An Analysis of Facebook Photo Caching

Qi Huang*, Ken Birman*, Robbert van Renesse*, Wyatt Lloyd†‡, Sanjeev Kumar‡, Harry C. Li‡
*Cornell University, †Princeton University, ‡Facebook Inc.

Abstract

This paper examines the workload of Facebook's photo-serving stack and the effectiveness of the many layers of caching it employs. Facebook's image-management infrastructure is complex and geographically distributed. It includes browser caches on end-user systems, Edge Caches at ~20 PoPs, an Origin Cache, and for some kinds of images, additional caching via Akamai. The underlying image storage layer is widely distributed, and includes multiple data centers.

We instrumented every Facebook-controlled layer of the stack and sampled the resulting event stream to obtain traces covering over 77 million requests for more than 1 million unique photos. This permits us to study traffic patterns, cache access patterns, geolocation of clients and servers, and to explore correlation between properties of the content and accesses. Our results (1) quantify the overall traffic percentages served by different layers: 65.5% browser cache, 20.0% Edge Cache, 4.6% Origin Cache, and 9.9% Backend storage, (2) reveal that a significant portion of photo requests are routed to remote PoPs and data centers as a consequence both of load-balancing and peering policy, (3) demonstrate the potential performance benefits of coordinating Edge Caches and adopting S4LRU eviction algorithms at both Edge and Origin layers, and (4) show that the popularity of photos is highly dependent on content age and conditionally dependent on the social-networking metrics we considered.

1 Introduction

The popularity of social networks has driven a dramatic surge in the amount of user-created content stored by Internet portals. As a result, the effectiveness of the stacks that store and deliver binary large objects (blobs) has become an important issue for the social network provider community [2, 3]. While there are many forms of digital content, media binaries such as photos and videos are the most prevalent and will be our focus here.

Our paper explores the dynamics of the full Facebook photo-serving stack, from the client browser to Facebook's Haystack storage server, looking both at the performance of each layer and at interactions between multiple system layers. As in many other settings [1, 7, 8, 11, 13, 17, 18, 19, 22, 23], the goal of this study is to gain insights that can inform design decisions for future content storage and delivery systems. Specifically, we ask (1) how much of the access traffic is ultimately served by the Backend storage server, as opposed to the many caching layers between the browser and the Backend, (2) how requests travel through the overall photo-serving stack, (3) how different cache sizes and eviction algorithms would affect the current performance, and (4) what object meta data is most predictive of subsequent access patterns.

Our study addresses these questions by collecting and correlating access records from multiple layers of the Facebook Internet hierarchy between clients and Backend storage servers. The instrumented components include client browsers running on all desktops and laptops accessing the social network website, all Edge cache hosts deployed at geographically distributed points of presence (PoP), the Origin cache in US data centers, and Backend servers residing in US data centers. This enabled us to study the traffic distribution at each layer, and the relationship between the events observed and such factors as cache effects, geographical location (for client, Edge PoP and data center) and content properties such as content age and the owner's social connectivity. This data set also enables us to simulate caching performance with various cache sizes and eviction algorithms. We focus on what we identified as key questions in shaping a new generation of content-serving infrastructure solutions.

1. To the best of our knowledge, our paper is the first study to examine an entire Internet image-serving infrastructure at a massive scale.
2. By quantifying layer-by-layer cache effectiveness, we find that browser caches, Edge caches and the Origin cache handle an aggregated 90% of the traffic. For the most-popular 0.03% of content, cache hit rates neared 100%. This narrow but high success

SOSP'13, Nov. 3–6, 2013, Farmington, Pennsylvania, USA.
ACM 978-1-4503-2388-8/13/11.
http://dx.doi.org/10.1145/2517349.2522722

This version fixes a typo in section 4 that is present in the official conference version. Updated: 11/12/2013

rate reshapes the load patterns for Backend servers, which see approximately Zipfian traffic but with Zipf coefficient α diminishing deeper in the stack.
3. By looking at geographical traffic flow from clients to Backend, we find that content is often served across a large distance rather than locally.
4. We identify opportunities to improve cache hit ratios using geographic-scale collaborative caching at Edge servers, and by adopting advanced eviction algorithms such as S4LRU in the Edge and Origin.
5. By examining the relationship between image access and associated meta-data, we find that content popularity rapidly drops with age following a Pareto distribution and is conditionally dependent on the owner's social connectivity.

The paper is organized as follows. Section 2 presents an overview of the Facebook photo serving-stack, highlighting our instrumentation points. Section 3 describes our sampling methodology. After giving a high level overview of the workload characteristics and current caching performance in Section 4, Sections 5, 6, and 7 further break down the analysis in three categories: geographical traffic distribution, potential improvements, and traffic association with content age and the content owners' social connectivity. Related work is discussed in Section 8 and we conclude in Section 9.

2 Facebook's Photo-Serving Stack

As today's largest social-networking provider, Facebook stores and serves billions of photos on behalf of users. To deliver this content efficiently, with high availability and low latency, Facebook operates a massive photo-serving stack distributed at geographic scale. The sheer size of the resulting infrastructure and the high loads it continuously serves make it challenging to instrument. At a typical moment in time there may be hundreds of millions of clients interacting with Facebook Edge Caches. These are backed by Origin Cache and Haystack storage systems running in data centers located worldwide. To maximize availability and give Facebook's routing infrastructure as much freedom as possible, all of these components are capable of responding to any photo-access request. This architecture and the full life cycle of a photo request are shown in Figure 1; shaded elements designate the components accessible in this study.

2.1 The Facebook Photo-Caching Stack

When a user receives an HTML file from Facebook's front-end web servers (step 1), a browser or mobile client app begins downloading photos based on the embedded URLs in that file. These URLs are custom-generated by web servers to control traffic distribution across the serving stack: they include a unique photo identifier, specify the display dimensions of the image, and encode the *fetch path*, which specifies where a request that misses at each layer of cache should be directed next. Once there is a hit at any layer, the photo is sent back in reverse along the fetch path and then returned to the client.

There are two parallel stacks that cache photos, one run by Akamai and one by Facebook. For this study, we focus on accesses originating at locations for which Facebook's infrastructure serves all requests, ensuring that the data reported here has no bias associated with our lack of instrumentation for the Akamai stack. The remainder of this section describes Facebook's stack.

There are three layers of caches in front of the backend servers that store the actual photos. These caches, ordered by their proximity to clients, are the client browser's cache, an Edge Cache, and the Origin Cache.

Browser The first cache layer is in the client's browser. The typical browser cache is co-located with the client, uses an in-memory hash table to test for existence in the cache, stores objects on disk, and uses the LRU eviction algorithm. There are, however, many variations on the typical browser cache. If a request misses at the browser cache, the browser sends an HTTP request out to the Internet (step 2). The fetch path dictates whether that request is sent to the Akamai CDN or the Facebook Edge.

Edge The Facebook Edge is comprised of a set of Edge Caches that each run inside points of presence (PoPs) close to end users. There are a small number of Edge Caches spread across the US that all function independently. (As of this study there are nine high-volume Edge Caches, though this number is growing and they are being expanded internationally.) The particular Edge Cache that a request encounters is determined by its fetch path. Each Edge Cache has an in-memory hash table that holds metadata about stored photos and large amounts of flash memory that store the actual photos [10]. If a request hits, it is retrieved from the flash and returned to the client browser. If it misses, the photo is fetched from Facebook's Origin Cache (step 3) and inserted into this Edge Cache. The Edge caches currently all use a FIFO cache replacement policy.

Origin Requests are routed from Edge Caches to servers in the Origin Cache using a hash mapping based on the unique id of the photo being accessed. Like the Edge Caches, each Origin Cache server has an in-memory hash table that holds metadata about stored photos and a large flash memory that stores the actual photos. It uses a FIFO eviction policy.

