Optimizing Data Center Traffic of Online Social Networks

Lei Jiao
University of Göttingen
Göttingen, Germany
jiao@cs.uni-goettingen.de

Jun Li
University of Oregon
Eugene, OR, USA
lijun@cs.uoregon.edu

Xiaoming Fu
University of Göttingen
Göttingen, Germany
fu@cs.uni-goettingen.de

***Abstract*—With a huge number of users and a very large scale of data, an Online Social Network (OSN) service has to partition its data among multiple servers inside a data center. As data are often partitioned randomly, the response time in accessing the data is however unpredictable. Researchers have proposed social locality to address this concern: if a server hosts the master replica of a user's data, it must also host a replica (either master or slave) of every friend of this user, thus enabling convenient access of all of them on the same server. However, doing so comes with two overheads: the replication storage and the traffic of maintaining replica consistency. Existing work focuses on the former, but overlooks the latter that can consume considerable network resources. In this paper, we study social-locality-aware partitioning of the OSN data while meeting diverse performance goals of data center networks. We formulate the traffic optimization problem and propose a new traffic-aware data partitioning algorithm. Through the evaluations with a large-scale, real-world Twitter trace, we further show that, compared with state-of-the-art algorithms, our algorithm significantly reduces traffic without deteriorating the load balance among servers and causing extra replication storage.**

I. Introduction

Online Social Networks (OSNs) are extremely popular destinations for Internet users nowadays, *e.g.*, Facebook had 1 billion users as of October 2012 [1]. With users of such a huge scale, it is imperative to implement a scalable backend system to support users' data storage and access. Current OSN data center infrastructures often adopt distributed DBMS (*e.g.*, MySQL) and/or key-value stores (*e.g.*, Cassandra [2]), which essentially distribute users' data among servers randomly. Though simple and efficient, random distribution fails to match the OSN data access patterns and can thus suffer from performance problems. For instance, in a typical OSN service such as Facebook News Feed or Twitter, to display a user's home page, the service must access and collect the data of this user's every friend from multiple servers, with unpredictable response time determined by the server with the highest latency. This problem becomes particularly severe when servers and the data center network are under heavy workload.

To address this issue, it has been proposed to replicate the data of a user's every friend to the server where this user's own data are stored, *i.e.*, maintaining *social locality* so that services such as News Feed can be resolved within a single server [3]. In this paradigm, each user has a single master replica, with which the replicas of friends' data are co-located on the same server; each user also has multiple slave replicas on different servers co-located with friends' master replicas. The overheads of social locality are two-fold: the replication storage of slave replicas and the traffic from master to slaves in order to maintain the consistency.

Existing work mainly focuses on optimizing the storage [3], [4], yet overlooks the traffic aspect. According to [3], the minimum average number of slave replicas per user to ensure social locality is up to 20 for an OSN service on a cluster of 512 servers. Given about 3.2 billion daily Facebook comments [5] and the average packet size of 1 KB, the traffic for synchronizing replicas can be up to about 60 TB per day, which could consume considerable data center network resources, not to mention other user-generated contents. In industrial data centers, the networks are often the bottleneck [6], [7], [8]; Besides, the user-facing service traffic (*e.g.*, the traffic between OSN service and OSN users) and the backend synchronization traffic shares a common data center network infrastructure, competing for network resources [9]. Therefore, optimizing the backend traffic can yield more network resources for the user-facing service, and can improve the salability of data center networks.

In this paper, we study the problem of social-locality-aware partitioning of the OSN data backend in a data center environment. While embracing social locality's advantages such as eliminating unpredictable inter-server response time, we aim to minimize its overhead in the traffic aspect without ruining the existing optimization (if any) of the storage aspect.

