Social Hash: an Assignment Framework for Optimizing Distributed Systems Operations on Social Networks

Alon Shalita[†], Brian Karrer[†], Igor Kabiljo[†], Arun Sharma[†], Alessandro Presta[†], Aaron Adcock[†], Herald Kllapi[*], and Michael Stumm[§]

[†]Facebook {alon,briankarrer,ikabiljo,asharma,alessandro,aadcock}@fb.com
[*]University of Athens herald@di.uoa.gr
[§]University of Toronto stumm@eecg.toronto.edu

Abstract

How objects are assigned to components in a distributed system can have a significant impact on performance and resource usage. *Social Hash* is a framework for producing, serving, and maintaining assignments of objects to components so as to optimize the operations of large social networks, such as Facebook's Social Graph. The framework uses a two-level scheme to decouple compute-intensive optimization from relatively low-overhead dynamic adaptation. The optimization at the first level occurs on a slow timescale, and in our applications is based on graph partitioning in order to leverage the structure of the social network. The dynamic adaptation at the second level takes place frequently to adapt to changes in access patterns and infrastructure, with the goal of balancing component loads.

We demonstrate the effectiveness of Social Hash with two real applications. The first assigns HTTP requests to individual compute clusters with the goal of minimizing the (memory-based) cache miss rate; Social Hash decreased the cache miss rate of production workloads by 25%. The second application assigns data records to storage subsystems with the goal of minimizing the number of storage subsystems that need to be accessed on multiget fetch requests; Social Hash cut the average response time in half on production workloads for one of the storage systems at Facebook.

1 Introduction

Almost all of the user-visible data and information served up by the Facebook app is maintained in a single directed graph called the *Social Graph* [2, 34, 35]. Friends, Checkins, Tags, Posts, Likes, and Comments are all represented as vertices and edges in the graph. As such, the graph contains billions of vertices and trillions of edges, and it consumes many hundreds of petabytes of storage space.

The information presented to Facebook users is primarily the result of dynamically generated queries on the Social Graph. For instance, a user's home profile page contains the results of hundreds of dynamically triggered queries. Given the popularity of Facebook, the Social Graph must be able to service well over a billion queries a second.

The scale of both the graph and the volume of queries makes it necessary to use a distributed system design for implementing the systems supporting the Social Graph. Designing and implementing such a system so that it operates efficiently is non-trivial.

A problem that repeatedly arises in distributed systems that serve large social networks is one of *assigning objects to components*; for example, assigning user requests to compute servers (*HTTP request routing*), or assigning data records to storage subsystems (*storage sharding*). How such assignments are made can have a significant impact on performance and resource usage. Moreover, the assignments must satisfy a wide range of requirements: e.g., they must (*i*) be amenable to quick lookup, (*ii*) respect component size constraints, and (*iii*) be able to adapt to changes in the graph, usage patterns and hardware infrastructure, while keeping the load well balanced, and (*iv*) limit the frequency of assignment changes to prevent excess overhead.

The relationship between the data of the social network and the queries on the social network is $m : n$ — a query may require several data items and a data item may be required by several queries. This makes finding a good assignment of objects to components non-trivial; finding an optimal solution for many objective functions is NP Hard [6]. Moreover, a target optimization goal, captured by an objective function, may conflict with the

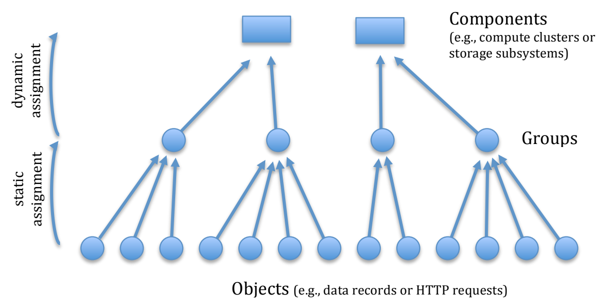
Components
(e.g., compute clusters or storage subsystems)
dynamic assignment
Groups
static assignment
Objects (e.g., data records or HTTP requests)

Figure 1: Social Hash Abstract Framework

goal of keeping the loads on the components reasonably well balanced. In the next subsection, we propose a two-level framework that allows us to trade off these two conflicting objectives.

Social Hash Framework

We have developed a general framework that accommodates the HTTP request routing and storage sharding examples mentioned above, as well as other variants of the assignment problem. In our Social Hash framework, the assignment of objects (such as users or data records) to components (such as compute clusters or storage subsystems) is done in two steps. (See Fig. 1.)

In the first step, each *object* is assigned to a *group*, where groups are conceptual entities representing clusterings of objects. Importantly, there are usually many more groups than components. This assignment is based on optimizing a given, scenario-dependent, objective function. For example, when assigning HTTP requests to compute clusters, the objective function may seek to minimize the (main memory) cache miss rate; and when assigning data records to disk subsystems, the objective function may seek to minimize the number of disk subsystems that must be contacted for multi-get queries. Because this optimization is typically computationally intensive, objects are re-assigned to groups only periodically and offline (e.g., daily or weekly). Hence, we refer to this as the *static assignment step*.

In the second step, each *group* is assigned to a *component*. This second assignment is based on inputs from system monitors and system administrators so as to rapidly and dynamically respond to changes in the system and workload. It is able to accommodate components going on or offline, and it is responsible for keeping the components' loads well balanced. Because the assignments at this level can change in real time, we refer to this as the *dynamic assignment step*.

A key attribute of our framework is the decoupling of optimization in the static assignment step, and dynamic adaptation in the dynamic assignment step. Our solutions to the assignment problem rely on being able to beneficially group together relatively small, cohesive sets of objects in the Social Graph. In the optimizations performed by the static assignment step, we use graph partitioning to extract these sets from the Social Graph or from prior access patterns. Optimization methods other than graph partitioning could be used interchangeably, but graph partitioning is expected to be particularly effective in the context of social networks, because most requests are social in nature where users that are socially close tend to consume similar data. The *Social* in Social Hash reflects this essential idea of grouping socially similar objects together.

Contributions

This paper describes the Social Hash framework for assigning objects to components given scenario-dependent optimization objectives, while satisfying the requirements of fine-grained load balancing, assignment stability, and fast lookup in the context of practical difficulties presented by changes in the workload and infrastructure.

The Social Hash framework and the two applications described in this paper have been in production use at Facebook for over a year. Over 78% of Facebook's "stateless" Web traffic routing occurs with this framework, and the storage sharding application involves tens of thousands of storage servers. The framework has also been used in other settings (e.g., to distribute vertices in a graph processing system, and to reorder data to improve compression rates). We do not describe these additional applications in this paper.

