The Little Engine(s) That Could: Scaling Online Social Networks

Josep M. Pujol, Vijay Erramilli, Georgos Siganos, Xiaoyuan Yang
Nikos Laoutaris, Parminder Chhabra, Pablo Rodriguez
Telefonica Research
{ jmps, vijay, georgos, yxiao, nikos, pchhabra, pablorr }@tid.es

ABSTRACT

The difficulty of scaling Online Social Networks (OSNs) has introduced new system design challenges that has often caused costly re-architecting for services like Twitter and Facebook. The complexity of interconnection of users in social networks has introduced new scalability challenges. Conventional vertical scaling by resorting to full replication can be a costly proposition. Horizontal scaling by partitioning and distributing data among multiples servers – *e.g.* using DHTs – can lead to costly inter-server communication.

We design, implement, and evaluate SPAR, a social partitioning and replication middle-ware that transparently leverages the social graph structure to achieve data locality while minimizing replication. SPAR guarantees that for all users in an OSN, their direct neighbor's data is co-located in the same server. The gains from this approach are multi-fold: application developers can assume local semantics, *i.e.*, develop as they would for a single server; scalability is achieved by adding commodity servers with low memory and network I/O requirements; and redundancy is achieved at a fraction of the cost.

We detail our system design and an evaluation based on datasets from Twitter, Orkut, and Facebook, with a working implementation. We show that SPAR incurs minimum overhead, and can help a well-known open-source Twitter clone reach Twitter's scale without changing a line of its application logic and achieves higher throughput than Cassandra, Facebook's DHT based key-value store database.

Categories and Subject Descriptors

D.3.4 Information Systems [**Information storage and retrieval**]: Systems and Software, Distributed systems; E.1 Data [**Data Structures**]: Graphs and networks

General Terms

Algorithms, Experiments, Performance

Keywords

Social Networks, Scalability, Partition, Replication

SIGCOMM'10, August 30–September 3, 2010, New Delhi, India.

1. INTRODUCTION

Recently, there has been a recent unprecedented increase in the use of Online Social Networks (OSNs) and applications with a social component. The most popular OSNs attract hundreds of millions of users – *e.g.* Facebook [16] – and deliver status updates at very high rates – *e.g.* Twitter [7]. OSNs differ from traditional web applications significantly in different ways: they handle highly personalized content; produce non-traditional workloads [11, 32]; and most importantly, they deal with highly interconnected data due to the presence of a strong community structure among their end users [23, 26, 27, 12]. All these factors create new challenges for the maintenance, management and scaling of OSN systems.

Scaling real systems is hard as it is, but the problem is particularly acute for OSNs, due to their interconnected nature and their astounding growth rate. Twitter, for instance, grew by 1382% between Feb and Mar 2009 [24] and was thus forced to redesign and re-implement its architecture several times in order to keep up with the demand. Other OSNs that failed to do so have virtually disappeared [2].

A natural and traditional solution to cope with higher demand is to upgrade existing hardware. Such *vertical scaling*, however, is expensive because of the cost of high performance servers. In some cases, vertical scaling can even be *technically infeasible*, *e.g.* Facebook requires multiple hundreds of Terabytes of memory across thousands of machines [3]. A more cost effective approach is to rely on *horizontal scaling* by engaging a higher number of cheap commodity servers and partitioning the load among them. The advent of cloud computing systems like Amazon EC2 and Google AppEngine has streamlined horizontal scaling by removing the need to own hardware and instead providing the ability to lease virtual machines (VMs) dynamically from the cloud. Horizontal scaling has eased most of the scaling problems faced by traditional web applications. Since the application front-end and logic is stateless, it can be instantiated on new servers on demand in order to meet the current load. The data back-end layer however, is more problematic since it maintains state. If data can be partitioned into disjoint components, horizontal scaling still holds. However, this last condition does not hold for OSNs.

In the case of OSNs, the existence of social communities [23, 26, 27], hinders the partitioning of the data back-end into clean, disjoint components[18, 17]. The problem in OSNs is that most of the operations are based on the data of a user and her neighbors. Since users belong to more than one community, there is no disjoint partition (*i.e.*

server) where users and all her neighbors can be co-located. This *hairball* that is the community structure causes a lot of inter-server traffic for resolving queries. The problem becomes particularly acute under random partitioning, which is the de-facto standard in OSNs [16, 30]. On the other hand, replicating user's data in multiple or all the servers eliminates the inter-server traffic for reads but increases the replication overhead. This has a negative impact on query execution times and network traffic for propagating updates and maintaining consistency across replicas. Scalability for OSNs is indeed a difficult beast to tame.

2. OUR CONTRIBUTION – SPAR

The main contribution of this work is the design, implementation and extensive evaluation of SPAR: a Social Partitioning And Replication middle-ware for social applications.

2.1 What does SPAR do?

Solves the Designer's Dilemma for early stage OSNs.

Designers and developers of an early-stage OSN are confronted with the Designer's Dilemma : "Should they commit scarce developer resources towards adding features or should they first ensure that they have a highly scalable system in place that can handle high traffic volume?". Choosing the first option can lead to "death-by-success" – users join attracted to appealing features, but if the infrastructure cannot support an adequate QoS, it results in frustrated users leaving the service en-masse – *e.g.* this was the story behind the demise of Friendster [2]. On the other hand, starting with a highly scalable system, similar to the ones that power established OSNs like Facebook and Twitter, requires devoting scarce resources to complex distributed programming and management issues. This comes at the expense of building the core of the application that attracts users in the first place.

SPAR avoids this dilemma by enabling transparent OSN scalability. SPAR constricts all relevant data for a user on a server. The enforcement of *local semantics* at the data level allows queries to be resolved locally on that server, creating the *illusion* that the system is running on one centralized server. This simplifies programming for developers and enables them to focus on the core features of the service.

Avoids performance bottlenecks in established OSNs.

SPAR avoids the potential performance problems of having to query multiple servers across a network by enforcing local semantics on the data. For instance, the de-facto standard random partitioning used by Twitter or Facebook, splits data across hundreds of data back-end servers. These servers are then queried with *multi-get* requests to fetch the neighbors' data (*e.g.*, all the friends' tweets). This can result in unpredictable response times, determined by the highest-latency server. The problem can be particularly acute under heavy data center loads, where network congestion can cause severe network delays.

In addition to potential network problems, individual servers could also suffer performance problems (*e.g.*, Network I/O, Disk I/O or CPU bottlenecks) and drive down the performance of the system. For instance, servers could become CPU bounded as they need to handle a larger number of query requests from other servers. When a server CPU is bound, adding more servers does not help serve more requests [1]. Using additional servers decreases the bandwidth per server. It does not, however, decrease the number of requests per server, which means that CPU usage stays roughly the same. SPAR reduces the impact of such multi-get operations by ensuring that all required data is kept local to the server, avoiding potential network and server bottlenecks and thus increasing throughput.

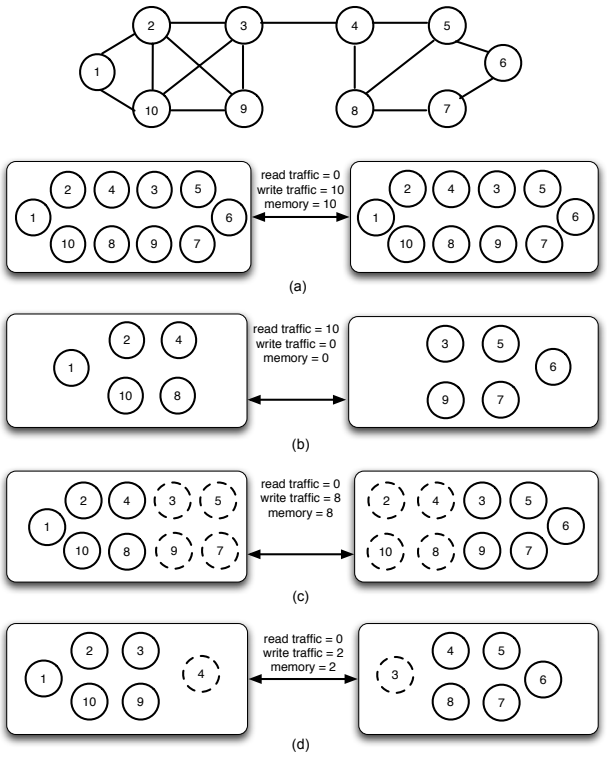

Figure 1: Sketch of social network to be partitioned in two servers using (a) Full Replication, (b) Partition using DHT (random partitioning), (c) Random Partitioning (DHT) with replication of the neighbors, (d) SPAR, socially aware partition and replication of the neighbors.