We explicitly consider data center network topologies (*e.g.*, tree [9], Clos topology [6], *etc.*) together with social locality. Different topologies have different features, we consequently define diverse network performance goals for the synchronization traffic to save network resource consumption. We further formulate the traffic optimization problem and propose a unified solution to achieve all network performance goals — our traffic-aware partitioning algorithm which is inspired by the fact that carefully swapping the roles of the master replica and a slave replica of a user can lead to traffic reduction. Trace-driven simulations with a large-scale, real-world Twitter dataset demonstrate that, compared with state-of-the-art algorithms, such as random placement (*i.e.*, the standard placement in MySQL and Cassandra), SPAR [3], and

METIS [10], our algorithm can reduce the synchronization traffic by approximately 30%-70% in a variety of data center network topologies with a number of servers, without affecting the existing load balance among servers and increasing the total replication storage.

II. Problem Statement

We briefly introduce the social locality paradigm and data center network topologies. For different topologies, we propose diverse network performance goals for the synchronization traffic. We then present the traffic optimization problem with diverse goals by a unified formulation.

A. Revisiting Social Locality

"Social locality" is a data *replication* scheme independent of data *partitioning*. The latter means dividing the whole dataset into separate subsets (*i.e.*, partitions), each of which is placed on a different server. In contrast, data can be replicated across servers. The social locality scheme chooses to replicate the data of a user's every friend to the server that hosts this user's own data, which has proved to be an effective approach to overcome the performance problems of OSN services.

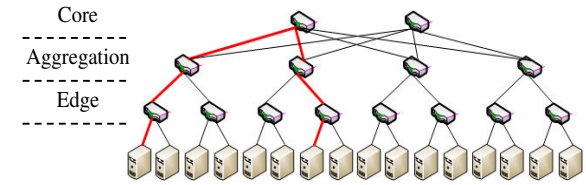
Social locality is a single-master-multi-slave paradigm. The partition that hosts a user's master is determined by the partitioning scheme; the partitions that host a user's slaves are determined by the social relations among users and also by the locations of the masters of this user's friends. The replica consistency is maintained by the synchronization traffic from a user's master to her slaves. Load balance in this context refers to balancing the number of masters among servers [3].

B. Defining Network Performance Goals

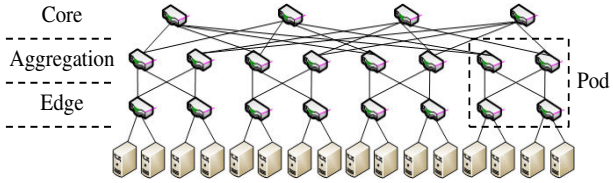
The *de facto* standard of data center network topology is the two- or three-layer tree [9], interconnecting servers by switches and/or routers. In a three-layer tree, at the bottom, servers in the same rack are connected to a top-of-rack or *edge* switch. Each edge switch is connected to an *aggregation* switch, and each aggregation switch is connected to one or multiple *core* switches. Given that each edge switch connects with k_1 servers and each aggregation switch connects with k_2 edge switches, the number of servers that are hosted by one aggregation switch is thus $k_1 k_2$. The tree topology is quite often *oversubscribed* in modern data centers in order to lower the total cost of such design [6].

A couple of full-capacity topologies (*e.g.*, fat-tree [6], BCube [11], *etc.*) have been proposed to overcome the oversubscription problem of the tree topology. Fat-tree is a design that is composed of servers and k-port switches. In a fat-tree, there are k *pod*s with $k/2$ edge switches and $k/2$ aggregation switches in each pod, $k^2/4$ core switches, and $k^3/4$ servers. The $k/2$ ports of each edge switch connect with $k/2$ servers, and the rest $k/2$ ports connect with different aggregation switches in the same pod. The rest $k/2$ ports of each aggregation switch connect with different core switches.

Fig. 1 depicts a tree and a fat-tree, with $k_1 = k_2 = 2$ and $k = 4$, respectively.