Haystack The backend, or Haystack, layer is accessed when there is a miss in the Origin cache. Because Origin servers are co-located with storage servers, the image can often be retrieved from a local Haystack server (step 4). If the local copy is held by an overloaded storage server or is

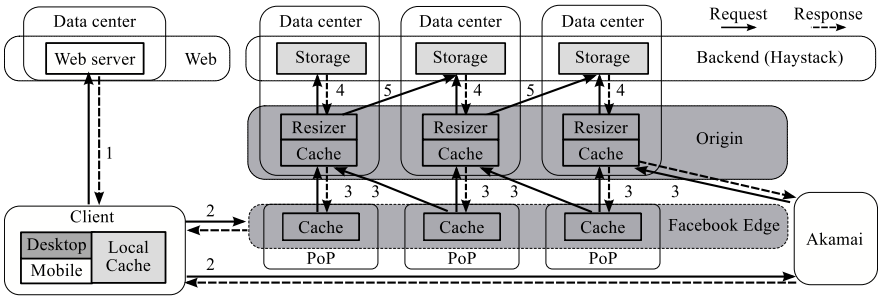


Figure 1: Facebook photo serving stack: components are linked to show the photo retrieval work-flow. *Desktop* and *Mobile* clients initiate request traffic, which routes either directly to the Facebook Edge or via Akamai depending on the fetch path. The Origin Cache collects traffic from both paths, serving images from its cache and resizing them if needed. The Haystack backend holds the actual image content. Shading highlights components tracked directly (dark) or indirectly (light) in our measurement infrastructure.

unavailable due to system failures, maintenance, or some other issue, the Origin will instead fetch the information from a local replica if one is available. Should there be no locally available replica, the Origin redirects the request to a remote data center.

Haystack resides at the lowest level of the photo serving stack and uses a compact blob representation, storing images within larger segments that are kept on log-structured volumes. The architecture is optimized to minimize I/O: the system keeps photo volume ids and offsets in memory, performing a single seek and a single disk read to retrieve desired data [2].

2.2 Photo Transformations

Facebook serves photos in many different forms to many different users. For instance, a desktop user with a big window will see larger photos than a desktop users with a smaller window who in turn sees larger photos than a mobile user. The resizing and cropping of photos complicates the simple picture of the caching stack we have painted thus far.

In the current architecture all transformations are done between the backend and caching layers, and thus all transformations of an image are treated as independent blobs. As a result, a single cache may have many transformation of the same photo. These transformations are done by Resizers (shown closest to the backend server in Figure 1), which are co-located with Origin Cache servers. The Resizers also transform photos that are requested by the Akamai CDN, though the results of those transformations are not stored in the Origin Cache.

When photos are first uploaded to Facebook they are scaled to a small number of common, known sizes, and copies at each of these sizes are saved to the backend Haystack machines. Requests for photos include not only the exact size and cropping requested, but also the original size from which it should be derived. The caching infrastructure treats all of these transformed and cropped photos as separate objects. One opportunity created by our instrumentation is that it lets us explore hypothetical alternatives to this architecture. For example, we evaluated the impact of a redesign that pushes all resizing actions to the client systems in Section 6.

2.3 Objective of the Caching Stack

The goals of the Facebook photo-caching stack differ by layer. The primary goal of the Edge cache is to reduce bandwidth between the Edge and Origin datacenters, whereas the main goal for other caches is *traffic sheltering* for its backend Haystack servers, which are I/O bound. This prioritization drives a number of decisions throughout the stack. For example, Facebook opted to treat the Origin cache as a single entity spread across multiple data centers. Doing so maximizes hit rate, and thus the degree of traffic sheltering, even though the design sometimes requires Edge Caches on the East Coast to request data from Origin Cache servers on the West Coast, which increases latency.

3 Methodology

We instrumented Facebook's photo-serving infrastructure, gathered a month-long trace, and then analyzed that trace using batch processing. This section presents our data gathering process, explains our sampling methodology, and addresses privacy considerations.

3.1 Multi-Point Data Collection

In order to track events through all the layers of the Facebook stack it is necessary to start by independently instrumenting the various components of the stack, collecting a representative sample in a manner that permits correlation of events related to the same request even when they occur at widely distributed locations in the hierarchy (Figure 1). The ability to correlate events across different layers provides new types of insights:

- **Traffic sheltering:** We are able to quantify the degree to which each layer of cache shelters the systems downstream from it. Our data set enables us to distinguish

hits, misses, and the corresponding network traffic from the browser caches resident with millions of users down through the Edge Caches, the Origin Cache, and finally to the Backend servers. This type of analysis would not be possible with instrumentation solely at the browser or on the Facebook Edge.

- **Geographical flow:** We can map the geographical flow of requests as they are routed from clients to the layer that resolves them. In some cases requests follow surprisingly remote routes: for example, we found that a significant percentage of requests are routed across the US. Our methodology enables us to evaluate the effectiveness of geoscale load balancing and of the caching hierarchy in light of the observed pattern of traffic.

Client To track requests with minimal code changes, we limit our instrumentation to desktop clients and exclude mobile platforms: (1) all browsers use the same web code base, hence there is no need to write separate code for different platforms; and (2) after a code rollout through Facebook's web servers, all desktop users will start running that new code; an app update takes effect far more slowly. Our client-side component is a fragment of javascript that records when browsers load specific photos that are selected based on a tunable sampling rate. Periodically, the javascript uploads its records to a remote web server and then deletes them locally.

The web servers aggregate results from multiple clients before reporting them to Scribe [15], a distributed logging service. Because our instrumentation has no visibility into the Akamai infrastructure, we limit data collection to requests for which Facebook serves all traffic; selected to generate a fully representative workload. By correlating the client logs with the logs collected on the Edge cache, we can now trace requests through the entire system.

Edge Cache Much like the client systems, each Edge host reports sampled events to Scribe whenever an HTTP response is sent back to the client. This allows us to learn whether the associated request is a hit or a miss on the Edge, along with other details. When a miss happens, the downstream protocol requires that the hit/miss status at Origin servers should also be sent back to the Edge. The report from the Edge cache contains all this information.

Origin Cache While the Edge trace already contains the hit/miss status at Origin servers, it does not provide details about communication between the Origin servers and the Backend. Therefore, we also have each Origin host report sampled events to Scribe when a request to the Backend is completed.

To ensure that the same photos are sampled in all three traces, our sampling strategy is based on hashing: we sample a tunable percentage of events by means of a deterministic test on the photoId. We explore this further in Section 3.3.

Scribe aggregates logs and loads them into Hive [21], Facebook's data warehouse. Scripts then perform statistical analyses yielding the graphs shown below.

3.2 Correlating Requests

By correlating traces between the different layers of the stack we accomplish several goals. First, we can ask what percentage of requests result in cache hits within the client browser. Additionally, we can study the paths taken by individual requests as they work their way down the stack. Our task would be trivial if we could add unique request-IDs to every photo request at the browser and then piggyback that information on the request as it travels along the stack, such an approach would be disruptive to the existing Facebook code base. This forces us to detect correlations in ways that are sometimes indirect, and that are accurate but not always perfectly so.

The first challenge arises in the client, where the detection of client-side cache hits is complicated by a technicality: although we do know which URLs are accessed, if a photo request is served by the browser cache our Javascript instrumentation has no way to determine that this was the case. For example, we can't infer that a local cache hit occured by measuring the time delay between photo fetch and completion: some clients are so close to Edge Caches that an Edge response could be faster than the local disk. Accordingly, we infer the aggregated cache performance for client object requests by comparing the number of requests seen at the browser with the number seen in the Edge for the same URL.

To determine the geographical flow between clients and PoPs, we correlate browser traces and Edge traces on a per request basis. If a client requests a URL and then an Edge Cache receives a request for that URL from the client's IP address, then we assume a miss in the browser cache triggered an Edge request. If the client issues multiple requests for a URL in a short time period and there is one request to an Edge Cache, then we assume the first request was a miss at browser but all subsequent requests were hits.

Correlating Backend-served requests in the Edge trace with requests between the Origin and Backend layers is relatively easy because they have a one-to-one mapping. If a request for a URL is satisfied after an Origin miss, and a request for the same URL occurs between the same Origin host and some Backend server, then we assume they are correlated. If the same URL causes multiple misses at the same Origin host, we align the requests with Origin requests to the Backend in timestamp order.

3.3 Sampling Bias

To avoid affecting performance, we sample requests instead of logging them all. Two sampling strategies were considered: (1) sampling requests randomly, (2) sam-

pling focused on some subset of photos selected by a deterministic test on photoId. We chose the latter for two reasons:

- **Fair coverage of unpopular photos:** Sampling based on the photo identifier enables us to avoid bias in favor of transiently popular items. A biased trace could lead to inflated cache performance results because popular items are likely stored in cache.
- **Cross stack analysis:** By using a single deterministic sampling rule that depends only on the unique photoId, we can capture and correlate events occurring at different layers.

A potential disadvantage of this approach is that because photo-access workload is Zipfian, a random hashing scheme could collect different proportions of photos from different popularity levels. This can cause the estimated cache performance to be inflated or deflated, reflecting an overly high or low coverage of popular objects. To quantify the degree of bias in our traces, we further downsampled our trace to two separate data sets, each of which covers 10% of our original photoIds. While one set inflates the hit ratios at browser, Edge and Origin caches by 3.6%, 2% and 0.4%, the other set deflates the hit ratios at browser and Edge caches 0.5% and 4.3%, resp. Overall, the browser and Edge cache performance are more sensitive to workload selection based on photoIds than the Origin. Comparing to Facebook's live monitoring data, which has a higher sampling ratio but lower sampling duration, our reported Edge hit ratio is lower by about 5% and our Origin hit ratio is about the same. We concluded that our sampling scheme is reasonably unbiased.