The three most important contributions we make in this paper are:

1. the two-step assignment hierarchy of our framework that decouples (*a*) optimization on the Social Graph or previous usage patterns from (*b*) adaptation to changes in the workload and hardware infrastructure;

2. our use of graph partitioning to exploit the structure of the social network to optimize HTTP routing in very large distributed systems;

3. our use of query history to construct bipartite graphs that are then partitioned to optimize storage sharding.

With respect to (1), the use of a multi-level scheme for allocating resources in distributed systems is not new, not even when used with graph partitioning [33]. In particular, some multi-tenant resource allocation schemes have used approaches that are in many respects similar to the one being proposed here [19, 26, 27, 28]. However, the specifics of our approach, especially as they relate to Facebook's operating environment and workload, are sufficiently interesting and unique to warrant a dedicated discussion and analysis. Regarding (2), edge-cut based graph partitioning techniques have been used for numerous optimization applications, but to the best of our knowledge not for making routing decisions to reduce cache miss rates. Similarly, for (3), graph partitioning has previously been applied to storage sharding [33], but partitioning bipartite graphs based on prior access patterns is, as far as we know, novel.

We show that the Social Hash framework enables significant performance improvements as measured on the production Social Graph system using live workloads. Our HTTP request routing optimization cut the cache miss rate by 25%, and our storage sharding optimization cut the average response latency in half.

2 Two motivating example applications

In this section, we provide more details of the two examples we mentioned in the Introduction. We discuss and analyze these applications in significantly greater detail in later sections.

HTTP request routing optimization. The purpose of HTTP request routing is to assign HTTP requests to compute clusters. When a cluster services a request, it fetches any required data from external storage servers, and then caches the data in a cluster-local main memory-based cache, such as TAO [2] or Memcache [24], for later reuse by other requests. For example, in a social network, a client may issue an HTTP request to generate the list of recent posts by a user's friends. The HTTP request will be routed to one of several compute clusters. The server will fetch all posts made by the user's friends from external databases and cache the fetched data. How HTTP requests are assigned to compute clusters will affect the cache hit rate (since a cached data record may be consumed by several queries). It is therefore desirable to choose a HTTP request assignment scheme which assigns requests with similar data requirements to the same compute cluster.

Storage sharding optimization. The purpose of storage sharding is to distribute a set of data records across several storage subsystems. A query which requires a certain record must communicate with the unique host that serves that record.[1] A query may consume several records, and a record may be consumed by several queries. For example, if the dataset consists of recent posts produced by all the users, a typical query might fetch the recent posts produced by a user's friends.

[1]To simplify our discussion, we disregard the fact that data is typically replicated across multiple storage servers.

The assignment of data records to storage subsystems determines the number of hosts a query needs to communicate with to obtain the required data. A common optimization is to group requests destined to the same storage subsystem and issue a single request for all of them. Additionally, since requests to different storage subsystems are processed independently, they can be sent in parallel. As a result, the latency of the slowest request will determine the latency of a multi-get query, and the more hosts a query needs to communicate with, the higher the expected latency (as we show in Section 6.1). It is thus desirable to choose a data record assignment scheme that collocates the data required by similar queries within a small number of storage subsystems.

3 The assignment problem

Assigning objects to system components is a challenging part of scaling an online distributed system. In this section, we abstract the essential features of our two motivating examples to formulate the problem we solve in this paper.

3.1 Requirements

We have the following requirements:

- *Minimal average query response time*: User satisfaction can improve with low query response times.
- *Load balanced components*: The better load-balanced the components, the higher the efficiency of the system; a poorly load-balanced system will reach its capacity earlier and in some cases may lead to increased latencies.
- *Assignment stability*: Assignments of objects to components should not change too frequently in order to avoid excessive overhead. For example, reassigning a query from one cluster to another may lead to extra (cold) cache misses at the new cluster.
- *Fast lookup*: Low latency lookup of the object-component assignment is important, given the online nature of our target distributed system.

3.2 Practical challenges

Meeting the requirements listed above is challenging for a variety of reasons:

• *Scale*: The assignment problem typically requires assigning a large number of objects to a substantially smaller number of components. The combinatorial explosion in the number of possible assignments prevents simple optimization methods from being effective.

• *Effects of similarity on load balance*: Colocating similar objects usually results in higher load imbalances than when colocating dissimilar objects. For example, similar users likely have similar hours of activity, browsing devices, and favorite product features, leading to load imbalance when assigning similar users to the same compute clusters.

• *Heterogenous and dynamic set of components*: Components are often heterogeneous and thus support different loads. Further, the desired load on each component can change over time; e.g., due to hardware failure. Finally the set of components will change over time as new hardware is introduced and old hardware removed.

• *Dynamic workload*: The relationship between data and queries can change over time. A previously rarely accessed data record could become popular, or new types of queries could start requesting data records that were previously not accessed. This can happen, for example, if friendship ties in the network are introduced or removed, or if product features change their data consumption pattern.

• *Addition and removal of objects*: Social networks change and grow constantly, so the set of objects that must be assigned changes over time. For example, users may join or leave the service.

The magnitude and relative importance of these practical challenges will differ depending on the distributed system being targeted. For Facebook, the scale is enormous; similar users do have similar patterns; and heterogeneous hardware is prevalent. On the other hand, changes to the graph occur at a (relatively) modest rate (in part because we often only consider subgraphs of the Social Graph); and rate of hardware failures is reasonably constant and predictable.

4 Social Hash Framework

In this section, we propose a framework called the Social Hash Framework which comprises a solution to the assignment problem and, moreover, addresses the practical challenges listed above.

In Section 1 we introduced the abstract framework with objects at the bottom, (abstract) groups in the middle, and components at the top. Recall that objects are queries, users, or data records, etc., and components are computer clusters, or storage subsystems, etc..

Objects are first assigned to groups in a optimization-based static assignment that is updated on a slow timescale of a day to a week. Groups are then assigned to components using an adaptation-based dynamic assignment that is updated on a much faster timescale. Dynamic assignment is used to keep the system load-balanced despite changes in the workload or changes in the underlying infrastructure. This two-level design is intended to accommodate the disparate requirements and challenges of efficiently operating a huge social network, as described in Section 3.

Below, we give more concrete details on the abstract framework, how it is implemented, and how it is used. In Sections 5 and 6 we will become even more concrete and present specific implementation issues for our two examples. We begin by presenting our rationale for using a two-level design.

4.1 Rationale

Our two-level approach for assigning objects to components is motivated by the observation that there is a conflict between the objectives of optimization and adaptation. In theory, one could assign objects to components directly, resulting in only one assignment step. However, this would not work well in practice because of difficulties adapting to changes: as mentioned, component loads often change unpredictably; components are added or removed from the system dynamically; and the similarity of objects that are naturally grouped together for optimization leads to unbalanced utilization of resources. Waiting to rerun the assignment algorithm would leave the system in a suboptimal state for too long, and changing assignments on individual objects without re-running the assignment algorithm would also be suboptimal.

