PARTITIONING SOCIAL NETWORKS FOR DATA LOCALITY ON A MEMORY BUDGET

BY

DAVID J. STEIN

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

Adviser:

Assistant Professor Yi Lu

ABSTRACT

Typical queries on online social network (OSN) applications are complex and require “feeds” to be compiled with timely information about many friends and friends’ friends, which may be stored across many servers. Partitioning the OSN social graph in such a way as to promote data locality, i.e. such that a user’s data will be stored on the same server as his friends’ data, has proven difficult to do, and many existing OSN partitioning systems do not even attempt this. However, recent work has demonstrated techniques that do achieve data locality for social network queries by placing replicas of user data. We show that exploiting temporal characteristics of user behavior can enable effective partitioning for data locality without replication. We then build on this concept and demonstrate improved data locality by placing replicas sparingly. The result is a system which allows one to allocate a memory budget for replication and in return get a commensurate improvement in data locality.

ACKNOWLEDGMENTS

I would like to thank my adviser, Professor Yi Lu, for her help and guidance throughout the course of my graduate study. I would also like to recognize my colleagues Mindi Yuan, Berenice Carrasco, and Joana M. F. Trindade for the very important role they played in this research.

Special thanks to the Trusted ILLIAC Center at the Information Trust Institute and Coordinated Science Laboratory for their generous contribution of use of their computing cluster to help complete this work.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The importance of online social networks has increased dramatically in recent years, introducing a host of new challenges. Key differences between social networks and traditional web services motivate the design of innovative architectures to support them. These innovative designs aspire to allow social networking services to scale more readily, to provide faster responses to queries, and to reduce the load that queries impose on the distributed systems that run them.

1.1 The Problem

Typical queries in social networking applications are demanding. A single page load may require that a user be provided with a "feed" compiled of recent information about that user's friends. In some social networking applications these feeds further allow users to see relevant updates and messages between their friends and their friends' friends. The responses to these queries are typically very sensitive to time and expire quickly; when a user checks her feed in the afternoon she may expect to see an entirely different feed than was shown to her that morning. Furthermore, users' sets of friends are largely unique. These characteristics of social networking applications conspire to make common queries not only expensive to process but largely non-repeatable, making them poorly suited for traditional performance-enhancing techniques such as caching. The fact that users expect their page requests to be handled nearly instantaneously only complicates the issue further.

The characteristics of the social graph further differentiate social networking applications from traditional web services. For example, in an online shopping service, a user may require to see his order history or shopping cart contents; this is information which is specific to that user and can easily be stored and retrieved accordingly. By contrast, a user of an online social network requires to see information about his entire network, which may include hundreds of friends. Because social networks are often very large (hundreds of millions of nodes), the user's friends and friends' friends are likely to be stored on many servers across the network. It is readily apparent that this would motivate a good partitioning of the social graph in which users and their friends are stored on the same server, but the highly interconnected structure of social graphs makes this difficult to achieve.

1.2 Previous Work

Common practice in industry is to employ horizontal partitioning, which provides fast access to information regarding a particular user but does not attempt to store connected users together to promote data locality. Twitter's Gizzard architecture makes use of range partitioning [1]. Facebook developed Cassandra, which partitions user data via consistent hashing of user IDs [2]. Both of techniques require fetching data from a large number of servers to compile feeds for a given user.

Consequently, we are motivated to seek better partitioning techniques which are able to achieve high locality of relevant user data. Such advanced algorithms would group users together in a partition by leveraging the structure of the social graph and of users' query patterns over time. Improved user data locality has the promise of faster response times [3], a decreased load required on the internal network (as a user's friends' data could be retrieved by polling fewer partitions), and improved scalability. [4] But effective partitioning to promote data locality is hard to do.

Nonetheless, recent work has shown promising results for advanced partitioning techniques. In [5], strict local semantics are guaranteed by ensuring that either a replica or the master copy of each of a user's friends is stored in the same partition as that user. As the social graph evolves, their algorithm adaptively makes partitioning decisions which result in the fewest required replicas. While their system provides substantially reduced network traffic and improved response times over traditional systems, the replication overhead it requires can be very large [3].

1.3 New Methods

While the method in [5] provides data locality by guaranteeing local semantics for every query, we examine loosening this strict requirement. A good partitioning of the social graph could ensure that relevant data is local for *most* queries. This way, resources would not be wasted by keeping excessive replicas of infrequently accessed data.

By observing trends in user behavior, it is possible to construct a model of an activity graph [3]. We will show how such an activity graph can be used to effectively partition the social network data in prediction of future query patterns. The resulting method will provide good data locality not for all possible query patterns but specifically for those which are considered most likely to occur in the near future, minimizing the interactions between partitions required to serve common queries. In contrast to [5], this is achieved without any replication at all.

Expanding on this idea further, we subsequently examine the potential for additional benefit by bringing replication back into the picture. By partitioning the social graph in general to maximize data locality for likely queries, while selectively adding replicas in specific situations in which it is advantageous to do so, we will demonstrate a superior algorithm which balances both replication and inter-partition interaction to find beneficial partition configurations for the social network.

Such a hybrid system has advantages over both approaches previously introduced. While the large factor of replication required by [5] may make it too expensive for use in some systems, an improved system would allow an administrator to allocate how much replication could be afforded, and in return get commensurate data locality while tolerating some degree of interaction to serve queries. Meanwhile, a system which merely minimizes interactions via strategic partitioning may be wasting extra memory which could otherwise be used for replicas to greatly improve data locality. The resulting system we will propose employs both interaction and replication to provide high data locality for common social network queries. It can also be tailored to the resources of a specific deployment.

The rest of this thesis proceeds as follows. The next chapter contains a review of relevant literature on social networks and partitioning. The subsequent chapter describes a method of partitioning a social network to provide high data locality for queries without replication by exploiting time-dependent characteristics of user behavior. The next chapter introduces an improved algorithm which provides enhanced data locality by leveraging replication as well as strategic partitioning. The final chapter draws conclusions from our research and provides recommendations for future work.

CHAPTER 2

LITERATURE REVIEW

There has been significant work in research literature on modeling online social network (OSN) graphs and finding effective strategies to partition and store them. The problem was discussed in [6], which mentioned many existing opportunities for optimization in current “feed-following” systems including OSNs. They surmise that the question of how to best balance the costs and benefits of clustering or partitioning in feed-following systems remains wide-open, and they suggest a clever combination of replication and clustering algorithms would be required to adequately handle the challenges presented by these kinds of systems.