(a) Three-layer tree



(b) Fat-tree

Fig. 1. Data center network topologies

Targeting at different data center topologies, we define diverse performance goals for the synchronization traffic. For the tree topology, it is desirable to localize the traffic to save the utilization of the oversubscribed upper-layer network links [8]. If a user's master and all her slaves are on servers in the same rack, synchronization only involves intra-rack traffic; otherwise, the synchronization traffic must go beyond the edge switch to upper layers in order to reach replicas on servers in another rack. Fig. 1(a) uses red lines to exemplify a path between servers via the core switch. We consequently define the following goal of traffic localization:

- **Goal #1** Minimize the core synchronization traffic, *i.e.*, the traffic traveling through core switch(es)

For full-capacity topologies, it is not necessary to localize the traffic due to the absence of oversubscription [8], but it is desirable to reduce the utilization of every switch and link to improve the network scalability [7]. We thus have the following goal, which is also applicable to the tree topology:

- **Goal #2** Minimize the total synchronization traffic, *i.e.*, the sum of the traffic perceived by every switch

Note that the performance goals of a data center network are not limited to the ones that we define here, *e.g.*, for a fat-tree, localizing the traffic also makes sense if upper-layer links suffer from congestion. As will be shown next, it is easy to use our model to express any network performance goal, and our proposed algorithm provides a general and efficient approach to reach all these goals.

C. Model Formulation

OSN is often modeled as a graph [12], where each user is represented by a vertex and each social relation between two users is represented by an edge between the corresponding two vertices. We additionally consider each user's traffic rate (*i.e.*, the size of the data generated by a user) in this paper.

Given such a social graph with each user's traffic rate, we are interested in the problem of partitioning the graph into N partitions, maintaining social locality for each user with

the synchronization traffic achieving our network performance goals. Additional inputs to the problem include pre-specified numbers of master replicas on each server and a pre-specified total number of slave replicas in the entire system. Our partitioning ensures that, the number of masters on each server equals to the corresponding pre-defined number for this server (*i.e.*, guaranteeing the load assignment across servers, or load balance if such number is the same for all servers), and the total number of slaves in the system does not exceed the pre-defined number (*i.e.*, guaranteeing the total replication storage within the given quota).

We introduce notations to formulate the problem. $G = (V, E)$ denotes the undirected social graph, where V is the set of vertices (*i.e.*, users) and E is the set of edges (*i.e.*, social relations). e_{ij} is the edge between user i and user j. t_i is the traffic rate of user i. $A(i, j)$ is the value representing the *adjacency* between server i and server j in the $N \times N$ control matrix A, where N is the total number of servers. M_j is the pre-defined number of masters on server j and S is the pre-defined total number of slaves in the system. $m(i, j)$ is binary, being 1 if the master of user i is assigned to server j and being 0 otherwise. Thus $m_i = \sum_{\forall j}(j \times m(i, j))$ is the server which hosts user i's master. $s(i, j)$ is similar to $m(i, j)$ but representing the assignment of salves to servers. We formulate the problem as follows.

minimize:

$$\sum_{\forall i} \sum_{\forall j \in \{j | s_{ij}=1, \forall j\}} (t_i \times A(m_i, j \times s(i, j)))$$

subject to:

$$\sum_{\forall j} m(i, j) = 1, \forall i \tag{1}$$

$$m(i', m_i) + s(i', m_i) = 1, \forall i, i', e_{ii'} \in E \tag{2}$$

$$\sum_{\forall i} m(i, j) = M_j, \forall j \tag{3}$$

$$\sum_{\forall i} \sum_{\forall j} s(i, j) \leq S \tag{4}$$

The objective is to minimize the traffic from masters to slaves, counted by a given control matrix. Each of our goals can be expressed by a particular control matrix. Thus our problem formulation applies uniformly to all the goals that are defined previously. Constraint (1) ensures a single master for every user. Constraint (2) ensures social locality for every user. Constraint (3) ensures that the distribution of masters on servers matches the pre-defined load assignment. Constraint (4) ensures that the total replication storage does not exceed the pre-defined quota.

The control matrix is used to count the traffic for a given performance goal in a given data center topology. For Goal #1, we only care about the core-layer traffic, and thus we only set the adjacency value between any two servers located under different aggregation switches to 1. For Goal #2, we count the traffic at every switch. If there are n switches in the communication path between any two servers, the adjacency value between these two servers are set to n. Aligned with the descriptions of data center network topologies in Section II-B, we present as follows the control matrices for the goals of the tree and the fat-tree topology, respectively.

