Gossip-based Partitioning and Replication Middleware for Online Social Networks

Muhammad Anis Uddin Nasir

Master of Science Thesis
Stockholm, Sweden 2013

TRITA-ICT-EX-2013:181
Degree Project for European Master in Distributed Computing

_Gossip-based Partitioning and Replication Middleware for Online Social Networks_

Author: Muhammad Anis Uddin Nasir
Supervisor: Sarunas Girdzijauskas
Examiner: Johan Montelius
Declaration

This thesis is an account of research undertaken between January 2013 and June 2013 at School of Information and Communication Technology, KTH Royal Institute of Technology, Sweden. I hereby certify that I have written this thesis independently and have only used the specified sources as indicated in the bibliography.

July 8, 2013
(Muhammad Anis uddin Nasir)
Abstract

The nature of Online Social Networks (OSNs) has created many scalability challenges for OSN providers in the last decade. The main challenge comes with a huge amount of data that is generated by OSNs and requires large data centers to handle the workload. The existence of strong community structure in social networks creates complex dependability patterns within the OSN data and makes it difficult to deploy over multiple servers without breaking the relation structure. Such systems may require communication among multiple servers and can generate high inter-server traffic. With no intelligent data allocation strategy, vital operations of OSNs like query processing and data management inevitably result in high and costly inter-server traffic. Existing solutions, i.e., distributed databases, key-value stores and auto scaling, may be inefficient and unable to scale for OSNs. In our work, we present a distributed partitioning and replication middleware that efficiently partitions an OSN graph and distributes it across the cluster in order to achieve high availability and scalability for OSNs. Our algorithm splits the social graph into equal sized partitions and periodically updates the partitions to achieve optimal placement of users. Additionally, our algorithm achieves fault tolerance and data locality at a low cost of redundancy through replication. We compared our algorithm with a de-facto random partitioning, state-of-the-art solution SPAR and a distributed graph partitioning algorithm called JA-BE-JA. In the experiments, we show that our approach generates four times lower replication overhead compared to random partitioning, and generates lower replication overhead by a factor of two compared to SPAR. We also demonstrate that our algorithm is able to tolerate high churn rates and scale for large OSNs.
Acknowledgment

I would like to express my deepest appreciation to my supervisor, Dr. Sarunas Girdzijauskas, who guided and encouraged me throughout this work. His methodology of supervision and guidance has really helped in achieving my targets while meeting the deadlines. It is not that I only learned technically from him, I have tremendously enjoyed working with him as a student and it really helped me to progress professionally. I would also like to express my sincere gratitude to my colleague Fatemeh Rahimian for her guidance and help during the thesis. Her ideas and advises played an important role in achieving different goals in the thesis. I would also like to thank PhD students at Swedish Institute of Computer Science, Martin Neumann and Niklas Ekström, for their insights, ideas and valuable feedback throughout this work. I would also like to express my gratitude to researchers at Swedish Institute of Computer Science, who have given their opinions and feedback on the work; Special thanks to Professor Jim Dowling for his guidance and suggestions. Also, I would like to show my gratitude for my examiner Johan Montelius, my opponent, Zygimantas Bruzgys, for their reviews and comments that helped me to improve greatly the final version of this document.

I would also like to thanks to Professor Leandro Navarro and all other professors from Politecnic University of Catalonla for their guidance and knowledge. Special thanks to my roommates Emmanouil Dimogerontakis and Faik Aras Tarhan, who really encouraged and helped me throughout my stay in Sweden. I would also like to thank all of my friends in Sweden, for making my stay full of cherishment and fun: Rizwan, Maria, Vaidas, Fahad, Atif, Zygimantas, Julia, Irfan, Asim, Faraz, and all classmates from European Master of Distributed Computing.

Finally, I would like to dedicate this thesis to my parents and brothers. Their continuous support and belief has played an important role in my life and really helped me to finish this European Master in Distributed Computing.

Stockholm, July 8, 2013
(Muhammad Anis Uddin Nasir)
# Contents

1 Introduction ........................................... 1
   1.1 Motivation ........................................ 2
   1.2 Contributions ..................................... 7
   1.3 Structure of the Thesis ............................ 8

2 Related Work .................................. 9
   2.1 Enterprise Solutions ............................... 9
   2.2 Graph Partitioning ................................. 10
   2.3 One-hop Neighborhood ............................. 11
   2.4 Activity Based Models ............................ 11
   2.5 Regional Servers ................................ 12
   2.6 Distributed Graph Processing Frameworks .......... 12

3 Problem Description .............................. 15
   3.1 Problem Statement ................................ 15
      3.1.1 Requirements ................................ 15
      3.1.2 Formulation .................................. 16
   3.2 Existing Scalability Models ....................... 18
      3.2.1 SPAR ........................................ 18
      3.2.2 JA-BE-JA .................................... 20

4 Gossip-based Partitioning and Replication Middleware 23
   4.1 System Design ..................................... 23
      4.1.1 Architecture ................................ 23
   4.2 System Operations .................................. 24
      4.2.1 Gossip Protocol ............................... 24
      4.2.2 Peer Selection ................................ 25
      4.2.3 Cost Function ................................ 25
      4.2.4 Simulated Annealing ......................... 26
   4.3 Algorithm .......................................... 26
      4.3.1 Greedy Algorithm ............................. 26
      4.3.2 Server Exchange .............................. 28
## 5 Tuning Parameters

5.1 Datasets .................................................. 29
5.2 Length of Random Walk ................................. 30
5.3 Peer Selection ........................................... 31
5.4 Simulated Annealing .................................... 32
  5.4.1 Noise Delta .......................................... 32
  5.4.2 Initial Temperature ................................. 39
5.5 Number of iterations ................................. 42

## 6 Evaluation

6.1 Experiments .............................................. 44
  6.1.1 Evaluation with Datasets ........................... 44
  6.1.2 Evaluation with Replication Factors ............... 45
  6.1.3 Evaluation with Servers ............................ 46
  6.1.4 Dynamic Behavior ................................. 48

## 7 Conclusion

7.1 Discussion ............................................. 51
7.2 Conclusion ............................................ 53

Bibliography .......................... 55
## List of Figures

1.1 An example showing that placement of high degree nodes may generate large number inter-partition edges. (a) a high degree node and (b) nodes equally distributed among two servers [1].

1.2 An example showing why min-edge cuts problem is different than minimum replica problem. The cloud in the figure represent communities, blue nodes represent the master copies of users and red nodes represent the replicas. (a) initial social graph with two communities, (b) the graph partitioned into two components by cutting minimum number of edges and (c) the graph partitioned into two components using minimum number of replicas [2]

1.3 (a) Sketch of an initial graph distributed across two servers, (b) Distribution of nodes and replicas after node addition and link creation using SPAR, i.e., SPAR requires five additional replicas to achieve data locality.

1.4 (a) Sketch of an initial graph distributed across two servers, figure (b) and (c) shows the distribution of nodes and replicas after node addition and link creation using SPAR and the proposed algorithm respectively, i.e., SPAR requires 5 additional replicas and our algorithm requires 3 additional replicas.

3.1 (a) initial network with three servers storing a social graph and a new edge is created between node 1 and 6, (b) network configuration after node 1 moves to server M3, (c) network configuration with no node movements, and (d) network configuration after node 6 moves to server M1 (copied from [2]).

4.1 Middleware integrated into three-tier web architecture

5.1 replication overhead for different length of random walks for random peer selection with $k=16$ and $f=2$

5.2 swap count and replication overhead for different values of noise delta for synthetic clustered graph with $k=16$ and $f=2$

5.3 swap count and replication overhead for different values of noise delta for synthetic highly clustered graph with $k=16$ and $f=2$

5.4 swap count and replication overhead for different values of noise delta for synthetic clustered graph with four clusters with $k=16$ and $f=2$. 


5.5 swap count and replication overhead for different values of noise delta for synthetic power law graph with $k=16$ and $f=2$. ................. 34
5.6 swap count and replication overhead for different values of noise delta for snap facebook graph with $k=16$ and $f=2$. ....................... 35
5.7 swap count and replication overhead for different values of alpha for synthetic clustered graph with $k=16$ and $f=2$. ....................... 36
5.8 swap count and replication overhead for different values of alpha for snap facebook graph with $k=16$ and $f=2$. ....................... 37
5.9 swap count and replication overhead for different values of exponential rate of change of temperature, for synthetic clustered graph with sixteen clusters with $k=16$ and $f=2$. ................................. 38
5.10 swap count and replication overhead for different values of exponential rate of change of temperature, for facebook dataset from SNAP with $k=16$ and $f=2$. ........................................ 38
5.11 swap count and replication overhead for different values of Initial Temperature with synthetic clustered graph with $k=16$ and $f=2$. .............. 39
5.12 swap count and replication overhead for different values of Initial Temperature with synthetic highly clustered graph with $k=16$ and $f=2$. .......................... 40
5.13 swap count and replication overhead for different values of Initial Temperature for synthetic low clustered graph with $k=16$ and $f=2$. .............. 40
5.14 swap count and replication overhead for different values of Initial Temperature for synthetic power law graph with $k=16$ and $f=2$. .............. 41
5.15 swap count and replication overhead for different values of Initial Temperature for SNAP facebook graph with $k=16$ and $f=2$. .............. 41
5.16 ratio of replication overhead for our algorithm and random partitioning for different number of iterations using synthetic graphs with $k=16$ and $f=2$. ........................................ 42
5.17 ratio of replication overhead for our algorithm and random partitioning for different number of iterations using real world graphs with $k=16$ and $f=2$. ........................................ 42
6.1 Comparison of replication overhead of our algorithm, random partitioning, SPAR and JA-BE-JA for different datasets with $k=16$ and $f=2$. ...... 44
6.2 Replication overhead for different fault tolerance replicas, i.e., $f=0, 2$ with $k=16$. ................................................................. 45
6.3 Replication overhead for different datasets and different number of servers with $f=2$. ................................................................. 47
6.4 Replication overhead versus number of iterations for SNAP Facebook graph with $f=2$ and $k=16$. ........................................ 48
6.5 Replication overhead versus number of iterations for WSON Facebook graph with $f=2$ and $k=16$. ........................................ 49
6.6 Replication overhead versus number of iterations for SNAP Twitter graph with $f=2$ and $k=16$. ........................................ 49
Acronyms and Abbreviations

**OSNs**  Online Social Networks
**SA**  Simulated Annealing
**DHTs**  Distributed Hash Tables
**SPAR**  Social Partitioning and Replication
**JA-BE-JA**  Distributed Graph Partitioning Algorithm
**SHA**  Simple Hash Algorithm
**APG**  Activity Prediction Graph
CHAPTER 1

Introduction

Social networks have gained tremendous popularity in the last decade and most of today’s popular web services are related to social networks [3]. Along with social network providers, many companies have used social networks to build various gaming and business applications [4]. The most popular Online Social Networks (OSNs), e.g., Facebook, Youtube and Twitter, attract millions of users and generate petabytes of data every day [5, 4, 6]. In order for OSN services to run smoothly under high workloads, OSN service providers are required to rely on large geo-distributed data centers. However, this acts as a barrier for small and medium enterprises to enter the OSN service market, since only well established OSN providers can afford to run large data centers. Such companies require efficient and low cost solutions to deal with OSNs and build applications. The main research interest of our work is to understand structural properties and patterns of OSNs and propose an efficient algorithm that can be utilized to store and process OSN data.