Facebook’s Cassandra [2] and Amazon’s Dynamo [7] are distributed storage systems which rely on distributed hashing to partition data across many servers. These systems are intended to be fault-tolerant and scalable. When used for OSNs, distributed hash partitioning can lead to poor performance due to lack of data locality. They also can create other problems such as Facebook’s “multi-get hole” [4] which can cause performance and scalability issues depending on which resources are constrained.

The “One Hop Replication” system in [8] tackles the problem of scalability by leveraging the community structure in OSNs and the fact that most of the information accessed is just “one-hop” away with regards to the friendship graph. Instead of replicating all the users (which would not be possible for a real system with a very large number of users), only the inter-partition activity links are replicated. Also the “bridge” users—users who have weak ties among them—are also replicated.

In SPAR [5], this idea is implemented in a form of middleware that transparently provides local semantics for social network application development, giving the appearance of a fully-replicated social network on each server in the distributed system. This is accomplished by guaranteeing that the server containing each user also contains a replica (if not the master copy) for each of that user's friends. They propose an algorithm which leverages social graph structure to choose a partition configuration which requires a minimum number of replicas to be maintained throughout the network. The local semantics, as well as promoting scalability, can also serve to improve performance via reduced network I/O requirements. However, it has been noted [3] that the factor of replication required by SPAR can be quite large, which could make it expensive to use in a production environment. It also does not exploit temporal characteristics of user behavior to make better partitioning decisions as other methods do [3].

In contrast, [3] demonstrates usage of temporal characteristics of user behavior to partition the graph in the time domain, so that users' information is grouped together for the time periods during which they are relevant to one another. This is done by constructing an activity prediction graph (APG) weighted to represent which edges between users are most important at the current point in time. This APG is easier to partition than the social graph itself, having a lighter tail on its power-law degree distribution. In general, it is known to be difficult to partition graphs with a power-law degree distribution in a balanced way [9]. Further discussion of this technique is provided in the next chapter.

Schism [10] partitions a distributed database based on workload and query patterns. Not specific to use with social networks, it works well where queries are static and repeated many times. However, it will not be able to predict future queries in social networks, in which both data and the network are changing over time.

TAMER [11] is another system that supports partitioning of large-scale OSN data for the purposes of broad distribution across geographic locations. Current OSNs replicate all user data in each geo-distributed data center, leading to redundancy and expensive synchronization. This is ameliorated by defining a threshold latency for information exchange such that only the minimum number of replicas satisfying this threshold across all the geo-distributed datacenters are maintained while others are discarded.

Partitioning, replication, and data locality in distributed database systems are not a topic unique to their application for OSNs. Distributed processing systems such as MapReduce and Dryad motivate advanced scheduling algorithms which promote data locality for distributed tasks. Quincy [12] schedules distributed jobs based on data locality and fairness constraints. Another system detailed in [13] employs “delay scheduling” to improve data locality while preserving fairness.

CHAPTER 3

DYNAMIC PARTITIONING

Most online social networks use distributed hashing [2, 7] to partition users' data across a distributed database. With this kind of partitioning, a user's friends' data will be stored in many different partitions. But characteristics of user behavior—how frequently they post messages and query their feeds—can be used to predict which users' information will be requested together to serve common queries. We can leverage this information to build a better partition configuration, ensuring that most queries will only require information from a few partitions.

To this end, we will construct an activity prediction graph (APG) which does not necessarily contain all of the "friendship" connections between users present in the full social graph, but instead contains only those connections which are "active"—that is, in which messages have been exchanged between the users recently. This APG will have a simpler topography than the full social graph, and its power-law degree distribution will have a much lighter tail [14]. This will make it easier to partition effectively.

Because characteristics of user behavior change over time, we will need to partition the social network data not only in the spatial domain of the nodes on the graph but also along the *time* dimension. That is, all messages between two users will not necessarily be stored on the same server, though they will be divided according to their time stamps and placed within a particular time range on a particular server.

When partitioning the APG, we will consider the *two-hop neighborhood* surrounding users and messages. This will supply greater data locality for applications which require consideration of the entire two-hop network,

Content in this chapter previously appeared in "Partitioning Social Networks for Fast Retrieval of Time-dependent Queries" by Mindi Yuan, David Stein, Berenice Carrasco, Joana M. F. Trindade, and Yi Lu [3]. © 2012 IEEE. Reprinted with permission.

such as Facebook's "news-feed" application in which users see not only messages between themselves and their friends but also between their friends and their friends' friends. For the algorithm we will describe, we also will assume that the friendship relationship is necessarily symmetrical, which is to say that if user a is a friend of user b, then user b must be a friend of user a.

3.1 Graph Model

Consider a graph G on the vertex sets U and V where U is the set of users in the social network and V is the set of messages between a pair of users. The message vertex $v \in V$ always has degree 2 and connects to the two user vertices who are interacting. We index a node v by the unordered pair of the indexes of the user vertices to which v is connected. Figure 3.1 shows an example of a graph with 6 user vertices and 5 message vertices.

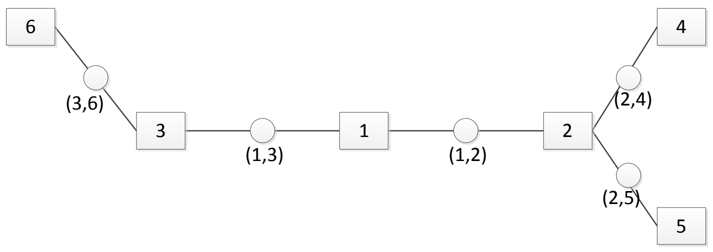

Figure 3.1: Graph with 6 user vertices and 5 message vertices.

We define the neighborhood set of a user vertex and a message vertex. Let $\mathcal{N}_u$ denote the neighborhood of user node u, where

$$\begin{aligned}\mathcal{N}_u = \;& \{u', (u,u') \text{ if } u < u', (u',u) \text{ if } u > u' \\ & : (u,u') \in V \text{ or } (u',u) \in V\}\end{aligned}$$

Let $\mathcal{N}_{u,u'}$ denote the neighborhood of message node (u, u'), where

$$\mathcal{N}_{u,u'} = \{u, u'\}$$

The neighborhood of a user node includes all the user nodes sharing a common message node and their common message nodes. The neighborhood of a message node includes the two user nodes connected to it. We now define the two-hop neighborhood of a user node and a message node.