- Control matrix of tree for Goal #1:

$$A_{t1}(i, j) = \begin{cases} 0, i = j \\ 0, i \neq j \wedge \left\lfloor \frac{i}{k_1} \right\rfloor = \left\lfloor \frac{j}{k_1} \right\rfloor \\ 0, i \neq j \wedge \left\lfloor \frac{i}{k_1} \right\rfloor \neq \left\lfloor \frac{j}{k_1} \right\rfloor \wedge \left\lfloor \frac{i}{k_1 k_2} \right\rfloor = \left\lfloor \frac{j}{k_1 k_2} \right\rfloor \\ 1, otherwise \end{cases}$$

- Control matrices of tree and fat-tree for Goal #2:

$$A_{t2}(i, j) = \begin{cases} 0, i = j \\ 1, i \neq j \wedge \left\lfloor \frac{i}{k_1} \right\rfloor = \left\lfloor \frac{j}{k_1} \right\rfloor \\ 3, i \neq j \wedge \left\lfloor \frac{i}{k_1} \right\rfloor \neq \left\lfloor \frac{j}{k_1} \right\rfloor \wedge \left\lfloor \frac{i}{k_1 k_2} \right\rfloor = \left\lfloor \frac{j}{k_1 k_2} \right\rfloor \\ 5, otherwise \end{cases}$$

$$A_{ft2}(i, j) = \begin{cases} 0, i = j \\ 1, i \neq j \wedge \left\lfloor \frac{2i}{k} \right\rfloor = \left\lfloor \frac{2j}{k} \right\rfloor \\ 3, i \neq j \wedge \left\lfloor \frac{2i}{k} \right\rfloor \neq \left\lfloor \frac{2j}{k} \right\rfloor \wedge \left\lfloor \frac{4i}{k^2} \right\rfloor = \left\lfloor \frac{4j}{k^2} \right\rfloor \\ 5, otherwise \end{cases}$$

III. Traffic-aware Partitioning

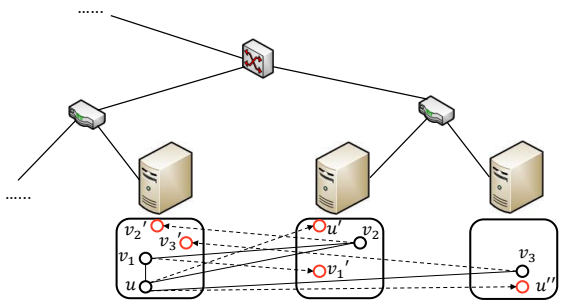
The traffic optimization problem is an Integer Linear Program that generally belongs to NP-hard problems [13]. We thus focus on developing a heuristic approach that works well in practice.

A. Overview

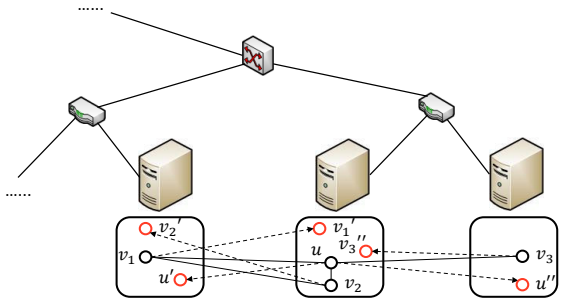
Starting with an initial solution (*i.e.*, a trial assignment of all replicas to servers), our algorithm tweaks this solution iteratively to search the solution space for the optimum. Each tweak operation changes the current assignment into a new one that has less traffic counted by the control matrix, without violating any of the constraints.