An assignment framework must therefore address both the optimization and adaptation objectives, and it must offer enough flexibility to be able to shift emphasis between these competing objectives at will. With a two-level approach, the static level optimizes the assignment to groups where, from the point of view of optimization, the group is treated as a virtual component. The dynamic level adapts to changes by assigning groups to components. Multiple groups may be assigned to the same component; however, all objects in the same group are guaranteed to be assigned to the same component. (See Figure 1.) As such, what is particularly propitious about our architecture is that dynamic reassignment of groups to components does not negate the optimization step because objects in a group remain collocated to the same component, even after reassignment.

We are able to seamlessly shift emphasis between static optimization and dynamic adaptation by means of the parameter n, the ratio of number of groups to number of components; that is $n := |G| \big/ |C|$. When $n = 1$, the emphasis is entirely on the static optimization. There is a $1:1$ correspondence between groups and components. As noted above, this may not work well for some applications because it may not be sufficiently adaptive. When $n \gg 1$, we trade off optimization for increased adaptation. When n is too large, the optimization quality may be severely degraded, and the overhead of dynamic assignment may be prohibitive. Clearly, the choice of n, and thus the tradeoff between optimization and adaptation, is best selected on a per-application basis; as we show in later sections, some applications require less aggressive adaptation than others, allowing more emphasis to be placed on optimization.

4.2 Framework Overview

In this subsection, we describe the main elements of the Social Hash framework, as depicted in Fig. 2: the static assignment algorithm, the dynamic assignment algorithm, the lookup method, and the missing key assignment. In the discussion that follows it is useful to note that objects are uniquely identified by a *key*.

The static assignment algorithm generates a static mapping from objects to groups using the following input: (*i*) a context dependent graph, which in our work can be either a unipartite graph (e.g., friendship graph) or a bipartite graph based on access logs (e.g., relating queries and accessed data records); (*ii*) type of object that is to be assigned to groups (e.g. data records, users, etc); (*iii*) an objective function; (*iv*) number of groups; and (*v*) permissible imbalance between groups. The output of the static partitioning algorithm is a hash table of (*key*, *group*) pairs, indexed by *key*. We refer to this hash table as the *Social Hash Table*.[2]

The dynamic assignment uses the following input: (*i*) current component loads, (*ii*) the desired maximum load per component, and possibly (*iii*) the historical loads per group. The desired load for each component is provided by system operators and monitoring systems, and the historical loads induced by each group can be derived from system logs. As the observed and desired loads change over time, the dynamic assignment shifts groups among components to balance the load. The output of the dynamic assignment is a hash table of (*group*, *component*) pairs, called the *Assignment Table*.[2]

[2]In practice, any key-value store that supports fast lookups can be used. We describe it as a hash table for ease of comprehension.

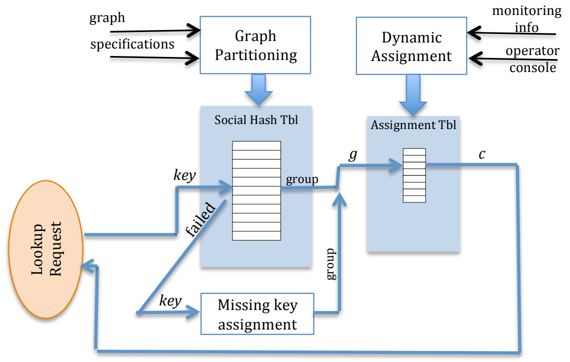

Figure 2: Social Hash Architecture

When a client wishes to look up which component an object has been assigned to, it will do so in two steps: first the object key is used to index into the Social Hash Table to obtain the target group, g; second, g is used to index into the Assignment Table to obtain the component id c. This is shown in Figure 2.

Because the Social Hash Table is constructed only periodically, it is possible that a target *key* is missing in the Social Hash table; for example, the key could refer to a new user or a user that has not had any activity in the recent past (and hence is not in the access log). When an object key is not found in the Social Hash Table, then the *Missing Key Assignment* rule does the exception handling and assigns the object to a group on the fly. The primary requirement is that these exceptional assignments are generated in a consistent way so that subsequent lookups of the same key return the same group. Eventually these new keys will be incorporated into the Social Hash Table by the static partitioning algorithm.

4.3 Static assignment algorithm

We use graph partitioning algorithms to partition objects into groups in the static assignment step. Graph partitioning algorithms have been well-studied [3], and a number of graph partitioning frameworks exist [5, 18]. However, social network graphs, like Facebook's Social Graph, can be huge compared to what much of the existing literature contemplates. As a result, an approach is needed that is amenable to distributed computation on distributed memory systems. We built our custom graph partitioning solution on top of the Apache Giraph graph processing system [1], in part because of its ability to partition graphs in parallel; other graph processing systems could have also potentially been used [10, 11, 20].

The basic strategy in obtaining good static assignments is the following graph partitioning heuristic. We assume the algorithm begins with an initial (weight-) balanced assignment of objects to groups represented as pairs (v, g_v), where v denotes an object and g_v denotes the group to which v is initially assigned. Next, for each v, we record the group g_v^* that gives the optimal assignment for v to minimize the objective function, assuming all other assignments remain the same. This step is repeated for each object to obtain a list of pairs (v, g_v^*). Each object can be processed in parallel. Finally, via a swapping algorithm, as many reassignments of v to g_v^* are carried out under the constraint that group sizes remain unchanged within each iteration; the swapping can again be done in parallel as long as it is properly coordinated (in our implementation with a centralized coordinator). This overall process is then repeated with the new assignments taken as the initial condition for the next iteration. The above process is iterated on until it converges or reaches a limit on the number of iterations.

The initial balanced assignment required by the static assignment algorithm is either obtained via a random assignment (e.g., when the algorithm is run for the very first time) or is obtained from the output of the previous iteration of the static assignment algorithm modulo the newly added objects that are assigned randomly.

The above procedure manages to produce high quality results for the graphs underlying Facebook operations in a fast and scalable manner. Within a day, a small cluster of a few hundred machines is able to partition the friendship graph of over 1.5B+ Facebook users into 21,000 balanced groups such that each user shares her group with at least 50% of her friends. And the same cluster is able to update the assignment starting from the previous assignment within a few hours, easily allowing a weekly (or even daily) update schedule. Finally, it is worth pointing out that the procedure is able to partition the graph into tens of thousands of groups, and it is amenable to maintaining stability, since each iteration begins with the previous assignment and it is easy to limit the movement of objects across groups.