3.4 Privacy Preservation

We took a series of steps to preserve the privacy of Facebook users. First, all raw data collected for this study was kept within the Facebook data warehouse (which lives behind a company firewall) and deleted within 90 days. Second, our data collection logic and analysis pipelines were heavily reviewed by Facebook employees to ensure compliance with Facebook privacy commitments. Our analysis does not access image contents or users profiles (however, we do sample some meta-information: photo size, age and the owner's number of followers). Finally, as noted earlier, our data collection scheme is randomized and based on photoId, not user-id; as such, it only yields aggregated statistics for a cut across the total set of photos accessed during our study period.

4 Workload Characteristics

Our analysis examines more than 70 TB of data, all corresponding to client-initiated requests that traversed the Facebook photo-serving stack during a one-month sampling period. Table 1 shows summary statistics for our trace. Our trace includes over 77M requests from 13.2M user browsers for more than 1.3M unique photos. Our analysis begins with a high level characterization of the trace, and then dives deeper in the sections that follow.

Table 1 gives the number of requests and hits at each successive layer. Of the 77.2M browser requests, 50.6M are satisfied by browser caches (65.5%), 15.4M by the Edge Caches (20.0%), 3.6M by the Origin cache (4.6%), and 7.6M by the Backend (9.9%). There is an enormous working set and the photo access distribution is long-tailed with a significant percentage of accesses are directed to low-popularity photos. Looking next at bytes being transferred at different layers, we see that among 492.2GB of photo traffic being delivered to the client, $492.2 - 250.6 = 241.6$GB were served by Edge caches, $250.6 - 187.2 = 63.4$GB were served by the Origin and 187.2GB were derived from Backend fetches, which corresponds to over 456GB of traffic between the Origin and Backend before resizing.

This table also gives the hit ratio at each caching layer. The 65.5% hit ratio at client browser caches provides significant traffic sheltering to Facebook's infrastructure. Without the browser caches, requests to the Edge Caches would approximately triple. The Edge Caches have a 58.0% hit ratio and this also provides significant traffic sheltering to downstream infrastructure: if the Edge Caches were removed, requests to the Origin Cache and the bandwidth required from it would more than double. Although the 31.8% hit ratio achieved by the Origin Cache is the lowest among the caches present in the Facebook stack, any hits that do occur at this level reduce costs in the storage layer and eliminate backend network cost, justifying deployment of a cache at this layer.

Recall that each size of a photo is a distinct object for caching purposes. The *Photos w/o size* row ignores the size distinctions and presents the number of distinct underlying photos being requested. The number of distinct photos requests at each tier remains relatively constant, about 1.3M. This agrees with our intuition about caches: they are heavily populated with popular content represented at various sizes, but still comprise just a small percentage of the unique photos accessed in any period. For the large numbers of unpopular photos, cache misses are common. The *Photos w/ size* row breaks these figures down, showing how many photos are requested at each layer, but treating each distinct size of an image as a separate photo. While the number decreases as we traverse the stack, the biggest change occurs in the Origin tier, suggesting that requests for new photo sizes are a source of misses. The Haystack Backend maintains each photo at four commonly-requested sizes, which helps explain why the count seen in the last column can exceed the number of unique photos accessed: for requests corresponding to these four sizes, there is no need to undertake a (costly) resizing computation.

	Inside browser	Edge caches	Origin cache	Backend (Haystack)
Photo requests	77,155,557	26,589,471	11,160,180	7,606,375
Hits	50,566,086	15,429,291	3,553,805	7,606,375
% of traffic served	65.5%	20.0%	4.6%	9.9%
Hit ratio	65.5%	58.0%	31.8%	N/A
Photos w/o size	1,384,453	1,301,972	1,300,476	1,295,938
Photos w/ size	2,678,443	2,496,512	2,484,155	1,531,339
Users	13,197,196	N/A	N/A	N/A
Client IPs	12,341,785	11,083,418	1,193	1,643
Client geolocations	24,297	23,065	24	4
Bytes transferred	N/A	492.2 GB	250.6 GB	456.5 GB (187.2 GB after resizing)

Table 1: Workload characteristics: broken down by different layers across the photo-serving stack where traffic was observed; *Client IPs* refers to the number of distinct IP addresses identified on the requester side at each layer, and *Client geolocations* refers to the number of distinct geographical regions to which those IPs map.

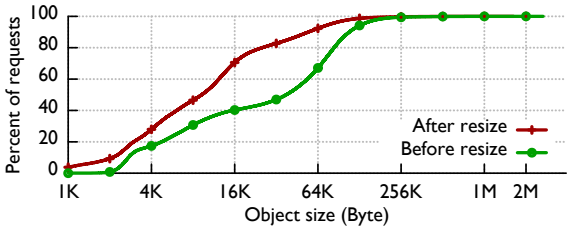


Figure 2: Cumulative distribution function (CDF) on object size being transferred through the Origin.

The size distribution of transferred photos depends upon the location at which traffic is observed. Figure 2 illustrates the cumulative distribution of object size transferred before and after going through the Origin Cache for all Backend fetches. After photos are resized, the percentage of transferred objects smaller than 32KB increases from 47% to over 80%.

The rows labeled *Client IPs* and *Client geolocations* in Table 1 offer measurements of coverage of our overall instrumentation stack. For example, we see that more than 12 million distinct client IP addresses covering over 24 thousand geolocations (cities or towns) used the system, that 1,193 distinct Facebook Edge caches were tracked, etc. As we move from left to right through the stack we see traffic aggregate from massive numbers of clients to a moderate scale of Edge regions and finally to a small number of data centers. Section 5 undertakes a detailed analysis of geolocation phenomena.

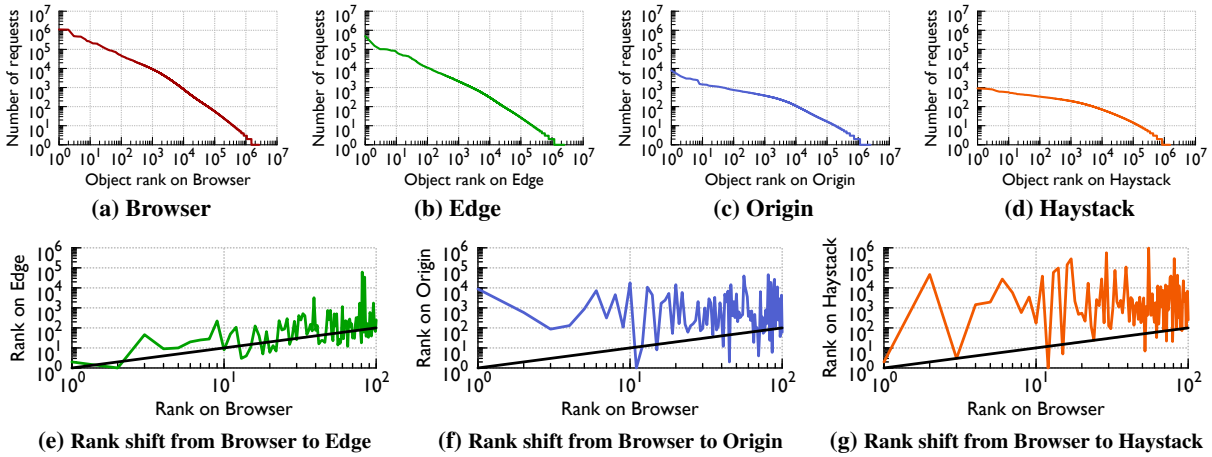
4.1 Popularity Distribution

A natural way to quantify object popularity is by tracking the number of repeated requests for each photo. For Haystack we consider each stored common sized photo as an object. For other layers we treat each resized variant as an object distinct from the underlying photo. By knowing the number of requests for each object, we can then explore the significance of object popularity in determining Facebook's caching performance. Prior studies of web traffic found that object popularity follows a Zipfian distribution [4]. Our study of browser access patterns supports this finding. However, at deeper levels of the photo stack, the distribution flattens, remaining mostly Zipf-like (with decreasing Zipf-coefficient α at each level), but increasingly distorted at the head and tail. By the time we reach the Haystack Backend, the distribution more closely resembles a *stretched exponential* distribution [12].

