DynaSoRe: Efficient In-Memory Store for Social Applications

Xiao Bai, Arnaud Jégou, Flavio P. Junqueira, Vincent Leroy

DynaSoRe: Efficient In-Memory Store for Social Applications

Xiao Bai[1], Arnaud Jégou[2], Flavio Junqueira[3], and Vincent Leroy[4]

[1] Yahoo! Research Barcelona xbai@yahoo-inc.com
[2] INRIA Rennes arnaud.jegou@inria.fr
[3] Microsoft Research Cambridge fpj@apache.org
[4] University of Grenoble - CNRS vincent.leroy@imag.fr

Abstract. Social network applications are inherently interactive, creating a requirement for processing user requests fast. To enable fast responses to user requests, social network applications typically rely on large banks of cache servers to hold and serve most of their content from the cache. In this work, we present *DynaSoRe*: a memory cache system for social network applications that optimizes data locality while placing user views across the system. DynaSoRe storage servers monitor access traffic and bring data frequently accessed together closer in the system to reduce the processing load across cache servers and network devices. Our simulation results considering realistic data center topologies show that DynaSoRe is able to adapt to traffic changes, increase data locality, and balance the load across the system. The traffic handled by the top tier of the network connecting servers drops by 94% compared to a static assignment of views to cache servers while requiring only 30% additional memory capacity compared to the whole volume of cached data.

1 Introduction

Social networking is prevalent in current Web applications. Facebook, Twitter, Flickr and Github are successful examples of social networking services that allow users to establish connections with other users and share content, such as status updates (Facebook), micro-blogs (Twitter), pictures (Flickr) and code (Github). Since the type of content produced across application might differ, we use in this work the term *event* to denote any content produced by a user of a social networking application. For this work, the format of events is not important and we consider each event as an application-specific array of bytes.

A common application of social networking consists of returning the latest events produced by the connections of a user in response to a read request. Given the online and interactive nature of such an application, it is critical to respond to user requests fast. Therefore, systems typically use an in-memory store to maintain events and serve requests to avoid accessing a persistent, often slower backend store. Events can be stored in the in-memory store in the form of materialized views. A view can be producer-pivoted and store the events produced by a given user, or it can be consumer-pivoted and store the events to be consumed by a given user (*e.g.*, the latest events produced by a user's connections) [16]. We only consider *producer-pivoted views* in this work.

When designing systems to serve online social applications, scalability and elasticity are critical properties to cope with a growing user population and an increasing demand of existing users. For example, Facebook has over 1 billion registered and active users. Serving such a large population requires a careful planning for provisioning and analysis of resource utilization. In particular, load imbalance and hotspots in the system may lead to severe performance degradation and a sharp drop of user satisfaction. To avoid load imbalances and hotspots, one viable design choice is to equip the system with a mechanism that enables it to dynamically adapt to changes of the workload.

A common way to achieve load balancing is to randomly place the views of users across the servers of a system. This however incurs high inter-server traffic to serve read requests since the views in a user's social network have to be read from a large amount of servers. SPAR is a seminal work that replicates all the views in a user's social network on the same server to implement highly efficient read operations [15]. Yet, this results in expensive write operations to update the large amount of replicated views. Moreover, SPAR assumes no bounds on the degree of replication for any given view, which is not practical since the memory capacity of each individual server is limited. Consequently, it is very important to make efficient utilization of resources.

In this work, we present *DynaSoRe* (Dynamic Social stoRe), an efficient in-memory store for online social networking applications that dynamically adapts to changes of the workload to keep the network traffic across the system low. We assume that a deployment of DynaSoRe comprises a large number of servers, tens to hundreds, spanning multiple racks in a data center. DynaSoRe servers create replicas of views to increase data locality and reduce communication across different parts of data centers. We assume a realistic tree structure for the networking substrate connecting the servers and aim to reduce traffic at network devices higher up in the network tree.

Our simulation results show that with 30% additional memory (to replicate the views), DynaSoRe reduces the traffic going through the top switch by 94% compared to a random assignment of views and by 90% compared to SPAR. With 100% additional memory, DynaSoRe only incurs 2% of the total traffic with the random assignment, significantly outperforming SPAR which incurs 35% of the traffic with the random assignment.

Contributions. In this paper, we make three main contributions:

- We propose *DynaSoRe*, an efficient in-memory store for online social applications, which dynamically adapts to changes of the workload to keep the network traffic across the system low.
- We provide simulation results showing that DynaSoRe outperforms our baselines, random assignment and SPAR, and that it is able to reduce the amount of network traffic across the system.
- We show that DynaSoRe is especially efficient compared to our baselines when assuming a memory budget, which is an important practical goal due to cost of rack space in modern data centers.

Roadmap. The remainder of this paper is structured as follows. We specify in Section 2 the model and requirements of an efficient in-memory store. We present the design of DynaSoRe in Section 3 and evaluate it in Section 4. We discuss related work in Section 5 and present our conclusions in Section 6.

2 Problem Statement

2.1 System model

DynaSoRe is a scalable and efficient distributed in-memory store for online social applications that enable users to produce and consume events. We assume that the social network is given and that it changes over time. The events users produce are organized into *views*, and the views are producer-pivoted (contain the events produced by a single user). A view is a list of events, possibly ordered by timestamps. DynaSoRe supports `read` and `write` operations. A `write` request from user u of an event e writes e to the view of u. A `read` request from user u reads the views of all connections in the social network of u. This closely follows the Twitter API. According to Twitter, status feeds represent by far the majority of the queries received[1]. Consequently, the benefits of DynaSoRe (that are only measured with the read/write operations) are significant even in a complete social application that also supports other kinds of operations.