We have successfully used the above heuristic on both unipartite and bipartite graphs, as we describe in more detail in Sections 5 and 6.

4.4 Dynamic assignment

The primary objective of dynamic assignment is to keep component loads well balanced despite changes in access patterns and infrastructure. Load balancing has been well researched in many domains. However, the specific load balancing strategy used for our Social Hash framework may vary from application to application so as to provide the best results. Factors that may affect the the choice of load balancing strategy include:

- *Accuracy in predicting future loads*: Low prediction accuracy favors a strategy with a high group-to-component ratio (e.g., $\gg 1,000$) and groups being assigned to components randomly. This is the strategy that is used for HTTP routing. On the other hand, the amount of storage used by data records is easier to predict (in our case), and hence warrants a low group-to-component ratio and non-random component assignment.
- *Dimensionality of loads*: A system requiring balancing across multiple different load dimensions (CPU, storage, queries per second, etc.) favors using a high group-to-component ratio and random assignment.
- *Group transfer overhead*: The higher the overhead of moving a group from one component to another, the more one would want to limit the rate of moves between components by increasing the load imbalance threshold that triggers a move.
- *Assignment memory*: It can be more efficient to assign a group back to an underloaded component it was previously assigned to in order to potentially benefit from the residual state that may still be present. This favors remembering recent assignments, or using techniques similar to consistent hashing.

Finally, we note that load balancing strategies used in other domains will need to be adapted to the Social Hash framework; e.g., load is transferred from one component to another in increments of a group; and the load each group incurs is not homogeneous, in part because of the similarity of objects within groups.

5 Social Hash for Facebook's Web Traffic Routing

In this section, we describe how we applied the Social Hash framework to Facebook's global web traffic routing to improve the efficiency of large cache services. This is Facebook's largest application using the framework and has been in production for over a year.

Facebook operates several worldwide data centers, each divided into front-end clusters containing web and cache tiers, and back-end clusters containing database and service tiers. To fulfill an HTTP request, a front-end web server may need to access databases or services in (possibly remote) back-end clusters. The returned data is cached within front-end cache services, such as TAO [2] or Memcache [24]. Clearly, the lower the cache miss rate, the higher the efficiency of hardware usage, and the lower the response times.

In addition, to reduce latencies for users, Facebook has “Point-of-Presence” (PoP) units around the world: small-scale computational centers which reside close to users. PoPs are used for multiple purposes, including peering with other network operators and media caching [12]. When an HTTP request to one of Facebook’s services is issued, the request will first go to a nearby PoP. A load balancer in the PoP then routes the request to one of the front-end clusters over fast communication channels.

5.1 Prior strategy

Prior to using Social Hash, routing decisions were based on user identifiers, using a consistent hashing scheme [15]. To make a routing decision, the user identifier was extracted from the request, where it was encoded within the request persistent attributes (i.e., cookie), and then used to index into a consistent hash ring to obtain the cluster id. The segments of the consistent hash ring corresponded in number and weight to the front-end clusters. The ring’s weights were dynamic and could be changed at any time, allowing dynamic changes to the cluster’s traffic load. The large number of users in comparison to the small number of clusters, along with the random nature of the hash ring, ensured that each cluster received a homogeneous traffic pattern. With fixed cluster weights, a user would repeatedly be routed to the same cluster, guaranteeing high hit rates for user-specific data. The consistent nature of the ring also ensured that changes to cluster weights resulted in relatively minor changes to the user-to-cluster mapping, reducing the number of cache misses after such changes.

5.2 Social Hash implementation

For the Social Hash static assignment, we used a unipartite graph with vertices representing Facebook’s users and edges representing the friendship ties between them. We partition the graph using the edge-cut optimization criterion, knowing that friends and socially similar users tend to consume the same data, and that they are therefore likely to reuse each other’s cached data records.

We use a relatively large number of groups for two reasons. First, the global routing scheme needs to be able to shift traffic across clusters in small quantities. Second, changes in HTTP request routing will affect many subsystems at Facebook, not just the cache tiers; and it is very difficult to predict how much load each group will incur on each subsystem. Hence, we have found the best strategy to balance the load overall is to use many groups and assign the groups to clusters randomly.

For the dynamic assignment step, we kept the existing consistent hash scheme, which is oblivious to the type of identifier it receives as input (either user- or group-id).

To be able to make an HTTP request routing decision at run time, it is necessary to access both the Social Hash Table and the Assignment Table. The latter is computed on-the-fly using the consistent hash mechanism, which requires a fairly small map between clusters and their weights; it is therefore easy to hold the map in the POP memories. The former, however, is large, consuming several gigabytes of storage space when uncompressed. We considered storing the Social Hash Table close to the PoP (in its own memory or in a nearby storage service), but decided not to do so due to added PoP complexity, fault tolerance considerations, and limited PoP resources that could be put to better use by other services. We also considered sending a lookup request to a data center, but rejected this idea due to latency concerns.

Instead, we encode the user assigned group within the request persistent attributes (i.e., as a cookie) and decode it to make a routing decision when a request arrives. Requests that do not have the group-id encoded in the header are routed to a random front-end cluster, where the session creation mechanism accesses a local copy of the Social Hash Table to fetch the group assigned to the user. Because the Social Hash Table is updated once a week, group-ids in the headers may become stale. For this reason, a user request will periodically (at least once an hour) trigger an update process where the group-id is updated with its latest value from the Social Hash Table. This allows long lasting connections to receive fresh routing information.

Our design eliminates the complexities and overhead of a Social Hash Table lookup at the PoPs, requiring just a single header read instead. The design is also more resilient to failure, because even if the data store providing the Social Hash Table is down, group-id’s will mostly be available in the request headers.

For technical reasons, some requests cannot be tagged properly with either the user or the group identifier (because the requests may have been issued by crawlers, bots or legacy clients). These requests are routed randomly, yet in a consistent manner, to one of the front-end clusters while respecting load constraints. In the past three months, 78% of the requests had a valid group-id that could be used for routing (and those that did not were not tagged with a user-id, a group-id, or any other identifier.).