The deployment of OSNs on huge data centers requires data to be partitioned into independent components in order to place the data on set of autonomous servers. Such partitions help to utilize the network bandwidth efficiently and reduce the inter-server traffic, as it places the group of strongly connected users on the same server and dependent queries, e.g., new feeds of a user, where it has to fetch data from all the neighbors, can be resolved locally from any server. However, OSNs mostly have to deal with connected data, which represent the relationships and interactions of OSN users with their friends and acquaintances. OSNs exhibit strong community structure, which is reflected in the social data that OSN providers have to store and process [7, 8]. The existence of community structure makes it difficult to split and distribute the data across multiple machines, thus creating challenges for management and scalability. There are several companies, e.g., Twitter and Facebook, which redesigned their infrastructure several times to satisfy user demands [2]. Other companies like Friendster, which were unable to adapt and scale, were moved out of business.
1.1 Motivation

OSNs deal with social graphs, where links are created among users based on relationships and acquaintances. Such networks exhibit small-world properties [9] and are capable of spreading information widely and rapidly. OSNs follow a power-law degree distribution, which means that an OSN graph contains huge number of users with low social degrees and contains small number of users with high social degrees [5, 4, 6]. Such power law degree distribution makes it difficult to partition the graph into equal sized clusters and requires special attention for placement of high degree nodes [1]. Figure 1.1 gives an example where placement of high degree node on two servers generates large number of inter-partition edges. Moreover, in OSNs, 80% of all information accesses remain within a one-hop neighborhood in the social network topology [5], and we can achieve high data locality and better utilization of network resources by replicating neighbors that are one-hop away. In our work, we benefit from the structural properties of OSNs and propose an approach for deployment of OSNs that provides high availability and guarantees data locality for one-hop neighbors through replication.

Figure 1.1: An example showing that placement of high degree nodes may generate large number inter-partition edges. (a) a high degree node and (b) nodes equally distributed among two servers [1].

A conventional way to handle high workload and satisfy user demands is to upgrade existing hardware. Such vertical scaling techniques are expensive and not feasible to handle petabytes of data [2]. An economical solution to handle the high workload is to rely on horizontal scalability utilizing higher number of cheap commodity servers. Horizontal scalability requires data to be partitioned into small components so that it can be distributed among multiple machines. However, OSNs deal with strongly connected data that is created based on relationships among users. It exhibits a strong community structure, where users are densely connected within the communities. The existence of strong community structure makes it difficult for OSN providers to partition and distribute the data across multiple machines.
Currently, most of the OSN providers utilize key-value stores, where random partitioning is used to split the data across multiple machines. Specifically, Cassandra [10] that is an open-source distributed database, which provides structured key-value store with tunable consistency. It uses Chord [11] like ring structure for network topology and leverages consistent hashing to assign users to a machine in the ring. The consistent hashing function takes the user-id and generates address of a server, using a hash function like SHA-1. Such hash functions are oblivious to the underlying relationship structure of OSNs and place the data related to a single user on different servers. For example, if a user \( u \) has a set of neighbors, the data of neighbors is likely to be placed on different servers based on the hash values. An update of user \( u \) will require pushing relevant messages to the servers hosting its neighbors. Such distribution will generate high inter-server traffic due to highly interconnected workload and may not work effectively in the case of OSNs.

In the past few years, researchers have started taking interest in management and scalability of OSNs, as typical queries in OSNs are usually demanding and usually require to accumulate data from more than one user [2, 12, 13]. For example, a read query for a user \( u \) may require the information of all the friends who have their birthdays in the next month and such queries may require accumulating data from more than one machine and are expensive in terms of network bandwidth. Recently, researchers have targeted the problem related to management and scalability of OSNs from different perspectives. Some solutions have been proposed targeting the one-hop neighborhood [2, 12, 14], as it accounts for 80% of information accesses across OSNs [5]. One-hop neighborhood represents information and updates related to friends that are directly connected to a user. Researchers have shown that we can reduce the network traffic through replication for the neighbors that are directly connected to a user [2]. Moreover, few solutions suggest utilizing activity based graphs instead of social graphs for replication of OSN users in a distributed environment [15, 16, 17]. Activity based graphs are created based on user activities and interaction in OSNs and help in reducing the replication overhead. Such graphs represent user relationships more efficiently compared to social graphs, as users do not interact with all of their connections [6]. Researchers have also considered using graph partitioning algorithms to partition the social graph into equal-sized clusters and then placing the independent clusters on different machines to efficiently utilize the network bandwidth [18, 19]. However, graph partitioning solutions may not work efficiently, in case of OSN graphs, due to existence of power law degree distribution in OSN graphs.
Figure 1.2: An example showing why min-edge cuts problem is different than minimum replica problem. The cloud in the figure represent communities, blue nodes represent the master copies of users and red nodes represent the replicas. (a) initial social graph with two communities, (b) the graph partitioned into two components by cutting minimum number of edges and (c) the graph partitioned into two components using minimum number of replicas [2]
1.1. Motivation

In our work, we compare our algorithm with two state of the art techniques JA-BE-JA [18] and SPAR [2], which are described below. JA-BE-JA is a distributed balanced graph partitioning algorithm, which continuously keeps the social graph partitioned into equal size clusters. It is a gossip based algorithm, which periodically adjusts the partitions utilizing the local information at every node and places the connected nodes together. JA-BE-JA efficiently partitions the graphs into equal size groups and reduces the edge cuts between the partitions. We utilize replication in order to achieve data locality for connected neighbors that reside on different servers, which means that for any user $u$, we create the copy of its master on all the servers that has its directly connected neighbors. JA-BE-JA provides better partitioning for the graphs by minimizing the edge cuts between the partitions and generates low replication overhead in case of no fault tolerance, i.e., there is no replication in the system to provide resilience against failures and the only replication in the system is to achieve data locality. However, replication overhead increases with the increase in replication factor, as the algorithm does not take into account the replicas while partitioning the social graph. Figure 1.2 illustrates how minimizing the edge-cut is not equivalent to minimizing the number of replicas. In particular, Figure 1.2 (b) shows that the minimum edge-cut will cut three edges and require five nodes to be replicated in order to achieve data locality. However, in case of minimizing the number of replicas in Figure 1.2 (c), the algorithm partitions the graph by cutting four edges, resulting in four replicas. In our work, we modify the cost function of JA-BE-JA in order to minimize the number of replicas, instead of minimizing the number of edge-cuts between the partitions.

![Figure 1.3](a) Sketch of an initial graph distributed across two servers, (b) Distribution of nodes and replicas after node addition and link creation using SPAR, i.e., SPAR requires five additional replicas to achieve data locality.

SPAR [2] addresses the problem of partitioning the social graph to provide scalability for OSNs. It guarantees data locality for connected neighbors, utilizing the social graph structure and creating eventually consistent replicas across servers. Data locality ensures that the data required by a user is always
present on a local server and the algorithm does not have to access any other server to resolve a query. SPAR creates replicas for neighbors that reside on other servers, so that it can access the information related to neighbors locally using the replicas. Each user data is stored in a system in the form of single master and set of replicas. Replicas are eventually consistent copies of the user master, which means that it is possible that replicas do not have the updated information corresponding to the user. SPAR improves network and CPU performance by distributing the workload equally among servers. SPAR uses a heuristic based on greedy optimization for placement of users and its replicas, which we describe below.

In SPAR, when a new user $u$ joins the network, it is assigned to a server with least number of users (masters). In case of link creation between user $u$ and any node $x$, the algorithm first checks if nodes are co-located, i.e., either nodes exist on same server or nodes have master-slave relationships on different servers. The algorithm does not require performing any operation if nodes are co-located. However, if nodes are not co-located, algorithm evaluates three different configurations, i.e., (a) no movement of users, (b) user $u$ is assigned to the server of user $x$, and (c) user $x$ is assigned to the server of user $u$. The algorithm greedily chooses the configuration that yields the minimum number of replicas. Moreover, it requires a load balancing condition to ensure that the workload is distributed equally among all the servers, which could restrict optimal assignment of the nodes to the servers. SPAR is a sequential algorithm, which may become inefficient for huge networks, dealing with millions of nodes and billions of edges (e.g., Facebook, Twitter), as it does not take into account the global picture of the social graph. In our work, we address this problem and utilize a distributed algorithm that can take into account the global picture of a graph while distributing the nodes across servers.