DynaSoRe spans multiple servers, as the views of all users cannot fit into the memory of a single server. Distributing the workload across servers is critical for scalability. Our applications reside in data centers. Servers inside a data center are typically organized in a three-level tree of switches, which has a core tier at the root of the tree, an intermediate tier, and an edge tier at the leaves of the tree [1,7,8] as shown in Figure 1. The core tier consists of the top-level switch (ST), which connects multiple intermediate switches. The intermediate tier consists of intermediate switches (SI) and each of them connects a subset of racks. The edge tier consists of racks and each rack is formed by of a set of servers connected by a rack switch (SR). The network devices, *i.e.*, switches at different levels of the tree architecture, only forward network traffic. The views of users are maintained in the servers (S), connected directly to rack switches. The servers have a bounded memory capacity and we established its capacity by the number of views it can host. We use b to denote the number of bytes we use for a view. Brokers (B) are also servers connecting directly to rack switches and they are in charge of reading and writing views on the different servers of the data center.

Note that DynaSoRe could be deployed on several data centers by adding a virtual switch representing communications between data centers, which would then be minimized by DynaSoRe . In practice, Web companies such as Facebook do not have applications deployed across data centers. Instead, they replicate the content of each data center through a master/slave mechanism[13]. Thus in this work we focus on the case of a single data center.

In the system, events are organized in views and stored as key-value pairs. Each key is a user id and the value is the user view comprising events the

[1] http://www.infoq.com/presentations/Twitter-Timeline-Scalability

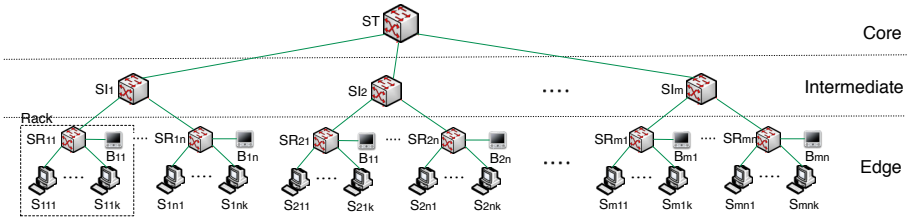


Fig. 1. System architecture of DynaSoRe.

user has produced. This memory store is back-ended by a persistent store that ensures data availability in the case of server crashes or graceful shutdowns for maintenance purposes. We focus in this work on the design of the memory store and a detailed discussion of the design of the persistent store is out of scope.

2.2 Requirements

To provide scalable and efficient `write` and `read` operations, DynaSoRe provides the following properties: locality, dynamic replication, and durability.

Locality. A user can efficiently read or write to a view if the network distance between the server executing the operation and the server storing the view is short. As one of our goals is to reduce the overall network traffic, we define the network distance between two servers to be the number of network devices (*e.g.*, switches) in the network path connecting them. DynaSoRe ideally ensures that all the views related to a user (*i.e.*, her own view and the views of her social connections) are placed close to each other according to network distance so that all requests can be executed efficiently. This requires flexibility with respect to selecting the server that executes the requests and the servers storing the views.

Dynamic replication. Online social networks are highly dynamic. The structure of the social network evolves as users add and remove social connections. User traffic can be irregular, with different daily usage patterns and flash events generating a spike of activity. Adapting to the behavior of users requires a mechanism to dynamically react to such changes and adjust the storage policy of the views impacted, for both number and placement of replicas. One goal for DynaSoRe is to trace user activity to enable an efficient utilization of its memory budget through accurate choices for the number and placement of replicas. Such a replication policy needs also to consider load balancing and to satisfy the capacity constraints of each server.

Durability and crash tolerance. We assume that servers can crash. Missing updates because of crashes, however, is highly undesirable, so we guarantee that updates to the system are durable. To do this, we rely upon a persistent store that works independently of DynaSoRe. Updates to the data are persisted before they are written to DynaSoRe to guarantee that they can be recovered in

the presence of faulty DynaSoRe servers. Since we replicate some views in our system, copies of data might be readily available even if a server crashes. In the case of a single replica, we need to fetch data from the persistent store to build it. In both cases, single or multiple replicas, crashes additionally require live servers to dynamically adjust the number of replicas of views to the new configuration.

2.3 Problem formulation

Given the system model and requirements, our objective is to generate an assignment of views to servers such that *(i)* each view is stored on at least one server and *(ii)* network usage is minimized. The first objective guarantees that any user view can be served from the memory store. We eliminate the trivial case in which the cluster does not have the storage capacity to keep a copy of each view. DynaSoRe is free to place views on any server, as long as it satisfies their capacity constraints. Any available space can be used to replicate a view and optimize the second objective. We define the amount of extra memory capacity in the system as follows : Given V the set of views in the system, and b the amount of memory required to store a single view, the system has $x\%$ extra memory if its total memory capacity is $(1 + x/100) \times |V| \times b$. To reduce network traffic, we need to assign views to servers such that it reduces the number of messages flowing across network devices. Note that a message between servers reaching the top switch also traverses two intermediate switches and two rack switches. Consequently, minimizing the number of messages going through the top switch is an important goal to reduce network traffic.

We show in Section 4 that DynaSoRe is able to dynamically adapt to workload variations and to use memory efficiently. In this work, we focus on the mechanisms to distribute user views across servers and on the creation and eviction of their replicas. Although important, fault tolerance is out of the scope of this work and we discuss briefly how one can tolerate crashes in Section 3.3.

3 System Design

In this section, we present the design of DynaSoRe. We first present its API, followed by the algorithm we use to make replication decisions. We end this section with a discussion on some software design issues.

3.1 API

DynaSoRe is an in-memory store used in conjunction with a persistent store. The API of DynaSoRe matches the one used by Facebook for memcache [13]. It consists of a `read` request that fetches data from the in-memory store, and a `write` request that updates the data in the memory store using the persistent store. Consequently, DynaSoRe can be used as a drop-in replacement of memcache to cache user views and generate social feeds.

`Read(u, L):` u is a user id and L is a list of user ids to read from. For each id u' in L, it returns *view*(u').

`Write(u):` u is a user id. It updates $view(u)$ by fetching the new version from the persistent store.