Figures 3a, 3b, 3c and 3d show the number of requests to each unique photo blob as measured at different layers, ordered by popularity rank in a log-log scale. Because each layer absorbs requests to some subset of items, the rank of each blob can change if popularity is recomputed layer by layer. To capture this effect visually, we plotted the rank shift, comparing popularity in the browser ranking to that seen in the Edge (Figure 3e), in the Origin tier (Figure 3f) and in Haystack (Figure 3g). In these graphs, the x-axis is the rank of a particular photo blob as ordered on browsers, while the y-axis gives the rank on the indicated layer for that same photo object. The type of blob is decided by the indicated layer. Had there been no rank shift, these graphs would match the straight black line seen in the background.

As seen in these plots, item popularity distributions at all layers are approximately Zipfian (Figures 3a-3d). However, level by level, item popularities shift, especially for the most popular 100 photo blobs in the Edge's popularity ranking. For example, when we look at the rank shift between browser and Edge (Figure 3e), where 3 top-10 objects dropped out of the highest-popularity ranking and a substantial fraction of the 10th-100th most popular objects dropped to around 1000th and even 10000th on the Edge ("upward" spikes correspond to items that were more popular in the browser ranking than in the Edge ranking).

As traffic tunnels deeper into the stack and reaches first the Origin Cache and then Haystack, millions of requests



(a) Browser **(b) Edge** **(c) Origin** **(d) Haystack**

(e) Rank shift from Browser to Edge **(f) Rank shift from Browser to Origin** **(g) Rank shift from Browser to Haystack**

Figure 3: Popularity distribution. Top: Number of requests to unique photos at each layer, ordered from the most popular to the least. Bottom: a comparison of popularity of items in each layer to popularity in the client browser, with the exact match shown as a straight line. Shifting popularity rankings are thus evident as spikes. Notice in (a)-(d) that as we move deeper into the stack, these distributions flatten in a significant way.

are served by each caching layer, hence the number of requests for popular images is steadily reduced. This explains why the distributions seen on the Edge, Origin and Haystack remain approximately Zipfian, but the Zipf coefficient, α, becomes smaller: the stream is becoming steadily less cacheable. Yet certain items are still being cached effectively, as seen by the dramatic popularity-rank shifts as we progress through the stack.

4.2 Hit Ratio

Given insight into the popularity distribution for distinct photo blobs, we can relate popularity to cache hit ratio performance as a way to explore the question posed earlier: *To what extent does photo blob popularity shape cache hit ratios?* Figure 4a illustrates the traffic share in terms of percent of client's requests served by each layer during a period of approximately one week. Client browsers resolved ~65% of the traffic from the local browser cache, the Edge cache served ~20%, the Origin tier ~5%, and Haystack handled the remaining ~10%. Although obtained differently, these statistics are consistent with the aggregated results we reported in Table 1.

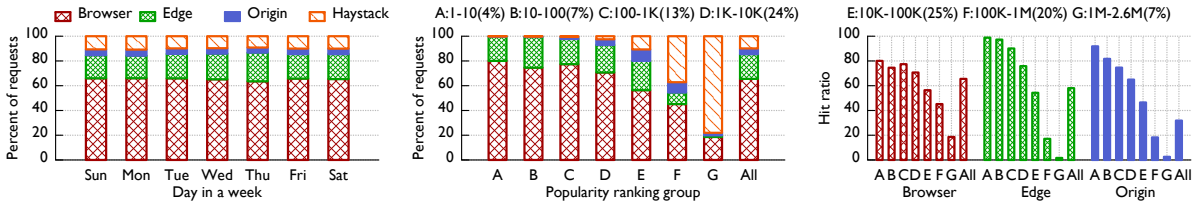
Figure 4b breaks down the traffic served by each layer into image-popularity groups. We assign each photo blob a popularity rank based on the number of requests in our trace. The most popular photo blob has rank 1 and the least popular blob has rank over 2.6M. We then bin items by popularity, using logarithmically-increasing bin sizes. The figure shows that the browser cache and Edge cache served more than 89% of requests for the hundred-thousand most popular images (groups A-E). As photo blobs become less popular (groups F then G) they are less likely to be resident in cache and thus a higher percentage of requests are satisfied by the Haystack Backend. In particular, we see that Haystack served almost 80% of requests for the least popular group (G). The Origin Cache also shelters the Backend from a significant amount of traffic, and this sheltering is especially effective for blobs in the middle popularity groups (D, E and F), which are not popular enough to be retained in the Edge cache.

Figure 4c illustrates the hit ratios binned by the same popularity groups for each cache layer. It also shows the percent of requests to each popularity group. One interesting result is the dramatically higher hit ratios for the Edge and Origin layers than the browser cache layer for popular photos (groups A-B). The explanation is straightforward. Browser caches can only serve photos that this particular client has previously downloaded, while the Edge and Origin caches are shared across all clients and can serve photos any client has previously downloaded. The reverse is true for unpopular photos (groups E-G). They have low hit ratios in the shared caches because they are quickly evicted for more generally popular content, but remain in the individual browser caches, which see traffic from only a single user.

Popularity group	# Requests	# Unique IPs	Req/IP ratio
A	5120408	665576	7.7
B	8313854	1530839	5.4
C	15497215	2302258	6.7

Table 2: Access statistics for selected groups. "Viral" photos are accessed by massive numbers of clients, rather than accessed many times by few clients, so browser caching is of only limited utility.

Looking closely at the hit ratios, it is at first counter-intuitive that the browser cache has a lower hit ratio for group B than the next less popular photo group C. The



(a) Traffic share for a week **(b) Traffic share for popularity groups** **(c) Hit ratio for popularity groups**

Figure 4: Traffic distribution. Percent of photo requests served by each layer, (a) aggregated daily for a week; (b) binned by image popularity rank on a single day. For (b), 1-10 represent the 10 most popular photos, 10-100 the 90 next most popular, etc. (c) shows the hit ratios for each cache layer binned by the same popularity group, along with each group's traffic share.

likely reason is that many photo blobs in this group are "viral," in the sense that large numbers of distinct clients are accessing them concurrently. Table 2 confirms this by relating the number of requests, the number of distinct IP addresses, and the ratio between these two for the top 3 popularity groups. As we can see, the ratio between the number of requests and the number of IP addresses for group B is lower than the more popular group A and less popular group C. We conclude that although many clients will access "viral" content once, having done so they are unlikely to subsequently revisit that content. On the other hand, a large set of photo blobs are repeatedly visited by the same group of users. This demonstrates that the Edge cache and browser cache complement one another in serving these two categories of popular images, jointly accounting for well over 90% of requests in popularity groups A, B and C of Figure 4b.

5 Geographic Traffic Distribution

This section explores the geographical patterns in request flows. We analyze traffic between clients and Edge Caches, how traffic is routed between the Edge Caches and Origin Cache, and how Backend requests are routed. Interestingly (and somewhat surprisingly), we find significant levels of cross-country routing at all layers.

5.1 Client To Edge Cache Traffic

We created a linked data set that traces activities for photo requests from selected cities to US-based Edge Caches. We selected thirteen US-based cities and nine Edge Caches, all heavily loaded during the period of our study. Figure 5 shows the percentage of requests from each city that was directed to each of the Edge Caches. Timezones are used to order cities (left is West) and Edge Caches (top is West).

Notice that each city we examine is served by all nine Edge Caches, even though in many cases this includes Edge Caches located across the country that are accessible only at relatively high latency. Indeed, while every Edge Cache receives a majority of its requests from geographically nearby cities, the largest share does not necessarily go to the nearest neighbor. For example, fewer Atlanta requests are served by the Atlanta Edge Cache than by the D.C. Edge Cache. Miami is another interesting case: Its traffic was distributed among several Edge Caches, with 50% shipped west and handled in San Jose, Palo Alto and LA and only 24% handled in Miami.

The reason behind this geographical diversity is a routing policy based on a combination of latency, Edge Cache capacity and ISP peering cost, none of which necessarily translates to physical locality. When a client request is received, the Facebook DNS server computes a weighted value for each Edge candidate, based on the latency, current traffic, and traffic cost, then picks the best option. The peering costs depend heavily on the ISP peering agreements for each Edge Cache, and, for historical reasons, the two oldest Edge Caches in San Jose and D.C. have especially favorable peering quality with respect to the ISPs hosting Facebook users. This increases the value of San Jose and D.C. compared to the other Edge Caches, even for far-away clients.