Let $\mathcal{H}_u$ denote the two-hop neighborhood of user node u, where

$$\mathcal{H}_u = \bigcup_{u' \in \mathcal{N}_u} \mathcal{N}_{u'}$$

Let $\mathcal{H}_{u,u'}$ denote the two-hop neighborhood of message node (u, u'), where

$$\mathcal{H}_{u,u'} = \mathcal{N}_u \bigcup \mathcal{N}_{u'}$$

The two-hop neighborhood of user i includes vertices whose content is visible to user i. A user can view the messages between him and his friends and all messages initiated or received by his friends. The messages initiated or received directly by user i are in the one-hop neighborhood centered at user i, while the messages initiated or received by all friends of user i reside in the two-hop neighborhood centered at user i. For example, the set of user vertices $\{2, 3, 4, 5, 6\}$ and the edges connecting them constitute the two-hop neighborhood centered at user 1. The two-hop neighborhood centered at a message vertex is the union of the one-hop neighborhood of its initiators and receivers. For example, the set of user vertices $\{1, 2, 3, 4, 5\}$ and the edges connecting them constitutes the two-hop neighborhood centered at message vertex $(1, 2)$.

Consider the following retrieval scenario. Each user is assigned an access frequency m_i and retrieves the data in the two-hop neighborhood. Each message node stores messages and is assigned a weight $w_{i,j}$. Potential values for $w_{i,j}$ are the number of messages at the node, or a weighted sum of the messages. The objective is to define weights $e^i_{i,j}$ and $e^j_{i,j}$, for the edges connecting user nodes i and j to the message node (i, j), so that minimizing the cross-partition edges in the graph will correspond to maximizing locality for accesses.

Let D_i denote the sum of all message node weights in the two-hop neighborhood of user i,

$$D_i = \sum_{(i,j) \in \mathcal{H}_i} w_{i,j}$$

Let $W^k_{(i,j),i}$ denote the message node weights in a remote partition if the edge between user vertex i and edge vertex (i,j) is cut. Let $G_{\sim(i,j),i}$ denote the graph with the edge between (i,j) and i removed, and let $\mathcal{N}^{\sim(i,j),i}$ and $\mathcal{H}^{\sim(i,j),i}$ be the neighborhoods in $G_{\sim(i,j),i}$, then

$$\begin{aligned} W^k_{(i,j),i} &= (\mathcal{H}_k - \mathcal{H}_k^{\sim(i,j),i}) \bigcup (i,j) \ \text{ if } \ i \in \mathcal{N}_k \\ &= (\mathcal{H}_k - \mathcal{H}_k^{\sim(i,j),i}) \bigcup i \ \text{ if } \ (i,j) \in \mathcal{N}_k \end{aligned}$$

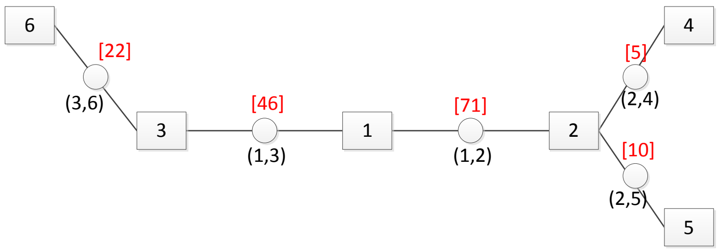

Figure 3.2: Graph with 6 user vertices and 5 message vertices, with weights on each message vertex.

For example, the total weight of messages accessible to user 1 in Figure 3.2 is

$$\begin{aligned} D_1 &= w_{3,6} + w_{1,3} + w_{1,2} + w_{2,4} + w_{2,5} \\ &= 22 + 46 + 71 + 5 + 10 = 154 \end{aligned}$$

and the total message weight in a remote partition for node 1 if the edge between $(1,2)$ and 1 is cut is

$$\begin{aligned} W^1_{(1,2),1} &= w_{1,2} + w_{2,4} + w_{2,5} \\ &= 71 + 5 + 10 = 86 \end{aligned}$$

The total message weight in a remote partition for node 3 if the edge between $(1,2)$ and 1 is cut is

$$W^3_{(1,2),1} = w_{1,2} = 71$$

We now define the edge weights. Let $e^i_{i,j}$ denote the weight on edge between nodes i and (i,j), then

$$e^i_{i,j} = \sum_{k \in \mathcal{H}_{i,j}} m_k \frac{W^k_{(i,j),i}}{D_k}$$

which is the sum of the fraction of remote message weights weighted by user frequency.

Consider accesses at each user node as an independent Poisson process with intensity m_i, and interaction between user i and j an independent Poisson process with intensity $w_{i,j}$.

If only considering the one-hop network, the $w_{i,j}$ can be directly used as the edge weight, i.e. $e^i_{i,j} = e^j_{i,j} = w_{i,j}$. However, this only captures the influence to the one-hop network of cutting an edge.

3.2 Periodic Algorithm

The periodic algorithm computes a partition configuration every month using the APG which has been updated to reflect the frequencies of recent interactions. We define the interaction node weights $w_{i,j}$ to be the discounted message frequency

$$w_{i,j} = C \sum_{k=1}^{K} f_k \; n^k_{i,j}$$

where K is the total number of past periods considered, C is a scaling constant, $n^k_{i,j}$ is the number of messages exchanged between users i and j in month k. We define f_k as the decay factor computed on a monthly basis for month k

$$f_k = \frac{|L_k \cap L|}{|L_k|}$$

in which L_k is the set of links in month k and L is the set of links in the current month. The cardinality of a set is denoted by $|\cdot|$.

The weight of each message vertex is used for computing balanced partitions as it is a prediction of the number of messages to expect at each link based on past queries. We use KMETIS, a software program from the METIS library [15], to partition the APG. KMETIS uses a multilevel k-way min-cut algorithm to produce partitions that balance vertex weights in each partition and minimize edge weights across partitions.

For a message whose corresponding node is present in the APG, it is stored in that message node's partition. For a message whose node is not present and is therefore not predicted by the APG, we use the following simple algorithm:

1. If both the initiator and receiver of the message exist in the APG, but no previous message exists, store the message with the user with a smaller value of D, as the new message will contribute a larger fraction of this user's future query.

2. If exactly one of the initiator and receiver of the message exists in the APG, store the new message in the same partition as that user.

3. If neither the initiator nor the receiver exists in the APG, store the new message in the partition with the least number of messages.

The values D_i are updated for each user i as new messages are stored in each partition.