3.2 Algorithm

Overview DynaSoRe is an iterative algorithm that optimizes view access locality. DynaSoRe monitors view access patterns to compute the placement of views and selects an appropriate broker for executing each request. Specifically, DynaSoRe keeps track, for each view, the rates it is read and written, as well as the location of brokers accessing it. When DynaSoRe detects that a view is frequently accessed from a distant part of the cluster, consuming large amounts of network resources, it creates a replica of this view and places it on a server close to those distant brokers. This improves the locality of future accesses and reduces network utilization. Similarly, when a broker executes a request, DynaSoRe analyzes the placement of the views accessed, and selects the closest broker as a proxy for the next instance of this operation.

Routing DynaSoRe optimizes view access locality by placing affine views on servers that are close according to network distance, and replicating some of the views on different cluster sub-trees to further improve locality. Using such tailored policies for view placement requires a routing layer to map the identifiers of requested views to the servers storing them.

Brokers. Each request submitted to DynaSoRe is executed by a broker. A request consists of a user identifier and an operation: `read` or `write`. DynaSoRe creates, for each user, a read proxy and a write proxy, each of them being an object deployed on a broker. The motivation of using two different proxies per user stems from the fact that they access different views. The write proxy updates the view of a user, while the read proxy reads the views of a user's social connections. These views may be stored in different parts of the cluster. Allowing DynaSoRe to select different brokers gives it more flexibility and impacts network traffic. The mapping of proxies to brokers is kept in a separate store and is fetched by the front-end as a user logs in. Once a front-end receives a user request, it sends it to the broker hosting the proxy for execution.

Routing policy. When multiple servers store the same view, the routing layer needs to select the most appropriate replica of the view for a given request. The routing policy of DynaSoRe favors locality of access. Following the tree structure of a cluster, a broker selects, among the servers storing a view, the closest one, *i.e.* the one with which it shares the lowest common ancestor. This choice reduces the number of switches traversed. When two replicas are at equal distance, the broker uses the server identifier to break ties.

Routing tables. The write proxy of a user is responsible for updating all the replicas of her view and for storing their locations. Whenever a new replica of

the view is created or deleted, the write proxy serves as a synchronization point and updates its list of replicas accordingly.

The read proxy of a user is in charge of routing her read requests. To this end, each broker stores in a routing table, for every view in the system, the location of its closest replica according to the routing policy described earlier. The routing table is shared by all the read proxies executed on a given broker. The write proxy of a view is also responsible for updating the routing tables whenever a view is created or deleted. As the routing policy is deterministic, only brokers affected by the change are notified.

Servers also store some information about routing. Each view stores the location of its write proxy, so that a server may notify a proxy in case of an eviction or replication attempt of its replica. When several replicas of a view exist, each replica also stores the location of the next closest replica. Both information are used to estimate the utility of a view that will be described in Section 3.2.

Proxy placement. To reduce the network traffic, the proxies should be as close as possible to the views they access. Whenever a request is executed, the proxy uses the routing table to obtain the location of the views and execute the operation. As a post-processing step, the proxy analyzes the location of these views and computes a position that minimizes the network transfers. Starting at the root of the cluster tree, the proxy follows, at each step, the branch from which most views were transferred, until it reaches a broker. If the obtained broker is different from the current one, the proxy migrates to the new broker for the next execution of this request. In the case of a write proxy, this migration involves in sending notification messages to view replicas.

Access statistics To dynamically improve view access locality, DynaSoRe gathers statistics about the frequency and the origin of each access to a view. This information is stored on the servers, along with the view itself. The origin of an access to a view is the switch from which the request accessing this view comes. Consequently, two brokers directly connected to the same switch correspond to the same origin. The writes to a given view are always executed in the location of its write proxy. However, reads can originate from any broker in the cluster. This explains why their origins should be tracked.

To reduce the memory footprint of access recording, DynaSoRe makes the granularity coarser as the network distance increases. Considering a tree-shaped topology, a server records accesses originating from all the switches located between the server and the top switch, as well as their siblings. For example, in Figure 1 the server S_{111} records the accesses originating from the switches SR_{11} (the accesses from the local broker) to SR_{1n} and from SI_2 to SI_m instead of an individual record for every switch. In this way, in a cluster of m intermediate switches and n rack switches per intermediate switch, every replica records maximum $m - 1 + n$ origins instead of $m \times n$ origins. While significantly reducing the memory footprint, this solution does not affect the efficiency of DynaSoRe. The algorithm still benefits from precise information in the last steps of the con-

vergence, and relies on aggregated statistics over sub-trees for decisions about more distant parts of the cluster.

DynaSoRe is a dynamic algorithm that is able to react to variations in the access patterns over time. We use rotating counters to record the number of accesses to views. Each counter is associated to a time period, and servers start updating the following counter at the end of the period. For example, to record the accesses during one day with a rotating period of one hour, we can use 24 counters of 1 byte. The number of counters, their sizes and their rotating periods can be configured depending on the reactivity we expect from the system, the accuracy of the logs and the amount of memory we can spend on it. It is possible to compress these counters efficiently. For instance, one may decrease the probability of logging an access as the counter increases to account for more accesses on 1 byte. One may also store these counters on the disk of the server to enable asynchronous updates of the counters. These optimizations are out of the scope of this work, and in the remainder of this paper we assume that the size of the counters is negligible with respect to the size of the views.

Storage management A DynaSoRe server is a in-memory key-value store implementing a memory management policy. A server has a fixed memory capacity, expressed as the number of views it can store. DynaSoRe manages the servers as a global pool of memory, ensuring that the view of each user is stored on at least one server. Each server stores several views, some of them being the only instance in the system, while others are replicated across multiple servers and therefore optional. The objective of DynaSoRe is to select, for each server, the views that will minimize network utilization, while respecting capacity constraint. We assume that the events generated by users have a fixed size, such as those of Twitter (140 characters). Heavy content (*e.g.*, pictures, videos, etc.) are usually not stored in cache but in dedicated servers.

View utility. Each server maintains read and write access statistics for the views it stores, as described in Section 3.2. Using these statistics, DynaSoRe can evaluate the utility of a view on a given server, *i.e.*, the impact of storing the view on this server in terms of network traffic. DynaSoRe uses the statistics about the origins of read requests to determine which of them are impacted by the view. It then computes the cost of routing them to the next closest replica instead of this server, which represents the read gains of storing the view on the server. The traffic generated by write requests represents the cost of maintaining this view, and is subtracted from the read gains to obtain the utility. The details of the utility computation are presented in Algorithm 1. The utility of a view is positive if its benefits in terms of read requests locality outweighs the cost of updating it when write requests occur. The goal of DynaSoRe is to optimize network utilization. Hence, views with negative utility are automatically removed.