Swapping the roles of a user's master and slave can be used as the tweak operation [14]. As an example, Figure 2 is a local view of part of the tree topology, where three servers are interconnected by two edge switches and one aggregation switch. The box below each server shows the current data replicas on this server. There are four users u, v_1, v_2 and v_3. Black circles are masters, and red ones are slaves that exist for maintaining social locality of masters. Solid lines represent social relations and dotted arrows represent the synchronization traffic. Let's consider the total traffic perceived by every switch, and let's assume all users have the same traffic rate of 1 unit, for example. The existing data assignment has the total traffic of 15 units, as in Figure 2(a). Figure 2(b) and Figure 2(c) perform the role-swaps. Firstly, we swap the roles of user u's master and her slave u', reducing the total traffic to 11 units. Secondly, we select v_2 and v_2', and swap the roles in order to maintain the existing load assignment. Social locality must be maintained after each role-swap. Overall, we achieve 4 units traffic reduction without altering the load assignment of the existing data placement (*i.e.*, the number of masters on each

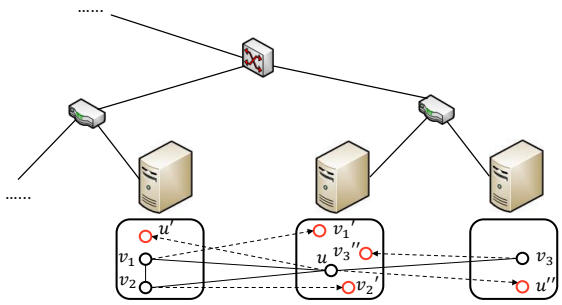
server remains the same before and after the two role-swaps) and without increasing the total replication storage (*i.e.*, the total number of slaves does not increase after the two role-swaps).



(a) Before swapping: traffic = 15



(b) After swapping the roles of u and u': traffic = 11



(c) After swapping the roles of v_2 and v_2': traffic = 11

Fig. 2. Using role-swaps to reduce the traffic

Note that, in order to always guarantee the load assignment, a single tweak must include two role-swaps, *i.e.*, we must select two users and do the role-swap for each of them. The two users' masters are on different servers, and each user's slave is co-located with the other user's master. Only two users satisfying this condition can be considered as candidates for a tweak operation.

B. Partitioning Algorithm

Algorithm 1 provides the pseudo codes of our partitioning algorithm. p_{start} is the starting solution; $Best$ maintains the best solution that has been found. For a tweak, two users u and v are selected. m_u is the server that hosts u's single master, and s_u is the server that hosts u's slave involved in this tweak. μ_u is the total number of slave replicas that can be reduced and τ_u is the amount of traffic (counted by the control matrix) that can be saved by swapping the roles of u's master on m_u and her slave on s_u. m_v, s_v, μ_v, and τ_v have similar meanings for user v. Δ denotes the total number of slave replicas that has been reduced so far compared with the starting solution.

Algorithm 1: partition(p_{start})

```
begin
    Best ← p_start, Δ ← 0;
    repeat
        (m_u, s_u, m_v, s_v) ← selectUsers();
        (μ_u, τ_u) ← getReduction(m_u, s_u);
        Best ← swapRole(Best, m_u, s_u);
        (μ_v, τ_v) ← getReduction(m_v, s_v);
        if Δ + μ_u + μ_v ≥ 0 and τ_u + τ_v > 0 then
            Best ← swapRole(Best, m_v, s_v);
            Δ ← Δ + μ_u + μ_v;
        else
            Best ← swapRole(Best, s_u, m_u);
    until Best is the ideal solution or we run out of time;
    return Best
```

Algorithm 2: getReduction(m_u, s_u)

```
begin
    μ_u ← 0, τ_u ← 0;
    Non_m_u ← ∅, Non_s_u ← ∅;
    Adjacency_m_u ← 0, Adjacency_s_u ← 0;
    Remove_m ← true, Remove_s ← true;
    for each v ∈ u's neighbors do
        if m_v ≠ m_u then
            Non_m_u ← Non_m_u ⋃ m_v;
            if u is v's only neighbor on m_u then
                μ_u ← μ_u + 1, τ_u ← τ_u + t_v × A(m_v, m_u);
            if m_v = s_u then
                Remove_s ← false;
        if m_v ≠ s_u then
            Non_s_u ← Non_s_u ⋃ m_v;
            if u is v's only neighbor on s_u then
                μ_u ← μ_u − 1, τ_u ← τ_u − t_v × A(m_v, s_u);
            if m_v = m_u then
                Remove_m ← false;
    if Remove_s = true then
        μ_u ← μ_u − 1;
    if Remove_m = true then
        μ_u ← μ_u + 1;
    for each i ∈ Non_m_u do
        Adjacency_m_u ← Adjacency_m_u + A(m_u, i);
    for each i ∈ Non_s_u do
        Adjacency_s_u ← Adjacency_s_u + A(s_u, i);
    τ_u ← τ_u + t_u × (Adjacency_m_u − Adjacency_s_u);
    return (μ_u, τ_u)
```