Figure 1.3 shows a graph partitioned between two servers. Later, node 9 joins the network and creates links with nodes 2, 3, 4 and 5. Figure 1.3 (b) shows how SPAR places the new user in the existing network. When node 9 joins the network, SPAR places the node 9 on server M2 as it contains fewer number of masters. In case of link creation between node 9 and node 2, the algorithm chooses configuration (a) and creates replicas of node 9 on server M1 and creates replica of node 2 on server M2. The algorithm chooses the same configuration for other new links maintaining the load balancing constraint across the servers. In particular, it shows that 5 additional replicas are required to maintain data locality. This behavior of SPAR is due to the fact that the nodes in SPAR are processed sequentially without taking into account the global picture. Moreover, the existence of load balancing constraint may not allow choosing optimal configuration for placement of nodes, hence, resulting in additional replication overhead.
1.2 Contributions

In this thesis, we propose a gossip based partitioning and replication middleware that deploys a social graph on top of data-stores with the goal of reducing number of replicas across servers. Like SPAR [2], our algorithm benefits from the social behavior of users in OSNs, i.e., 80% of user accesses remain within a one-hop neighborhood [5], and provides data locality for one-hop neighbors. The algorithm follows a node centric approach, where each node periodically selects a neighbor and share information about the replicas that are required for their existence on their servers. Users exchange their servers if the server exchange will result in overall less number of replicas in the system. We assume that every node is an independent processing unit and is capable of executing the algorithm utilizing the local information. Our algorithm is a heuristic-based solution that minimizes the number of replicas across the system. We prevent our algorithm from becoming stuck in a local optima by employing a simulated annealing technique [20]. Figure 1.4 (c) illustrates that our proposed algorithm requires lower number of replicas compared to SPAR. We implement our algorithm and compared its performance with de-facto random partitioning, state-of-the-art solution SPAR and a distributed graph partitioning algorithm called JA-BE-JA. We performed extensive experiments with different social networks graphs, replication factors and different number of servers. Results show that our approach reduces the replication overhead four times as compared to de-facto random partitioning.
ing and reduces the replication overhead by factor of two compared to SPAR in case of high clusterization in the social graph. Moreover, we demonstrate that our algorithm in capable of handling dynamic nature of OSNs and provides better scalability using a distributed algorithm instead of a centralized one.

1.3 Structure of the Thesis

We divide our work in chapters. In chapter 2 we discuss the related work of the thesis. Chapter 3 describes the problem statement, formulates the optimization problem for minimizing number of replicas and discusses the existing scalability models that we implement in our thesis. Further, Chapter 4 describes our algorithm and and its architecture. In chapter 5, we discuss the datasets that we use in our experiments and tune different parameters for experiments. Chapter 6 discusses the evaluation of the algorithm. Finally, chapter 7 concludes the thesis and suggests future work directions.
CHAPTER 2

Related Work

2.1 Enterprise Solutions

Most of the modern OSN providers rely on Distributed Hash Tables (DHTs) [21] and noSQL databases [22], which provide horizontal scalability by randomly partitioning and placing the data on multiple servers. Moreover, many companies have implemented customized solutions to handle high workload that is generated by OSNs. For example, Twitter in its early days, implemented Gizzard [23], which uses range partitioning to distribute the data across cluster. Facebook developed Cassandra [10], which provides high scalability and good performance with tunable consistency. Amazon implemented Dynamo [24], a highly available key-value store, to store user shopping carts. Such data-stores promise high scalability, due to their distributed design, and randomly partition and distribute the data across the network [22]. However, such solutions do not take care of social structure in OSN graph and poorly partition the data, which may place the connected users on different servers. As a result, processing a single query may require gathering data from multiple machines and utilize high network bandwidth. Therefore, Key-Value stores can lead to poor performance and may generate high inter-server traffic [2].

Data distribution is a trivial way to achieve better performance and availability for most of the applications. Various distributed databases, i.e., MySQL, Bayou [25] are available that provides high availability and eventual consistency for the data. Similarly distributed file systems, i.e., Ficus [26], Coda [27] promises high availability through replication. All such systems distribute the data randomly among the servers and have to gather the information from multiple servers, which may make such systems inefficient for OSNs. Graph databases provide an attractive solution for deployment of social graphs [28]. However, graph databases are designed for complex data and do not promise high scalability for large scale dynamic graphs [29]. Moreover, many cloud providers like Amazon and Google AppEngine lend auto scaling services for deployment of web applications [30]. Such services allow the application to
scale automatically depending on the workload, but they require applications to be stateless and data to be independent. However, OSNs contain connected data and such frameworks are not the optimal choice in case of large scale OSNs.

2.2 Graph Partitioning

An alternative way to deploy OSNs on any data-store is to partition the data into disjoint components and place the disconnected components on different servers to reduce the communication overhead. The existence of strong community structure in OSNs makes it difficult to partition the data, as users usually belong to more than one community [31]. Large amount of research has been dedicated to graph partitioning during the last decades [32, 33, 34]. However, most of these algorithms work with offline graphs, and they are insufficient for OSNs, which exhibit a dynamic behavior [2]. Additionally, these algorithms require the global view of the graph, hence making such algorithm not feasible for large scale OSNs. Recently, Fatemeh et al. [18] proposed a distributed graph partitioning algorithm called JA-BE-JA, which takes the global picture of the graph into account and partitions the graph into equal sized clusters while minimizing the edge-cuts between the partitions. However, such partitioning algorithms minimize the edge-cut between partitions (as we discussed in the previous chapter), and might not be efficient in reducing the replication overhead [2].

Moreover, Yang et al. [19] proposed Sedge, a Self Evolving Distributed Graph Management system that minimizes the inter-server traffic, which is generated during graph query processing in a cluster environment. Sedge uses offline and online partition schemes to partition the graph and reduce the inter-server traffic. The offline partitioning initially partitions the graph and distributes it to the workers across the cluster. For online partitioning, Sedge introduces a two-level partition management architecture with complimentary primary partitions and dynamic on-demand secondary partitions, which adapts in real time depending on the workload. Complimentary partitioning repartitions the graph that is partitioned in the offline partitioning part. It partitions the graph in a way that inter-partition edges become the internal edges of any of the partition. Further, Sedge creates on-demand secondary partitions by replicating the portion of primary partitions that contains the intensive workload. Using such partitioning schemes can reduce the inter-server communication for dynamic graphs. However, such algorithm will remain in transition due to dynamic nature of OSNs and will require intensive processing to handle the OSN workload.
2.3 One-hop Neighborhood

Pujol et al. [35] proposed a replication scheme considering one-hop neighborhood for scaling online social networks. Further, they continued their work and proposed a middleware called SPAR [2], which utilizes the social topology of OSN users to distribute a social graph in a distributed environment. SPAR operates on top of data-stores, i.e., MySQL, and Cassandra, and provides scalability for OSNs. It aims to provide data locality for one-hop neighbors by creating eventually consistent replicas of users across the servers. Such replication can help to resolve a query locally from any server. It improves the system performance and reduces the inter-server traffic by paying the replication cost (see chapter 3 for description of SPAR).

Some researchers have also put forward procedures for efficient replication that can help to reduce the network traffic in a distributed cluster. S-CLONE [12] proposes a social based replication scheme for OSN providers with fixed budget of disk space. It suggests to take into account the social relationship of users while replicating for fault tolerance, by creating replicas on the server with maximum number of neighbors. S-CLONE divides its replication procedure into two phases, i.e., (a) Replicate phase, (2) Adjust phase. In the first phase, the algorithm greedily places the replicas of a node on a server with maximum number of neighbors. In the second phase, the algorithm adjusts the number of replicas in the system to maintain the fault tolerance condition, i.e., there should always be $f$ replicas in the system to recover in the case of failure. Such algorithm reduces the network traffic as replication improves the data locality. Similarly, Nguyen et al. [36] suggested to preserve social locality in data replication for OSNs, as it helps in improving the performance and data locality of the system.

2.4 Activity Based Models

Further, unlike SPAR [2] that provides data locality for every user, Yuan et al. [15] proposed to partition the network in time domain. The proposed approach was influenced by the study of Wilson et al. [6], that claims that users do not interact with all of its friends or neighbors and for majority of users, 20% of their friends account for 70% for interactions. The proposed algorithm provides data locality for frequent or “active” users, who have exchanged messages recently. It uses activity prediction graph (APG) that keeps the updated data in all the partitions that are likely to be accessed by users. The author preferred two-hop neighborhood instead on 1-hop neighborhood, as it cover major portion of accessed data. In the experimentd authors showed that partitioning on the two-hop provide data locality for at least 19% of the queries. Chen et al. [37] and Huang et al. [16] in their works also proposed a similar strategy to use interaction graph instead of the social graph for data replication for deployment of OSNs. However, activity network evolves rapidly and to exploit the strong time correlation, the algorithm needs to be
adaptive and dynamic, as the data is OSNs has strong time dependence [17]. Such requirements make the algorithm CPU intensive, as it has to maintain an updated APG at very small time scale.

Mondal et al. [38] proposed a complete in-memory, distributed graph data management system that can support processing of large scale dynamically changing graphs and uses disk as a backup storage. In their work, they proposed aggressive replication of nodes in the graph to handle low latency queries. Further, they have investigated three different replication schemes targeting communication overhead and storage requirement in the system. The first replication scheme is a hybrid, adaptive replication policy that performs eager or lazy replication by creating a predictive model of node-level read-write frequencies. The second scheme uses a clustering-based approach to reduce the cost of replication across the network. In this scheme the algorithm group together nodes with similar access pattern. The last proposed scheme uses a fairness criterion to guide the replication decisions in the network. The fairness criteria states that at least $\tau \leq 1$ neighbors are present on the local server. In our work, we propose a simple partitioning strategy, where we partition the graph by minimizing the number of replicas across the servers.

2.5 Regional Servers

Witie et al. [39] proposed a different approach to improve the network performance by exploiting the locality for interest for OSN users. As most of the information accesses in OSNs are within a geographical region, authors proposed to use TCP proxies and caching in a distributed architecture to makes the queries processed through regional servers. Such deployment of regional servers will improve responsiveness and efficiency of an OSN service. It reduces overall cost of network bandwidth and workload on central servers, as most of the queries can be resolved through the regional servers.

2.6 Distributed Graph Processing Frameworks

Recently, researchers have started showing much interest in development of distributed graph processing frameworks to simulate large scale graphs, i.e., Google Pregel [40], GraphLab [41, 42], GraphChi [43] and PowerGraph [1]. All these graph-parallel abstractions consist of a sparse graph $G=(V,E)$ and a vertex program that simulates on each vertex $v \in V$ in parallel. Pregel [40] is a bulk synchronous message passing abstraction in which all vertex programs run simultaneously to achieve maximum parallelism and communicates with each other through message passing. Each vertex program consists of several steps, i.e., (1) receiving messages from all the neighbors, (2) processing those messages, and (3) sending messages to all the neighbors. A barrier is imposed between steps in order to ensure that all programs finish their processing be-
fore executing a new step. Pregel also introduces commutative associative message combiners that merge the messages originating from the same vertex to reduce the network traffic.