Some may argue that the decreased miss rates achieved with Social Hash leads to a fault tolerance issue, because the data records are less likely to be present in

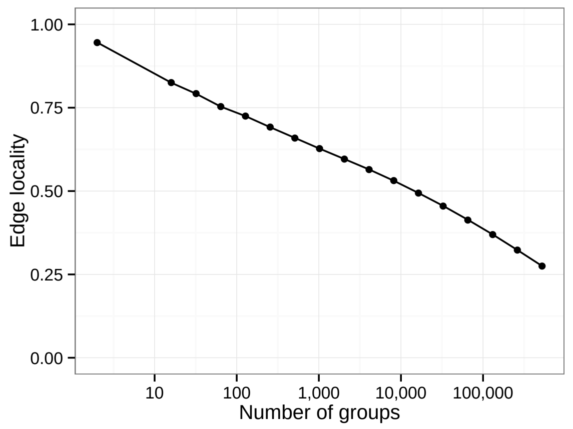

Figure 3: Edge locality (fraction of edges within groups) vs. the number of groups for Facebook's friendship graph.

multiple caches simultaneously. This could be a concern as the recovery of a cache failure would overwhelm the backing storage systems with excessive traffic and thus lead to severely degraded overall performance. However, our experience indicates that a failure of the main-memory caches within a cluster only causes a temporary load increase on the backend storage servers that stays within the normal operational load thresholds.

5.3 Operational observations

To get a sense of how access patterns of friends relate, we sampled well over 100 million accesses to TAO data records from access logs. We found that when two users access the same data record, there is a 15% chance they are friends. This is millions of times larger than the probability of two random users being friends. We conclude that co-locating the processing of friends' HTTP requests as much as possible is an effective strategy.

Figure 3 depicts edge locality vs. the number of groups used to partition the 1.5B+ Facebook users. Edge locality measures the fraction of "friend" edges connecting two users that are both assigned to the same group (thus, the goal of static assignment would be to maximize edge locality). It is not a surprise that edge locality decreases with the number of groups. Perhaps a bit more unexpected is the observation that edge locality remains reasonably large even when the number of groups increases significantly (e.g., >20% with 1 million groups); intuitively, this is because the friendship graph contains many small relatively dense communities. We chose the smallest number of groups that would satisfy our main requirement for dynamic assignment, namely to be able to balance the load by shifting only small amounts of traffic between front-end clusters. Repeating the process of assigning different numbers of groups into components offline and examining the resulting imbalance on known loads led us to use 21,000 groups on a few 10's of clusters; our group-to-component ratio is thus quite high.

The combination of new users being added to the system and changes to the friendship graph causes edge locality to degrade over time. We measured the decrease of edge locality from an initial, random assignment of users to one of 21,000 groups over the course of four weeks. We observed a nearly linear 0.1% decrease in edge locality per week. While small, we decided to update the routing assignment once a week so as to minimize a noticeable decrease in quality. At the same time, we did not observed a decrease in cache hit rate between updates, implying that 0.1% is a negligible amount. The decrease in edge locality implies that a longer update schedule would also be satisfactory, and that Social Hash can tolerate a long maintenance breakdown without altering Facebook's web traffic routing quality.

For the past three months, the Social Hash Table used for Facebook's routing has maintained an edge locality of over 50%, meaning half the friendships are within each of the 21,000 groups. This edge locality is slightly higher than the exploratory values shown in Figure 3, because we iterated longer in the graph partitioning algorithm on the production system than we did in the experiments from which we obtained the figure. The static assignment is well-balanced, with the largest group containing at most 0.8% more users than the average group. Each weekly update by the static assignment step resulted in around 1.5% of users switching groups from the previous assignment. All of these updates were suitably small to avoid noticeable increases in the cache miss rate when the updates were introduced into production.

5.4 Live traffic experiment

To measure the effectiveness of Social Hash-based HTTP routing optimization, we performed a live traffic experiment on two identical clusters with the same hardware, number of hosts and capacity constraints. These clusters are typical of what Facebook uses in production. Each cluster had many hundred TAO servers, which served the local web tier with cached social data.

For our experiment, we selected a set of groups randomly from the Social Hash Table. We then routed all HTTP requests from users assigned to these groups to one "test" cluster, while HTTP requests from a same number of other users were routed to the second, "control" cluster. Hence, the control cluster saw traffic with

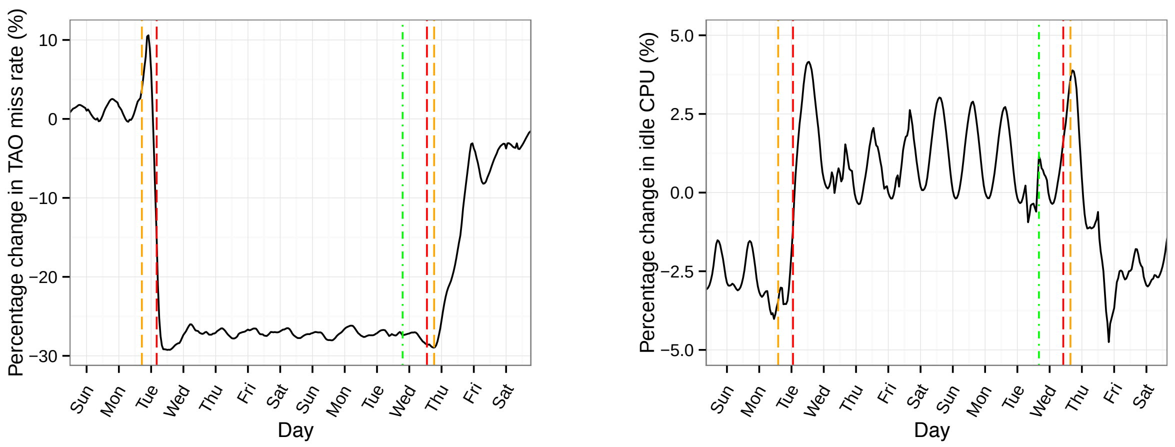

Figure 4: Percentage change in TAO miss rate (left, where lower is better) and CPU idle rate (right, where higher is better) on the Social Hash cluster relative to the cluster with random assignment. Area between red dashed lines: period of the test. Orange dashed lines: traffic shifts. Green dot-dash line: Social Hash Table is updated. The values on the days traffic was shifted (Tuesday and Wednesday, respectively) are not representative

attributes very similar to the traffic it received with the prior strategy: the traffic with the prior strategy was sampled from all users, while the traffic for the control cluster was sampled from all users except those associated with the test cluster. We ran the experiment for 10 days. During this time, operational changes that would affect hit rates on the two clusters were prevented.

The left hand side of Figure 4 shows the change in cache miss rate between the test and control clusters. It is evident that the miss rate drops by over 25% when assigning groups to a cluster as opposed to just users.

The right hand side of Figure 4 shows the change in average CPU idle rate between the test and the control cluster. The test cluster had up to 3% more idle time compared to the control cluster.

During the experiment, we updated the Social Hash Table by applying an updated static assignment. The time at which this occurred is shown with a vertical green dot-dash line. We note that the cache miss rate and the CPU idle time are not affected by the update, demonstrating that the transition process is smooth.

Figure 5 compares the daily working set size for TAO objects at both clusters. The daily working set of a cluster is the total size of all objects that were accessed by the TAO instance on that front-end cluster at least once during that day. The figure shows that the working set size dropped by as much as 8.3%.