A side effect of Facebook's Edge Cache assignment policy is that a client may shift from Edge Cache to Edge Cache if multiple candidates have similar values, especially when latency varies throughout the day as network dynamics evolve. We examined the percentage of clients served by a given number of Edge Caches in our trace: 0.9% of clients are served by 4 or more Edge Caches, 3.6% of clients are served by 3 or more Edge Caches, and 17.5% of clients are served by 2 or more Edge Caches. Client redirection reduces the Edge cache hit ratio because every Edge Cache reassignment brings the potential for new cold cache misses. In Section 6, we discuss potential improvement from collaborative caching.

5.2 Edge Cache to Origin Cache Traffic

Currently Facebook serves user-uploaded photos at four regional data centers in the United States. Two are on the East Coast (in Virginia and North Carolina) and two others are on the West Coast (in Oregon and California). In addition to hosting Haystack Backend clusters, these data centers comprise the Origin Cache configured to handle

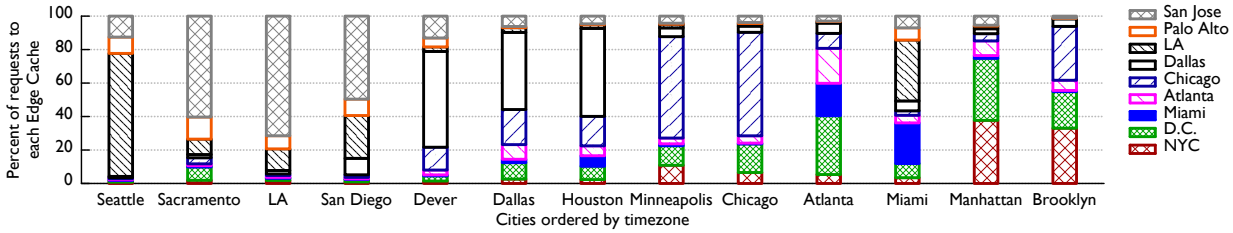


Figure 5: Traffic share from 13 large cities to Edge Caches (identified in legend at right).

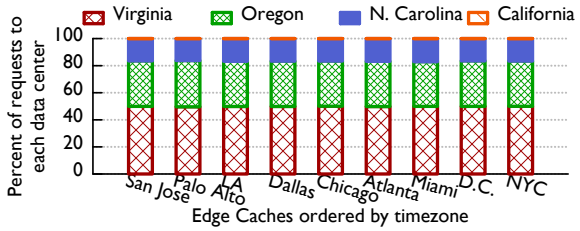


Figure 6: Traffic from major Edge Caches to the data centers that comprise the Origin Cache.

requests coming from various Edge Caches. Whenever there is an Edge Cache miss, the Edge Cache will contact a data center based on a consistent hashed value of that photo. In contrast with the Edge Caches, all Origin Cache servers are treated as a single unit and the traffic flow is purely based on content, not locality.

Figure 6 shows the share of requests from nine Edge Caches to the four Origin Cache data centers. The percentage of traffic served by each data center on behalf of each Edge Cache is nearly constant, reaffirming the effects of consistent hashing. One noticeable exception, California, was being decommissioned at the time of our analysis and not absorbing much Backend traffic.

5.3 Cross-Region Traffic at Backend

In an ideal scenario, when a request reaches a data center we would expect it to remain within that data center: the Origin Cache server will fetch the photo from the Backend in the event of a miss, and a local fetch would minimize latency. But two cases can arise that break this common pattern:

- **Misdirected resizing traffic:** Facebook continuously migrates Backend data, both for maintenance and to ensure that there are adequate numbers of backup copies of each item. Continuously modifying routing policy to keep it tightly aligned with replica location is not feasible, so the system tolerates some slack, which manifests in occasional less-than-ideal routing.
- **Failed local fetch:** Failures are common at scale, and thus the Backend server holding some local replica of a desired image may be offline or overloaded. When a request from an Origin Cache server to its nearby Backend fails to fetch a photo quickly, the Origin Cache server will pick a remote alternative.

Table 3 summarizes the traffic retention statistics for each data center. More than 99.8% of requests were routed to a data center within the region of the originating Origin Cache, while about 0.2% of traffic travels over long distances. This latter category was dominated by traffic sent from California, which is also the least active data center region in Figure 6 for Edge Caches. As noted earlier, this data center was being decommissioned during the period of our study.

Origin Cache Server Region	Backend Region		
	Virginia	North Carolina	Oregon
Virginia	99.885%	0.049%	0.066%
North Carolina	0.337%	99.645%	0.018%
Oregon	0.149%	0.013%	99.838%
California	24.760%	13.778%	61.462%

Table 3: Origin Cache to Backend traffic.

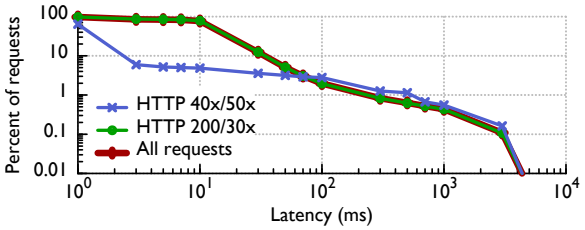


Figure 7: Complementary cumulative distribution function (CCDF) on latency of requests from Origin Cache servers to the Backend.

Figure 7 examines the latency of traffic between the Origin Cache servers and the Backend, with lines for successful requests (HTTP error code 200/30x), failed requests (HTTP error code 40x/50x), and all requests. While successful accesses dominate, more than 1% of requests failed. Most requests are completed within tens of milliseconds. Beyond that range, latency curves have two inflection points at 100ms and 3s, corresponding to the minimum delays incurred for cross-country traffic between eastern and western regions, and maximum timeouts currently set for cross-country retries. When a successful re-request follows a failed request, the latency is aggregated from the start of the first request.

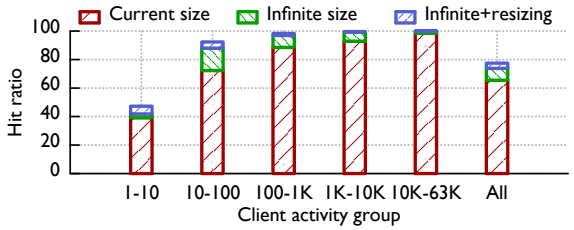


Figure 8: Measured, ideal, and resize-enabled hit ratios for clients grouped by activity. 1-10 groups clients with ≤ 10 requests, 10-100 groups those reporting 11 to 100 requests, etc. *All* groups all clients.

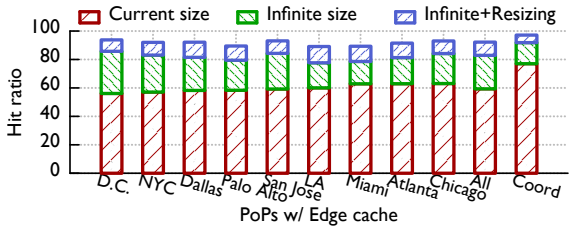


Figure 9: Measured, ideal, and resize-enabled hit ratios for the nine largest Edge Caches. *All* is the aggregated hit ratio for all regions. *Coord* gives the results for a hypothetical collaborative Edge Cache.

6 Potential Improvements

This section closely examines Browser, Edge, and Origin Cache performance. We use simulation to evaluate the effect of different cache sizes, algorithms, and strategies.

6.1 Browser Cache

Figure 8 shows the aggregated hit ratio we observed for different groups of clients. The "all" group includes all clients and had a aggregated hit ratio of 65.5%. This is much higher than the browser cache statistics published by the Chrome browser development team for general content: they saw hit ratios with a Gaussian distribution around a median of 35% for unfilled caches and 45% for filled caches [6].

The figure also breaks down hit ratios based on the observed activity level of clients, i.e., how many entries are in our log for them. The least active group with 1-10 logged requests saw a 39.2% hit ratio, while a more active group with 1K-10K logged requests saw a 92.9% hit ratio. The higher hit ratio for more active clients matches our intuition: highly active clients are more likely to access repeated content than less active clients, and thus their browser caches can achieve a higher hit ratio.

Browser Cache Simulation Using our trace to drive a simulation study, we can pose *what-if* questions. In Figure 8 we illustrate one example of the insights gained in this manner. We investigate what the client browser hit ratios would have been with an infinite cache size. We use the first 25% of our month-long trace to warm the cache and then evaluate using the remaining 75% of the trace. The infinite cache size results distinguish between cold (compulsory) misses for never-before-seen content and capacity misses, which never happen in an infinite cache. The infinite size cache bar thus gives an upper bound on the performance improvements that could be gained by increasing the cache size or improving the cache replacement policy. For most client activity groups this potential gain is significant, but the least active client group is an interesting outlier. These very inactive clients would see little benefit from larger or improved caches: an unbounded cache improved their hit ratio by 2.6% to slightly over 41.8%.