Algorithm 1 adopts a hill-climbing strategy by requiring that *every* tweak must reduce the traffic (*i.e.*, $\tau_u + \tau_v > 0$). We are aware of other design options, *e.g.*, Simulated Annealing [15] which in our case allows tweaks with traffic increase, *etc.* While such techniques may discover better solutions or approximate closer to the theoretical optimum(s), it is easy to integrate them to our algorithm, and we leave this as possible future work since we find that hill-climbing can already achieve significant traffic reductions in practice, as will

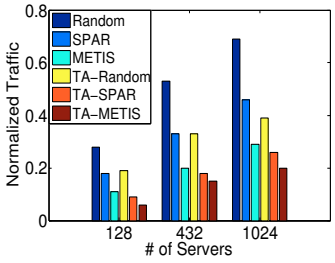


Fig. 3. Core-layer traffic

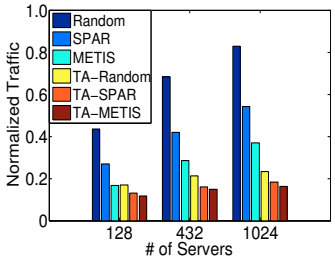


Fig. 4. Perceived traffic (tree)

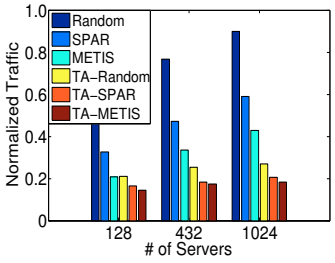


Fig. 5. Perceived traffic (fat-tree)

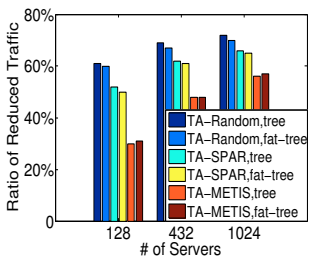


Fig. 6. Traffic reduction ratio

be shown in Section IV-B.

Algorithm 2, invoked by Algorithm 1, specifies the calculation of the replica number reduction and the traffic reduction that can be achieved by a given role-swap. An intuitive alternative to get the reductions is calculating the total replica number and the total traffic before and after a tweak, respectively, and then calculating the difference for each of them. However, compared with Algorithm 2 which only accesses the neighborhood of the selected user, this intuitive approach needs to access every user in the system and can cause considerable computation overhead for a large social graph.

C. Complexity Considerations

The design of our algorithm is partially guided by our complexity considerations. The time complexity of Algorithm 2 is $O(|V| + 2 \times N) = O(|V|)$, where $|V|$ is the total number of users and N is the total number of servers, given $N \ll |V|$. Without Algorithm 2, the intuitive method of calculating the reductions as mentioned above will be of $O(|V|^2)$.

In Algorithm 1, we need to select two users with their role-swaps. Different selection strategies usually have different trade-offs between time complexity and the amount of traffic reduction. A greedy selection may have a good traffic reduction, but it takes more time, because after a role-swap is performed, we must re-calculate the reductions of role-swaps of all this selected user's neighbors, which takes $O(|V|^2)$ if we do all calculations by Algorithm 2, and then sort all the role-swaps again, which takes $O(|E| \log |E|)$ where $|E|$ is the total number of social relations. Therefore, the greedy selection has a complexity of $O(|V|^2 + |E| \log |E|)$. In contrast, a random selection only has $O(1)$, but may achieve less reductions than greedy. We take random selection in this paper.