Replication of views. Servers regularly update the utility of the views they store and use this information to maintain an admission threshold so that a sufficient

Algorithm 1 Estimate Profit

```
1: function ESTIMATE PROFIT(logs, server, nearest)
2:     serverReadCost ← 0
3:     nearestReadCost ← 0
4:     for all < source, reads >∈ logs.reads() do
5:         serverReadCost + = reads · COST(source, server)
6:         nearestReadCost + = reads · COST(source, nearest)
7:     serverWriteCost ← writes · COST(broker, server)
8:     serverProfit ← nearestReadCost − serverReadCost − serverWriteCost
9:     return serverProfit
```

amount (*e.g.,* 90%) of their memory is occupied by views whose utility is above the admission threshold. If less memory is used, the admission threshold is 0. These admission thresholds are disseminated throughout the system using a piggybacking mechanism. Each broker maintains the admission threshold of the servers located in its rack, and transmits the lowest threshold to other racks upon accessing them. Thus, each server receives regular updates containing the lowest access threshold in other racks. A replica of a view on a given server serves either the brokers of the whole cluster, when this is the unique replica, or the brokers of a sub-tree of the cluster, when multiple replicas exist. Upon receiving a request for a view, a server updates its access statistics and evaluates the possibility of replicating it on another server of this sub-tree. This procedure is detailed in Algorithm 2. The utility of the replica is computed by simulating its addition on one of the servers, following the approach described previously. If the utility exceeds the admission threshold of the server, a message is sent to the write proxy of the view to request the creation of a replica.

When no replicas can be created, the server attempts to migrate the view to a more appropriate location. The computation of the utility of the view at the new location is slightly different from the replication case, since it assumes the deletion of the view on the current server and therefore generates higher scores. Algorithm 3 details this procedure. The migration of the view is subject to the admission threshold. Using the admission threshold avoids the migration of views rarely accessed to servers with high replication demand.

Eviction of views. To easily deploy new views on servers, DynaSoRe ensures that each server regularly frees memory. When the memory utilization of a server exceeds a given threshold (*e.g.,* 95%), a background process starts evicting the views that have the least utility. Views that have no other replica in the system have infinite utility and cannot be evicted. Since multiples servers could try to evict the different replicas of the same view simultaneously, DynaSoRe relies on the write proxy of the view as a synchronization point to ensure at least one replica remains in the system. Servers typically manage to evict a sufficient amount of views to reach 95% capacity. One exception happens when the full DynaSoRe cluster reaches it maximum capacity, in which case there is no memory left for view replication. This proactive eviction policy decouples the eviction of replicas from the reception of requests, thus ensuring that memory can be freed at any time even when some replicas do not receive any requests.

Algorithm 2 Evaluate Creation of Replica

```
1: procedure Evaluate Creation of Replica(logs)
2:     newReplica ← ∅
3:     bestProfit ← 0
4:     for all < source, reads >∈ logs.reads() do
5:         profit ← Estimate Profit(logs, source, this)
6:         threshold ← Admission threshold(source)
7:         if profit > threshold & profit > bestProfit then
8:             newReplica ← Least loaded server(source)
9:             bestProfit ← profit
10:    if newReplica ≠ ∅ then
11:        Send(newReplica) to broker
```

Algorithm 3 Compute Optimal Position of Replica

```
1: procedure Compute Optimal Position of Replica(logs)
2:     nearest ← nearest replica
3:     bestPosition ← this
4:     bestProfit ← Estimate Profit(logs, this, nearest)
5:     for all < source, reads >∈ logs.reads() do
6:         profit ← Estimate Profit(logs, source, nearest)
7:         threshold ← Admission threshold(source)
8:         if profit > bestProfit & profit > threshold then
9:             bestPosition ← Least loaded server(source)
10:            bestProfit ← profit
11:    if bestProfit < 0 then
12:        Send(removeThis) to broker
13:    else
14:        if bestPosition ≠ this then
15:            Send(bestPosition, removeThis) to broker
```

3.3 Software Design

Durability DynaSoRe complies with the architecture of Facebook, and relies on the same cache coherence protocol [13]. When a user writes an event, this command is first processed by the persistent store to generate the new version of the view of a user. The persistent store then notifies DynaSoRe by sending a `write` request to the write proxy of the user, which fetches the new version of the view from the persistent store and updates the replicas. The persistent store logs `write` requests before sending them, so they can be re-emitted in case of a crash. If a server crashes, the views can be safely recovered from the persistent store. Also, frequently accessed views are likely to be already replicated in the memory of other servers, allowing faster recovery and avoiding cache misses during the recovery process. We have chosen this design because memory is limited, and replicating frequently accessed views leads to higher performance compared to replicating rarely accessed views for faster recovery. However, if a large amount of memory is available, DynaSoRe can also be configured to keep multiple replicas of each view on different servers. In that case, the threshold for infinite utility is set to the minimum number of replicas and recovery is fully performed from memory. The state of brokers and the location of the proxies of users are persisted in a high performance disk-based write-ahead log such as BookKeeper [10], so that the setup of DynaSoRe is also recoverable.

Cluster modification The configuration of the cluster on top of which DynaSoRe is running may change over time. For example, the number of servers allocated to DynaSoRe can grow as the number of users increase. There are three different ways a server can be added to the system:

1. The additional server is added into an existing rack. In this case, the new server will become the least loaded server in the rack, and all the new replicas deployed into this rack are stored in this new server until it becomes as loaded as the other servers in the rack.

2. A new rack is added below an existing intermediate switch. The same reasoning for the previous case applies here. The new rack is automatically used to reduce the traffic of the top router.