We also simulated the effect of moving some resizing to the client: clients with a cached full-size image resize that object rather than fetching the required image size. While client-side resizing does not result in large improvements in hit ratio for most client groups, it does provide a significant 5.5% improvement even relative to an unbounded cache for the least active clients.

6.2 Edge Cache

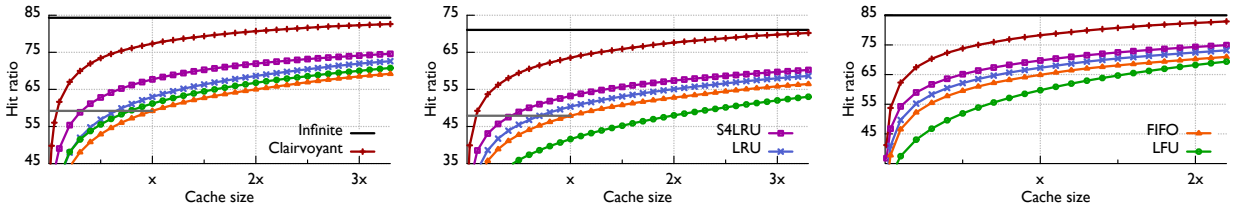
To investigate Edge cache performance at a finer granularity, we analyzed the hit ratio for nine heavily used Edge Caches. Figure 9 illustrates the actual hit ratio observed at each Edge Cache, a value aggregated across all regions, denoted "All", and a value for a hypothetical collaborative cache that combines all Edge Caches into a single Edge Cache. (We defer further discussion of the collaborative cache until later in this subsection.) We also estimated the highest possible hit ratio for perfect Edge Caches by replaying access logs and assuming an infinite cache warmed by the first 25% of our month-long trace. We then further enhanced the hypothetical perfect Edge Caches with the ability to resize images. The results are stacked in Figure 9, with the actual value below and the simulated ideal performance contributing the upper portion of each bar.

The current hit ratios range from 56.1% for D.C. to 63.1% in Chicago. The upper bound on improvement, infinite size caches, has hit ratios from 77.7% in LA to 85.8% in D.C.. While the current hit ratios represent significant traffic sheltering and bandwidth reduction, the much higher ratios for infinite caches demonstrate there is much room for improvement. The even higher hit ratios for infinite caches that can resize photos makes this point even clearer: hit ratios could potentially be improved to be as high as 89.1% in LA, and to 93.8% in D.C..

Edge Cache Simulation Given the possibility of increases as high as 40% in hit ratios, we ran a number of *what-if* simulations. Figures 10a and 10b explores the effect of different cache algorithms and cache sizes for

FIFO	A first-in-first-out queue is used for cache eviction. This is the algorithm Facebook currently uses.
LRU	A priority queue ordered by last-access time is used for cache eviction.
LFU	A priority queue ordered first by number of hits and then by last-access time is used for cache eviction.
S4LRU	Quadruply-segmented LRU. Four queues are maintained at levels 0 to 3. On a cache miss, the item is inserted at the head of queue 0. On a cache hit, the item is moved to the head of the next higher queue (items in queue 3 move to the head of queue 3). Each queue is allocated $1/4$ of the total cache size and items are evicted from the tail of a queue to the head of the next lower queue to maintain the size invariants. Items evicted from queue 0 are evicted from the cache.
Clairvoyant	A priority queue ordered by next-access time is used for cache eviction. (Requires knowledge of the future.)
Infinite	No object is ever evicted from the cache. (Requires a cache of infinite size.)

Table 4: Descriptions of the simulated caching algorithms.



(a) Object-Hit Ratio at San Jose **(b) Byte-Hit Ratio at San Jose** **(c) Byte-Hit Ratio for a Collaborative Edge**

Figure 10: Simulation of Edge Caches with different cache algorithms and sizes. The object-hit ratio and byte-hit ratio are shown for the San Jose Edge Cache in (a) and (b), respectively. The byte-hit ratio for a collaborative Edge Cache is given in (c). The gray bar gives the observed hit ratio and size x approximates the current size of the cache.

the San Jose Edge Cache. We use San Jose here because it is the median in current Edge Cache hit ratios and the approximate visual median graph of all nine examined Edge Caches. The horizontal gray bar on the graph corresponds to the observed hit ratio for San Jose, 59.2%. We label the x-coordinate of the intersection between that observed hit ratio line and the FIFO simulation line, which is the current caching algorithm in use at Edge Caches, as size x. This is our approximation of the current size of the cache at San Jose.

The different cache algorithms we explored are explained briefly in Table 4. We first examine the results for object-hit ratio. Our results demonstrate that more sophisticated algorithms yield significant improvements over the current FIFO algorithm: 2.0% from LFU, 3.6% from LRU, and 8.5% from S4LRU. Each of these improvements yields a reduction in downstream requests. For instance, the 8.5% improvement in hit ratio from S4LRU yields a 20.8% reduction in downstream requests.

The performance of the Clairvoyant algorithm demonstrates that the infinite-size-cache hit ratio of 84.3% is unachievable at the current cache size. Instead, an almost-theoretically-perfect algorithm could only achieve a 77.3% hit ratio.[1] This hit ratio still represents a very large potential increase of 18.1% in hit ratio over the current FIFO algorithm, which corresponds to a 44.4% decrease in downstream requests. The large gap between the best algorithm we tested, S4LRU, and the Clairvoyant algorithm demonstrates there may be ample gains available to still-cleverer algorithms.

The object-hit ratios correspond to the success of a cache in sheltering traffic from downstream layers, i.e., decreasing the number of requests (and ultimately disk-based IO operations). For Facebook, the main goal of Edge Caches is not traffic sheltering, but bandwidth reduction. Figure 10b shows byte-hit ratios given different cache sizes. These results, while slightly lower, mostly mirror the object-hit ratios. LFU is a notable exception, with a byte-hit ratio below that of FIFO. This indicates that LFU would not be an improvement for Facebook because even though it can provide some traffic sheltering at the Edge, it increases bandwidth consumption. S4LRU is again the best of the tested algorithms with an increase of 5.3% in byte-hit ratio at size x, which translates to a 10% decrease in Origin-to-Edge bandwidth.

Figure 10 also demonstrates the effect of different cache sizes. Increasing the size of the cache is also an effective way to improve hit ratios: doubling the cache size increases the object-hit ratio of the FIFO algorithm by 5.8%, the LFU algorithm by 5.6%, the LRU algorithm by 5.7%, and the S4LRU algorithm by 4.3%. Similarly, it increases the byte-hit ratios of the FIFO algorithm by 4.8%, the LFU algorithm by 6.4%, the LRU algorithm by 4.8%, and the S4LRU algorithm by 4.2%.

Combining the analysis of different cache algorithms and sizes yields even more dramatic results. There is an inflection point for each algorithm at a cache size smaller than x. This translates to higher-performing algorithms

[1] The "Clairvoyant" algorithm is not theoretically perfect because it does not take object size into account. It will choose to store an object of size $2x$ next accessed at time t over storing 2 objects of size x next accessed at times $t+1$ and $t+2$.

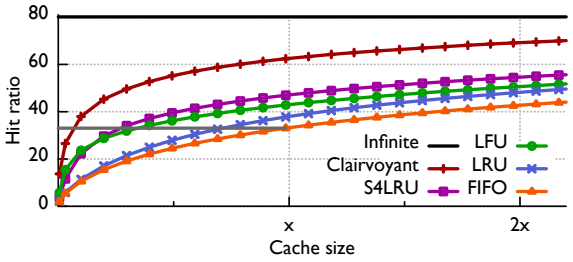


Figure 11: Simulation of Origin Cache with different cache algorithms and sizes.

being able to achieve the current object-hit ratio at much smaller cache sizes: LFU at 0.8x, LRU at 0.65x, and S4LRU at 0.35x. The results are similar for the size needed to achieve the current byte-hit ratio: LRU at 0.7x and S4LRU at 0.3x. These results provide a major insight to inform future static-content caches: a small investment in Edge Caches with a reasonably sophisticated algorithm can yield major reductions in traffic. Further, the smaller a cache, the greater the choice of algorithm matters.

Collaborative Edge Cache We also simulated a collaborative Edge Cache that combines all current Edge Caches into a single logical cache. Our motivation for this *what-if* scenario is twofold. First, in the current Edge Cache design, a popular photo may be stored at every Edge Cache. A collaborative Edge Cache would only store that photo once, leaving it with extra space for many more photos. Second, as we showed in Section 5, many clients are redirected between Edge Caches, resulting in cold misses that would be avoided in a collaborative cache. Of course, this hypothetical collaborative Edge Cache might not be ultimately economical because it would incur greater peering costs than the current system and would likely increase client-photo-load latency.