GraphLab [41] is an asynchronous distributed shared memory abstraction in which all vertex programs communicate using a shared memory. GraphLab ensures serializability by blocking the neighboring vertex programs to execute simultaneously and schedule such programs to execute in the future. Moreover, GraphChi [43] is a disk based graph processing framework that is capable of processing large graphs with billions of edges on a single machine. It uses a parallel sliding window to break the graph into small parts and process those small parts in memory. PowerGraph [1] combines the best features of Pregel [40] and GraphLab [41], and proposes GAS decomposition, i.e., Gather, Apply and Scatter, to factor vertex programs over edges. It borrows data graph and shared memory concept from GraphLab. From pregel, it acquires commutative, associative gather concept. It supports both the high parallel bulk-synchronous Pregel model and computationally efficient asynchronous GraphLab model. It also introduces the concept of delta caching to cache the results from previous iterations that can be used in future iterations. All these frameworks provide perfect platform to simulate algorithm on large scale graphs containing billions of nodes and edges. Keeping the recent trend in mind we propose a gossip based algorithm that can easily be simulated using such frameworks. Implementations of our algorithm on these frameworks is left as a future work.
3.1 Problem Statement

In this section, we define the requirements that should be addressed by a solution in order to achieve scalability for OSNs. Moreover, we define the minimum replica problem that can be used to store OSNs in any data storage system. Finally, we discuss the existing models that we implement for comparison with our algorithm.

3.1.1 Requirements

Following are the set of requirements that our algorithm needs to address:

- **availability**: OSNs consist of users and their relationships, where users interact with each other and receive updates within their group of friends and communities. Such networks are also used for advertisement and product related services to distribute the information across the group of users. Any service that is developed with the goal of storage and processing for OSN information should be able to provide high availability for its users.

- **data locality**: The system should be able to provide data locality for users in one-hop neighborhood, which means that data related to one-hop neighbors should be present in the local server. Such data locality will help to respond user queries locally without accessing any other server. To achieve data locality, we create copies of user master, which are called replicas. There can be two types of replicas in the system, i.e., (1) data locality replicas, (2) faulte tolerance replicas. Data locality replicas improve the network performance by creating local copies of the neighbors that are used to access neighbor information locally from any server. Other replicas that are present for fault tolerance, make system resilient against failure and helps to recover user information in case of failures.
• **fault tolerance**: The system should be able to cope with machine failures, which means that the system should have store user information at different places where it can recover the information in case of failure. A storage system usually achieves fault tolerance using $f$ replicas of a user, which are stored randomly across the network.

• **k-way partitioning**: Let $G=(V,E)$ denotes an undirected social graph with $V$ vertices representing the users and $E$ edges, which represent relationships among users. Let $k$ be the total number of servers to host the social graph. We need to divide the social graph into $k$ independent partitions, where each $v \in V$ is assigned exactly to one of the $k$ servers.

• **load balance**: A balanced partition means that each partition share almost equal amount of workload, so that no partition gets overloaded and can end up as a bottleneck in the network. Such partitioning allows system to achieve better utilization of resources and minimize the response time for read and write queries.

• **distributed algorithm**: Decentralized algorithms are popular to handle high workload and provide scalability for different applications. Such systems are capable of handling dynamic nature of applications, i.e., high churn rate, and self management in case of failure. Keeping the stable nature of decentralized systems in mind, we want the algorithm to work in a distributed manner that each system in the network can execute algorithm with only local information. Such conditions help algorithm to process large scale dynamic graphs, which are impossible to process on a single machine.

• **stability**: The system should be resilient to churn and different dynamicity of OSNs. It should be able to handle high churn rates, where users continuously joins and leaves the network. Similarly, the system should be stable to provide data locality for the new connections that are formed between users, without affecting the network topology of the existing system.

• **replication overhead**: As discussed previously, our algorithm will use replication in order to achieve data locality and fault tolerance for users. It promises consistency for a single master copy of a users and perform all updated read and writes to that master copy. The algorithm updates all the replicas of a user eventually and promises eventual consistency for the replicas. To minimize the replication overhead, the algorithm should achieve data locality and fault tolerance using the minimum number of replicas across the network.

### 3.1.2 Formulation

We can now formulate the problem as an optimization problem of minimizing the number of replicas across the network. We model the problem using an
undirected graph $G = (V, E)$, where $V$ are the vertices and $E$ are the edges connecting those vertices. In addition, we have $k$ servers to place those vertices, where each vertex $i \in V$ should be placed on any of the $k$ servers. For every user, each of his neighbors should exist on the same server either as a master or as a replica.

Suppose $m_{ij}$ denotes a binary decision variable which is equal to one, if master of user $i \in V$ resides on server $j \in 0 \cdot \cdot \cdot k - 1$. Similarly, we have a binary decision variable $r_{ij}$ for replica of a user, which is equal to one, if replica of user $i \in V$ exists on server $j \in 0 \cdot \cdot \cdot k - 1$. We need to ensure that every edge $e(i,l) \in E$, where $i,l \in V$, follows the data locality requirement and the system has minimum $f$ number of replicas corresponding to each user for fault tolerance. We can write the optimization problem as follows.

\[
\begin{align*}
\text{minimize} & \quad \sum_i \sum_j r_{ij} \\
\text{subject to} & \quad \sum_j m_{ij} = 1, \quad \forall i \in V \quad (3.1) \\
& \quad m_{ij} \leq m_{ij} + r_{ij} + 1, \quad \forall i,l \in V, \forall j, \forall e(i,l) \in E \quad (3.2) \\
& \quad \sum_i m_{ij} = \sum_i m_{i(j+1)} + \epsilon, \quad \forall j \in 0 \cdot \cdot \cdot k - 1 \quad (3.3) \\
& \quad \sum_j r_{ij} \geq f, \quad \forall i \in V \quad (3.4)
\end{align*}
\]

We need four different constraints to fulfill the requirements for the system.

- Constraint (3.1) ensures that every user has exactly one master on the servers.

- Constraint (3.2) ensures the data locality condition that for every edge, information should exist on the local server, either as master or replica.

- Constraint (3.3) ensures the load balancing condition among all the servers. It guarantees that each server handles almost equal number of masters, where the workload may vary depending on the coefficient of variation ($\epsilon$) [44], which is the ratio of standard deviation to the mean (of the server sizes). This condition makes sure that when a new user joins the network, it is assigned in a way that workload is equally distributed among all the servers. As replicas are eventually consistent, we only put the constraint on the master copy of a user.
Chapter 3. Problem Description

- Constraint (3.4) ensures the fault tolerance condition that at least $f$ number of replicas of every user should exist in the system.

However, the mentioned optimization problem is an NP-Hard problem and can not be solved using linear programming. [2].

3.2 Existing Scalability Models

In the Chapter 2, we discussed existing solutions that can offer scalability for OSNs. In this section we describe in details two of the solutions that we consider and implement in our thesis for comparison with our algorithm.

3.2.1 SPAR

SPAR is a social partitioning and replication middle-ware that was proposed by Pujol et al. [2] to provide scalability for OSNs. It utilizes social graph structure and guarantees data locality for one-hop neighbors through replication. SPAR stores users as a master and creates copy of those master, called replicas, across the network to maintain the local semantics, i.e., required information related to a user and its direct neighbors should be local to any server. Such local semantics ensures that the algorithm does not require fetching information from any other machine on the network. It helps in achieving high network performance and efficient bandwidth utilization. Moreover, it facilitates developers to build their applications as it is developed on top of a single machine.

SPAR focuses on providing data locality while creating minimum number of replicas across the network. It uses a heuristic based on greedy optimization that utilizes the local information at every node. SPAR algorithm is triggered by six different operations, i.e., node addition/removal, edge addition/removal and server addition/removal. In SPAR, when a new node joins a network it is assigned to a partition with fewest number for masters and required number of replicas are created for fault tolerance across the network. In case of node removal, master and all replicas are removed and all the neighbors of the user are updated.

For edge additions, algorithm begins by checking if the nodes are collocated, i.e., either masters of both nodes are on same server or both nodes have master-slave relationship on their servers. If the nodes are not collocated, algorithm evaluates three different configurations, i.e., (a) no movement of users, (b) first user is assigned to the server of second user, and (c) opposite of configuration (b). Further, the algorithm selects the configuration that yields minimum number of replicas. To demonstrate edge addition, we can look at Figure 3.1 (a), where a small social graph is distributed across three servers. Further, an edge is created between node 1 and node 6. As both nodes are not collocated, algorithm will evaluate three different configurations. Figure
3.2. Existing Scalability Models

3.1 (c) represents the first configuration, where no master will move from its server and it requires two additional replicas. Figure 3.1 (b) represents the second configuration where node 1 moves from server M1 to server M3. In this scenario the algorithm requires two more replicas to maintain the data locality, as replicas of node 2, 3 and 4 will be created on server M3 and replica of node 5 will be removed from server M1. Further, 3.1 (d) shows the third configuration, where node 6 moves from server M3 to M1 and it will remove one replica as it will not require the replica of node 5 on server M3. The algorithm in this scenario will choose the third configuration, as it results in minimum number of replicas.

![Diagram](image)

Figure 3.1: (a) initial network with three servers storing a social graph and a new edge is created between node 1 and 6, (b) network configuration after node 1 moves to server M3, (c) network configuration with no node movements, and (d) network configuration after node 6 moves to server M1 (copied from [2]).