We conclude from this experiment that Social Hash is effective at improving the efficiency of the cache for HTTP requests: fewer requests are sent to backend systems, and the hardware is utilized in a more efficient way.

6 Storage sharding

In this section, we describe in detail how we applied the Social Hash framework to sharded storage systems at Facebook. The assignment problem is to decide how to assign data records (the objects) to storage subsystems (the components).

6.1 Fanout vs. Latency

The objective function we optimize is *fanout*, the number of storage subsystems that must be contacted for multi-get queries. We argue and experimentally demonstrate that fanout is a suitable objective function, since lower fanout is closely correlated with lower latencies [7].

Multiget queries are typically forced to issue requests to multiple storage subsystems, and they do so in parallel. As such, the latency of a multi-get query is determined by the slowest request. By reducing fanout, the probability of encountering a request that is unexpectedly slower than the others is reduced, thus reducing the latency of the query. This is the fundamental argument for using fanout as the objective function for the assignment problem in the context of storage sharding. Another argument is that lower fanout reduces the connection overhead per data record.

To further elaborate the relevance of choosing fanout as the objective function, consider this abstract scenario. Suppose 1% of individual requests to storage servers incur a significant delay due to unanticipated system-specific issues (CPU thread scheduling delays, system

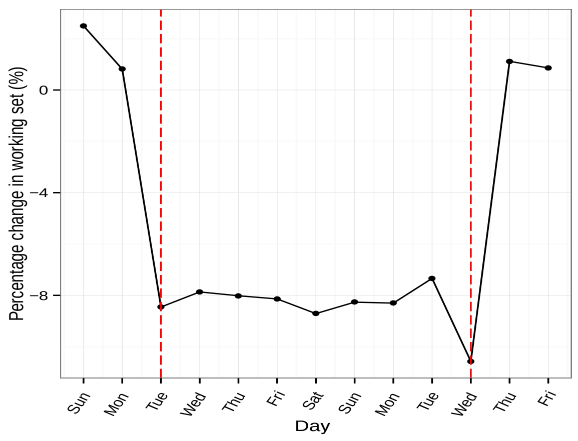

Figure 5: Percentage change in daily TAO working set size on the Social Hash cluster relative to the cluster with random assignment. The red dashed lines indicate the first and last days of the test where the test was running only during part of the day (so the values for these two days may not be representative).

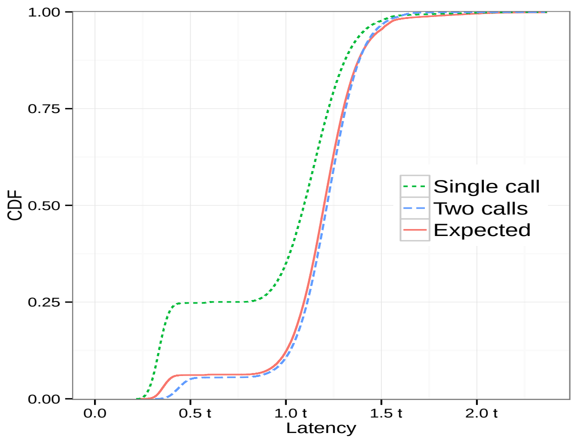

Figure 6: Cumulative distribution of latency for a single request, two requests in parallel, and the expected distribution from two independent samples from the single request distribution, where t is the average latency of a single call

.

interrupts, etc.). If a query must contact 10 storage servers, then one can calculate that the multi-get request has a 9.6% chance an individual sub-request will experience a significant delay. If the fanout can be reduced, one can reduce the probability of incurring the delay.

We ran a simple experiment to confirm our understanding of the relationship between fanout and latency. We issued trivial remote requests and measured the latency of a single request (fanout=1) and the latency of two requests sent in parallel (fanout=2). Figure 6 shows the cumulative latency distribution for both cases. A fanout of 1 results in lower latencies than a fanout of 2. If we calculate the expected distribution computed from two independent samples from the single request distribution, then the observed overall latency for two parallel requests matches the expected distribution quite nicely.

One possible caveat to our analysis of the relationship between fanout and latency is that reducing fanout generally increases the size of the largest request, which could increase latency. Fortunately, storage subsystems today have processors with many cores that can be exploited by the software to increase the parallelism in servicing a single, large request.

6.2 Implementation

For the static assignment we apply bipartite graph partitioning to minimize fanout. We create the bipartite graph from logs of queries from the dynamic operations of the social network.[3] The queries and data records accessed by the queries are represented by two types of vertices. A query vertex is edge-connected to a data vertex iff the query accesses the data record. The graph partitioning algorithm is then used to partition the data vertices into groups so as to minimize the average number of groups each query is connected to.

Clearly, most data needs to be replicated for fault tolerance (and other) reasons. Many systems at Facebook do this by organizing machines storing data into nonoverlapping sets, each containing each data record exactly once. We refer to such a set as a replica. Since assignment is independent between replicas, we will restrict our analysis to scenarios with just one replica.

6.3 Simplified sharding experiment

We consider the following simple experiment. We use 40 stripped down storage servers, where data is stored in a memory-based, key-value store. We assume that there is one data record per user. We run this setup in two configurations. In the first, “random” configuration, data records are distributed across the 40 storage servers using a hash function, which is a common practice. In the second, “social” configuration, we use our Social Hash framework to minimize fanout.

We then sampled a live traffic pattern, and issued the same set of queries to both configurations, and we measured fanout and latency. With the random configuration,

[3]In some cases, prior knowledge of which records each query must retrieve is sufficient to create the graph.

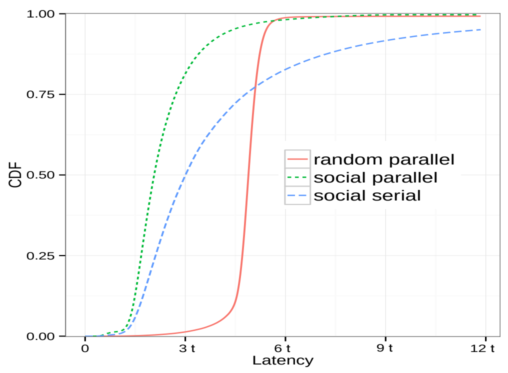

Figure 7: Cumulative latency distribution for fetching data of friends, where t is the average latency of a single call.

the queries needed to issue requests to 38.8 storage subsystems on average. With the social configuration, the queries needed to issue requests to only 9.9 storage subsystems on average. This decrease in fanout resulted in a 2.1X lower average latency for the queries.