Figure 10c gives the byte-hit ratio for different cache algorithms and sizes for a collaborative Edge Cache. The size x in this graph is the sum of the estimated cache size we found by finding the intersection of observed hit ratio and FIFO simulation hit ratio for each of the nine Edge Caches. At the current cache sizes, the improvement in hit ratio from going collaborative is 17.0% for FIFO and 16.6% for S4LRU. Compared to the current individual Edge Caches running FIFO, a collaborative Edge Cache running S4LRU would improve the byte-hit ratio by 21.9%, which translates to a 42.0% decrease in Origin-to-Edge bandwidth.

6.3 Origin Cache

We used our trace of requests to the Origin Cache to perform a *what-if* analysis for different cache algorithms and sizes. We again evaluated the cache algorithms in Table 4. The results are shown in Figure 11. The observed hit ratio for the Origin Cache is shown with a gray line and our estimated cache size for it is denoted size x.

The current hit ratio relative to the Clairvoyant algorithm hit ratio is much lower at the Origin Cache than at the Edge Caches and thus provides a greater opportunity for improvement. Moving from the FIFO cache replacement algorithm to LRU improves the hit ratio by 4.7%, LFU improves it by 9.8%, and S4LRU improves it by 13.9%. While there is a considerable 15.5% gap between S4LRU and the theoretically-almost-optimal Clairvoyant algorithm, S4LRU still provides significant traffic sheltering: it reduces downstream requests, and thus Backend disk-IO operations, by 20.7%.

Increasing cache size also has a notable effect. Doubling cache size improves the hit ratio by 9.5% for the FIFO algorithm and 8.5% for the S4LRU algorithm. A double-sized S4LRU Origin Cache would increase the hit ratio to 54.4%, decreasing Backend requests by 31.9% compared to a current-sized FIFO Origin Cache. This would represent a significant improvement in the sheltering effectiveness of Facebook's Origin Cache. Combining the analysis of different cache sizes and algorithms, we see an inflection point in the graph well below the current cache size: the current hit ratio can be achieved with a much smaller cache and higher-performing algorithms. The current hit ratio (33.0%, in the portion of the trace used for simulation) can be achieved with a 0.7x size LRU cache, a 0.35x size LFU cache, or a 0.28x size S4LRU cache.

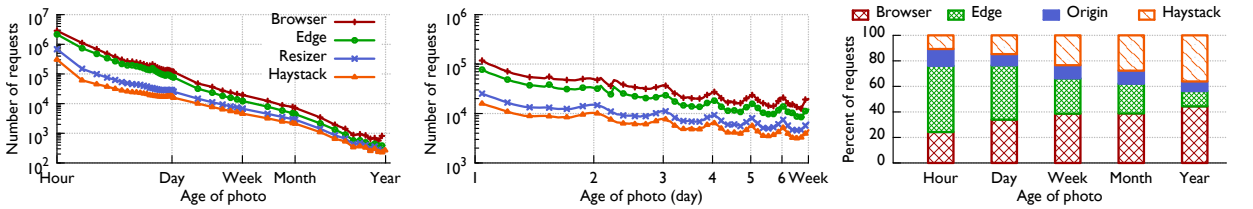
We omit the byte-hit ratio for the Origin Cache, but the difference is similar to what we see at the Edge Caches. The byte-hit ratio is slightly lower than the object-hit ratio, but the simulation results all appear similar with the exception of LFU. When examined under the lens of byte-hit ratio LFU loses its edge over LRU and performs closer to FIFO. The S4LRU algorithm is again the best for byte-hit rate with a 8.8% improvement over the FIFO algorithm, which results in 11.5% less Backend-to-Origin bandwidth consumption.

7 Social-Network Analysis

This section explores the relationship between photo requests and various kinds of photo meta-information. We studied two properties that intuitively should be strongly associated with photo traffic: the age of photos and the number of Facebook followers associated with the owner.

7.1 Content Age Effect

It is generally assumed that new content will draw attention and hence account for the majority of traffic seen within the blob-serving stack. Our data set permits us to evaluate such hypotheses for the Facebook image hierarchy by linking the traces collected from different layers to the meta-information available in Facebook's photo database. We carried out this analysis, categorizing re-



(a) Age spans from 1 hour to 1 year (b) Age spans from 1 day to 1 week (c) Traffic share for non-profile photos by age

Figure 12: Traffic popularity and requests served by layer for photos at different age. The number of requests to each image, categorized by age of requested photos in hours, broken down at every layer across the stack.

quests for images by the age of the target content, then looking at the way this information varies at each layer of the stack. Photo age (in hours) was determined by subtracting the photo creation time from the request time. Thus, even a photo uploaded the same day will have associated requests sorted into 24 hourly categories. This analysis excludes profile photos because Facebook's internal storage procedures for them precludes determining their ago precisely.

Figure 12 plots the number of requests at each layer for photos of different ages. In Figure 12a, we consider a range of ages from 1 hour to 1 year. As content ages, the associated traffic diminishes at every layer; the relationship is nearly linear when plotted on a log-log scale. Figure 12b zooms into the mid-range age scales, focusing on a week. We see a noticeable daily traffic fluctuation. We traced this to a fluctuation in photo creation time, determining that users create and upload greater numbers of photos during certain periods of the day. This creation-time effect carries through to induce the striking photo-access-by-age pattern observed for smaller ages.

Our analysis reveals that traffic differences between caches deployed close to clients (browser, Edge Cache) and storage Backend (including the Origin Cache) are more pronounced for young photos than for old ones. This matches intuition: fresh content is popular and hence tends to be effectively cached throughout the image serving hierarchy, resulting in higher cache hit ratios. Figure 12c clearly exhibits this pattern. The age-based popularity decay of photos seen in Figure 12a is nearly Pareto, suggesting that an age-based cache replacement algorithm could be effective.

We should note that although our traces include accesses to profile photos, and we used them in all other analyses, we were forced to exclude profile photos for this age-analysis. The issue relates to a quirk of the Facebook architecture: when a user changes his or her profile photo, Facebook creates a new profile object but reuses the same name as for the previous versions. Profile objects can be distinguished by looking at the ownerId, which Facebook sets to the underlying photoId, but we can not determine the time of creation. None of our other analyses are impacted, but we were forced to exclude profile objects in our age analysis. The effect is to slightly reduce the computed traffic share for caches close to clients, especially in the categories associated with young and popular photos.

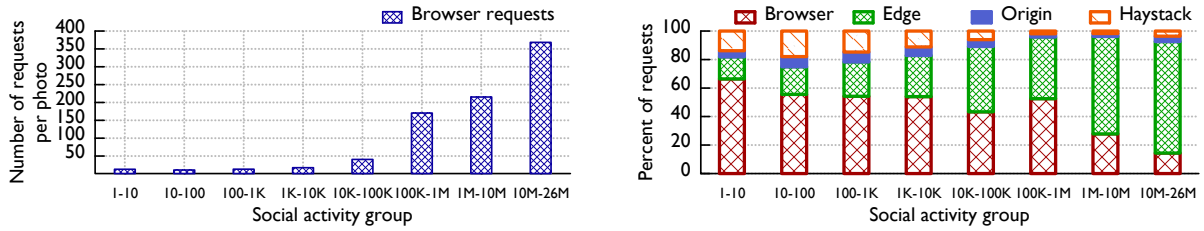
7.2 Social Effects

Intuitively, we expect that the more friends a photo owner has, the more likely the photo is to be accessed. We observed this phenomenon in our study, but only when we condition on owner type. We binned owners by the number of followers (friends for normal users, fans for public page owners), creating "popularity groups", and graphed photo requests by their owners' groups, yielding the data seen in Figure 13. For each photo request, the photo owner's friend count was fetched on the day when the access happened, thus requests for one photo may be split into multiple groups when an owner's popularity changes. We include profile photos in this analysis.

Figure 13a graphs the number of requests for each photo against the photo owner's popularity group. Most Facebook users have fewer than 1000 friends, and for that range the number of requests for each photo is almost constant. For public page owners who can have thousands or millions of fans, each photo has a significantly higher number of requests, determined by the size of the fan base. Figure 13b further breaks down the traffic distribution at each layer of the photo-serving stack for different social activity groups. For normal users with fewer than 1000 followers (friends), the caches absorb ~80% of the requests for their photos; but for public page owners, more followers (fans) drives higher percentages of traffic being absorbed by caches. However, browser caches tend to have lower hit ratios for owners with more than 1 million followers. This is because these photos fall into the "viral" category discussed earlier in Section 4.