Minimizes the effect of provider lock-ins.

The Designer's Dilemma has prompted several providers of cloud services to develop and offer scalable "Key-Value" stores (so-called NoSQL) that run on top of DHTs. They offer transparent scalability at the expense of sacrificing the full power of established RDBMS, losing an expressive query language like SQL, powerful query optimizers and the encapsulation and abstraction of data related operations, *etc.* Further, systems such as Amazon's SimpleDB and Google's BigTable require using APIs that are tied to a particular cloud provider and thus suffer from poor portability that can lead to architectural lock-ins [9].

Cross platform Key-Value stores like Cassandra or CouchDB do not cause lock-in concerns, but suffer from the aforementioned shortcomings in addition to performance problems as we will argue shortly.

SPAR is implemented as a middle-ware that is platform agnostic and allows developers to select its preferred data-store, either be a Key-Value store or a relational database.

2.2 How does SPAR do it?

Through *joint partitioning and replication*. On the partition side, SPAR ensures that the underlying community structure is preserved as much as possible. On the replication side, SPAR ensures that data of all one-hop neighbors of a user hosted on a particular server is co-located on that same server, thereby guaranteeing local semantics of data. Note that most of the relevant data for a user in an OSN is one-hop away (friends, followers, *etc.*).

Fig. 1 is a toy-example highlighting the operation and benefits of SPAR. At the top of Fig. 1, we depict a social graph with 10 users (nodes) and 15 edges (bidirectional friendship relationships). The social graph contains an evident strong community structure: two communities that are connected through the "bridge" nodes 3 and 4. Then, we depict the physical placement of users on 2 servers under four different solutions. We summarize the memory (in terms of user profiles/data) and network cost assuming unit-sized profiles and a read rate of 1 for all profiles.

Random partition (b) – the de-facto standard of Key-Value stores – minimizes the replication overhead (0 units), and thus has the lowest memory (either RAM or Disk) footprint on the servers. On the downside, random partition imposes the highest aggregate network traffic due to reads (10 units), and thus increases the network I/O cost of the servers and the networking equipment that interconnects them. Replicating the neighbors as shown in (c) will eliminate the read traffic across servers, but will increase the memory in return. Another widely used approach is Full Replication (a). In this case, network read traffic falls to 0, but the memory requirements are high (10 units). Full replication also results in high write traffic to maintain consistency. Our proposed solution, SPAR (d), performs the best overall.

Summary of results.

The above toy example is a preview of the performance results of Sec. 5 based on workloads from Twitter, Orkut, and Facebook. We summarize the results here:

- SPAR provides local semantics with the least overhead (reduction of 200% over random in some cases), while reducing inter-server read traffic to zero.
- SPAR handles node and edge dynamics observed in OSNs with minimal overhead in the system. In addition, SPAR provides mechanisms for addition/removal of servers and handle failures gracefully.
- In our implementation, SPAR serves 300% more req/s than Cassandra while reducing network traffic by a factor of 8. We also show substantial gains when we implement SPAR with MySQL.

2.3 What does SPAR not do?

SPAR is not designed for the distribution of content such as pictures and videos in an OSN. This problem is well studied in the area of Content Distribution Networks (CDN). SPAR is an On-line Transaction Processing system (OLTP), it is not a solution for storage or for batch data analysis such as Hadoop (MapReduce). SPAR is not intended to characterize or aid in computing properties of the OSN graph, although we leave this for future work.

3. PROBLEM STATEMENT

We first describe the requirements that SPAR has to address. Next, we formulate the problem solved by SPAR. Finally, we discuss why existing social partition based solutions are inadequate to meet our set of requirements.

3.1 Requirements

The set of requirements that SPAR must fulfill are:

Maintain local semantics: Relevant data for a user in OSNs is her own and that of her direct neighbors (e.g. followees' tweets, friends' status updates, *etc.*). To achieve local semantics we need to ensure that for every *master replica* of a user, either a *master* or a *slave replica* of all her direct neighbors is co-located in the same server. We use the term *replica* to refer to a copy of the user's data. We differentiate between the *master* replica, serving all application level read/write operations; and the *slave* replica, required for redundancy and to guarantee data locality.

Balance loads: Application level read and write requests of a user are directed only to her *master* replica. Write operations need to be propagated to all her *slave* replicas for consistency. However, since masters handle much more load than slaves, we can obtain an approximately balanced load by doing an even distribution of masters among servers.

Be resilient to machine failures: To cope with machine failures, we need to ensure that all *masters* have at least K *slave* replicas that act as redundant copies.

Be amenable to online operations: OSN are highly dynamic; new users constantly join and there is a process of graph densification due to new edges formed between users [22]. Further, the infrastructure hosting the OSN may change through the addition, removal or upgrade of servers. To handle such dynamics, the solution needs to be *responsive* to such a wide range of events, and yet *simple* to quickly converge to an efficient assignment of users to servers.

Be stable: Given that we are operating in a highly dynamic environment, we need to ensure that the solution is stable. For example, addition of a few edges should not lead to a cascade of changes in the assignment of *masters* and *slaves* to servers.

Minimize replication overhead: The overall performance and efficiency of the system is strongly correlated to the number of replicas in the system. We require a solution that keeps replication overhead – defined as the average number of *slave* replicas created per user – as low as possible.

3.2 Formulation

Given the above requirements, we can formulate the solution as an optimization problem of minimizing the number of required replicas. For this purpose, we use the following notation. Let $G = (V, E)$ denote the social graph representing the OSN, with node set V representing users, and edge set E representing (friendship) relationships among users. Let $N = |V|$ denote the total number of users and M the number of available servers.

We could cast the problem as an integer linear program where p_{ij} denotes a binary decision variable that becomes 1 if and only if the primary of user i is assigned to partition j, $1 \leq j \leq M$. Also r_{ij} denote a similar decision variable for a replica of user i assigned to partition j. Finally, let the constants $\epsilon_{ii'} = 1$ if $\{i, i'\} \in E$ capture the friendship relationships. We state the MIN_REPLICA problem as follows:

$$\min \sum_i \sum_j r_{ij}$$

$$\text{s.t.} \quad \sum_{\forall j} p_{ij} = 1 \tag{1}$$

$$p_{ij} + \epsilon_{ii'} \leq p_{i'j} + r_{i'j} + 1, \forall i, j, i' \tag{2}$$

$$\sum_i (p_{ij}) = \sum_i (p_{i(j+1)}), 1 \leq j \leq M-1 \tag{3}$$

$$\sum_j r_{ij} \geq k, \forall i \in V \tag{4}$$

Constraint 1 in the above formulation ensures that there is exactly one master copy of a user in the system. Constraint 2 ensures that all neighbors of a user (be it masters or slaves) are in the same machine. Constraint 3 tries to distribute equal number of master replicas across the machines and Constraint 4 encodes the redundancy requirement.

LEMMA 1. *Min_Replica is NP-Hard*

A simple reduction from the graph Min-bisection problem [14] can be used to prove this. We skip the description of the formal proof for lack of space.