3. A new branch is added to the cluster by adding a new intermediate switch. In this case, DynaSoRe has no incentive to add data to the new servers since no requests will originate from there. When adding a new branch to the data center, we consequently need to move some views and proxies onto the new servers to bootstrap it. This procedure is, however, not detailed in this paper.

Removing servers on the other hand requires the views hosted by the servers to be relocated. Before removing a server, the views that have no other replica should be moved to a near server. The views that exist on multiple servers can simply be deleted as DynaSoRe will recreate them if needed.

Managing the social network As described earlier, DynaSoRe does not maintain the social network, and instead receives the list of users from which data need to be retrieved when executing a `read` request. Consequently, the only direct impact of a modification to the social network is the modification of the list of views accessed by reads. The addition of a link between users u_1 and u_2 increases the probability to have either u_2's view replicated near u_1's read proxy, or u_1's read proxy migrated closer to a replica of u_2's view. DynaSoRe adapts to the modifications to the social network transparently, without requiring any specific action. When a new user enters the system, DynaSoRe needs to allocate a read proxy, a write proxy, and a view on a server for this user. The server chosen is the least loaded one at the time of the entrance of the user, and the two proxies are selected to be as close as possible to this server.

4 Evaluation

4.1 Baseline

Random In-memory storage systems, such as Memcached and Redis, rely on hash functions to randomly assign data to servers. This configuration is static in the sense that it is not affected by the request traffic. For this scheme, the proxies of a user are deployed on the broker located in the same rack in which the user view is located. This is the simplest baseline we compare against, as it ignores the topology of the data center, the structure of the social graph, and does not leverage free memory through replication.

METIS Graph data can be statically partitioned across servers using graph partitioning. This leverages the clustering properties of social graphs and increases the probability that social friends are assigned to the same sever. We rely on the METIS library to generate partitions, and randomly assign each of them to a server. The read and write proxies of users are deployed on the broker located in the rack hosting their view. This solution does not take into account the hierarchy of the cluster, and does not perform replication. It also does not handle modifications to the social graph, and needs to re-partition the whole social graph to integrate them.

Hierarchical METIS We improve the standard graph partitioning to account for the cluster structure. We first generate one partition for each intermediate switch, and then recursively re-partition them to assign views to rack switches and then servers. Compared to directly partitioning across servers, this solution significantly reduces the network distance of views of social friends assigned to different servers.

SPAR SPAR [15] is a middleware that ensures the views of the social friends of a user are stored on the same server as her own view. SPAR assumes that it is always possible to replicate a view on a server, without taking into account memory limitations. We adapt SPAR to limit its memory utilization. The views of the friends of a user are copied to her server as long as storage is available. When the server is full, these views are not replicated. Similarly to the graph partitioning case, the proxies of a user are located in the rack hosting her view.

4.2 Datasets

Social graphs We evaluate the performance of DynaSoRe by comparing it against our baselines on three different social networks (summarized in Table 1):

- a sample from the Twitter social graph from August 2009 [3]
- a sample from the Facebook social graph from 2008 [17]
- a sample from the LiveJournal social graph [2]

	# users	# links
Twitter	1.7M	5M
Facebook	3M	47M
LiveJournal	4.8M	69M

Table 1. Number of users and links in each dataset

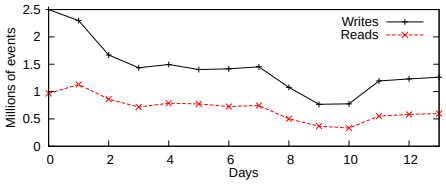


Fig. 2. Number of reads and writes in the Yahoo! News Activity dataset

Request log In this section, we rely on two different kinds of request logs for our experiments, a synthetic one and a real one. The real one is obtained from Yahoo! News. We discuss the logs we used in more details below.

Synthetic logs. Huberman et al. [9] argue that the read and write activity of users in social networks is proportional to the logarithm of their in and out degrees in the social graphs. Silberstein et al. [16] observe that there are approximately 4 times more reads than writes in a social system. Using this information, we create a random traffic generator matching these distributions and obtain, for each social graph, a request log. We additionally assume that each user issues on average one write request per day and that requests are evenly distributed over time. Compared to real workloads, these synthetic workloads show low variation, which enables DynaSoRe to accurately estimate read and write rates.

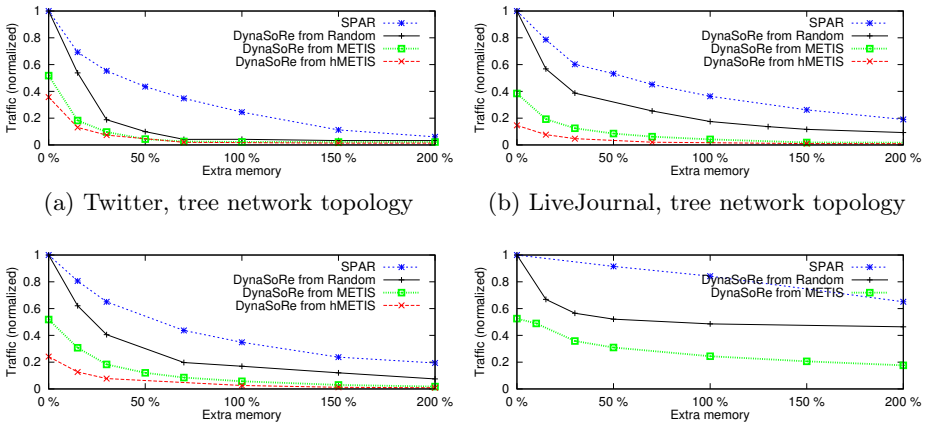
Real user traffic. Yahoo! News Activity is a proprietary social platform that allows users to share (`write`) news articles, and view the articles that their Facebook friends read (`read`). We use a two-week sample of the Yahoo! News Activity logs as a source of real user traffic in the experiments. We focus in this experiment on users who performed at least one read and one write during the two weeks. This selection results in a dataset of 2.5M users with 17M writes and 9.8M reads. Figure 2 depicts the distribution of read and write activities per day. Users can consult the activity of their friends both on the Yahoo! website, or on Facebook. In the latter case, the reads are not processed by the Yahoo! website and do not appear in the log, which explains the prevalence of writes in our dataset. Because we do not have access to the Facebook social graph, we map the users of Yahoo! News Activity to the users in the Facebook social graph presented in Section 4.2. We rank both lists of users according to their number of writes and their number of friends, respectively, and connect users with the same rank. Because the Facebook social graph has more users, we only consider the first 2.5 million users according to the number of friends.