The cumulative distribution of latencies for the random and social configurations are shown in Figure 7, where we also include the social configuration's latency distribution after disabling parallelism within each machine. Without parallelism, the average latency is still lower then with the random configuration, but only by 23%. Furthermore, the slowest 25% queries on the social configuration without parallelism exhibited substantially higher latencies than the 25% slowest queries on the random configuration. This figure confirms the importance of using parallelism within each system.

6.4 Operational observations

After we deployed storage sharding optimized with Social Hash to one of the graph databases at Facebook, containing thousands of storage servers, we found that measured latencies of queries decreased by over 50% on average, and CPU utilization also decreased by over 50%.

We attribute much of this improvement in performance to our method of assigning data records to groups, using graph partitioning on bi-partite graphs generated from prior queries. The solid line in Figure 8 shows the average fanout as a function of the number of groups when using our method. The dotted line shows the average fanout when using standard edge-cut optimization criteria on the (unipartite) friendship graph.

After analyzing expected load balance, we decided on a group-to-component ratio of 8; the dynamic assignment algorithm then selects which 8 groups to assign to the same storage subsystem, based on the historical load patterns. This allowed us to keep fanout small, while still

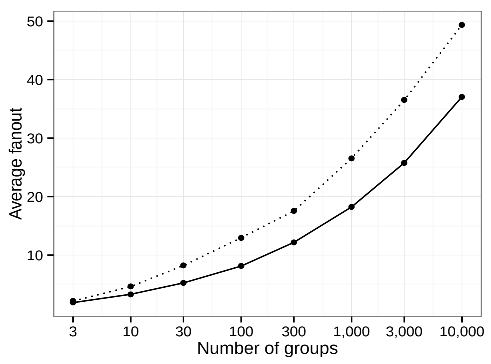

Figure 8: The average fanout versus number of groups on Facebook's friendship graph when using edge locality optimization (dotted curve) and our fanout optimization (solid curve), respectively.

being able to maintain good load balance.

In practice, fanout degrades over time. For the 40 group solution we used in the simplified application, we observed a fanout increase of about 2% on average over the course of a week. A single static assignment update sufficed to bring the fanout back to what it was previously, requiring only 1.85% of the data records to have to be moved. With such low impact, we decided static assignment updates were only necessary every few months, relying on dynamic assignment to move groups when necessary in between. Even then, we found that dynamic assignment updates were not necessary more than once a week on average. We used the same static assignment for all replicas, but made dynamic assignment decisions independently for each replica.

7 Related work

As discussed in Section 4.3, graph partitioning has an extensive literature, and our optimization objectives, edge locality and fanout, correspond to edge cut and hypergraph edge cut. A recent review of graph partitioning methods can be found online [3]. Many graph partitioning systems have been built and are available. For example, Metis [16, 18] is one that is frequently used.

A Giraph-based approach to graph partitioning called "Spinner" was recently announced [21]. Our work is distinct in that their application was optimizing batch processing systems, such as Giraph itself, via increased edge locality, and our graph partitioning system is embedded in the Social Hash framework.

Average fanout in a bipartite graph, when presented as a hypergraph, with vertices being one side of the bipartite graph, and hyper-edges representing the vertices from the other side, directly translates into the hypergraph partitioning problem. Hypergraph partitioning also has an extensive literature [4, 17], and one of the publicly available parallel solutions is PHG [9], which can be found in the Zoltan toolkit [8].

Partitioning online social networks has previously been used to improve performance of distributed systems. Ugander and Backstrom discuss partitioning large graphs to optimize Facebook infrastructure [33]. Stein considered a theoretical application of partitioning to Facebook infrastructure [29]. Tran and Zhang considered a multi-objective optimization problem based on edge cut motivated by read and write behaviors in online social networks [31, 32].

Other research has considered data replication in combination with partitioning for sharding data for online social networks. Pujol et al. studied low fanout configurations via replication of data between hosts [25] and Wang et al. suggested minimizing fan-out by random replication and query optimization [36]. Nguyen et al. considered how to place additional replicas of users given a fixed initial assignment of users to servers [22, 30].

Dean and Barroso [7] investigated the effect of latency variability on fanout queries in distributed systems, and suggested several means to reduce its influence. Jeon et al. [14] argued for the necessity of parallelizing execution of large requests, in order to tame latencies.

Our contribution differs from these lines of research by presenting a realized framework integrated into production systems at Facebook. A production application to online social networks is provided by Huang et al. who describe improving infrastructure performance for Renren through a combination of graph partitioning and data replication methods [13]. Sharding has been considered for distributed social network databases by Nicoara, et al. who propose Hermes [23].

8 Concluding Remarks

We introduced the Social Hash framework for producing, serving, and maintaining assignments of objects to components in distributed systems. The framework was designed for optimizing operations on large social networks, such as Facebook's Social Graph. A key aspect of the framework is how optimization is decoupled from dynamic adaptation, through a two-level scheme that uses graph partitioning for optimization at the first level and dynamic assignment at the second level. The first level leverages the structure of the social network and its usage patterns, while the second level adapts to changes in the data, its access patterns and the infrastructure.

We demonstrated the effectiveness of the Social Hash framework with the HTTP request routing and storage sharding applications. For the former, Social Hash was able to decrease the cache miss rate by 25%, and for the latter, it was able to cut the average response time in half, as measured on the live Facebook system with live traffic production workloads. The approaches we took with both applications was, to the best of our knowledge, novel; i.e., graph partitioning the Social Graph to optimize HTTP request routing, and using query history to construct bipartite graphs that are then partitioned to optimize storage sharding.

Our approach has some limitations. It was designed in the context of optimizing online social networks and hence will not be suitable for every distributed system. To be successful, both the static and dynamic assignment steps rely on certain characteristics, which tend to be fulfilled by social networks. For the static step, the underlying graph must be conducive to partitioning, and the graph must be reasonably sparse so that the partitioning is computationally tractable; social graphs almost always meet those characteristics. The social graph cannot be changing too rapidly; otherwise the optimized static assignment will be obsolete too quickly and the attendant exception handling becomes too computationally complex. For the dynamic step, we assume that the workload and the infrastructure does not change too rapidly.

While we have been able to obtain impressive efficiency gains using the Social Hash framework, we believe there is much room for further improvement. We are currently: (*i*) working on improving the performance of our graph partitioning algorithms, (*ii*) considering using historical query patterns and bi-partite graph partitioning to further improve cache miss rates, (*iii*) incorporating geo-locality considerations for our HTTP routing optimizations, and (*iv*) incorporating alternative replication schemes for further reducing fanout in storage sharded systems.