3.3 Why graph/social partitioning falls short

An obvious set of candidates that can be used to address the problem described in the previous section include graph partitioning algorithms [10, 19] and modularity optimization algorithms [23, 26, 27, 12]. These algorithms either aim to find equal sized partitions of a graph such that the number of inter-partition edges is minimized, or try to maximize modularity[27]. There are, however, several reason why these methods are inadequate for our purpose:

• Most graph partitioning algorithms are not incremental (offline) [12, 27, 10]. This poses a problem when dealing with highly dynamic social graphs, as they require costly re-computation of the partition. An incremental (online) algorithm is more suited for this scenario.

• Algorithms based on community detection are known to be extremely sensitive to input conditions. Small changes to the graph structure can lead to very different placement of nodes into partitions [20]. In other words, they do not produce *stable* solutions.

• It can be argued that directly reducing the number of inter-partition edges is equivalent to a reduction of the number of replicas. However, it is not the case. Consider the example depicted in Fig. 2. Minimizing the number of inter-partition edges will result in partitions P_1 and P_2, with only 3 inter-partition edges but this requires 5 replicas to maintain locality. On the other hand, partitions P_3 and P_4 result in 4 inter-partition edges, but this requires one less replica. As we will show later in Sec. 5 minimizing inter-partition edges indeed leads to worse results. Motivated by the shortcomings of the existing solutions, we present our online solution in the next section.

4. SPAR: JOINT PARTITIONING AND REPLICATION ON THE FLY

Having described the MIN_REPLICA problem and the requirements that it must fulfill, we now present our heuristic solution based on a greedy optimization using local information.

4.1 Overview

We assume that users represent nodes in a graph, and edges are formed when users create links among them. The algorithm runs in the Partition Manager module as explained in Sec. 6. In the average case, the required information for the algorithm is proportional to the product of the average node degree and the number of servers. The worst case computational complexity of the algorithm is proportional to the highest node degree. The algorithm is triggered by any one of the six following possible events: addition or removal of either nodes, edges, or servers.

4.2 Description

Node addition: A new node (user) is assigned to the partition with the fewest number of masters replicas. In addition, K slaves are created and assigned to random partitions.

Node removal: When a node is removed (a user is deleted), the master and all the slaves are removed. The states of the nodes that had an edge with it are updated.

Edge addition: When a new edge is created between nodes u and v, the algorithm checks whether both masters are already co-located with each other or with a master's slave. If so, no further action is required.

If not, the algorithm calculates the number of replicas that would be generated for each of the three possible configurations: 1) no movements of masters, which maintains the *status-quo*, 2) the master of u goes to the partition containing the master of v, 3) the opposite.

Let us start with configuration 1). Here, a replica is added if it does not already exist in the partition of the master of the complementary node. This may result in an increase of 1 or 2 replicas depending on whether the two masters are already present in each other's partitions. This can occur if nodes v or u already have relationships with other nodes in the same partition or if there already exist extra slaves of v or u for redundancy.

In configuration 2), no slave replicas are created for u and v since their masters will be in the same partition. However, for the node that moves, in this case u, we will have to create a slave replica of itself in its old partition to service the master of the neighbors that were left behind in that partition. In addition, the masters of these neighbors will have to create a slave replica in the new partition – if they do not already have one– to preserve the local semantics of u. Finally the algorithm removes the slave replicas that were in the old partition only to serve the master of u, since they are no longer needed. The above rule is also subject to maintaining a minimum number of slave replicas due to the K redundancy: the old partition slave will not be removed if the overall system ends up with less than K slaves for that particular node. Configuration 3) is complementary to 2).

The algorithm greedily chooses the configuration that yields the smallest aggregate number of replicas subject to the constraint of load-balancing the master across the partitions. More specifically, configuration 2) and 3) also need to ensure that the movement does not cause load unbalancing. That is, this movement either happens to a partition with fewer masters, or to a partition for which the savings in terms of number of replicas of the best configuration to the second best one is greater than the current ratio of load imbalance between partitions.

Fig. 3 illustrates the steps just described with an example. The initial configuration (upper-left subplot) contains

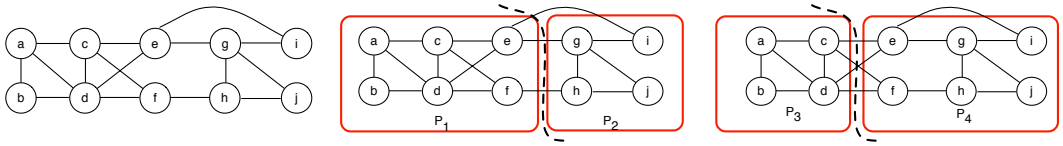

Figure 2: **Illustrative example on why minimizing edges between partitions (inter-partition) does not minimize replicas. The partition P_1 and P_2 results in 3 edges between partitions and 5 nodes to be replicated (e to i). The partitions P_3 and P_4 results in 4 inter-partition edges but only 4 nodes need to be replicated (c to f).**

6 nodes in 3 partitions. The current number of replicated nodes (empty circles) is 4. An edge between nodes 1 and 6 is created. Since there is no replica of 1 in M3 or replica of 6 in M1 if we maintain *status quo*, two additional replicas will have to be created to maintain the *local semantics*.

The algorithm also evaluates the number of replicas that are required for the other two possible configurations. If node 1 were to move to M3, we would need three new replicas in M3 since only 2 out of the 5 neighbors of node 1 are already in M3. In addition, the movement would allow removing the slave of node 5 from M1 because it is no longer needed. Consequently, the movement would increase the total number of replicas by 3-1=2, yielding a new total of 6 replicas, which is worse that maintaining the *status quo*.

In the last step, the algorithm evaluates the number of replicas for the third allowed configuration: moving the master of node 6 in M1. Here, the replica of node 5 in M3 can be removed because it already exists in M1 and no other node links to it in M3. Thus, no replica needs to be created. The change in the number of replicas is -1, yielding a total of 3 replicas.

Moving 6 to M1 minimizes the total number of replicas. However, such a configuration violates the load balancing condition and hence, cannot be performed. Thus, the final action is not to move (status quo) and create an additional 2 replicas.[1]

Edge removal: When an edge between u and v disappears, the algorithm removes the replica of u in the partition holding the master of node v if no other node requires it, and vice-versa. The algorithm checks whether there are more than K slave replicas before removing the node so that the desired redundancy level is maintained.

Server addition: Unlike the previous cases, server addition and removal do not depend on the events of the social graph but are triggered externally by system administrators or detected automatically by the system management tools.

There are two choices when adding a server: 1) force redistribution of the masters from the other servers to the new one so that all servers are balanced immediately, or 2) let the re-distribution of the masters be the result of the node and edge arrival processes and the load-balancing condition.

In the first case, the algorithm will select the $\frac{N}{M^2+M}$ least replicated masters from the M server and move them to the new server $M+1$. After the movement of the masters, the

[1]One might wonder what happens with power-users like Oprah. Must all her followers be replicated to the server hosting her master? Only the direct neighbors from whom a user is reading must be co-located. Power-users might have to be replicated across multiple servers but their large audience does not affect the system negatively. Also, OSNs impose a limit to the number of people that you can follow or befriend to avoid spammers and bots, *e.g.* 5000 for Facebook.