3.3 Adaptive Local Algorithm

The periodic algorithm has two drawbacks: (1) It changes the placement of a large number of message nodes at the end of a period, creating artificial remote accesses for subsequent retrievals as two messages on the same message node can reside in different partitions; (2) It fails to take advantage of the strong time correlation of messages as no repartitioning takes place within the period. This motivates the design of a local adaptive algorithm.

We propose a local algorithm that is triggered when a retrieval results in remote accesses. We define the boundary pairs

$$B \;=\; \{\, (i,(i,j)) : i \text{ and } (i,j) \text{ are in different partitions} \,\}$$

Only the subset of boundary pairs for which the weights in the two-hop network have changed since the last repartitioning will be considered. Changes in message weights outside the two-hop network can have an effect on the boundary pair, but are ignored to reduce complexity as the effect is usually small. We recompute the edge weights in the APG updated with current messages. For each pair in the boundary set, we consider the following reward function:

$$Z \;=\; -\Delta E \;-\; M$$

where ΔE is defined as the change in cross-boundary cost in the APG if the node in consideration is moved. The threshold parameter M represents a base cost of movement for one node. Movement occurs only when $Z > 0$. For a message node (i, k) currently in the same partition as i,

$$\Delta E = e^{i}_{i,k} \;-\; e^{k}_{i,k}$$

is used to decide whether to move it to the same partition as k. For a user node i, which can connect to multiple message nodes in different partitions, we define ΔE as

$$\Delta E = \max_{P \in \{\text{adjacent partitions}\}} \sum_{(i,k) \in P} e^{i}_{i,k} \;-\; \sum_{(i,j)\ \text{local}} e^{i}_{i,j}$$

Thus we represent the reward for moving the user node into the best choice among its adjacent partitions.

3.4 Evaluation

The dataset for our evaluation was produced from an event trace from a subgraph of the Facebook social graph in New Orleans between 2005 and 2006. We train the APG with the events from January 2005 through November 2006, and we test the algorithms with messages in the month of December 2006. Each user is assigned an access process that retrieves the most recent 6 messages in the two-hop neighborhood. We choose the number of messages to be 6 as our data set is relatively small: the data in December 2006 contains a total of 13948 messages with 8640 active users. We evaluate the performance of both the periodic and local algorithms.

3.4.1 Periodic Algorithm

We compare our algorithm to two hash-based horizontal partitioning algorithms. These are the algorithms used in commercial online social networks. The first algorithm, "hash_p1," hashes the initiator ID of a message. As a result, all messages generated by the same user are grouped in one partition. The second algorithm, "hash_p1p2," hashes the unordered sender-receiver pair of a message. All messages exchanged between a particular pair of users are grouped in one partition. We compare the experiments with different numbers of partitions up to 20. We did not experiment with a larger number of partitions as there are only 8640 active users for December 2006, and we are considering the locality of messages in a two-hop neighborhood. We also show the results from a retrospective algorithm, denoted by "retro", where the actual messages in December 2006 were used to train the APG. This is the optimal result for a static partitioning algorithm. We use $C = 12$ and $K = 23$ to construct the APG, where C is a scaling constant and K is the total number of past periods considered. We experimented with other values of C and K and the result is not sensitive to the change of the values. We associate a frequency m_i to each user i. For this experiment, we let $m_i = \sum_{(i,j)\in G} w_{i,j}$, which is the sum of all weights on message nodes connected to user node i. This assumes that the frequency of reads are proportional to the number of messages sent or received by a user. We did not consider the balance of accesses across partitions in this experiment, but it can be readily integrated by assigning weights to user vertices proportional to its frequency.

Figure 3.3 compares the proportion of queries that have all 6 most recent messages in a single partition for the three algorithms. With 5 partitions, the periodic algorithm produces 50% of all queries with all 6 messages in one partition, whereas both hashing algorithms have less than 10% of local queries. With 20 partitions, the periodic algorithm achieves 34% of local queries as some two-hop neighborhoods need to be cut to keep the balance of the data storage, which is still over 12 times better than the hashing algorithms, each achieving 2.8% and 2.6% of local queries. In all cases, the performance of the periodic algorithm is within 80% of the retrospective algorithm, showing a good prediction quality of the APG.

Figure 3.4 compares the proportion of queries that have all 6 most recent

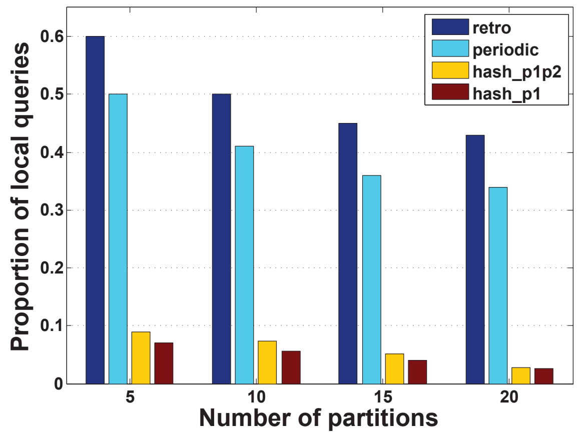

Figure 3.3: Proportion of queries that access only 1 partition. Comparison of the periodic algorithm with hashing the initiator ID (hash_p1) and the unordered initiator-receiver pair (hash_p1p2).

messages in at most 3 partitions. For all numbers of partitions, more than 90% of queries access at most 3 partitions with the periodic algorithm. For the hashing algorithms, while 71% of all queries access at most 3 partitions when there are a total of 5 partitions, the fraction decreases to less than 40% when there are a total of 20 partitions. The performance of the periodic algorithm is within more than 95% of the retrospective algorithm.

3.4.2 Local Algorithm

Figures 3.5 and 3.6 show the performance of the local algorithm with $M = 10$. Recall that M is the constant cost for moving one node. With 5 partitions, the local algorithm results in 20% more local queries than the periodic algorithm and almost 6 times more than the hash algorithms. With 20 partitions, the local algorithm achieves 30% better than the periodic algorithm and 13 times better than the hash. Both the local and periodic algorithms have more than 90% queries accessing at most 3 partitions, with the local algorithm performing slightly better. The total number of movements for the local algorithm with 5 partitions is 1122, evenly distributed across time. This amounts to 40 movements daily, which is small. The number of movements increases with the number of partitions, reaching 1859 at 20 partitions.