IV. Experimental Evaluation

A. Experimental Settings

OSN Dataset. By crawling Twitter in a breadth-first searching manner in March 2010, we collected a dataset of 107,734 users with 2,744,006 social relations. For each crawled user, we have her profile, tweets, and the followers list. We use the total size of each user's tweets published in February 2010 as the user's traffic rate.

Data center network topology. We simulate switches of 8, 12, and 16 ports, respectively, and use them to organize a tree and a fat-tree topology. In a fat-tree, the total number of switches and that of servers are determined by the number of ports per switch, as mentioned in Section II-B; In a tree, we always use 2 switches in the core layer. Table I contains the details of the network configurations. For each topology, we build the corresponding control matrix by the formulas presented in Section II-C.

TABLE I
Data center network configurations

k, k_1, k_2		8	12	16
# of servers		128	432	1024
# of switches	tree	20	41	70
	fat-tree	80	180	320

Initial assignment. The initial assignment serves as the starting solution in our evaluations. For our social graph with each user's traffic rate, we use random placement, SPAR [3], and METIS [10] to generate an initial assignment of master replicas to servers, respectively, and slave replicas are then assigned to servers to maintain social locality for each user. We implement SPAR on our own, strictly following [3]; METIS has an open-source implementation that we can directly use.

B. Evaluation Results

The results are illustrated in Fig. 3, 4, 5, and 6, where "TA-" denotes our traffic-aware partitioning, starting with a specified initial assignment, and traffic values have been normalized.

Fig. 3 shows the traffic that passes the core-layer switches in the tree topology. We use our algorithm to reduce such core-layer traffic to achieve Goal #1. Fig. 4 and 5 compare the total traffic perceived by every switch in a tree and a fat-tree, respectively. We apply our algorithm to reduce such traffic to achieve Goal #2. We find that, for any given topology with a fixed number of servers, users placed by METIS tend to incur less core-layer traffic and total perceived traffic than by random and SPAR. This meets our expectation. METIS explicitly minimizes the inter-server traffic, while SPAR minimizes the total number of slave replicas, equivalent to minimizing the inter-server traffic with the assumption that each user has the same traffic rate, *i.e.*, SPAR essentially ignores the difference of users' traffic rates. As the number of servers increases, the number of users per server drops and users tend to be placed under different aggregation switches, which explains why the core-layer traffic and the total perceived traffic grows.

We notice that our traffic-aware partitioning algorithm can significantly reduce the core-layer traffic and the total per-

ceived traffic on top of random, SPAR and METIS, respectively. Our algorithm makes no assumption about the initial assignment and the underlying data center network topology, and can work effectively. Fig 6 makes clear the ratio of the reduced traffic over the total perceived traffic. It is easy to see that our algorithm is not sensitive to the type of data center topologies since tree and fat-tree have similar traffic reduction ratios for a fixed number of servers. It also shows that random has the largest traffic reduction ratio than SPAR and METIS, implying that random has a larger room for our algorithm to optimize. Both SPAR, which optimizes the total replication storage, and METIS, which optimizes the inter-server traffic, cannot automatically meet our Goal #1 or Goal #2 (*i.e.*, they do not place replicas to achieve specific network performance goals), while our traffic-aware partitioning algorithm can minimize the traffic in order to align with the network performance goals.

V. Related Work

Commercial OSN services usually adopt distributed hashing to partition the data backend [2], [16], which can lead to poor performance (*e.g.*, response time) due to the inter-server multi-get operations caused by OSN data access patterns. To address this, recent work proposes to eliminate multi-get operations by maintaining social locality, *i.e.*, replicating friends across servers [3], [4], or reduce such operations by carefully placing users' data on servers without replication, *e.g.*, exploring self-similarities of user interactions [17].

Carrasco *et al.* partition OSN along the time dimension, *i.e.*, ensuring data locality for different users at different time periods in order to save storage [18] over time. Cheng *et al.* partition the YouTube social media network by preserving social relations and balancing viewing workload among servers [19]. Curino *et al.* partition relational DBMS aiming at minimizing the number of distributed transactions [20]. Similar to [3], tuple replication across servers is applied if multiple transactions access the same tuple simultaneously.