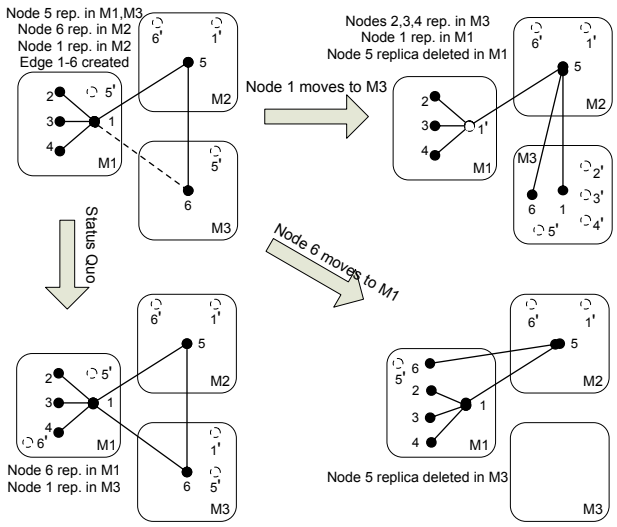

Figure 3: **SPAR Online Sketch**

algorithm will ensure that for all the masters moved to the new server, there is a slave replica of their neighbors to guarantee the local data semantics. This mechanism guarantees that the masters across all the $M+1$ servers are equally balanced. However, it may not provide a minimum replication overhead. For this reason, for a fraction of the edges of the masters involved, the algorithm also triggers a system-replay edge creation event, which reduces the replication overhead. As we will see later in the evaluation section, this provides good replication overhead even under dynamic server events.

In the second case, the algorithm does nothing else to increase the number of available servers. The edge/user arrival will take care of filling the new server with new users that in turn attract old users when edges are formed to them. This leads to an eventual load balancing of the masters across servers without enforcing movement operations. The only condition is that the OSN continues to grow.

Server removal: When a server is removed, whether intentionally or due to a failure, the algorithm re-allocates the $\frac{N}{M}$ master nodes hosted in that server to the remaining $M-1$ servers equally. The algorithm decides the server in which a slave replica is promoted to master, based on the ratio of its neighbors that already exist on that server. Thus, highly connected nodes, with potentially many replicas to be moved due to local data semantics, get to first choose the server they go to. The remaining nodes are placed wherever they fit, following simple water-filling strategy. As we will see in the evaluation section, this strategy ensures equal repartition of the failed masters while maintaining a small replication overhead.

5. MEASUREMENT DRIVEN EVALUATION

In this section, we evaluate the performance of SPAR in terms of replication overhead and replica movements (migrations).

5.1 Evaluation methodology

5.1.1 Metrics

Our main quantitative evaluation metric is the *replication overhead* r_o that stands for the number of slave replicas that need to be created to guarantee local semantics while observing the K-redundancy condition. We also analyze the number of node movements, i.e. replica migrations between servers during the operation of SPAR online.

5.1.2 Datasets

We use three different datasets, each serving a different purpose to evaluate individual performance metrics.

Twitter: We collected a dataset by crawling Twitter between Nov 25 - Dec 4, 2008. It comprises $2,408,534$ nodes and $48,776,888$ edges. It also contains 12M tweets generated by the 2.4M users during the time of the collection. Although there exists larger datasets [21], they are topic specific and do not contain all the tweets of individual users. To the best of our knowledge, this is the largest dataset with complete user activity. This data allows us to have a good approximation to what Twitter was as of Dec. 2008.

Facebook: We used a public dataset of the New Orleans Facebook network [34]. It includes nodes, friendship links, as well as wall posts and was collected between Dec 2008 and Jan 2009. The data consists of 60,290 nodes and 1,545,686 edges. This dataset includes edge creation timestamps between users by using the first wall post. This information, however, is not available for all users. Therefore, we filtered the dataset and retained a sub-network containing complete information for 59,297 nodes and 477,993 edges.

Orkut: Collected between Oct 3 and Nov 11 2006, this dataset consists of $3,072,441$ nodes and $223,534,301$ edges. Further information about this dataset can be found in [25] and is the largest of the three datasets.

5.1.3 Algorithms for Comparison

We compare SPAR against the following partitioning algorithms:

Random Partitioning: Key-Value stores like Cassandra, MongoDB, SimpleDB, *etc.* partition data randomly across servers. Random partition is the de-facto standard used in most commercial systems [30].

Graph Partitioning: There exist several offline algorithms for partitioning a graph into a fixed number of equal sized partitions in such a way that the number of inter-partition edges are minimized [10, 19]. We use METIS [19], which is known to be very fast and yields high quality partitions for large social graphs [23].

Modularity Optimization (MO+) Algorithms: We also consider an offline community detection algorithm [12] built around the modularity metric [27, 26]. We modified it in order to be able to create a fixed number of equal sized partitions [28]. Our modified version, called MO+, operates by grouping the communities in partition sequentially until a given partition is full. If a community is larger than the predefined size, we recursively apply MO [12] to the community.

5.1.4 Computation of results

The evaluation procedure is as follows:

- The input consists of a graph (one of the datasets described earlier), the number of desired partitions M and the desired minimum number of replicas per user's profile K.
- The partitions are produced by executing each of the algorithms on the input.
- Since we require local semantics, we process the partitions obtained by the candidate algorithms in the offline algorithms case. We then add replicas when the master of a user is missing some of its neighbors in the same partition.
- For Facebook, we generate the edge creation trace using the exact timestamps. For Twitter and Orkut, the timestamps are not available. So, we create random permutations of the edges in order to have an ordered edge creation trace. In the case of the exact timestamp and a random permutation of the edge give the same qualitative results, therefore, we can assume that the same applies for Orkut and Twiter. Furthermore, there is virtually no quantitative difference across multiple random permutations for any dataset.

5.2 Evaluation of replication overhead

Fig. 4 summarizes the replication overhead of the different algorithms over the different datasets for $K = 0$ and 2 as well as for different numbers of servers, from 4 up to 512. We see that SPAR online generates much smaller replication overhead than all the other algorithms, including traditional offline graph partitioning (METIS) and community detection algorithms (MO+). The relative ranking of algorithms from best to worse is, SPAR, MO+, METIS, and Random. Looking at the absolute value of the replication overhead, we see that it increases sub-linearly with the number of servers – note that the x-axis is logarithmic. This means that adding more servers does not lead to a proportional increase in per-server resource requirement (as given by r_o).

SPAR naturally achieves a low replication overhead since this is the optimization objective. The competing algorithms optimize for minimal inter-server edges. Therefore, they end up needing more replicas to add the missing nodes for guaranteeing local semantics in each partition in the post-processing step.

Low-Cost Redundancy: An important property of SPAR is that *it achieves local semantics at a discounted cost by reusing replicas that are needed for redundancy anyway (or the other way around).* To see this, lets focus on the example of 32 servers. If no fault tolerance is guaranteed ($K = 0$), then the replication overhead of SPAR to ensure local semantics is 1.72. If we require its profile to be replicated at least $K = 2$ times, then the new replication overhead is 2.76 (rather than 3.72). Out of the 2.76 copies, 2 are inevitable due to the redundancy requirement. Thus, the local semantics were achieved at a lower cost 2.76-2=0.76 instead of 1.72. The gain comes from leveraging the redundant replicas to achieve locality, something only SPAR does explicitly.

We now focus our comparison on Random vs SPAR, since Random is the de-facto standard. In Fig. 5, we depict the ratio between the overhead of Random and that of SPAR. In the case of the Twitter dataset, we see improvements varying from 12% for a partition of 4 servers – it is low because we have $K = 2$, which means 3 replicas of each node to be

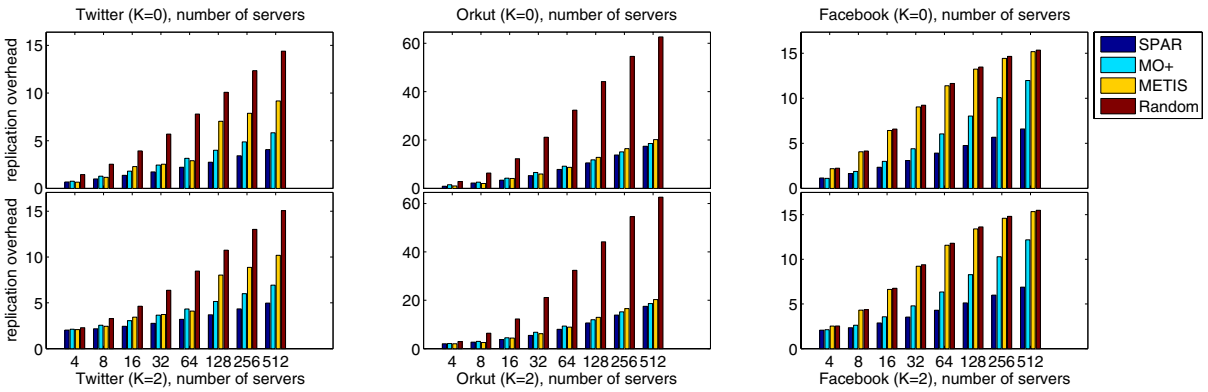

Figure 4: Replication overhead (r_o) for SPAR, METIS, MO+ and Random for Twitter, Orkut and Facebook. The upper graph show the results for $K = 0$, the lower graph shows the results for $K = 2$, K is the number of replicas for redundancy.