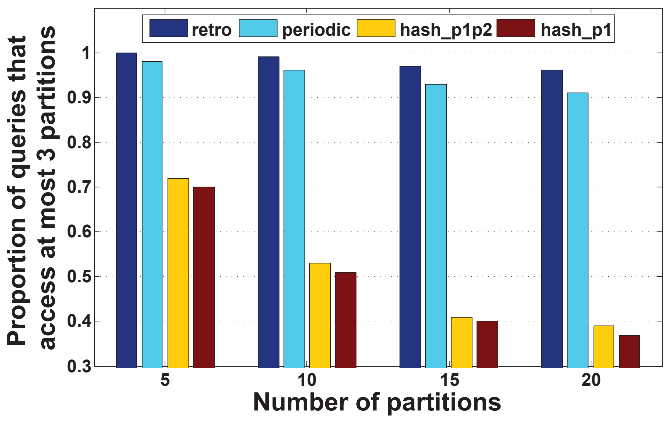

Figure 3.4: Proportion of queries that access at most 3 partitions. Comparison of the periodic algorithm with hashing the initiator ID (hash_p1) and the unordered initiator-receiver pair (hash_p1p2).

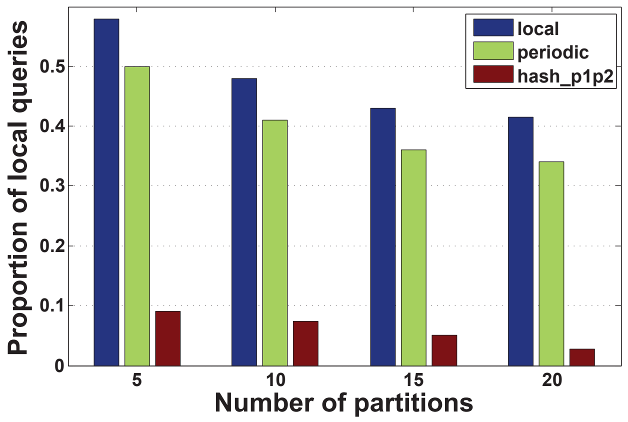

Figure 3.5: Locality tests for the period, adaptive local, and hash_p1p2 algorithms. The vertical axis shows the percent of queries which require data from only a single partition.

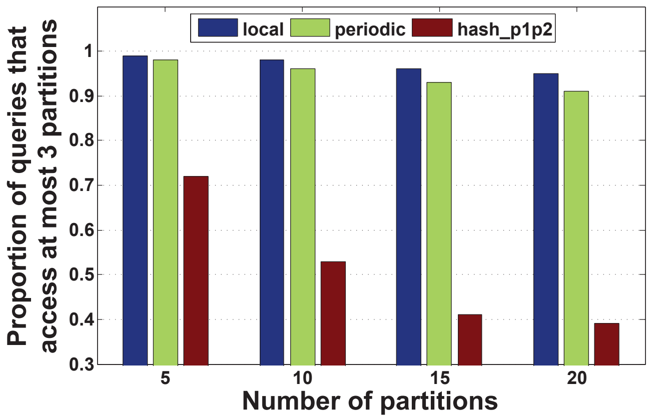

Figure 3.6: Locality tests for the period, adaptive local, and hash_p1p2 algorithms. The vertical axis shows the percent of queries which require data from at most 3 partitions.

CHAPTER 4

DYNAMIC PARTITIONING WITH REPLICATION

A significant drawback of the partitioning algorithm in the previous chapter is that it cannot place replicas of important data in multiple partitions, regardless of how beneficial such a configuration would be. Figure 4.1 shows an example social network in which the ability to replicate would be advantageous.

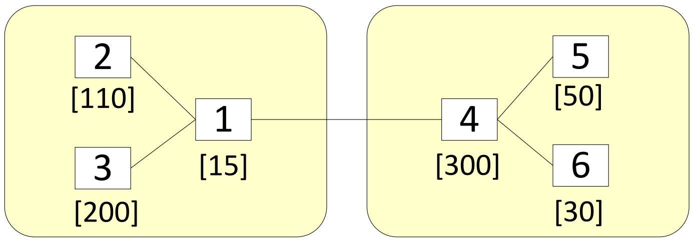

Figure 4.1: A partitioned social graph with 6 users. The numbers in brackets indicate how often that user will query for his feed each month.

We notice that user 1's friends query for their feeds frequently. Queries which require user 1's information come from user 1's partition 310 times per month, while queries from his friend user 4's partition which require user 1's information come 300 times per month. Regardless of which partition user 1 were to be placed in, a large amount of inter-partition interaction will occur to serve his friends' queries. Moving user 4 into user 1's partition also will not help, as user 4 also frequently demands information from his friends users 5 and 6, from whom he would be cut off if he moved. For the purposes of maintaining evenness among partitions, we do not allow the trivial configuration which would place all users in the same partition. We can see that in this example no way of partitioning the graph is very good.

However, if we replicate user 1's information in user 4's partition, we can dramatically reduce the number of queries which require information from a remote partition. This configuration is shown in Figure 4.2. If the extra memory required to store the replica of user 1 can be afforded, the number of inter-partition requests needed to serve queries in one month can be reduced from 315 to only 15. (User 1 will still need to access user 4's information during the few times when he will query for his feed.)

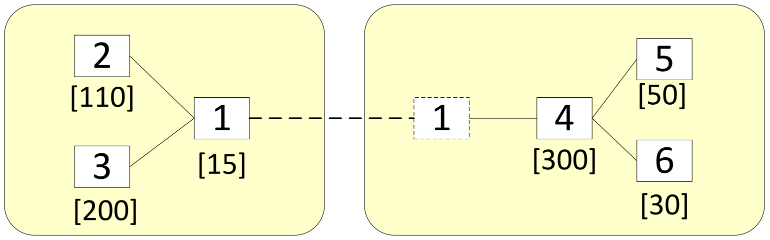

Figure 4.2: A replica of user 1 has been placed in the other partition. The numbers in brackets indicate how often a user will query for his feed each month.

This realization motivates us to describe an algorithm which not only partitions the social graph to minimize requests across partition boundaries but also strategically places replicas where they will be most helpful. Such an algorithm can be constrained by the amount of memory allocated for storing replicas—what we will call the replication budget.

The replication budget can be parameterized by a target replication factor r_{target} and a cap replication factor r_{cap}—the latter a threshold which the algorithm will not cross. The *replication factor* denotes the quantity of replicas in place in the social network. If we make 50 replicas on a graph containing 100 master nodes, we say the replication factor is 1.5.