4.3 Simulator and cluster configuration

We implement a cluster simulator in Java to evaluate the different view management protocols on large clusters. The simulator represents all the servers and network devices in order to simulate their message exchanges and measure them. The virtual data center used in our experiment is composed of a top switch, 5 intermediate switches, each connected to 5 rack switches, for a total of 25 racks containing 10 machines each. In every rack, 1 machine is broker while the 9 others are servers to store views. Servers keep view access logs using a sliding counter of 24 slots shifted every hour. After each shift, the replica utility is recomputed and the server's admission threshold is updated. Each server has the same memory capacity, and the total memory capacity is a parameter of each simulator run. Finally, we assume that each application message, *i.e.,* `read`, `write` request and their answer, is 10 times longer than a protocol message. In fact, most protocol messages do not carry any user data and are therefore much smaller.

4.4 Initial data placement and performance

The random placement and graph partitioning approaches produce static assignments of views to servers, which persists during the whole experiment. SPAR



(a) Twitter, tree network topology

(b) LiveJournal, tree network topology

(c) Facebook, tree network topology

(d) Facebook, flat network topology

Fig. 3. Top switch traffic with varying memory capacity

places views as the structure of the social network evolves. We first create one replica for each user, and we simulate the addition of all the edges of the social graph to obtain the its view placement. Once the memory of all servers has been used, the view layout remains constant. For DynaSoRe, the system is deployed on an existing social platform and uses this configuration as an initial setup. It then modifies this initial view placement by reacting to the request traffic.

We consider three different view placement strategies when initializing DynaSoRe: Random, METIS and hierarchical METIS (hMETIS). Using the synthetic request log, we evaluate the performance of each system after convergence, *i.e.,* once the content of the servers stabilizes. Figures 3a, 3b and 3c depict the traffic of the top switch for the 3 different social graphs. The traffic is normalized with respect to the traffic of Random. On the x-axis, we vary the extra memory capacity of the cluster. $x = 0\%$ means the capacity matches exactly the space required to store all the views without replication. With $x = 100\%$ memory capacity doubles, so the algorithms can replicate views up to 2 times on average.

Considering the initial data assignment ($x = 0\%$), we can clearly see that graph partitioning approaches (METIS and hMETIS) outperform Random data placement. Furthermore, hierarchical partitioning leads to a two-fold improvement over standard clustering. These results are expected: partitioning increases the probability that views of social friends will be stored on the same rack, which reduces the traffic of the top switch upon accessing them. hMETIS further improves this result by taking into account the hierarchy of the cluster. Thus, when the views of 2 friends are not located on the same rack, they are likely to be communicated through an intermediate switch rather than the top switch.

As we increase the memory capacity, both DynaSoRe and SPAR are able to replicate and move views. Yet, the results indicate that DynaSoRe is much more efficient than SPAR for using the available memory space. For example, in the

case of Twitter, with 30% memory in addition to the amount of data stored, SPAR reduces the traffic by 42% compared to a Random, while DynaSoRe reduces it by 80%. These figures also demonstrate the importance of the initial data placement in the case of DynaSoRe. As DynaSoRe relies on heuristics to place views in the cluster, a good initial placement allows it to converge to better overall configurations, while a random placement converges to slightly worse performance. As the amount of available memory further increases, the performance of DynaSoRe converges and part of the memory remains unused. Indeed, DynaSoRe detects that replicating some views does not provide an overall benefit, since the cost of writing to the extra replicas outweighs the benefits of reading them locality, which induces higher network traffic.

Table 2 and Table 3 present the average switch traffic at the top, intermediate, and rack levels for two memory configurations. We normalize the traffic value by the equivalent switch traffic using Random. DynaSoRe is initialized using hMETIS. Note that network traffic drops more significantly for the top switch which is the most loaded with Random. As fewer requests access different racks, rack switches also benefit from DynaSoRe, but to a lesser extent. Comparing absolute values, the traffic of the top switch almost drops to the level of a rack switch. Ultimately, DynaSoRe is able to relax the performance requirements for top and intermediate switches.

Figure 4 shows the traffic on the top switch for the Facebook graph using the real user traffic extracted from Yahoo! News Activity. For space reasons, we only display the performances achieved by SPAR and DynaSoRe starting from the placement generated by Random and METIS with 50% extra memory. The figure shows the evolution of the traffic over time, and we can see that the traffic on the top switch follows the request pattern observed in Figure 2. This figure shows that DynaSoRe is able to converge to an efficient view placement configuration, even in the case with high variance traffic. DynaSoRe still clearly outperforms the baseline, confirming the results obtained with the synthetic logs. Our results (not shown here for space reasons) show that the performance of DynaSoRe is consistently better (3 times when starting from Random, 9 times when starting from METIS) than Random independently of the traffic variation, confirming the robustness of DynaSoRe under high traffic.

	Facebook	Twitter	Live J.
Top switch DynaSoRe	.07	.06	.04
Top switch SPAR	.65	.55	.60
Inter switch DynaSoRe	.14	.11	.08
Inter switch SPAR	.77	.61	.70
Rack switch DynaSoRe	.60	.59	.57
Rack switch SPAR	.94	.84	.90

Table 2. Switch traffic, 30% extra memory

	Facebook	Twitter	Live J.
Top switch DynaSoRe	.01	.01	.01
Top switch SPAR	.24	.11	.26
Inter switch DynaSoRe	.03	.02	.02
Inter switch SPAR	.39	.13	.37
Rack switch DynaSoRe	.54	.53	.53
Rack switch SPAR	.77	.60	.75

Table 3. Switch traffic, 150% extra mem.