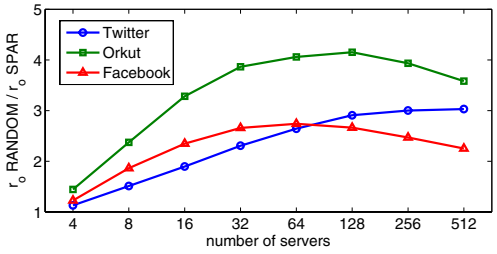

Figure 5: SPAR versus Random for $K = 2$

placed in 4 servers – to 200% in the case of 512 servers. For Orkut and Facebook the improvement ranges from 22% and 44% to 174% and 315% respectively. For large number of servers, the ratio starts decreasing. However, this is likely not representative of what would happen in a real system, since the number of replicas per server is artificially small (given the number of users and the number of servers).

Such small overheads allow SPAR to achieve a much higher throughput, as we will demonstrate in Sec.7 by running a real implementation of SPAR on top of Cassandra and MySQL.

5.3 Dynamic operations and SPAR

So far, we have shown that SPAR online outperforms existing solutions when measuring replication overhead. We now turn to other system requirements stated in Sec.3.1.

5.3.1 *The Delicate Art of Balancing Loads*

We first focus on how replicas are distributed across users. We take the example of Twitter with 128 servers and K=2. In this case, the average replication overhead is 3.69: 75.8% of the users have 3 replicas, 90% of the users have 7 or less replicas and 99% of the users have 31 or less replicas. Out of the 2.4M users, only 139 need to be replicated across the 128 servers. Next we look at the impact of such replication distribution on read and write operations.

Reads and Writes: Read operations are only conducted on masters. Therefore, we want to analyze whether the aggregate read load of servers, due to read of the master by their users, is balanced. This load depends on the distribution of master among servers, and on the read patterns of the users. SPAR online yields a partition in which the

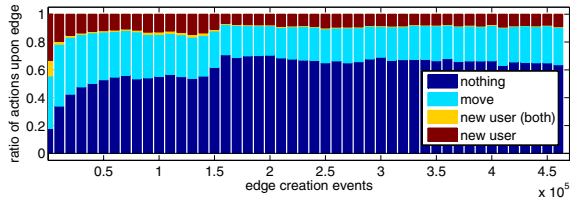

Figure 6: Ratio of actions performed by SPAR as edge creation events occur for Facebook

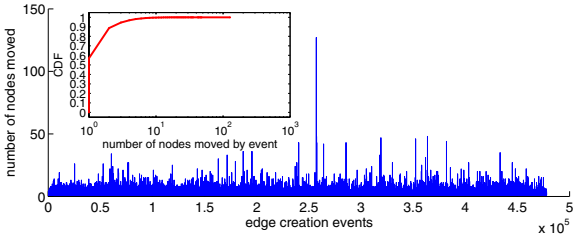

Figure 7: Number of movements per edge creation event for Facebook

coefficient of variation (COV) of masters is 0.0019. Thus, our online heuristic is successful in balancing masters across servers. To test against workload imbalances, we examine write patterns. The COV of writes per server is 0.37, which indicates that they are fairly balanced across all servers and no single server is the source of a high proportion of writes. We expect reads to be even more balanced across users: systems such as Twitter handle reads automatically through API calls via periodic polling (90% of Twitter traffic is generated via its API). Further, there is a very low correlation between heavy-writers and the number of their slave replicas. So, they do not present a big problem for the system. We have therefore shown that SPAR handles both writes and reads well in terms of balancing them across servers as per requirements.

5.3.2 *Moving Nodes Around*

Next, we turn our attention to the footprint of SPAR in terms of user's data movements (migrations). In Fig. 6, we show a stacked bin time-series of action taken by SPAR on-

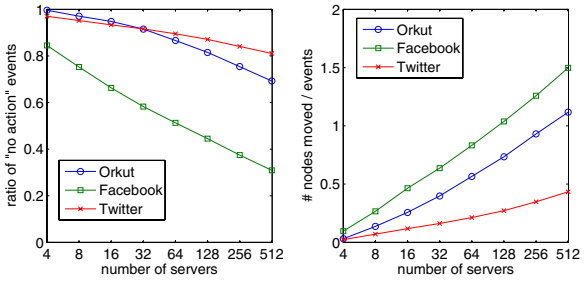

Figure 8: Movement costs for various datasets.

line upon each edge arrival event for the Facebook dataset with $K = 2$ and 16 servers. We see that following a transient phase during which the network builds up, SPAR online enters a steady-state phase in which 60% of edge arrivals do not create any movements. In the remaining 40% of the cases, data gets moved.

In Fig. 7, we plot the number of transmissions per edge arrival, with its CDF as an inset. We see that whenever a movement occurs, in an overwhelming majority (90%) of cases, it involves the data of only two users or less. The largest movement involves moving data of 130 users (nodes).

Fig. 8 summarizes the movement costs on the system for Facebook, Orkut and Twitter from 4 servers to 512. The left plot in Fig. 8 depicts the average fraction of *do nothing* actions that involve no movement of nodes (users' data), discounted the transient phase. The right plot in Fig. 8 depicts the total of number of movements divided by the number of edges. These figures show that the footprint remains low for all configurations of datasets and number of servers.

5.3.3 Adding/Removing Servers

Adding servers: When a server is added, we can use one of two policies: (1) wait for new arrivals to fill up the server, or (2) re-distribute existing masters from other servers into the new server. We will study the overhead of such policies in terms of replication costs.

We start with the first case, where we go from 16 to 32 servers by adding one server every 150K new users. This strategy yields a marginal increase of r_o (2.78), compared to the r_o (2.74) that we would have obtained if the system was dimensioned for 32 servers from the beginning. This shows that SPAR is able to achieve an efficient state independently of how servers are added. Although it does not do an explicit re-distribution, the COV of the number of masters per server remains very low at the end on the trace (0.004). This shows that we can gracefully add new servers without extra overhead.

In the second experiment, we tested the effect of an extensive upgrade of the infrastructure. We double the number of servers at once and force re-distribution of the masters. This reduces the number of masters per server by half while being load-balanced. We tested the addition of 16 servers in two cases, first after 50% of the edge creation trace is replayed, and second after 100%. Doubling the number of servers leads to the expected transmission of half the masters. SPAR, however, needs to move additional slaves to maintain local semantics. Adding a server also causes an increase in the final replication overhead. For instance, doubling the initial 16 servers at 50% of the trace produces a transient increase in the replication overhead by 10% . This overhead

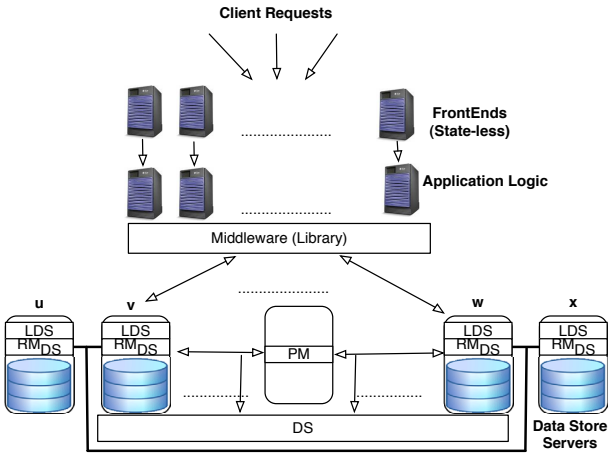

Figure 9: Sketch of SPAR's architecture

is progressively reduced as new edges are added, reaching 2.82, only a 2% higher than if we started with 32 servers. With proactive reshuffling of edges (replay old edges as if new), described in Sec. 4, the additional overhead caused by addition of servers on-the-fly becomes almost insignificant, although the number of movements is higher due to the proactive reshuffling.