In this chapter we present an algorithm which uses both replication and strategic partitioning to promote data locality for common queries. We simplify our discussion of this algorithm by addressing the one-hop network only. We also will not consider message nodes as we did in the previous chapter, and will simply assume that all messages are stored with the recipient at his user node. Finally, unlike the system discussed in the previous chapter, this algorithm will support asymmetric friendship relationships in the social graph.

4.1 Graph Model

Let a social network be represented as a directed graph G on vertex set V. Let each node in V represent one user on the social network. Let each directed edge $(i, j) \in G$ represent the relationship "user i follows user j," which is to say that user j's recent information will appear in user i's feed.

We further define for each user node i both an access frequency m_i and a message weight w_i. The access frequency m_i of a given user node denotes how often she will query for her feed in the next period of time. The message weight w_i of a given user node denotes how many messages will be posted at that user node in the next period of time. Both m_i and w_i can be considered the intensities of independent Poisson processes for the arrival of query and message events, respectively, for user i.

We define a set of partitions $P = \{p_1, p_2, \ldots, p_k\}$ and a mapping $p : V \to P$ indicating in which partition a node resides. Every node in V is in exactly one partition. Each partition also may contain replicas, and we define the function $R : V \times P \to \{0, 1\}$ such that $R(i, p_a) = 1$ if and only if a replica of user node i is in p_a. We further assert that if $p(j) = p_b$, then it also must hold that $R(j, p_b) = 0$, meaning that a partition cannot hold both a node and its replica.

By nature of its access frequency, we say that a user node exerts *demand* on all nodes whom she follows. The weight of this demand also depends on the message weight of the node being demanded. If node i follows node j, then we define the *demand* from i to j as follows:

$$D(i, j) = m(i)w(j)$$

4.2 Algorithm

The partitioning algorithm will seek to minimize the following objective function:

$$Z = \text{interaction} + \text{evenness} + \text{replication}$$

in which

$$\text{interaction} = \sum_{i \in V} \sum_{\substack{(i,j) \in E \\ p(i) \neq p(j) \\ R(j,p(i))=0}} D(i,j) \tag{4.1}$$

and in which the evenness index is a cost function which penalizes the increasing variance among the partitions of message weights, access frequencies, and/or replicas and their weights; and in which the replication index is a cost function which penalizes an increasing count of replicas beyond a target replication rate r_{target}. Optimal formulas and coefficients for the evenness and replication indexes are expected to depend on application-specific resource characteristics and constraints, such as internal bandwidth and the cost of memory.

The cost of a replica is understood to be derived from the fact that there is scarcity of space for them in memory, not from any cost required to keep them up to date. Social networking applications are well-suited for a loose consistency model; we do not require feed information regarding all friends to be accessible instantly. Granted this, we assume that the maintenance of replicas can be put on a low priority, and we therefore consider it "free."

We describe an iterative greedy algorithm which searches for a more optimal configuration of partitions and replicas. For simplicity in our demonstration, w_i was set to 1 for all i.

We define two functions, *pull* and *push*, which will be useful in implementing our greedy algorithm.

$$\text{pull}(i, p_a) = \sum_{\substack{(j,i) \in E \\ p(j)=p_a}} D(j,i)$$

$$\text{push}(i, p_a) = \sum_{\substack{(i,j) \in E \\ p(j)=p_a}} D(i,j)$$

Pull describes the aggregate demand exerted onto a node by all of her followers in a given partition. *Push* describes the aggregate demand exerted by a node onto all nodes she follows in a given partition.

In each iteration, potential rewards are evaluated from within two categories:

1. **move** v **from** $p(v)$ **to** p_b

for which the reward Z is defined

$$\begin{aligned} Z = &\ \text{push}(i, p_b) + \text{pull}(i, p_b) + \text{push}(i, p(i)) \\ &+ \text{pull}(i, p(i)) + C_R R(i, p_b) - C_E(p_b) + C_E(p(i)) - T \end{aligned}$$

where C_R is the cost (here treated as reward) for freeing v's replica in partition p_b, and $C_E(p_b)$ is the "evenness" cost—the cost (or reward) of moving a node into partition p_b based on that partition's size relative to the sizes of the other partitions. T is a threshold set to prevent trivial moves. For more details about the replication and evenness cost terms, see Appendix A.

2. **create a replica of v in p_b**
   for which the reward Z is defined

$$Z = \text{pull}(i, p_b) - \min(C_R, C_R^*) - T_R$$

where C_R^* is the *lowest cost replica* in partition p_b. T_R is a threshold set to prevent trivial replicas from being created.

For each node in the graph, the rewards are computed for each option described above and for each partition. The results are sorted so that the most advantageous move or replication and destination partition are chosen for each node. The changes are then committed to the graph in descending order of reward. Nodes whose best potential action has negative reward are left untouched.

When a replica is created, the reward value associated with it is saved. After reward values are computed for all replication options, these data are used to prune out replicas which are no longer worthwhile by the end of the iteration. If a replica's new reward value is negative, the replica will be deleted.

4.3 Evaluation

Our algorithm was tested using four datasets derived from an event trace of Facebook data from the New Orleans network between 2005 and 2006. In this chapter we discuss the metrics and methodology used.

4.3.1 Metrics

At the start of each test and at the end of each iteration, we stored the interaction index, replication index, and evenness index of the graph.

The interaction index, defined in Equation 4.1, denotes the average number of *additional* partitions (other than a user's home partition) which need to be accessed in order to serve a request. For the purposes of evaluating this algorithm, it is assumed that all friends whom a user is following will be mentioned in that user's feed for all feed queries. (This assumption could be removed by using non-uniform weights w_i.)

The replication index for evaluation of this experiment was defined as follows:

$$\text{replication} = \frac{1}{|V||P|} \sum_{i \in V} \sum_{p_a \in P} R(i, p_a)$$

where $|\cdot|$ denotes the cardinality of a set. The replication index can be intuitively understood as the factor by which the number of nodes is multiplied to obtain the number of nodes and replicas.

The evenness index for evaluation is defined as:

$$\text{evenness} = \frac{\sum_{p_a} \left| \sum_{\substack{i \in V \\ p(i) = p_a}} w_i - \frac{\sum_{j \in V} w_j}{|P|} \right|}{|V|}$$

The evenness index measures the spread in sum weights among the partitions. It is intended to be kept low.

4.3.2 Methodology

The algorithm was implemented as a simulation on a single machine. Four test sets were run numerous times under different parameters to achieve different target replication factors. Each test used a given social graph as its initial state and ran 20 iterations of the algorithm. After each iteration, the interaction, replication, and evenness indexes were stored so that they could be plotted versus the iteration number.