Acknowledgements

We would like to thank Tony Savor, Kenny Lau, Venkat Venkataramani and Avery Ching for their support, Alex Laslavic, Praveen Kumar, Jan Jezabek, Alexander Ramirez, Omry Yadan, Michael Paleczny, Jianming Wu, Chunhui Zhu, Deyang Zhao and Pavan Athivarapu for helping integrate with Facebook systems, Sanjeev Kumar, Kaushik Veeraraghavan, Dionysis Logothetis, Romain Thibaux and Rajesh Nishtala for their feedback on early drafts, Badr El-Said and Laxman Dhulipala for their contributions to the framework, and Dimitris Achlioptas for discussions on graph partitioning. We would also like to thank the reviewers for their constructive and helpful comments.

References

[1] Apache Giraph. http://giraph.apache.org/.

[2] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook's distributed data store for the Social Graph. In *Proc. 2013 USENIX Annual Technical Conference (USENIX ATC'13)*, pages 49–60, 2013.

[3] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. Recent advances in graph partitioning. *CoRR*, abs/1311.3144, 2013.

[4] U. V. Catalyurek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Trans. on Parallel and Distributed Systems*, 10(7):673–693, 1999.

[5] C. Chevalier and F. Pellegrini. PT-Scotch: A tool for efficient parallel graph ordering. *CoRR*, abs/0907.1375, 2009.

[6] R. Cohen, L. Katzi, and D. Raz. An efficient approximation for the generalized assignment problem. *Information Processing Letters*, 100(4):162–166, 2006.

[7] J. Dean and L. A. Barroso. The tail at scale. *CACM*, 56(2):74–80, Feb. 2013.

[8] K. Devine, E. Boman, L. Riesen, U. Catalyurek, and C. Chevalier. Getting started with Zoltan: A short tutorial. In *Proc. 2009 Dagstuhl Seminar on Combinatorial Scientific Computing*, 2009. Also available as Sandia National Labs Tech Report SAND2009-0578C.

[9] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and U. V. Catalyurek. Parallel hypergraph partitioning for scientific computing. In *Proc. Intl. Parallel and Distributed Processing Symposium (IPDPS)*, pages 10–20, 2006.

[10] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proc. 10th Symp. on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, 2012.

[11] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *Proc. 11th Symp. on Operating Systems Design and Implementation (OSDI 14)*, pages 599–613, Broomfield, CO, Oct. 2014.

[12] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li. An analysis of facebook photo caching. In *Proc. 24th Symp. on Operating Systems Principles (SOSP'13*.

[13] Y. Huang, Q. Deng, and Y. Zhu. Differentiating your friends for scaling online social networks. In *Proc. IEEE Intl. Conf. on Cluster Computing (CLUSTER'12)*, pages 411–419, Sept 2012.

[14] M. Jeon, S. Kim, S.-w. Hwang, Y. He, S. Elnikety, A. L. Cox, and S. Rixner. Predictive parallelization: Taming tail latencies in Web search. In *Proc. 37th Intl. ACM SIGIR Conference on Research & Development in Information Retrieval (SIGIR'14)*, pages 253–262, 2014.

[15] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proc. 29th Annual ACM Symp. on Theory of Computing (STOC'97)*, pages 654–663, 1997.

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

[17] G. Karypis and V. Kumar. Multilevel k-way hypergraph partitioning. *VLSI design*, 11(3):285–300, 2000.

[18] D. Lasalle and G. Karypis. Multi-threaded graph partitioning. In *Proc. IEEE 27th Intl. Symp. on Parallel and Distributed Processing (IPDPS'13)*, pages 225–236, 2013.

[19] H. Lin, K. Sun, S. Zhao, and Y. Han. Feedback-control-based performance regulation for multi-tenant applications. In *Proc. 15th Intl. Conf. on Parallel and Distributed Systems (ICPADS'09)*, pages 134–141, Dec 2009.

[20] 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 *Proc. ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'10)*, pages 135–146, 2010.

[21] C. Martella, D. Logothetis, and G. Siganos. Spinner: Scalable graph partitioning for the cloud. *CoRR*, abs/1404.3861, 2014.

[22] K. Nguyen, C. Pham, D. Tran, F. Zhang, et al. Preserving social locality in data replication for online social networks. In *Proc. 31st Intl. Conf. on Distributed Computing Systems Workshops (ICDCSW)*, pages 129–133, 2011.

[23] D. Nicoara, S. Kamali, K. Daudjee, and L. Chen. Hermes: Dynamic partitioning for distributed social network graph databases. In *Proc. 18th Intl. Conf. on Extending Database Technology (EDBT'15)*, pages 25–36, Mar. 2015.

[24] 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 *Proc. 10th USENIX Conf. on Networked Systems Design and Implementation (NSDI'13)*, pages 385–398, 2013.

[25] J. M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez. The little engine(s) that could: Scaling online social networks. *SIGCOMM ComputĊommun. Rev.*, 40(4):375–386, Aug. 2010.

[26] D. Shue, M. J. Freedman, and A. Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *Proc. 10th USENIX Symp. on Operating Systems Design and Implementation (OSDI 12)*, pages 349–362, 2012.

[27] D. D. C. Shue. *Multi-tenant Resource Allocation For Shared Cloud Storage*. PhD thesis, Princeton University, 2014.

[28] Y. Song, Y. Sun, and W. Shi. A two-tiered on-demand resource allocation mechanism for VM-based data centers. *IEEE Trans. on Services Computing*, 6(1):116–129, 2013.

[29] D. Stein. Partitioning social networks for data locality on a memory budget. Master's thesis, University of Illinois, Urbana-Champaign, 2012.

[30] D. A. Tran, K. Nguyen, and C. Pham. S-CLONE: Socially-aware data replication for social networks. *Computer Networks*, 56(7):2001–2013, 2012.

[31] D. A. Tran and T. Zhang. Socially aware data partitioning for distributed storage of social data. In *Proc. IFIP Networking Conference*, pages 1–9, May 2013.

[32] D. A. Tran and T. Zhang. S-PUT: An EA-based framework for socially aware data partitioning. *Computer Networks*, 75:504–518, Dec. 2014.

[33] J. Ugander and L. Backstrom. Balanced label propagation for partitioning massive graphs. In *Proc. 6th ACM Intl. Conf. on Web Search and Data Mining (WSDM-13)*, pages 507–516, 2013.

[34] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow. The anatomy of the Facebook Social Graph. *CoRR*, abs/1111.4503, 2011.

[35] V. Venkataramani, Z. Amsden, N. Bronson, G. Cabrera III, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, J. Hoon, et al. TAO: How Facebook serves the Social Graph. In *Proc. 2012 ACM SIGMOD Intl. Conf. on Management of Data*, pages 791–792, 2012.

[36] R. Wang, C. Conrad, and S. Shah. Using set cover to optimize a large-scale low latency distributed graph. In *Proc 5th USENIX Workshop on Hot Topics in Cloud Computing*, 2013.