SPAR imposes a load balancing constraint across the network, so that the workload is distributed across multiple machines. The load balancing constraint may restrict the move mentioned in above example, as third configuration will result in no workload on server 3 and may possibly choose first configuration to balance the workload. For edge removal, SPAR removes the edge and the replicas while maintaining the $f$ replicas for fault tolerance. We will not discuss the server addition and removal, as we do not consider server addition and removal in our work.
3.2.2 JA-BE-JA

We also consider using a graph partitioning algorithm to split the social graph in equal size partitions. Such partitioning algorithms can help to find clusters in a graph, which can be placed on independent servers to reduce the inter-server traffic. In our work, we considered a distributed graph partitioning algorithm called JA-BE-JA that is proposed by Fatemeh et al. [18]. It is a gossip based heuristic algorithm that partitions social graphs into equal size clusters using the local information at every node. It utilizes simulated annealing and helps to achieve load balance among servers. In JA-BE-JA, each node of the graph is considered as an independent processing unit. As a gossip based algorithm, each node in the graph selects a neighbor and gossip about its neighbors on the same server. Depending on the information, nodes may exchange their servers and place itself close to its neighbors. In our work, we used JA-BE-JA to split the social graph and create replicas for the nodes that are not local, so that we maintain data locality for all one-hop neighbors.

JA-BE-JA follows a local search algorithm, where each node initially selects a color uniformly at random. The color of a node and its neighbors define the energy of a node, i.e., if a node and most of its neighbors have same color, it will have low energy. Each node in the social graph tries to acquire low energy by moving to the same server with its neighbors. For example, suppose $c_p$ indicates the color of node $p$ and $N_p$ denotes the neighbor of node $p$. We can define $x_{p(c)}$ as the number of neighbors of node $p$ having the same color $c$. Two nodes $p$ and $q$ with colors $c_p$ and $c_q$ only exchange their servers if they satisfy equation 3.6, where $\alpha$ is a parameter of the energy function [18].

\[
x_{p(c)} = \{ q \in N_p : c_q = c \} \quad (3.5)
\]

\[
x_{p(c_q)}^{\alpha} + x_{q(c_p)}^{\alpha} > x_{p(c_p)}^{\alpha} + x_{q(c_q)}^{\alpha} \quad (3.6)
\]

To achieve clusterization, each node periodically selects another node from the graph using one of the peer selection strategies, i.e., Local, Random and Hybrid. Local search selects any of the direct neighbors from a node. For random peer selection, it uses a random walk of fixed length to get a distant neighbor and hybrid peer selection is a combination of both local and random search in which algorithm initially selects a direct neighbor and calculate the pair-wise benefit for server exchange. In case, when the algorithm does not achieve a better placement, it tries to select a distant neighbor to perform the server exchange. After peer selection, both nodes calculate their energy and exchange their servers if it results in lower energy for both the nodes. As JA-BE-JA is a local search algorithm, it is probable that the algorithm may get stuck in local optima. To avoid local optima it uses simulated annealing technique. Each node in the graph execute the same algorithm and achieve the optimal placement after the fixed number of iterations. We perform the experiments to tune different parameters for JA-BE-JA and details regarding
those parameters are mentioned in Chapter 6.
Gossip-based Partitioning and Replication Middleware

In previous chapter, we defined requirements of the system and illustrated a minimum replica problem that can be used to fulfill the requirements. In this chapter we present our heuristic, that fulfill aforementioned requirements. Our heuristic is a gossip-based partitioning and replication algorithm, in which nodes perform gossip in order to reduce the number of replicas across the system. It is a dynamic local search algorithm, which periodically adjusts the partitions utilizing the local information at every node. In this chapter, we discuss the system design and different operations that we used in our algorithm.

4.1 System Design

We propose a gossip-based local search algorithm that runs on top of data-stores and efficiently places the users across the system. The algorithm aims to provide high availability for OSN users while compromising consistency for replicas. It periodically executes and adjusts the partitions in order to reduce the number of replicas. We assume that each user in the network is an independent processing unit, e.g., in case of 100,000 users and 10 servers, each server will act on behalf of 10,000 users. Our approach is capable of executing in both distributed and central environment and is capable of handling millions of users and connections.

4.1.1 Architecture

Figure 4.1 shows the integration of the proposed middleware in a three-tier web architecture, which is influenced by the architecture of SPAR. The middleware is implemented on top of data-stores, e.g., Cassandra or MySQL, and can only be triggered by an application. It contains a directory service that is used by applications to communicate with middleware and to get the address of a specific user on the data-store. Applications can use the specific drivers,
like MySQL driver or Cassandra API, to interact with the data-store and can parse all the queries locally from any server. The middleware takes the responsibility of placement of users and their replicas, thus managing the distributed architecture and providing transparent scalability to the application.

Figure 4.1: Middleware integrated into three-tier web architecture

4.2 System Operations

In this section we discuss different operations that are used by our algorithm to achieve its goal of minimizing the number of replicas across the system.

4.2.1 Gossip Protocol

In our work, we propose a gossip-based protocol in which nodes communicate with each other to achieve a common goal of minimizing the number of replicas. Gossip protocols play an important role in modern large-scale distributed systems. Recently, gossip-based protocols have achieved popularity in peer-to-peer community due to its scalability, simplicity and robustness. In gossip-based protocols information spread in the same way as rumors are spread in a society. Such protocols are also called epidemic protocols as the information spread in a similar to viral infection among group of people. The frequencies of interaction for topology management in gossip-based protocols
are less compared to other messages so that the communication cost is low. Such protocols can be used in various applications like message broadcasting, overlay management, aggregations, and clustering. As OSNs exhibit small world properties [9], it helps gossip-based protocols to spread the information widely and rapidly and make gossip protocols an attractive choice for OSNs.

4.2.2 Peer Selection

Each node in the network executes a gossip-based algorithm, where it periodically selects a peer to perform a server exchange. This server exchange helps nodes to move to the servers that can reduce the number of replicas in the system. We used three different schemes, i.e., 1) Local, 2) Random and 3) Hybrid, for peer selection and are mentioned below:

- **Local**: In local peer selection, the algorithm selects any of the direct neighbor of a user uniformly at random.

- **Random**: Different algorithms exists to get the random view of a graph, starting from a node without having knowledge of the complete graph. For example, there are different algorithms that perform random walks based on Markov Chains [45]. These algorithms start from a node and walk random number of steps along the edges, and returns the last visited node. In our algorithm, we used random walk of fixed length to select a distant neighbor in the social graph.

- **Hybrid**: Hybrid selection policy is a combination of local and random selection policy. In this method the algorithm first selects a direct neighbor and tries to calculate the pair-wise benefit. In case of no gain, algorithm selects a distant neighbor using a random walk to perform the server exchange [18].

4.2.3 Cost Function

In this section, we discuss the cost function that algorithm uses to calculate the cost of a node on a server. The cost of a user on any server is represented by the number of replicas that the node requires to maintain the data locality. We have two different cost function to calculate the cost, i.e., 1) cost on the server where user already exists and 2) cost for the new server. Suppose we have a graph $G = (V,E)$, and the algorithm randomly selects a node $p \in V$. Node $p$ has multiple copies across the servers ($0...k-1$), i.e., master copy $s_p$ and set of replicas $S'_p$. As graph $G$ represents a social graph, node $p$ may have set of neighbor $N_p$. Below mentioned functions are used to calculate the cost of a node on its existing server.

$$L(s) = \sum_{i \in N_p} 1 (s_i = s) \quad (4.1)$$
Equation 4.1 calculates the sum of neighbors of a node that exists as a master on a server \( s \). Equation 4.2 denotes a decision variable, which is equal to 1 if a user \( i \) exists on server \( s \) as a replica. Equation 4.3 denotes a decision variable, which is equal to 1 if no neighbor of user \( i \) has more than one master on server \( s \). Equation 4.4 denotes the cost function that is the sum of all the neighbors that exists as a replica and has no other neighbor as master on the same server. These equations will calculate the cost of a node if it already exists on a server. For cost of a node on a new server, the algorithm counts the neighbors that do not exist on a new server, either as master or a replica.

### 4.2.4 Simulated Annealing

Our algorithm continuously keeps the social graph partitioned into equal size clusters. To avoid the algorithm getting stuck into local optima, we used simulated annealing technique [20]. Simulated annealing is a metaheuristic that is used to solve the global optimization problem of finding a global optima in a large search space. It works in a way that it introduces a noise into the system so that the system can perform moves that are not allowed in the normal condition and escape from the local optima. The noise eventually fades out from the system depending on the cooling rate (\( \delta \)) and the algorithm again converges to the value close to the global optima. We performed experiments (see chapter 5) to set up parameters for simulated annealing, i.e., cooling rate (\( \delta \)) and initial temperature (\( T_0 \)). For cooling rate, we used two different types of schemes, i.e., (1) constant and (2) exponential. Cooling rate is directly proportional to the convergence time of the algorithm and can be varied depending on the requirements.

### 4.3 Algorithm

#### 4.3.1 Greedy Algorithm

The algorithm initiates by taking an input graph in the form of edge list and distributes the users equally among \( k \) servers. When a new node joins the network, the algorithm assigns it to the server with minimum number of masters and creates the required number of replicas across the servers, for fault tolerance. In case of edge addition, the algorithm checks if both nodes
Algorithm 1 Peer Selection

Require: any node \( p \) in the graph has following methods:
- \( \text{getLocalPeer}(p) \): returns any neighbor of node \( p \) uniformly at random.
- \( \text{getDistantPeer}(p) \): return a distant neighbor using random walk.