The datasets were produced from an event trace from a subgraph of the Facebook social graph in New Orleans between 2005 and 2006; it is the same dataset which was used in [3]. All tests feature a graph with the same topology of 7945 nodes and 18163 undirected edges. Tests were run starting from both random and structured partitionings and having both 15 and 20 partitions. The structured partition configurations were generated using KMETIS, a program in the METIS library [15]. KMETIS produced balanced partitions which minimized edge-cut in the social graph, where we defined edge weights $e_{(i,j)}$ to be

$$e_{(i,j)} = D(i,j) + D(j,i)$$

This experiment tested the algorithm's ability to find better partition configurations from a given initial state, both by placing replicas and by rearranging nodes among partitions to minimize interaction costs. The experiment did not specifically explore the algorithm's effect along the temporal dimension, but it could be adapted easily to a scenario as described in [3]. At each time period, several iterations of our algorithm could be run. Changes in graph topology could be handled as in [5] and [3]. Changes in user activity (such as changes in access frequencies) could be handled as in [3] where a decay rate causes old data to have an increasingly diminished effect on access frequencies m_i and message weights w_i.

4.3.3 Results

Figure 4.3 shows the effect after 20 iterations when the algorithm was run on the test graph with 15 partitions. The test graph had an initial partition configuration generated using the method described in [3]. The greedy algorithm quickly adds replicas, increasing the replication factor but decreasing the interaction index. The target replication factor was set to 1.5, demonstrating a reduction in the interaction index from 1.78 to 0.32. This means that queries would now require communication with an average of 1.32 partition servers (including the user's home partition server), down from an average of 3.78 before—a 65% reduction—in return for a 52% increase in memory subscribed for storing replicas.

Figure 4.4 shows the same experiment run on the same graph, except with a different, randomly-generated initial partition configuration. While the state after 20 iterations shows considerable improvements from the initial state (0.96 interaction index down from 5.97—an 83% reduction in

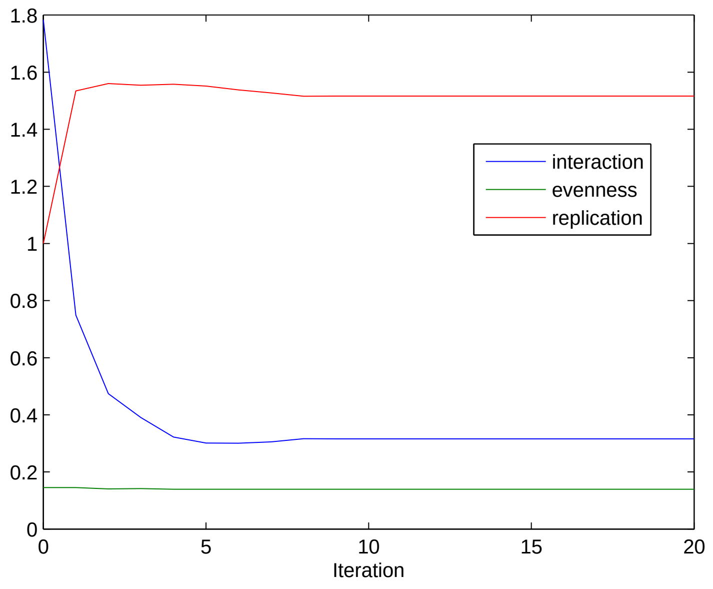

Figure 4.3: Simulation with 15-partition graph for 20 iterations.

average number of servers to serve a request—for only a 51% increase in replication), it is not as good as the result from the previous experiment which began with a better-partitioned graph.

These results show that, though the algorithm yields a benefit, namely leveraging additional space for replicas to reduce the interaction index and improve data locality for queries, it shows itself to converge towards local optima when far better configurations may be possible.

Figure 4.5 displays the trade-off between interaction and replication seen in the experiments. While the simulation confirms the intuition that we can build a system which delivers an improvement in data locality commensurate with the amount of memory which can be afforded for replication, our algorithm requires a good initial partitioning to be able to produce its best results.

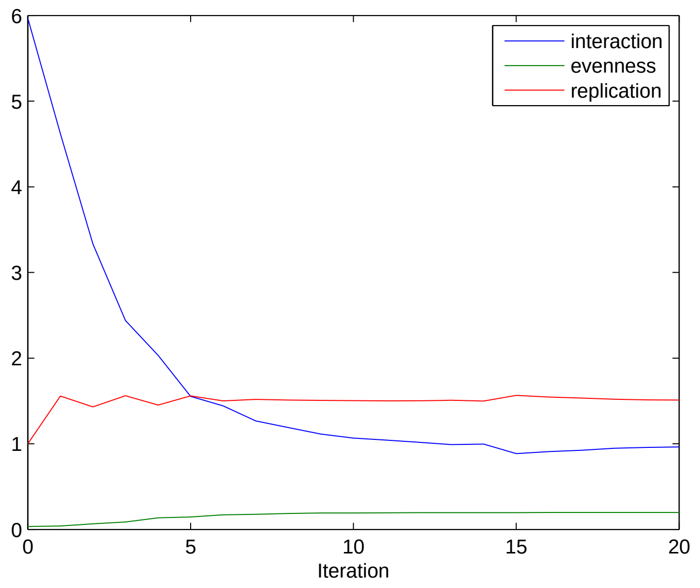

Figure 4.4: Simulation with randomized 15-partition graph.

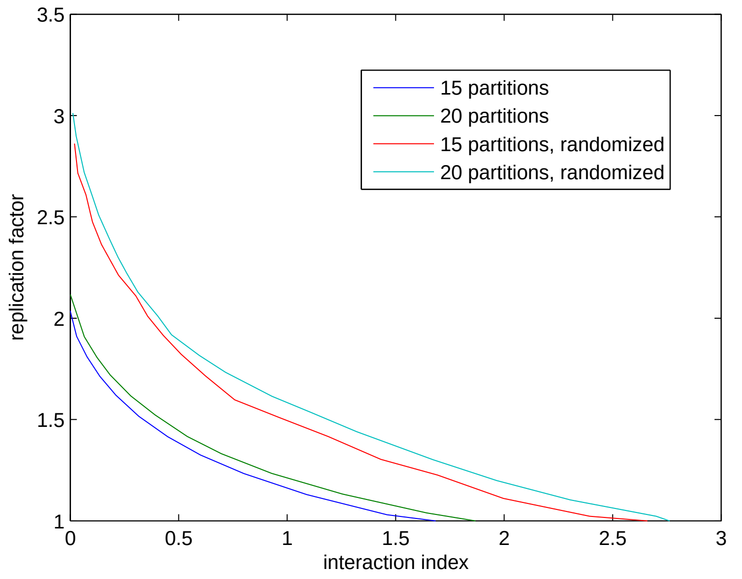

Figure 4.5: Experimental interaction-replication trade-off relationships for both 15- and 20-partition graphs, after 20 iterations.

