Karypis. Multilevel algorithms for partitioning power-law graphs. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, pages 124–124, Washington, DC, USA, 2006. IEEE Computer Society.

[10] Selim G. Akl. An optimal algorithm for parallel selection. *Information Processing Letters*, 19(1):47 – 50, 1984.

[11] Konstantin Andreev and Harald Räcke. Balanced graph partitioning. *Theory Comput. Syst.*, 39(6):929–939, 2006.

[12] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. Linkbench: a database benchmark based on the facebook social graph. In *SIGMOD Conference*, pages 1185–1196, 2013.

[13] Lars Backstrom. Anatomy of facebook. https://www.facebook.com/notes/facebook-data-team/anatomy-of-facebook/10150388519243859.

[14] P. A. Bernstein and E. Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann, 2nd edition, 2009.

[15] Rishan Chen, Mao Yang, Xuetian Weng, Byron Choi, Bingsheng He, and Xiaoming Li. Improving large graph processing on partitioned graphs in the cloud. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 3:1–3:13, New York, NY, USA, 2012. ACM.

[16] Alex Cheng and Mark Evans. An in-depth look inside the twitter world. http://www.sysomos.com/insidetwitter/.

[17] M. Ciglan, A. Averbuch, and L. Hluchy. Benchmarking traversal operations over graph databases. In *Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on*, pages 186–189, April 2012.

[18] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

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

[20] Feldmann Andreas Emil. Fast balanced partitioning is hard even on grids and trees. In *Proc. MFCS*, pages 372–382, 2012.

[21] Guy Even, Joseph Naor, Satish Rao, and Baruch Schieber. Fast approximate graph partitioning algorithms. *SIAM J. Comput.*, 28(6):2187–2214, 1999.

[22] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference*, DAC '82, pages 175–181, Piscataway, NJ, USA, 1982. IEEE Press.

[23] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

[24] J. Golbeck. *Analyzing the Social Web*. Morgan Kaufmann, 2013.

[25] Borislav Iordanov. Hypergraphdb: A generalized graph database. *WAIM*, pages 25–36, 2010.

[26] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. 1996.

[27] George Karypis and Vipin Kumar. Metis unstructured graph partitioning and sparse matrix ordering system. 1995.

[28] George Karypis and Vipin Kumar. Multilevel graph partitioning schemes. In *Proc. 24th Intern. Conf. Par. Proc., III*, pages 113–122. CRC Press, 1995.

[29] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, December 1998.

[30] George Karypis and Vipin Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *J. Parallel Distrib. Comput.*, 48(1):71–95, January 1998.

[31] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell Sys. Tech. J.*, 49(2):291–308, 1970.

[32] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 169–182, New York, NY, USA, 2013. ACM.

[33] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *Procs. WWW*, pages 591–600, 2010.

[34] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 31–46, Berkeley, CA, USA, 2012. USENIX Association.

[35] MichaelM. Lee, Indrajit Roy, Alvin AuYoung, Vanish Talwar, K.R. Jayaram, and Yuanyuan Zhou. Views and transactional storage for large graphs. In David Eyers and Karsten Schwan, editors, *Middleware 2013*, volume 8275 of *Lecture Notes in Computer Science*, pages 287–306. Springer Berlin Heidelberg, 2013.

[36] Frank Lin and William W. Cohen. Power iteration clustering. *Proceedings of the 27th International Conference on Machine Learning*, 2010.

[37] Ulrike Luxburg. A tutorial on spectral clustering. *Statistics and Computing*, 17(4):395–416, December 2007.

[38] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proc. SIGMOD*, pages 135–146, 2010.

[39] N. Martínez-Bazan, V. Muntés-Mulero, Sergio S. Gómez-Villamor, J. Nin, M. Sánchez-Martínez, and J. Larriba-Pey. Dex: High-performance exploration on large graphs for information retrieval. *CIKM*, pages 573–582, 2007.

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

[41] Jayanta Mondal and Amol Deshpande. Managing large dynamic graphs efficiently. In *Proc. SIGMOD*, pages 145–156, 2012.

[42] Joel Nishimura and Johan Ugander. Restreaming graph partitioning: simple versatile algorithms for advanced balancing. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '13, pages 1106–1114, New York, NY, USA, 2013. ACM.

[43] Vol Nr and Per-Olof Fjällström. Algorithms for Graph Partitioning: A Survey, 1998.

[44] Chao-Wei Ou and Sanjay Ranka. Parallel incremental graph partitioning. *IEEE Trans. Parallel Distrib. Syst.*, 8(8):884–896, August 1997.

[45] Josep M. Pujol, Vijay Erramilli, Georgos Siganos, Xiaoyuan Yang, Nikos Laoutaris, Parminder Chhabra, and Pablo Rodriguez. The little engine(s) that could: scaling online social networks. *SIGCOMM Comput. Commun. Rev.*, 40(4):375–386, August 2010.

[46] Alessandra Sala, Lili Cao, Christo Wilson, Robert Zablit, Haitao Zheng, and Ben Y. Zhao. Measurement-calibrated graph models for social network experiments. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 861–870. ACM, 2010.

[47] M. Sarwat, S. Elnikety, Y. He, and G. Kliot. Horton: Online query execution engine for large distributed graphs. In *IEEE*, pages 1289–1292, 2012.

[48] K. Schloegel, G. Karypis, and V. Kumar. Multilevel diffusion schemes for repartitioning of adaptive meshed. *TR 97-013, U. Minnesota, Dept of Computer Science*, 1997.

[49] Kirk Schloegel, George Karypis, and Vipin Kumar. Parallel multilevel diffusion algorithms for repartitioning of adaptive meshes. *Journal of Parallel and Distributed Computing 47*, 1997.

[50] Kirk Schloegel, George Karypis, and Vipin Kumar. Parallel multilevel algorithms for multi-constraint graph partitioning (distinguished paper). In *Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, Euro-Par '00, pages 296–310, London, UK, UK, 2000. Springer-Verlag.

[51] Jianbo Shi. Learning segmentation by random walks. In *In Advances in Neural Information Processing*, pages 470–477. MIT Press, 2000.

[52] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22(8):888–905, August 2000.

[53] Isabelle Stanton and Gabriel Kliot. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '12, pages 1222–1230, New York, NY, USA, 2012. ACM.

[54] Charalampos E. Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. 2012.

[55] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[56] Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn Wilkins. A comparison of a graph database and a relational database. *Proceedings of the 48th Annual Southeast Regional Conference*, 2010.

[57] Lu Wang, Yanghua Xiao, Bin Shao, and Haixun Wang. How to partition a billion-node graph. In *Proceedings of ICDE*, 2014.

[58] Christo Wilson, Bryce Boe, Alessandra Sala, Krishna P.N. Puttaswamy, and Ben Y. Zhao. User interactions in social networks and their implications. In *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, pages 205–218. ACM, 2009.

[59] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *CoRR*, abs/1205.6233, 2012.

[60] S. Yang, X. Yan, B. Zong, and A. Khan. Towards effective partition management for large graphs. In *Proc. SIGMOD*, pages 517–528, 2012.

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