Removing servers: Next, we test what happens when a server is removed. The average number of movements is 485K, with a marginal increase in r_o from 2.74 to 2.87. We can further reduce r_o to 2.77, at the cost of additional 180K transmissions, if we replay the edges of the nodes affected by the server removal. One might argue that server removal seldom happens as systems usually do not scaled down. The case of server removals due to failures is discussed in Sec. 6.2.

Overall, we have demonstrated that SPAR is able to distribute load efficiently, and cope with both social network graph dynamics as well as physical (or virtual) machine dynamics with low overhead. Next, we describe the SPAR system architecture.

6. SPAR SYSTEM ARCHITECTURE

We now describe the basic architecture and operations of SPAR. Fig. 9 depicts how SPAR integrates into a typical three-tier web architecture. The only interaction between the application and SPAR is through the Middle-ware (MW). The MW needs to be called by the application to know the address of the back-end server that contains the user's data to be read or written. Once the application has obtained the address, it uses the common interface of any data-store e.g. a MySQL driver, Cassandra's API, *etc.* The application logic can be written as if centralized since it is agnostic to the number of back-end servers, how to operate them and how to scale out. Making those operations scalable and transparent is the task of the remaining components of SPAR: the Directory Service DS, the Local Directory Service LDS, the Partition Manager PM and the Replication Manager RM.

6.1 System Operations

Data Distribution: The data distribution is handled by the DS, which returns the server that hosts the master

replica of a user through a key table look-up: $H_m(key) \rightarrow u$. Additionally, the DS also resolves the list of all servers where the user has replicas in: $H_s(i) \rightarrow \{u, v, ..., w\}$. The Directory Service is implemented as a DHT with consistent caching and is distributed across all servers in the data back end. The Local Directory Service (LDS) contains a partial view of the DS. Specifically, $\frac{N(1+r_o)}{M}$ of the look-up table H_m and $\frac{N}{M}$ of look-up table H_s. The LDS only acts as cache of the DS for performance reasons and is invalidated by the DS whenever the location of a replica changes.

Data Partitioning: The PM runs the SPAR algorithm described in Sec. 4 and performs the following functions: (i) map the user's key to its replicas, whether master or slaves, (ii) schedule the movement of replicas, and (iii) re-distribute replicas in the event of server addition or removal. SPAR algorithm would allow the PM to be distributed. However, for simplicity, we implemented a centralized version of PM and run multiple mirrors that act on the failure of the main PM to avoid single point of failure[2]. PM is the only component that can update the DS. Note that this simplifies the requirements to guarantee global consistency on the DS because only the main PM can write to it.

Data Movements: Data movements (migrations of replicas) takes place when a replica needs to be moved (migrated) from one server to another. After a movement, all replicas undergo reconciliation to avoid inconsistencies that could have arisen during the movement, i.e. the user writing data while its master changes its location. The same applies when a replica is scheduled to be removed. We do not propose a new mechanism to handle such reconciliation events, but instead rely on the semantic reconciliation based on versioning proposed in other distributed systems (e.g. Dynamo [13]).

Data Replication and Consistency: The Replication Manager RM runs on each server of the data back-end. The main responsibility of RM is to propagate the writes that take place on a user's master to all her slaves using an eventual consistency model, which guarantees that all the replicas will – with time – be in sync. Note that since SPAR relies on a single master and multiple slaves configuration. This avoids the inconsistencies arising from maintaining multiple masters. A single master has the additional benefit that inconsistencies can only arise due to failures or due to movements produced by edge, node or server arrivals and not as part of the regular operations of read and writes. Note that the RM is not replacing the data-store, but different data-stores need different implementations of the RM to adapt to its interface. In this first iteration of SPAR we provide the implementation of RM for MySQL and Cassandra although other data-stores could be equally supported (i.e. Postgres, Memcached, MongoDB *etc.*).

Adding and Removing Servers: The PM also controls the addition and/or removal of servers into the back-end cluster on demand. This process is described in Sec. 4 and evaluated in 5.

Handling Failures: Failure on the servers running the data back-end are bond to happen, either because of server specific failures, e.g. disk, PSU failure, etc. or because of failures at the data-center level, e.g. power outages, network issues, etc.

SPAR relies on a *heartbeat*-like system to monitor the health of the data back-end servers. When a failure is detected the PM decides the course of action based on the failure management policy set by the administrator. We consider two types of policies, one that reacts to *transient* failures and another one to *permanent* failures.

A permanent failure is treated as a server removal. In this case, slave replicas of the master that went down are promoted to masters and all their neighbors are recreated. For more details see Section 4 and 5.

For transient failure events (short lived), one potential option is to promote one of the slave replicas whose master went down without triggering the recreation of her neighbors. Consequently, the *local data semantics* is temporally sacrificed. In this case, the system would only suffer a graceful degradation since we can leverage a nice property of SPAR. This property entails that the server hosting a slave replica of one of the failed masters will also contain a large portion of the neighbors of such master. Therefore, while the promotion of a slave does not guarantee local data semantics, it does still provide access to most of the user's neighborhood. To better illustrate this point let us take the example of the Twitter dataset for 16 partitions. In this case, 65% of users have more than 90% of their direct neighbors present in the server that hosts the best possible slave replica, which is the candidate to be promoted in the case of failure. This solution should only be applied for extremely short lived outages. Otherwise, the user experience of the OSN would suffer and the system administrator would be better off implementing the permanent failure scenario.

Current SPAR replication is not meant for *high-availability* but for redundancy and to guarantee local data semantics. However, high-availability in SPAR could be obtained in two ways: one is to modify the formulation of the problem so that there are at least K' master replicas where the *local data semantics* is maintained while minimizing MIN_REPLICA. This approach, however, is left for future work. The other option is a simple brute force approach that mirrors the SPAR system under $K = 0$ as many times as desired.[3]

6.2 Implementation Details

The RM sits on top of a data-store and controls and modifies its operations. The read events (*e.g.* selects) are forwarded directly to the data-store without delay. However, the write events (*e.g.*updates, inserts, deletes) need to be analyzed and can be altered both for replication and performance reasons.

Let us illustrate the inner workings of the RM with an example of a write operation in MySQL. A user i wants to create a new event w, which generates the following command to be inserted in MySQL: *insert into event(i_{id},w_{id},w)* , where i_{id} and w_{id} are the pointers to the user and the event's content. The RM will react to this command by obtaining the target table *event*. The RM knows, through simple configuration rules, that the *event* table is partitioned, and thus the *insert* needs to be replayed in all servers hosting the slave replicas of user i_{id}. The RM queries its LDS to obtain the list of servers to propagate the insert, and issue the same *insert* command to the local MySQL. Addition-

[2]The current centralized version of PM can handle 52 edge creations per second in a commodity server.

[3]For instance, in the case of Twitter for 128 server, we could obtain via mirroring three master replicas (with full local semantics) with a replication overhead r_o of 6.63. The one master plus two slave replicas configuration ($K = 2$) would result in a r_o of 3.20.

ally, this event will be broadcast to all the neighbors of the user. For each contact j of i the application will generate a *insert into event-inbox(i_{id},j_{id},w_{id})* and the RM_{mysql} will replay the same command to all the appropriate servers.