A number of literatures study the graph (re)partitioning problems. Graph partitioning refers to dividing a weighted graph into a given number of partitions in order to minimize either the inter-partition edge weights or the inter-partition communication volume while balancing the vertex weights of each partition [21]. Graph repartitioning further considers the existing partitioning and additionally minimizes the extra cost of migrating vertices across partitions [22].

The problem we study in this paper does not map to any of these existing work. OSN and DBMS partitioning focus on server performance and neglect the networks. Graph (re)partitioning problems, though aiming at minimizing inter-partition communication, is also inapplicable for our case. Firstly, they do not have social locality. Secondly, minimizing the total inter-server communication does not necessarily minimize the traffic traveling via the core-layer switches or the total traffic perceived by every switch, as shown in this paper.

VI. Conclusion

As the OSN data have to be partitioned among multiple servers in a data center, in this paper we studied how to partition them not only with social locality, but also with optimized inter-server traffic. We model the problem, propose a traffic-aware partitioning algorithm, and evaluate how our algorithm works with real-world Twitter data. Our experiments show that regardless of the data center topology and the initial data assignment, our algorithm can significantly reduce the core-layer traffic and the total traffic that is perceived by every switch. Its performance is superior to not only random data partitioning, but also state-of-the-art algorithms including SPAR and METIS.

References

[1] "Key facts - facebook newsroom," http://newsroom.fb.com/Key-Facts.

[2] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[3] J. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez, "The little engine(s) that could: Scaling online social networks," in *SIGCOMM*, 2010.

[4] D. Tran, K. Nguyen, and C. Pham, "S-clone: Socially-aware data replication for social networks," *Computer Networks*, vol. 56, no. 7, pp. 2001–2013, 2012.

[5] "How to use facebook for business marketing," http://www.facebook.com/business/overview.

[6] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *SIGCOMM*, 2008.

[7] X. Meng, V. Pappas, and L. Zhang, "Improving the scalability of data center networks with traffic-aware virtual machine placement," in *INFOCOM*, 2010.

[8] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, "Improving datacenter performance and robustness with multipath tcp," in *SIGCOMM*, 2011.

[9] *Cisco Data Center Infrastructure 2.5 Design Guide*. Cisco Systems, Inc., Nov. 2, 2011.

[10] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1999.

[11] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, "Bcube: a high performance, server-centric network architecture for modular data centers," in *SIGCOMM*, 2009.

[12] A. Mislove, M. Marcon, K. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and analysis of online social networks," in *IMC*, 2007.

[13] M. Gary and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness*. W.H. Freeman, Jan. 15, 1979.

[14] L. Jiao, J. Li, T. Xu, and X. Fu, "Cost optimization for online social networks on geo-distributed clouds," in *ICNP*, 2012.

[15] S. Kirkpatrick, C. Gelatt, and M. Vecchi, "Optimization by simmulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[16] "Twitter engineering: Introducing gizzard, a framework for creating distributed datastores," http://engineering.twitter.com/2010/04/introducing-gizzard-framework-for.html.

[17] H. Chen, H. Jin, N. Jin, and T. Gu, "Minimizing inter-server communications by exploiting self-similarity in online social networks," in *ICNP*, 2012.

[18] B. Carrasco, Y. Lu, and J. Trindade, "Partitioning social networks for time-dependent queries," in *EuroSys SNS*, 2011.

[19] X. Cheng and J. Liu, "Load-balanced migration of social media to content clouds," in *NOSSDAV*, 2011.

[20] C. Curino, E. Jones, Y. Zhang, and S. Madden, "Schism: a workload-driven approach to database replication and partitioning," in *VLDB*, 2010.

[21] A. Abou-Rjeili and G. Karypis, "Multilevel algorithms for partitioning power-law graphs," in *IPDPS*, 2006.

[22] K. Schloegel, G. Karypis, and V. Kumar, "Wavefront diffusion and lmsr: Algorithms for dynamic repartitioning of adaptive meshes," *IEEE TPDS*, vol. 12, no. 5, pp. 451–466, 2001.