1: \begin{procedure} \text{selectPeer}(p) \end{procedure}
2: \hspace{1em} \text{neighbor} \leftarrow \text{getLocalPeer}(p)
3: \hspace{1em} \text{if} \ \text{swap}(p, \text{neighbor}) \neq \text{True} \ \text{then}
4: \hspace{2em} \text{neighbor} \leftarrow \text{getDistantPeer}(p)
5: \hspace{2em} \text{swap}(p, \text{neighbor})
6: \hspace{1em} \text{end if}
7: \hspace{1em} T_r \leftarrow T_r - \delta
8: \hspace{1em} \text{if} \ T_r < 1 \ \text{then}
9: \hspace{2em} T_r \leftarrow 1
10: \hspace{1em} \text{end if}
11: \text{end procedure}

Algorithm 2 Swap Servers

Require: any node \( p \) in the graph has following methods:
- \( \text{getCost}(p) \): get cost for node \( p \) on its server
- \( \text{getCost}(p, \text{server}) \): get cost for node \( p \) on new server \( s \)

1: \begin{procedure} \text{Swap}(p, q) \end{procedure}
2: \hspace{1em} u_{p}(s_p) \leftarrow \text{getCost}(p)
3: \hspace{1em} u_{q}(s_q) \leftarrow \text{getCost}(p)
4: \hspace{1em} u_{p}(s_q) \leftarrow \text{getCost}(p, s_q)
5: \hspace{1em} u_{q}(s_p) \leftarrow \text{getCost}(p, s_p)
6: \hspace{1em} \text{if} \ u_{p}(s_q) + u_{q}(s_p) + 2 < (u_{p}(s_p) + u_{q}(s_q)) * T_r \ \text{then}
7: \hspace{2em} \text{move}(p, s_q)
8: \hspace{2em} \text{move}(q, s_p)
9: \hspace{2em} \text{return True}
10: \hspace{1em} \text{else}
11: \hspace{2em} \text{return False}
12: \hspace{1em} \text{end if}
13: \text{end procedure}

Algorithm 3 Move a node

Require: any node \( p \) exists on server \( s_p \) requires methods:
- \( \text{isMaster}(p, s) \): returns True if master of node \( p \) exists on server \( s \)
- \( \text{insertMaster}(p, s) \): creates master of node \( p \) on server \( s \)
- \( \text{replicaExists}(p, s) \): returns True if replica of node \( p \) exists on server \( s \)
- \( \text{insertReplica}(p, s) \): creates a replica of node \( p \) on server \( s \)
- \( \text{deleteReplica}(p, s) \): removes the replica of node \( p \) from server \( s \)

1: \begin{procedure} \text{Move}(p, s_q) \end{procedure}
2: \hspace{1em} \text{for all} \ \text{neighbor} \in N_p, \ \text{do}
3: \hspace{2em} \text{if} \ \text{isMaster}(\text{neighbor}, s_q) \ \text{then}
4: \hspace{3em} \text{insertReplica}(\text{neighbor}, s_q)
5: \hspace{2em} \text{else}
6: \hspace{3em} \text{if} \ \text{isMaster}(\text{neighbor}, s_q) \neq \text{True} \ \text{then}
7: \hspace{4em} \text{insertReplica}(\text{neighbor}, s_q)
8: \hspace{3em} \text{end if}
9: \hspace{2em} \text{end if}
10: \hspace{1em} \text{end for}
11: \hspace{1em} \text{if} \ \text{replicaExists}(p, s_q) \ \text{then}
12: \hspace{2em} \text{deleteReplica}(p, s_q)
13: \hspace{2em} \text{insertMaster}(p, s_q)
14: \hspace{2em} \text{else}
15: \hspace{3em} \text{insertMaster}(p, s_q)
16: \hspace{2em} \text{end if}
17: \text{end procedure}
exist on the same server and in case of different servers algorithm creates their replicas to maintain data locality.

### 4.3.2 Server Exchange

Our proposed algorithm is a periodic algorithm in which each node periodically selects a neighbor to perform gossip. After the peer selection, both the user and its selected neighbor calculates the number of replicas that are required for data locality on their servers. For example, suppose node $A$, which has its master on server 1 and node $B$, which has its master on server 2, are two nodes in the graph and node $A$ selects node $B$ for the server exchange. The algorithm utilizes the cost function (which we discuss above) to calculate number of replicas for both the nodes $A$ and $B$ to exists on both the servers 1 and 2. The nodes only exchange their servers in case of server exchange resulting in lower number of replicas. Our algorithm uses equation 4.7 to decide if it has to perform the server exchange between node $p$ and $q$. The equation calculates the cost of both the nodes on their existing and new servers. Moreover, the equation also contains an additional parameter ($\text{adjust}$), which accounts for the required replicas of the nodes on their existing server in case of their movement to the new server. Algorithms 1, 2 and 3 show algorithms for peer selection, swapping servers and moving node from one server to another server respectively.

\[
\text{adjust}(p,q) = \begin{cases} 
0 & \text{if } e(p,q) \in E \\
t(p,s_p,s_q) + t(q,s_q,s_p) & \text{otherwise} 
\end{cases} 
\quad (4.5)
\]

\[
t(i,s_p,s_q) = \begin{cases} 
0 & L(s_p) > 0 \text{ and } L(s_q) > 0 \\
1 & L(s_p) > 0 \text{ and } L(s_q) = 0 
\end{cases} 
\quad (4.6)
\]

\[
X_p(s_p) + X_q(s_q) + \text{adjust}(p,q) > X_p(s_q) + X_q(s_p) \quad p, q \in N 
\quad (4.7)
\]
Tuning Parameters

5.1 Datasets

We used three different type of datasets in our experiments, i.e., 1) Facebook Graphs, 2) Twitter Graph and 3) Synthetic Graphs.

1. Facebook Graph

Facebook is among the top most OSNs, with user base of 1,110 million [46]. Facebook users are more connected than Orkut users, e.g., 37% of Facebook users have more than 100 friends, compared to 20% for Orkut [6]. However, both these networks show small world properties and follow power-law social degree distribution. For our experiments we used two different facebook graphs. We took the first graph from Stanford Large Network Dataset Collection [47], which contains 4,039 nodes and 88,234 edges. We took the second graph from Online Social Networks Research @The Max Planck Institute of Software Systems [48], which contains 60,290 nodes and 1,545,686 edges.

2. Twitter Graph

Twitter differs from other OSNs in network topology. Instead of relationship twitter users follow each other. Twitter social graph does not show power-law distribution for social degrees, have a short effective diameter and show very low reciprocity [49]. For our experiments, we took the twitter graph from Stanford Large Network Dataset Collection [47], which contains 81,306 nodes and 1,768,149 edges.

3. Synthetic Graph

We have generated five different synthetic graphs, which are described below.

- **clustered graph (Synth-C)**: This graph contains of 2000 nodes, indexed from 1 to 2000. Each nodes has 10 different neighbors and the graph is divided into sixteen clusters, where each node connects to a node inside its cluster with 75% probability.
• **highly clustered graph (Synth-HC):** This graph is similar to Synth-C, in terms of number of nodes, clusters and node degree. However, in this graph each node connects with a node inside its cluster with 95% probability.

• **clustered graph (Synth-LC):** This graph contains 1000 nodes and generated on the same pattern like Synth-C graph, expect that there exist four clusters in this graph, instead of sixteen.

• **highly clustered graph (Synth-LHC):** It is a highly synthesized graph of 1000 nodes with four clusters, where each nodes connects to a node inside its cluster with 95% probability. Each node in the graph connects with 10 other nodes, which means that the average degree of the graph is 20.

• **power-law graph (Synth-PL):** we generated a power-law graph using Python Web Graph Generator [50]. Power Law graphs are random graphs with power law degree distribution, in which nodes connects to other nodes uniformly at random, while maintaining the power law degree distribution.

Table 5.1 summarizes all the graphs that we use in our experiments.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Nodes</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synth-LC</td>
<td>1,000</td>
<td>10,000</td>
</tr>
<tr>
<td>Synth-LHC</td>
<td>1,000</td>
<td>10,000</td>
</tr>
<tr>
<td>Synth-C</td>
<td>2,000</td>
<td>20,000</td>
</tr>
<tr>
<td>Synth-HC</td>
<td>2,000</td>
<td>20,000</td>
</tr>
<tr>
<td>Synth-PL</td>
<td>2,000</td>
<td>20,000</td>
</tr>
<tr>
<td>Snap-Facebook</td>
<td>4,039</td>
<td>88,234</td>
</tr>
<tr>
<td>WSON-Facebook</td>
<td>60,290</td>
<td>1,545,686</td>
</tr>
<tr>
<td>Snap-Twitter</td>
<td>81,306</td>
<td>1,768,149</td>
</tr>
</tbody>
</table>

Table 5.1: Description of Datasets

### 5.2 Length of Random Walk

As discussed in chapter 4, our algorithm requires to perform random walk for random peer selection. In this experiment, we perform random walks for a different number of steps \(m\) and tune the length of random walks that can give us optimal results. We have used three different datasets (from the graphs mentioned above), and distributed the graphs on sixteen servers with replication factor of two. This experiment was performed without simulated annealing. Figure 5.1 shows the replication overhead for different size of random walks. As can be seen, random walk for length greater than six yields optimal results for peer selection. We can observe the replication overhead
for three different datasets. The power law graph generates the maximum replication overhead compared to other graphs due to existence of randomness in the graph. The synthetic low clusterized graph generates high replication overhead, as the graph contains four clusters. Such distribution for graph with four clusters on sixteen servers generates many edge-cuts due to existence of randomness within the clusters. Facebook graph generates the minimum replication overhead due to existence of small world properties within the real world graphs, i.e., short average diameter and high clustering coefficient. We used six \((m=6)\) as length of random walk for rest of the experiments. This experiment also validate the existence of small world properties in OSNs, as it proves that the existence of six degree separation among the OSN users.

![Graph](image)

**Figure 5.1:** replication overhead for different length of random walks for random peer selection with \(k=16\) and \(f=2\).

### 5.3 Peer Selection

In this experiment, we compared different peer selection techniques that we discussed in chapter 4, i.e., 1) Local, 2) Random, and 3) Hybrid. Table 5.2 shows the replication overhead for the three schemes, using different datasets. For this experiment, datasets were distributed on sixteen servers with replication factor of two. The algorithm achieves optimal results with hybrid peer selection policy, which is combination of local and random peer selection. However, improvements from random peer selection to hybrid peer selection are not very high. As local peer selection is not very expensive, we choose hybrid peer selection to perform further experiments.
Table 5.2: Replication overhead for different peer selection strategies with \( m=6, k=16 \) and \( f=2 \).