7. SPAR IN THE WILD: EVALUATION

In this section, we study the performance of SPAR on two data-stores: MySQL (a traditional RDBMS) and Cassandra (a Key-Value used in production). More specifically, we compare SPAR against random partitioning (for Cassandra) and full replication (for MySQL).

As the reference OSN application to be scaled we use an open-source Twitter-clone called Statusnet [5], which relies on centralized architecture (PHP and MySQL/Postgres).

Our testbed consists of a cluster of 16 low-end commodity servers: the so-called "little engine(s)". These servers are interconnected by a Gigabit-Ethernet switch. Each server has a Pentium Duo CPU at 2.33GHz with 2GB of RAM and a single hard drive. The data of what Twitter was as of end of 2008 (Sec. 5.1.2) is loaded in the data-stores.

7.1 Evaluation with Cassandra

Statusnet is designed to run on top of MySQL or Postgres. Therefore, to evaluate SPAR with Cassandra we need to reproduce the functionality on Statusnet for the data model specific of Cassandra (version 0.5.0). We define a data scheme that contains information about users, updates (tweets) and the list of update streams that the users subscribe to. We implement the data scheme using different *columns* and *super columns*.

To implement SPAR on top of Cassandra, first, we disable the default random partitioner of Cassandra by creating isolated independent instances of Cassandra. The Cassandra nodes in our system do not communicate with each other. So, we have full control of the data placement (users and updates). We implement the Directory Service DS on top of the same Cassandra nodes showing that we can piggyback on the underlying data-store infrastructure. The DS is distributed across all servers to avoid bottlenecks.

The SPAR middle-ware for Cassandra is written using a Thrift interface. We describe what the middle-ware does for the canonical operation of Statusnet; retrieving the last 20 updates (tweets) for a given user. The middle-ware performs three operations: 1) randomly select a Directory Service node and request the location of the master replica of the user by using the *get* primitive of Cassandra. 2) connect to the node that hosts the master and perform a *get-slice* operation to request the *update-id*s of the list of the 20 status updates to be shown to the user, and finally 3) do a *multi-get* to retrieve the content of all the status updates and return to Statusnet.

As noted earlier in our evaluation we compare the performance of Statusnet with the SPAR instantiation on Cassandra and the standard vanilla Cassandra (with random partition). We are interested in answering the following two question questions: (i) What impact does SPAR have on the response time as compared to random partitioning? and (ii) How much does SPAR reduce network traffic?

To answer these two questions, we perform the following set of experiments. We randomly select 40K users out of the Twitter dataset and issue requests to retrieve the last 20 status updates at a rate of 100, 200, 400 and 800 requests per second. Note that this requests are not primitive *get/set* operations on Cassandra but application level requests – a user getting the status updates from her friends.

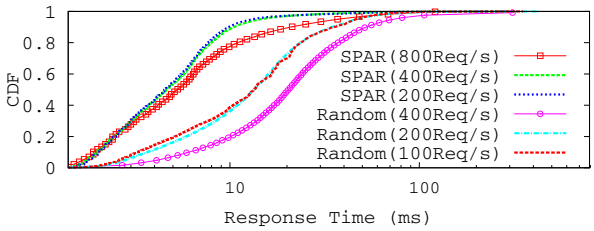

Figure 10: Response time of SPAR on top of Cassandra.

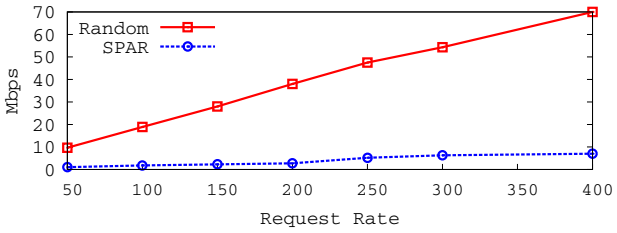

Figure 11: Network activity in SPAR on top of Cassandra.

Examining Response Times: Fig. 10 shows the response time of SPAR and the vanilla Cassandra using random partitioning. We can see that SPAR reduces the average response time by 77% (400 requests/second). However, what it is most interesting is the throughput in the aggregate number of request per second given a realistic quality of service. SPAR can support 800 req/s with the 99th percentile response time below 100ms. Cassandra with random partitioning can only support 1/4 of the request rate with the same quality of service, about 200 req/s.

Why does SPAR outperform the random partitioning of Cassandra? There are multiple reasons and we discuss each one in brief. First, Cassandra is affected by the delay of the worst performing server. This is due to the heavy inter-server traffic for remote reads.Our setting runs on Gigabit Ethernet switches without background traffic and hence network bandwidth is not a bottleneck, as Fig. 11 also shows. The commodity servers, however, often hit network I/O and CPU bottlenecks in trying to sustain high rates of remote reads. With SPAR, as all relevant data is local by design, remote reads are not necessary. A second and less obvious reason for the better performance of SPAR has to do with its improved memory hit ratio that comes as a byproduct of the non-random partitioning of masters. Indeed, a read for a user brings in memory from the disk the data of the user as well as those of her friends. Given that masters of her friends are on the same server with high likelihood, and that reads are directed to the masters, there is a good chance that a read for one of these friend will find most of the required data already in memory because of previous reads. The random partitioning scheme of Cassandra destroys such correlations and thus suffers from a lower memory hit ratio and the consequent disk I/O penalties.

Analyzing the Network Load: Fig. 11 depicts the aggregate network activity of the cluster under various request rates. For random partitioning, requests are spread among multiple nodes and *multi-get* operations significantly increases the network load. Compared to a vanilla Cassan-

dra implementation, SPAR reduces the network traffic by a factor of 8 (for 400 reqs/sec).

7.2 Evaluation with MySQL

We now turn our attention to MySQL, a traditional RDBM system. The first question we want to answer is: can SPAR scale an OSN application using MySQL? This is important as it allows developers to continue using the familiar RDBMS framework without worrying about scaling issues. Specifically, we want to answer if we can make Statusnet deal with the demand of Twitter as of Dec. 2008.

We use MySQL version 5.5 together with the SQL data scheme that is provided by Statusnet [5]. The schema contains SQL tables related to the users (table *user* and *profile*), the social graph (*subscription*), updates (*notice*) and the list of updates per user (*notice_inbox*). We adapted our Twitter dataset to the Statusnet data scheme, so that it contains all information about users and status updates. We retrieve the last 20 status updates (tweets) for a given user by performing a single query using a join on the *notice* and *notice_inbox* tables.

To stress-test our setup we use *Tsung* [6] and two servers that we use to emulate the activity for thousands of concurrent users. We generate both application read requests (retrieve the last 20 status updates for the user) and application write requests (a user generates a new status update and updates the inboxes for all her friends). Our experimental evaluation consists of multiple 4 minute sessions where we query for the last status updates of a random subset of users with a constant request rate. We make sure that every user is queried only once per session, and that the requests are spread evenly among servers.

Comparison to Full Replication: First, we check if a scheme based on *Full* Replication can work in practice. This would mean loading the entire Twitter dataset on all 16 servers and measuring the number of users that the system can serve. The average 95th percentile of the response time is 113ms for 16 req/s (1 request per second per machine), 151ms for 160 req/s, and 245ms for 320 req/s. As expected, the 99th percentiles are even higher with 152ms for 16 req/s. On the other hand, when we use SPAR, the cluster can serve more than $2,500$ req/s with a 99th percentile of less than 150ms. This shows that SPAR using a MySQL data store is able to withstand Twitter-scale read loads with a small cluster of commodity machines, whereas a full replication system cannot cope.

Note that this experiment shows that the same centralized code of Statusnet that knows nothing about distributed systems can still support Twitter-scale loads when using SPAR.