8 Related Work

Many measurement studies have examined web access patterns for services associated with content delivery, storage, and web hosting. To the best of our knowledge, our paper is the first to systematically instrument and analyze a real-world workload at the scale of Facebook, and to successfully trace such a high volume of events throughout a massively distributed stack. The most closely re-



(a) Client requests per photo

(b) Traffic share for social activity groups

Figure 13: Photo popularity organized owner popularity. (a) Requests per photo categorized by the number of followers for the photo's owner. (b) Traffic distribution by layer for different social activity groups.

lated prior work that we identified is a classic study by Saroiu et al. [19] that compared the characteristics of four different Internet content delivery mechanisms using a network trace captured between University of Washington and the rest of the Internet. That work was undertaken some time ago, however, during a period when peer-to-peer networks were a dominant source of Internet traffic. A follow-on paper [11] took the analysis further, comparing the media popularity distribution seen in the campus trace with that associated with traditional web traffic. Our focus on blob traffic induced by social networking thus looks at a different question, and on a much larger scale.

Additional work involved studies of the *flash crowd phenomenon* in content delivery networks (CDNs) [23, 16, 20]. These efforts focused on data obtained by monitoring aggregated network traffic. Such work yields broad statistics, but we gain only limited insight into application properties that gave rise to the phenomena observed. An exception is Freedman's investigation [8] of a 5-year system log of the Coral CDN [9], studying its behavior with detailed insight into its operational properties and architecture. Our work covers both sides, enabling us to break behaviors down in a manner not previously possible.

Whereas our focus here was on the network side of the image-processing stack, the lowest layer we considered is the backend storage server. Here, one can point to many classic studies [1, 7, 13, 18, 22] and also to the recent detailed architecture and performance evaluation of Haystack, the Facebook blob storage server [2]. Detailed network and caching traffic traces can inform the design of future storage systems, just as they enabled us to study how different caching policies might reduce loads within the Facebook infrastructure. However, constraints of length and focus forced us to limit the scope of the present paper, and we leave this for future investigation.

Numerous research projects have explored the modelling of web workload, and several recent papers [14, 5] monitor web traffic over extended time periods, to the point of evaluating workload changes as the web itself evolved. Breslau et al. [4] explore the impact of Zipf's law with respect to web caching, showing that Zipf-like popularity distributions cause cache hit rates to grow logarithmically with population size, as well as other effects. In contrast, Guo et al. [12] argue that access to media content often has a significantly distorted head and tail relative to a classic Zipf distribution. We found that caches closest to the client browser have a purely Zipf popularity distribution, but that deep within the Facebook architecture, the Haystack Backend experiences a workload that looks very much like what [12] characterize as a stretched exponential distribution.

9 Conclusions & Future Directions

We instrumented the entire Facebook photo-serving stack obtaining traces representative of Facebook's full workload. A number of valuable findings emerge from this integrated perspective, including the workload pattern, traffic distribution and geographic system dynamics, yielding insights that should be helpful to future system designers.

Our work also points to possible caching options that merit further study. It may be worthwhile to explore collaborative caching at geographic (nationwide) scales, and to adopt S4LRU eviction algorithms at both Edge and Origin layers. We also identified an opportunity to improve client performance by increasing browser cache sizes for very active clients and by enabling local photo resizing for less active clients.

Our paper leaves a number of questions for future investigation. One important area concerns the placement of resizing functionality along the stack, which is essential to balance the cost between network and computation. Another important area is designing even better caching algorithms, perhaps by predicting future access likelihood based on meta information about the images.

Acknowledgements We are grateful to the SOSP program committee, our shepherd Helen Wang, Kaushik Veeraraghavan, Daniel A. Freedman, and Yun Mao for their extensive comments that substantially improved this work. Bryan Alger, Peter Ruibal, Subramanian Muralidhar, Shiva Shankar P, Viswanath Sivakumar and other Facebook infrastructure engineers offered tremendous support for our instrumentation. This work was supported in part by DARPA, NSF, and the Facebook Graduate Fellowship.

References

[1] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a Distributed File System. In *Proc. Symposium on Operating Systems Principles (SOSP)*, Pacific Grove, California, USA, 1991. ACM.

[2] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a Needle in Haystack: Facebook's Photo Storage. In *Proc. Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, BC, Canada, Oct. 2010. USENIX.

[3] A. Bigian. Blobstore: Twitter's in-house photo storage system. http://tinyurl.com/cda5ahq, 2012.

[4] L. Breslau, P. Cue, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proc. International Conference on Computer Communications (INFOCOM)*. IEEE, Mar. 1999.

[5] M. Butkiewicz, H. V. Madhyastha, and V. Sekar. Understanding website complexity: measurements, metrics, and implications. In *Proc. SIGCOMM Internet Measurement Conference (IMC)*, Berlin, Germany, Nov. 2011. ACM.

[6] W. Chan. Chromium cache metrics. http://tinyurl.com/csu34wa, 2013.

[7] Y. Chen, K. Srinivasan, G. Goodson, and R. Katz. Design Implications for Enterprise Storage Systems via Multi-Dimensional Trace Analysis. In *Proc. Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, 2011. ACM.

[8] M. J. Freedman. Experiences with CoralCDN: A five-year operational view. In *Proc. 7th Symposium on Networked Systems Design and Implementation (NSDI 10)*, San Jose, CA, Apr. 2010.

[9] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with Coral. In *Proc. Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, California, Mar. 2004. USENIX.

[10] A. Gartrell, M. Srinivasan, B. Alger, and K. Sundararajan. McDipper: A key-value cache for Flash storage. http://tinyurl.com/c39w465, 2013.

[11] K. P. Gummadi, R. J. Dunn, S. Saroiu, S. D. Gribble, H. M. Levy, and J. Zahorjan. Measurement, Modeling, and Analysis of a Peer-to-Peer File-Sharing Workload. In *Proc. Symposium on Operating Systems Principles*, Bolton Landing, NY, USA, Dec. 2003. ACM.

[12] L. Guo, E. Tan, S. Chen, Z. Xiao, and X. Zhang. The stretched exponential distribution of internet media access patterns. In *Proc. Symposium on Principles of distributed computingi (PODC)*, Toronto, Canada, Aug. 2008. ACM.

[13] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications. In *Proc. Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, 2011. ACM.

[14] S. Ihm and V. S. Pai. Towards Understanding Modern Web Traffic. In *Proc. SIGCOMM Internet Measurement Conference (IMC)*, Berlin, Germany, Nov. 2011. ACM.

[15] R. Johnson. Facebook's Scribe technology now open source. http://tinyurl.com/d7qzest, 2008.

[16] J. Jung, B. Krishnamurthy, and M. Rabinovich. Flash crowds and Denial of Service attacks: Characterization and implications for CDNs and web sites. In *Proc. International World Wide Web conference (WWW)*, Honolulu, Hawaii, USA, May 2002. ACM.

[17] S. Kavalanekar, B. L. Worthington, Q. Zhang, and V. Sharda. Characterization of storage workload traces from production Windows Servers. In *International Symposium on Workload Characterization (IISWC)*, Seattle, Washington, USA, Sept. 2008. IEEE.

[18] J. K. Ousterhout, H. Da Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proc. Symposium on Operating Systems Principles (SOSP)*, Orcas Island, Washington, USA, 1985. ACM.

[19] S. Saroiu, K. P. Gummadi, R. J. Dunn, S. D. Gribble, and H. M. Levy. An analysis of Internet Content Delivery Systems. In *Proc. Symposium on Operating System Design and Implementation (OSDI)*, Boston, Massachusetts, USA, Dec. 2002. USENIX.

[20] S. Scellato, C. Mascolo, M. Musolesi, and J. Crowcroft. Track globally, deliver locally: improving content delivery networks by tracking geographic social cascades. In *Proc. International World Wide Web conference (WWW)*, Hyderabad, India, Mar. 2011. ACM.

[21] A. Thusoo. Hive - A Petabyte Scale Data Warehouse using Hadoop. http://tinyurl.com/bprpy5p, 2009.

[22] W. Vogels. File system usage in Windows NT 4.0. In *Proc. Symposium on Operating Systems Principles (SOSP)*, Charleston, South Carolina, USA, 1999. ACM.

[23] P. Wendell and M. J. Freedman. Going viral: flash crowds in an open CDN. In *Proc. SIGCOMM Internet Measurement Conference (IMC)*, Berlin, Germany, Nov. 2011. ACM.