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Local</th>
<th>Random</th>
<th>Hybrid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synth-LC</td>
<td>8847</td>
<td>6814</td>
<td>6785</td>
</tr>
<tr>
<td>Synth-LHC</td>
<td>9027</td>
<td>5219</td>
<td>4894</td>
</tr>
<tr>
<td>Synth-C</td>
<td>16531</td>
<td>11794</td>
<td>11895</td>
</tr>
<tr>
<td>Synth-HC</td>
<td>16125</td>
<td>8077</td>
<td>8136</td>
</tr>
<tr>
<td>Synth-PL</td>
<td>15918</td>
<td>14666</td>
<td>14579</td>
</tr>
<tr>
<td>Snap-Facebook</td>
<td>29271</td>
<td>18194</td>
<td>17702</td>
</tr>
<tr>
<td>WSON-Facebook</td>
<td>366690</td>
<td>228988</td>
<td>228905</td>
</tr>
<tr>
<td>Snap-Twitter</td>
<td>349890</td>
<td>197172</td>
<td>197061</td>
</tr>
</tbody>
</table>

5.4 Simulated Annealing

In this section, we describe experiments that are performed to tune parameters for simulated annealing, i.e., (1) the cooling rate \( \delta \) and (2) Initial Temperature.

5.4.1 Noise Delta

Constant Noise Delta

This experiment was performed for constant cooling rate \( \delta \), i.e., the cooling rate [51]. Initially, noise is introduced in the system, which is removed from the system at a constant rate \( \delta \), as described in the equation (5.1). In this experiment, we used different noise delta \( \delta \) and compute the replication overhead and number of swaps that are required in order to minimize the replication overhead. Swap count is directly proportional to the time required for algorithm to reach the global optima. Figures 5.2, 5.3, 5.4, 5.5 and 5.6 show graphs for five different datasets, i.e., Synth-C, Synth-HC, Synth-LC, Synth-PL and Snap-Facebook. In this experiment, these graphs were distributed on sixteen servers with replication factor of two. As can be seen in the graphs that lower delta values generates lower replication overhead, but takes longer execution time and higher noise delta finishes the processing faster but generates higher replication overhead.

\[
T(t) = T(t-1) - \delta
\]  (5.1)
Figure 5.2: swap count and replication overhead for different values of noise delta for synthetic clusterized graph with $k=16$ and $f=2$.

Figure 5.3: swap count and replication overhead for different values of noise delta for synthetic highly clusterized graph with $k=16$ and $f=2$. 
Figure 5.4: swap count and replication overhead for different values of noise delta for synthetic clusterized graph with four clusters with $k=16$ and $f=2$.

Figure 5.5: swap count and replication overhead for different values of noise delta for synthetic power law graph with $k=16$ and $f=2$. 
Figure 5.6: swap count and replication overhead for different values of noise delta for snap facebook graph with $k=16$ and $f=2$. 
Exponential Noise Delta

In this experiment, we used exponential rate of change of temperature \( t \), i.e., the cooling rate \([51]\). Equation 5.2 describes the equation for rate of change of temperature. The algorithm initially introduces a noise into the system and allows system to perform moves to escape from the local optima. Later, the system cools down depending on the rate of change of temperature \( t \) and approaches the global optima. We perform experiments to find values for two different parameters: (1) alpha \( (\alpha) \) and rate of change of temperature \( t \). Figure 5.7 and 5.8 show graphs with different alpha values for two different datasets, i.e., Synth-C and Snap-Facebook. As can be seen, changing alpha values does not have larger impact on the replication overhead and the swap count.

\[
T(t) = T_f + T_o \alpha^{-t}
\] (5.2)

Figure 5.7: swap count and replication overhead for different values of alpha for synthetic clusterized graph with \( k=16 \) and \( f=2 \).
Figure 5.8: swap count and replication overhead for different values of alpha for snap facebook graph with $k=16$ and $f=2$.

In second part of experiment we vary the exponential rate of change temperature and observe the replication overhead and number of swaps. Figures 5.9 and 5.10 show graphs for two different datasets, i.e., Synth-C and Snap Facebook. As can be seen, in both the graphs, lower delta values generates lower replication overhead, but takes longer execution time to reach the global optima and higher noise delta converges to global optima faster but generates higher replication overhead. After comparison of both the results, i.e., (1) constant cooling rate and (2) exponential cooling rate. We can conclude that there is no unique choice for rate of change of temperature for simulated annealing and we can choose different schemes depending if we want faster convergence, or lower replication overhead. Exponential rate of change of temperature may yield low replication overhead, but will require higher execution time (number of swaps) to reach the global optima. Constant rate of change of temperature reaches the lower replication overhead with less number of swaps compared to exponential rate of change. Therefore, we use constant rate of change of temperature in further experiments.
Figure 5.9: swap count and replication overhead for different values of exponential rate of change of temperature, for synthetic clusterized graph with sixteen clusters with $k=16$ and $f=2$.

Figure 5.10: swap count and replication overhead for different values of exponential rate of change of temperature, for facebook dataset from SNAP with $k=16$ and $f=2$. 
5.4.2 Initial Temperature

In this experiment, we used constant rate of change of temperature with constant noise delta ($\delta=0.003$) and varied the initial temperature for simulated annealing. Figures 5.11, 5.12, 5.13, 5.14 and 5.15 show the results for different initial temperature for five different datasets, i.e., Synth-C, Synth-HC, Synth-LC, Synth-PL and Snap Facebook. As can be seen, that synthetic graphs do not show much deviation in replication overhead with change initial temperature. However, Facebook graph shows improvements in results for different initial temperature. Based on experiments we used $T_0=2$ in further experiments.

![Graph showing swap count and replication overhead for different values of Initial Temperature with synthetic clusterized graph with $k=16$ and $f=2$.](image)

Figure 5.11: swap count and replication overhead for different values of Initial Temperature with synthetic clusterized graph with $k=16$ and $f=2$. 
Figure 5.12: swap count and replication overhead for different values of Initial Temperature with synthetic highly clusterized graph with $k=16$ and $f=2$.

Figure 5.13: swap count and replication overhead for different values of Initial Temperature for synthetic low clusterized graph with $k=16$ and $f=2$. 
5.4. Simulated Annealing

Figure 5.14: swap count and replication overhead for different values of Initial Temperature for synthetic power law graph with $k=16$ and $f=2$.

Figure 5.15: swap count and replication overhead for different values of Initial Temperature for SNAP facebook graph with $k=16$ and $f=2$. 
5.5 Number of iterations

In this experiment, we fix simulated annealing parameters, i.e., noise delta \( \delta = 0.003 \) and initial temperature \( T_0 = 2 \), and vary the number of iterations. Figures 5.16 and 5.17 show the ratio of replication overhead for our algorithm and random partitioning for different number of iterations. As can be seen, the algorithm reaches global optima after 200 iterations for all the graphs. We achieved more gain in replication overhead in real world graphs as compared to synthetic graphs. This behavior can be attributed to the lack of small world properties in the synthetic graphs.

![Figure 5.16: ratio of replication overhead for our algorithm and random partitioning for different number of iterations using synthetic graphs with \( k=16 \) and \( f=2 \).](image1)

![Figure 5.17: ratio of replication overhead for our algorithm and random partitioning for different number of iterations using real world graphs with \( k=16 \) and \( f=2 \).](image2)
We have implemented our algorithm, SPAR and JA-BE-JA in Python. Additionally, we compared our algorithms with de-facto random partitioning system. We used replication overhead as a metric for comparison between algorithms. Table 6.1, 6.2 and 6.3 show values for different parameters that were used during experiments.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial Temperature ($T_o$)</td>
<td>2</td>
</tr>
<tr>
<td>Final Temperature ($T_f$)</td>
<td>1</td>
</tr>
<tr>
<td>Cooling Factor ($\delta$)</td>
<td>0.003</td>
</tr>
<tr>
<td>Length of Random Walk ($m$)</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 6.1: Parameters for our algorithm

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial Temperature ($T_o$)</td>
<td>2</td>
</tr>
<tr>
<td>Final Temperature ($T_f$)</td>
<td>1</td>
</tr>
<tr>
<td>Cooling Factor ($\delta$)</td>
<td>0.003</td>
</tr>
<tr>
<td>Energy Function Parameter ($\alpha$)</td>
<td>2</td>
</tr>
<tr>
<td>Length of Random Walk ($m$)</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 6.2: Parameters for JA-BE-JA

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Balancing Factor</td>
<td>0.2</td>
</tr>
</tbody>
</table>

Table 6.3: Parameters for SPAR

Our algorithm initiates by taking as input a social graph in the form of edge list. The algorithm partitions the graph and distributes the nodes across
the servers. Each node is assumed to be an independent processing unit, and it periodically executes the algorithm. Therefore, when a new node joins the network, it eventually places itself in the existing structure with minimum number of replicas.

6.1 Experiments

We designed four different experiments to compare the replication overhead of our proposed algorithm with random partitioning, JA-BE-JA and SPAR, i.e., (a) with different datasets, (b) with different replication factors, (c) with different number of servers, and (d) dynamic behavior.

6.1.1 Evaluation with Datasets

![Graph comparison of replication overhead](image)

Figure 6.1: Comparison of replication overhead of our algorithm, random partitioning, SPAR and JA-BE-JA for different datasets with $k = 16$ and $f = 2$.

Figure 6.1 shows the graph comparing replication overhead of our algorithm with random partitioning, SPAR and JA-BE-JA for different datasets. The experiment was performed on 16 servers ($k = 16$) with replication factor of two ($f = 2$). As can be seen, our algorithm generates minimum replication overhead compared to all the other algorithms. Specially, the gain compared to random partitioning, which is the de-facto standard, is more than 3x for some of the datasets. Our algorithm reduces the replication overhead by factor of two compared to SPAR in case of high clusterization in the graph. This behavior was expected, as our algorithm takes into account the global picture of the graph and avoids the load balancing constraint that is the bottleneck in
6.1. Experiments

SPAR algorithm. The graph partitioning algorithm generates low replication overhead in case of synthetic graphs, due to existence of prominent clusters. However, it generates higher replication overhead for real world graphs, as it solves the minimum edge cut problem, which is different from minimum replica problem. Our algorithm performs better than all the other algorithms and generates lower replication overhead for both synthetic and real world graphs.

6.1.2 Evaluation with Replication Factors

In this experiment, we observe the behavior of our algorithm for different replication factors, i.e., $f = 0, 2$. In the graph, we report replication overhead that is required for data locality, while ignoring the necessary replication overhead that exists due to replication factor (fault tolerance). This experiment was performed on 16 servers ($k = 16$). Figure 6.2 shows the replication overhead for both the scenarios. As can be seen, replication overhead decreases with the increase of replication factor for all the datasets, as the replicas that exist for fault tolerance in the system also help in achieving data locality. Additionally, we can observe that replication overhead for the real time graphs are very low in all the cases. Therefore, we can induce that our algorithm gives better results in case of OSNs due to existence of small world properties in OSN graphs.