Adding Writes: To further stress-test the system, we introduce insertions of updates. We evaluate the effect of the insertion of 16 updates/s (1 update/s per machine). In this part we evaluate only SPAR (with MySQL), as full replication using MySQL performs very poorly. Note that the insertion of a new status update to the system can generate thousands of updates, since the system needs to insert to the *notice_inbox* table one entry for every user that should receive the status update (to all the followers in Twitter terminology). How we treat these inserts is crucial for the overall performance. A naive implementation, that performs single or multi inserts into the *notice_inbox* can completely saturate the system.

We group the inserts and control the rate at which we introduce them in the system. Under this scenario, we show that we can achieve a 95th percentile response time below 260ms for 50 read req/s and below 380ms for 200 read req/s while a constant rate of 16 updates/s. We should note here that the median response time in both cases is very low, around 2ms. Performance will only get better as we add more machines.

In this section, we have shown that using SPAR leads to high throughput (reqs/sec) and better scalability when compared to full replication solutions and random partitioning, at the cost a very modest replication overhead.

8. RELATED WORK

To the best of our knowledge this is the first work to address the problem of scalability of the data back-end for OSNs. In this section we compare and contrast the approach presented in this paper (and our prior work in the area [28, 29]) with related work in the literature.

Scaling Out: Scaling out web applications is one of the key features offered by Cloud providers such as Amazon EC2 and Google AppEngine. They allow developers to effortlessly add more computing resources on demand or depending on the current load of the application [4]. This scaling out however, is only transparent as long as the application is stateless. This is the case of typical Web front-end layer, or when the data back-end can be partitioned into independent components. We deal with scaling of the application back-end when data is *not* independent as in the case of OSNs by providing means to ensure local semantics at the data level.

Key-Value Stores: Many popular OSNs today rely on DHTs and Key-Value stores to deal with the scaling problems of the data back-end (e.g. Twitter is migrating to Cassandra). While Key-Value stores do not suffer from scalability due to their distributed design, they rely on random partition of the data across the back-end server. This can can lead to poor performance in the case of OSN workloads (as we show in this paper). SPAR improves performance multi-fold over Key-Value stores as it minimizes network I/O by keeping all relevant data for a given request local to the server. Keeping data local helps prevent issues like the 'multi-get' hole observed in the typical operation of Key-Value stores[1].

Distributed File Systems and Databases: Distributing data for the sake of performance, availability and resilience has been widely studied in the file system and database systems community. Ficus [15] and Coda [31] are distributed file systems that replicate files for high availability. Farsite is a distributed file system that achieves high availability and scalability using replication [8]. Distributed RDBMS systems like MySQL cluster and Bayou [33] allow for disconnected operations and provide eventual data consistency. SPAR takes a different approach as it does not distribute data, but maintains it locally via replication. This approach is more suitable for OSNs since their typical operation requires fetching data from multiples servers in a regular basis.

9. CONCLUSIONS

Scaling OSNs is a hard problem because the data of users is highly interconnected, and hence, cannot be subjected to a clean partition. We present SPAR, a system based on partitioning of the OSN social graph combined with a user-level replication so that the local data semantics for all users

is guaranteed. By local semantics, we mean that all relevant data of the direct neighbors of a user is co-located in the same server hosting the user. This enables queries to be resolved locally on a server, and consequently, breaks the dependency between users that makes scalability of OSNs so problematic.

Preserving local semantics has many benefits. First, it enables transparent scaling of the OSN at a low cost. Second, the performance benefit in throughput (requests per second) served increases multifold as all relevant data is local and network I/O is avoided. Third, network traffic is sharply reduced.

We designed and validated SPAR using real datasets from three different OSNs. We showed that replication overhead needed to achieve local data semantics is low using SPAR. We also demonstrated that SPAR can deal with the dynamics experienced by an OSN gracefully. Further, we implemented a Twitter-like application and evaluated SPAR on top of a RDBMS (MySQL) and a Key-Value store (Cassandra) using real traces from Twitter. We showed that SPAR offers significant gains in throughput (req/s) while reducing network traffic.

Acknowledgments

We thank the anonymous reviewers and our shepherd Yin Zhang for their valuable feedback. Special thanks to Evan Weaver, Zografoula Vagena and Ravi Sundaram for their early comments and feedback.

10. REFERENCES

[1] Facebook's memcached multiget hole: More machines != more capacity. http://highscalability.com/blog/2009/10/26/facebooks-memcached-multiget-hole-more-machines-more-capacit.html.

[2] Friendster lost lead due to failure to scale. http://highscalability.com/blog/2007/11/13/friendster-lost-lead-because-of-a-failure-to-scale.html.

[3] Notes from scaling mysql up or out. http://venublog.com/2008/04/16/notes-from-scaling-mysql-up-or-out/.

[4] Rightscale. http://www.rightscale.com.

[5] Status net. http://status.net.

[6] Tsung: Distributed load testing tool. http://tsung.erlang-projects.org/.

[7] Twitter architecture. http://highscalability.com/scaling-twitter-making-twitter-10000-percent-faster.

[8] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, Jon, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *OSDI 02*.

[9] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28.

[10] S. Arora, S. Rao, and U. Vazirani. Expander flows, geometric embeddings and graph partitioning. *J. ACM*, 56(2):1–37, 2009.

[11] F. Benevenuto, T. Rodrigues, M. Cha, and V. Almeida. Characterizing user behavior in online social networks. In *Proc. of IMC '09*, pages 49–62, New York, NY, USA, 2009. ACM.

[12] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *J.STAT.MECH.*, page P10008, 2008.

[13] 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.*, 41(6):205–220, 2007.

[14] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[15] R. G. Guy, J. S. Heidemann, W. Mak, T. W. Page, Jr., G. J. Popek, and D. Rothmeier. Implementation of the Ficus replicated file system. In *USENIX Conference Proceedings, 1990*.

[16] J. Hamilton. Geo-replication at facebook. http://perspectives.mvdirona.com/2008/08/21/GeoReplicationAtFacebook.aspx.

[17] J. Hamilton. Scaling linkedin. http://perspectives.mvdirona.com/2008/06/08/ScalingLinkedIn.aspx.

[18] HighScalability.com. Why are facebook, digg and twitter so hard to scale? http://highscalability.com/blog/2009/10/13 /why-are-facebook-digg-and-twitter-so-hard-to-scale.html.

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

[20] H. Kwak, Y. Choi, Y.-H. Eom, H. Jeong, and S. Moon. Mining communities in networks: a solution for consistency and its evaluation. In *ACM IMC '09*.

[21] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? 2010.

[22] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Transactions on KDD*, 1:1, 2007.

[23] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *CoRR*, abs/0810.1355, 2008.

[24] N. Media. Growth of twitter. http://blog.nielsen.com/nielsenwire/online_mobile/twitters-tweet-smell-of-success/.

[25] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. In *ACM IMC '07*.

[26] M. Newman and J. Park. Why social networks are different from other types of networks. *Phys. Rev. E*, 68:036122, 2003.

[27] M. E. J. Newman. Modularity and community structure in networks. *PNAS*, 103:8577, 2006.

[28] J. M. Pujol, V. Erramilli, and P. Rodriguez. Divide and conquer: Partitioning online social networks. http://arxiv.org/abs/0905.4918v1, 2009.

[29] J. M. Pujol, G. Siganos, V. Erramilli, and P. Rodriguez. Scaling online social networks without pains. In *Proc of NETDB*, 2009.

[30] J. Rothschild. High performance at massive scale - lessons learned at facebook. http://cns.ucsd.edu/lecturearchive09.shtml#Roth.

[31] M. Satyanarayanan. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39:447–459, 1990.

[32] F. Schneider, A. Feldmann, B. Krishnamurthy, and W. Willinger. Understanding online social network usage from a network perspective. In *IMC '09*.

[33] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *ACM SOSP '95*.

[34] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi. On the evolution of user interaction in facebook. In *Proc of WOSN'09)*.