4.5 Behavior in flat network topologies

The results presented above assume a tree topology for the network of the data center. This setup is common in data centers, hence DynaSoRe was specifically tailored for it. For the sake of fairness (as the baselines are designed without considering any network topology of data centers), we also evaluate DynaSoRe on a flat network topology. In this case, all of the 250 servers act as both caches and brokers, and are directly connected to a single switch. This configuration is similar to the one used to evaluate SPAR in [15]. Figure 3d shows that the performances of DynaSoRe and SPAR on the Facebook social graph using the synthetic request logs. Given that DynaSoRe was specifically tailored to tree topologies, the performance gap between DynaSoRe and SPAR is not as large as that presented in Figure 3c. DynaSoRe still clearly outperforms SPAR, in particular in the configurations of low memory, thanks to its better replication policy. In the remainder of the evaluation, we focus on the tree network topology.

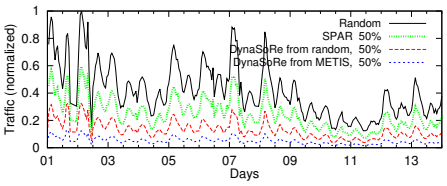


Fig. 4. Top switch traffic with Yahoo! News Activity requests, Facebook

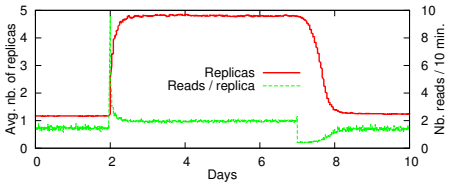
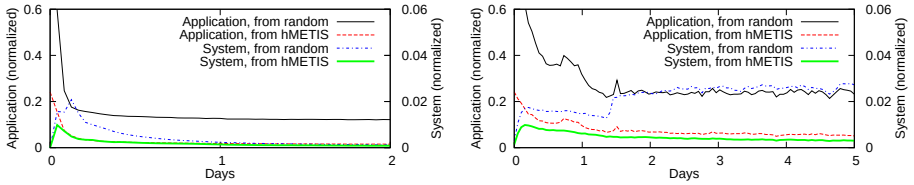


Fig. 5. Flash event : Addition of 100 followers, Facebook, 30% extra memory

4.6 Flash Events

Online social networks are often subject to flash events, in which the activity of a subset of users suddenly spikes. To evaluate the reactivity of DynaSoRe, we simulate a sudden increase of popularity of some users, and measure the evolution of their number of replicas and the number of requests each of them processes. More precisely, at time $t = 2$ days in the simulation run, we randomly select a user and make this user popular by adding 100 random followers to read her view. Five days later ($t = 7$ days) all these additional followers are removed. We repeat this experiment 100 times on the Facebook dataset with a 30% extra memory capacity. We present the average results in Figure 5.

At the beginning of the experiment, the user is not particularly active, and has 1.15 replicas on average. As new followers arrive, DynaSoRe detects that the number of reads of the view increases, and starts replicating it on other servers. DynaSoRe stabilizes in a configuration close to 5 replicas for this view. Given that the users reading this view are selected at random, they originate from all racks of the cluster and DynaSoRe generates a replica per intermediate switch. After replication, the average number of reads per replica is very close to the



(a) Synthetic requests (b) Real requests

Fig. 6. Top switch traffic over time, Facebook, 150% extra memory

initial situation. The utility of replicas is high enough to be maintained by the system, but additional replicas do not pass the admission threshold. At the end of this period, the number of reads per replica drops sharply. DynaSoRe is able to detect and adjust the utility of the replicas, which leads to their eviction before the end of the following day. These results illustrate DynaSoRe's ability to react quickly to flash events, and evict replicas once they become useless.

4.7 Convergence time

SPAR and the three static approaches to assign views only require the social graph to determine the assignment of views. They do not react dynamically to traffic changes, and consequently, they do not require any time to converge as long as the social graph is stable. For DynaSoRe, however, it is important to evaluate the convergence time, using both stable synthetic traffic and real traffic. Before converging to a stable assignment, DynaSoRe replicates views regularly. This traffic of replicas generates messages that also consume network resources. Once the system stabilizes, the overhead of system messages becomes negligible.

Figure 6a shows the traffic of the top switch when running DynaSoRe on the Facebook social graph using synthetic traffic with an extra memory of 150%. We separate the application traffic and the system traffic to study the convergence of DynaSoRe over time (x axis). After a few hours of traffic, DynaSoRe has almost reached its best performance, starting both from a random placement and from a placement based on graph partitioning. The amount of system messages sent by the protocol rapidly drops and reaches its minimum after one day. Note that less memory capacity makes the time to converge shorter, since DynaSoRe performs fewer replication operations. Figure 6b displays the results of the same experiments executed using the real request trace from Yahoo! News Activity. As the workload presents more variation, DynaSoRe does not fully converge, and the system traffic remains at a noticeable level as views are created and evicted. The request rate of the real workload is lower than the synthetic one, which explains the slower convergence: DynaSoRe is driven by requests. Initializing DynaSoRe using graph partitioning, however, induces an initial state that is more stable, allowing the system traffic to remain low. Despite slower convergence, the application traffic still reaches its best performance after one day.

5 Related Work

DynaSoRe enables online data placement in in-memory store for social networking applications. We review in this section related work on in-memory storage systems, offline data placement algorithms, online data placement algorithms, and discuss their differences with DynaSoRe.

In-memory storage RAMCloud [14] is a large-scale in-memory storage system that aggregates the RAM of hundreds of servers to provide a low-latency key-value store. RAMCloud does not currently implement any data placement policy, and could benefit from the algorithms used in DynaSoRe. RAMCloud recovers from failures using a distributed log accessed in parallel on multiple disks. This is similar to write-ahead logging approach described in Section 3.3 and could be also used in DynaSoRe.

Offline data placement Curino *et al.* describe Schism [4], a partitioning and replication approach for distributed databases to minimize the amount of transactions executed across multiple servers. Schism uses an offline standard graph partitioning algorithms on the request log graph to assign database tuples to servers. DynaSoRe is an online strategy, creating and placing views dynamically. As a consequence, it is much easier to react to changes in access patterns that frequently occur in social applications. DynaSoRe benefits from graph clustering techniques similar to those used in Schism to generate more effective initial placement of views for faster convergence to ideal data placement.