![Figure 6.2](image)

Figure 6.2: Replication overhead for different fault tolerance replicas, i.e., $f = 0, 2$ with $k=16$. 
6.1.3 Evaluation with Servers

In this experiment, we measure the performance of our algorithm, random partitioning, SPAR and JA-BE-JA for different number of servers, i.e., \( k = 8, 16, 32, \) and 64. The replication factor is two, i.e., at least two replicas for each node should exist in the system. Figure 6.3 shows multiple graphs for different datasets, where each graph is generated by varying the number of servers. As shown, our algorithm performs better than all the other algorithms in most of the cases. We achieve reduction up to four times in case of random partitioning in the real world graphs. This trend can be due to the fact that our algorithm takes benefit from the social structure of the graph. The gain compared to SPAR and JA-BE-JA is also prominent. Our algorithm generates lower replication overhead compared to SPAR, as SPAR is a sequential algorithm, which does not take into account the global picture of the graph and contains the load balancing constraint which results into high replication overhead. Moreover, JA-BE-JA solves the problem for the minimum edge cut, which is not similar to minimum number of replicas and generates higher number of replicas compared to our proposed algorithm. Additionally, we can observe that the replication overhead is not linearly proportional to the number of servers and replication overhead reduces with the increase of servers. Hence, our algorithm scales with the increase of the number of servers and generates lower replication overhead compared to all the other algorithms.
Figure 6.3: Replication overhead for different datasets and different number of servers with $f = 2$. 
6.1.4 Dynamic Behavior

In this experiment, we evaluate the performance of our algorithm in case where new edges are continuously created among nodes in the social network. We use real world graphs to perform this experiment using sixteen servers with replication factor of two, i.e., $k=16$ and $f=2$. We divide the social graphs into small components, where edges are created between nodes periodically to simulate the dynamic behavior of OSNs. For example, if we have 100,000 edges we divide it into 10 components and add 10,000 edges in the graph periodically until it covers the complete graph. Figure 6.4, 6.5 and 6.6 show the replication overhead for the social graphs. We add new edges in the graph after every 50 cycles, which create a spike in the graph in terms of replication overhead. Our algorithm runs periodically and tries to adjust the partition, whenever new edges are added to the graph. It takes benefits from the distributed nature of algorithm and adjusts efficiently with dynamic behavior of the graph without affecting the existing structure of the graph. Further, we can observe that algorithm converges rapidly after the addition of new edges and achieve the minimum replication overhead after very few iterations.

Figure 6.4: Replication overhead versus number of iterations for SNAP Facebook graph with $f=2$ and $k=16$. 

![Figure 6.4: Replication overhead versus number of iterations for SNAP Facebook graph with $f=2$ and $k=16$.](image)
Figure 6.5: Replication overhead versus number of iterations for WSON Facebook graph with \( f = 2 \) and \( k = 16 \).

Figure 6.6: Replication overhead versus number of iterations for SNAP Twitter graph with \( f = 2 \) and \( k = 16 \).
7.1 Discussion

OSNs have gained enormous popularity in the last decade and created many new business opportunities for new companies. OSNs differ from other web applications due to presence of high workload and distinct interaction structure among its users. Many solutions have been proposed targeting different issues, i.e., scalability, availability, consistency, and management, which exist due to complex nature of OSNs. In our work, we proposed a gossip based partitioning and replication middleware that also targeted the problem related to management and scalability for OSNs. We utilized the social structure of OSNs to partition the social graph for storage and processing. Our proposed approach promises high availability and guarantees data locality for OSN users within one-hop neighborhood, i.e., the information related to a user and its friends will always be available on a local server. We achieved data locality by creating local copies of neighbors that were located on other servers. However, this approach has a potential for further improvements, as OSN users do not interact with all of their friends, i.e., for 90% of Facebook users, 70% of their interaction is generated from top 20% of their friends [6]. In our work, we focus on cheap replica maintenance schemes (e.g., eventual consistency) and trade off strong consistency for availability. For future work, we will look at broader range of consistency models and adaptive consistency.

An alternative way to partition OSNs is to use the interaction graph, instead of a social graph [52]. Such interaction or activity based graphs are created based on interaction among users. Partitioning based on such graphs will avoid the excessive replication that was caused in the case of partitioning based social structure. However, we have to consider the fact that only 30% of Facebook users interact consistently over time and the interaction patterns keeps on changing during the life time of a user [17]. Hence, it creates challenges to maintain an activity based graph, which continuously changes over time. Moreover, modern OSNs deal with billion of users and maintaining large scale graphs that evolves with time may require additional resources.
An attractive solution for OSNs can be a combination graph partitioning and activity based graph, where we can only replicate nodes which produce interactions greater than a certain threshold. For other nodes we can use any graph partitioning algorithm to reduce the edge cuts between the partitions.

Benevenuto et al. [5] identified that OSNs follow a skewed power-law degree distribution, which means that there are many users in a social graph with very few neighbors and small amount of users exist with high social degrees. For example, one percent of twitter users are adjacent nearly to half of the twitter users [1]. Gonzalez et al. [1] in their work, on PowerGraph, proposed to partition the social graph by cutting the nodes instead of edges, which means that by replicating high degree nodes we can achieve high the data locality for OSNs. We believe that we can also take benefit from the existence of power-law degree distribution in social graphs and improve our algorithm by only replicating the nodes with higher social degrees. Such data distribution will not promise data locality for all the users but will reduce the replication overhead by great amount.

In the experiments we showed that our algorithm reduces the replication overhead compared to other algorithms. Our algorithm achieved four times less replication overhead compared to random partitioning, as it takes benefit from the social structure of the graph. It also reduces the replication overhead compared to SPAR, as SPAR is a sequential algorithm and it may get stuck in the local optima. Moreover, SPAR uses three different configurations to evaluate different node placement during edge addition. However, existence of load balance constraint in SPAR restricts two of the three configurations in most of the scenarios (see chapter 3 for description). Our algorithm uses a gossip protocol to solve the optimization problem for minimizing the number of replicas and achieve better scalability due to the distributed design. Moreover, our algorithm generates low replication overhead compared to SPAR as it takes into account the global picture of the graph and uses simulated annealing to escape from the local optima. Furthermore, our algorithm performs better than graph partitioning algorithm as it solves the problem of minimizing the number of replicas across the network, which is different from minimizing the edge-cuts between the partitions (see chapter 1 for description). We demonstrated that our algorithm is capable of handling dynamic nature of OSNs, as the distributed model of our algorithm helps it to converge faster.

As our proposed algorithm is a heuristic based solution, we used simulated annealing to escape the local optima. We performed extensive experimentation with different parameters of simulated annealing. Results showed that simulated annealing can help in reducing the number of replicas across the network. We performed experiments with constant and exponential rate of change of temperature. Exponential rate of change of temperature converges faster compared to the constant rate of change of temperature, but it
generates higher replication overhead. Whereas, constant rate of change of temperature generates lesser replication overhead and takes longer time to converge. We can not conclude which scheme is optimal for choice for OSNs, as both can be used depending on the requirements. We also observed that our algorithm was unable to gain benefit from simulated annealing in case of random graphs, as random graphs does not contain explicit clusters and there are no valid moves to improve the replication overhead.

In our work, we used eventually consistent copies of user data (replicas) and distributed those copies across the network to achieve data locality. We only guarantee consistency for a single copy (master) of a user in the network and use that copy to achieve load balance across the servers. Each master copy is responsible for recent reads and writes and later, those writes are propagated across the network to all the replicas. However, balancing the replicas across the servers, along with master copies, may result in different partitions and can be used to implement tunable consistency models for the system. We will address this problem of achieving load balance between the partitions using both masters and replicas in the future. Furthermore, we tested our algorithm for undirected graphs, as SPAR and JA-BE-JA both works for undirected graph. Our algorithm is capable of handling directed graph and reducing the replication overhead for directed graphs, e.g., Twitter graph.

In experiments, we evaluated all the algorithms using different type of datasets. We noticed that the replication overhead varies depending on the nature of a social graph and such properties of the graph can play an important role in providing a better design for its management. We used synthetic graphs with explicit clusters to verify if our algorithm is capable of partitioning the social graph into independent components. We also used random graphs with power law degree distribution and our algorithm verified that random graphs are difficult to partition and generate high replication overhead. Our experiments with real world graphs validated the existence of small world properties in the social graph and generated lower replication overhead. This behavior is due to the fact that social graphs contain hierarchical clustering, which helps in achieving better partitioning. We can conclude that most of modern social networks exhibit small world properties, which can help in better partitioning and distribution of social graph in a network. However, it is necessary to understand the nature of the graph prior to its deployment.

## 7.2 Conclusion

We proposed a distributed partitioning and replication middleware, which efficiently places the OSN data across the network in order to provide high availability and scalability for OSNs. We demonstrated that our proposed algorithm provides fault tolerance and guarantees data locality within one-hop
neighborhood through replication. We designed and implemented our algorithm and compared its performance with random partitioning, SPAR and a distributed graph partitioning algorithm, for three different types of datasets, i.e., (1) Facebook graphs, (2) Twitter graphs, and (3) Synthetic graphs. Results showed that our algorithm outperformed random partitioning strategy and generates four times less replication overhead. Moreover, our algorithm reduced the replication overhead by a factor of two compared to SPAR in case of high clusterization in the social graph. Our algorithm was able to scale on larger number of servers with lower replication overhead, compared to SPAR, due to its distributed architecture. We have also demonstrated that our algorithm is capable of handling the dynamic nature of OSNs and is able to handle high churn rates that are present in OSNs. Our experiments also showed that random graphs are difficult to partition and generates higher replication overhead as compared to OSN graphs. We implemented the algorithm for functionality testing and tested our algorithm using graphs up to 2 million edges. However, we left the implementation and the simulation of our algorithm using distributed graph processing frameworks, e.g., GraphLab and Apache Giraph, with larger datasets for future work.
Bibliography


[50] Saket Sathe. *Python Web Graph Generator*.