CHAPTER 5

DISCUSSION AND CONCLUSIONS

The results of our experiments demonstrate that our algorithms succeed in partitioning social networks for data locality. Strategic partitioning using an activity prediction graph populated with temporal characteristics of user behavior dramatically reduces the number of servers required to serve a typical query compared to traditional hash-based techniques. Adding the possibility for replication further reduces inter-partition interaction.

Unlike [5] which guarantees strict local semantics via complete local replication (which can be very expensive), our algorithm is a demonstration that there can be a middle ground between replication-only and interaction-only paradigms.

While [3] stressed the importance of partitioning for data locality for the two-hop neighborhood, we stayed with a one-hop model in our replication algorithm as was done in [5]. We believe our concepts of *demand*, *push*, and *pull* could be generalized to allow for consideration for the user's extended neighborhood more than one hop away.

5.1 Future Work

Formal analysis of the proposed algorithms is beyond the scope of this thesis. Questions of whether and under what conditions the greedy algorithms converge towards optima are left to future work.

We expect that the minimum factor of replication required to obtain acceptable locality would depend on characteristics of the graph's structure, such as the clustering coefficient. Whether and how graph structure predicts the required amount of replication is left to future work.

The replication algorithm's requirement to iterate over the entire social graph could make it poorly suited for real-world deployments where social graphs are very large. It is conceivable that a decentralized version of this system could be designed in which the partitions communicate with one another as independent agents to make node-movement and replication decisions. This is in contrast to the current algorithm which requires knowledge of all nodes' information to be stored and processed at a central location. Development of a distributed version of this algorithm would be an important step towards a real deployment of our system.

APPENDIX A

COST FUNCTIONS

A.1 Evenness Cost

In our implementation we use the following definition for evenness cost $C_E(p_a)$:

$$\begin{aligned} t(p_a) &= \sum_{\substack{i \in V \\ p(i)=p_a}} w_i \\ \hat{t} &= \frac{\sum_{p \in P} t(p)}{|P|} \\ \hat{e}(p_a) &= \frac{\frac{\sum_{p \in P} t(p)}{|P|} - t(p_a)}{\sum_{p \in P} t(p)} \frac{\sum_{i \in V} m_i}{|V|} \\ C_E(p_a) &= \beta |\hat{e}(p_a)| \hat{e}(p_a) \end{aligned}$$

A.2 Replication Cost

The following is used for replication cost C_R. Note that the current replication index is represented as I_R, and the target replication factor parameter is r_{target}. The list of recorded current replicas' costs is H.

$$C_R = \begin{cases} \min\left(\left\{x \mid y \in H, |\{y < x\}| > \left(\frac{I_R - r_{\text{target}}}{I_R}|H|\right)\right\}\right) & \text{if } I_R > r_{\text{target}} \\ 0 & \text{otherwise} \end{cases}$$

Note that at the end of each iteration, as an additional step, the replica count $|H|$ is checked against $(r_{\text{cap}} - 1)|V|$, where r_{cap} is the maximum allowed replication factor I_R. If it is greater, the least valuable replicas are removed to bring I_R back down to r_{cap}.

REFERENCES

[1] N. Kallen, R. Pointer, E. Ceaser, and J. Kalucki, “Introducing Gizzard, a framework for creating distributed datastores,” *Twitter Engineering Website*, vol. 40, April 2006. [Online]. Available: http://engineering.twitter.com/2010/04/introducing-gizzard-framework-for.html

[2] A. Lakshman and P. Malik, “Cassandra: a decentralized structured storage system,” *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1773912.1773922

[3] M. Yuan, D. Stein, B. Carrasco, J. M. F. Trindade, and Y. Lu, “Partitioning social networks for fast retrieval of time-dependent queries,” 2012, presented at the 3rd International Workshop on Graph Data Management: Techniques and Applications (GDM ’12).

[4] T. Hoff, “Facebook’s memcached multiget hole: More machines != more capacity,” *High Scalability*, October 2009. [Online]. Available: http://highscalability.com/blog/2009/10/26/facebooks-memcached-multiget-hole-more-machines-more-capacit.html

[5] J. M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez, “The little engine(s) that could: scaling online social networks,” *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 4, pp. 375–386, Aug. 2010. [Online]. Available: http://doi.acm.org/10.1145/1851275.1851227

[6] A. Silberstein, A. Machanavajjhala, and R. Ramakrishnan, “Feed following: the big data challenge in social applications,” in *Databases and Social Networks*, ser. DBSocial ’11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/1996413.1996414 pp. 1–6.

[7] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 205–220, Oct. 2007. [Online]. Available: http://doi.acm.org/10.1145/1323293.1294281

[8] J. Pujol, G. Siganos, V. Erramilli, and P. Rodriguez, "Scaling online social networks without pains," presented at Workshop on Networking Meets Databases (NetDB) in cooperation with SOSP 2009.

[9] P.-O. Fjallstrom, "Algorithms for graph partitioning: A survey," *Linkoping Electronic Articles in Computer Information Science*, 1998.

[10] C. Curino, E. Jones, Y. Zhang, and S. Madden, "Schism: a workload-driven approach to database replication and partitioning," *Proc. of the VLDB Endowment*, vol. 3, 2010.

[11] L. Jiao, T. Xu, J. Li, and X. Fu, "Latency-aware data partitioning for geo-replicated online social networks," in *Proceedings of the Workshop on Posters and Demos Track*, ser. PDT '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2088960.2088975 pp. 15:1–15:2.

[12] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: fair scheduling for distributed computing clusters," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629601 pp. 261–276.

[13] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proceedings of the 5th European conference on Computer systems*, ser. EuroSys '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1755913.1755940 pp. 265–278.

[14] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graph evolution: Densification and shrinking diameters," *ACM Trans. Knowl. Discov. Data*, vol. 1, March 2007.

[15] G. Karypis and V. Kumar, "Metis: Unstructured graph partitioning and sparse matrix ordering system," 2009. [Online]. Available: http://www.cs.umn.edu/˜metis