Zhong *et al.* consider the case of object placement for multi-object operations [19]. Using linear programming, they place correlated objects on the same nodes to reduce the communication overhead. However, this solution focuses on correlation and does not take access frequencies into account. It does not account for the hierarchy of network either.

Duong *et al.* analyze the problem of statically sharding social networks to optimize read requests [5]. They demonstrate the benefits of social-network aware data placement strategies, and obtain moderate performance improvements through replication. Nonetheless, these results are limited by the absence of write requests in the cost model. In addition, it only supports static social networks and does not account for network topology.

There are a few graph processing engines [6,12,18] that split graphs over several machines using offline partitioning algorithms. Messages exchanged between partitions are the results of partial computations, which can be further reduced through the use of combiners. While these approaches lead to important gains, they cannot be applied to all kind of requests, and they mostly benefit long computational tasks rather than low latency systems considered in this paper.

Online data placement SPAR [15] is a middleware for online social networking systems that ensures that the server containing the view of a user also contains those of her friends. This favors reads, but sacrifices writes as all the replica of a user's view need to be updated. Similar to DynaSoRe, SPAR uses an online

algorithm that reacts to the evolution of the social network. The main differences between SPAR and DynaSoRe stems from the assumption on the storage layer. SPAR assumes that storage is cheap enough to massively replicate views, up to 20 times for 512 servers, largely exceeding fault tolerance requirements. DynaSoRe is much more flexible, and operates at a sweet spot, trading a small storage overhead for high network gains. By default, DynaSoRe does not guarantee that each view is replicated multiple times, and relies on the stable storage to ensure durability. Yet, DynaSoRe can be configured to provide an in-memory replication equivalent to SPAR, as explained in Section 3.3.

Silberstein *et al.* propose to measure users' events production and consumption rates to devise a push-pull model for social feeds generation [16]. The specialized data transfer policy significantly reduces the load of the servers and the network. DynaSoRe is inspired by this work and also relies on the rates of reads and writes of events to decide when to replace views. However, DynaSoRe addresses different problem and focuses on determining where to maintain the views, which will lead to performance gain in addition to this approach.

DynPart [11] is a data partitioning algorithm triggered upon inserting tuples in a database. DynPart analyzes requests matching a tuple and places the tuple on the servers that are accessed when executing these requests. While DynPart handles insertion of data, it never reverts previous decisions and therefore cannot deal with new requests or changes in request frequency. Social networks are frequently modified, leading to different requests, and are subject to unpredictable flash events. For these reasons, DynaSoRe is a better fit for social applications.

6 Conclusion

Adapting to workload variations and incorporating detail of the underlying network architecture are both critical for serving social networking applications efficiently. Typical designs that randomly and statically place views across servers induce a significant amount of load to top tiers of tree-based network layouts. DynaSoRe is an in-memory view storage system that instead adapts to workload variations and uses the network distance between servers to reduce traffic at the top tiers. DynaSoRe analyzes request traffic to optimize view placement and substantially reduces network utilization. DynaSoRe leverages free memory capacity to replicate frequently accessed views close to the brokers reading them. It selects the brokers that serve each request and places them close to the views they fetch according to network distance. In our evaluation of DynaSoRe, we used different social networks and showed that with only 30% additional memory, the traffic of the top switch drops by 94% compared to a static random view placement, and 90% compared to the SPAR protocol.

Acknowledgement

This work was supported by the LEADS project (ICT-318809), funded by the European Community, the Torres Quevedo Program from the Spanish Ministry

of Science and Innovation, co-funded by the European Social Fund; and the ERC Starting Grant GOSSPLE number 204742.

References

1. M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*, pages 63–74, 2008.
2. L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: membership, growth, and evolution. In *SIGKDD*, pages 44–54, 2006.
3. M. Cha, H. Haddadi, F. Benevenuto, and K. P. Gummadi. Measuring User Influence in Twitter: The Million Follower Fallacy. In *In ICWSM*, 2010.
4. C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *the VLDB Endowment*, 3(1-2):48–57, 2010.
5. Q. Duong, S. Goel, J. Hofman, and S. Vassilvitskii. Sharding social networks. In *WSDM*, pages 223–232, 2013.
6. J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, 2012.
7. A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: a scalable and flexible data center network. In *SIGCOMM*, pages 51–62, 2009.
8. U. Hoelzle and L. A. Barroso. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 2009.
9. B. A. Huberman, D. M. Romero, and F. Wu. Social networks that matter: Twitter under the microscope. *First Monday*, 14(1), 2009.
10. F. P. Junqueira, I. Kelly, and B. Reed. Durability with bookkeeper. *ACM SIGOPS Operating Systems Review*, 47(1):9–15, 2013.
11. M. Liroz-Gistau, R. Akbarinia, E. Pacitti, F. Porto, and P. Valduriez. Dynamic workload-based partitioning for large-scale databases. In *DEXA*, volume 7447, pages 183–190, 2012.
12. G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.
13. R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at facebook. In *NSDI*, pages 385–398, 2013.
14. D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in ramcloud. In *SOSP*, pages 29–41, 2011.
15. J. M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez. The little engine(s) that could: scaling online social networks. In *SIGCOMM*, pages 375–386, 2010.
16. A. Silberstein, J. Terrace, B. F. Cooper, and R. Ramakrishnan. Feeding frenzy: selectively materializing users' event feeds. In *SIGMOD*, pages 831–842, 2010.
17. C. Wilson, B. Boe, A. Sala, K. P. Puttaswamy, and B. Y. Zhao. User interactions in social networks and their implications. In *Eurosys*, pages 205–218, 2009.
18. S. Yang, X. Yan, B. Zong, and A. Khan. Towards effective partition management for large graphs. In *SIGMOD*, pages 517–528, 2012.
19. M. Zhong, K. Shen, and J. Seiferas. Correlation-aware object placement for multi-object operations. In *ICDCS*, pages 512–521, 2008.