Simulating Facebook Photo Caching

Deep Desai and Jasmeet Singh

Department of Computer Science & Engineering, Texas A&M University
{dkdnco,jasmeet13n}@tamu.edu

Abstract—This project simulates the Facebook photo caching infrastructures and validates the results presented in the paper titled An Analysis of Facebook Photo Caching [4]. Facebook's photo serving infrastructure is huge, complex and geographically distributed. It includes caching at various levels starting from client browser, 9 edge caches at PoPs (Point of Presence), 4 origin caches at 4 data centers. The underlying database storage layer called haystack is widely distributed and is spread across 4 data centers in US alone. We developed a framework which can simulate a caching infrastructure spread across different layers, and is geographically distributed. The overall strategy is to generate traffic patterns similar to Facebook's photo access traffic and send those requests to our simulated infrastructure. We generate around 70 million requests characterized by a Zipfian distribution. The simulation results permit us to study traffic patterns, cache access patterns and help us validate our results against actual results obtained by running such tests on actual Facebook infrastructure presented in the referenced paper [4]. Our results quantify (1) the overall traffic percentages served by different caching layers, (2) the hit ratios at different caching layers, (3) the changes in hit-ratios by employing different cache eviction algorithms at different layers, (4) effects of changing the size of caches on percentage traffic served and hit ratios.

Index Terms—caching, facebook photo caching, simulation

1 Introduction

Caching improves access times and reduces data traffic to data sources that have limited throughput. Online social networks like Facebook, Twitter serve billions of photos uploaded by their users. To give a perspective of the scale, Facebook revealed in a white paper that its users have uploaded more than 250 billion photos, and are uploading 350 million new photos each day [5]. At a typical moment in time there may be hundreds of millions of clients interacting with Facebook Edge Caches [4]. However, most photos are not accessed by many users whereas some photos are accessed by millions of users (can be termed as viral photos). Number of people accessing some photos might also reduce as the photos gets old. This makes a perfect condition to cache photos that are recent or accessed regularly by many users in a certain time window to shield requests from reaching the back-end databases that tend to be slow in locating and retrieving photos from huge databases which can be geographically distributed.

In this project we are trying to simulate the Facebook's photo caching infrastructure. The simulator generates photo requests using a request generator which imitates real world web access patterns. These requests are sent to the simulated caching infrastructure and the request trace through the stack is logged. The logged data is then aggregated to produce results as presented in the original paper [4]. The percentage traffic served and hit ratio numbers obtained by us are very similar to the numbers presented in the original paper [4]. This serves as a validation for our methodology and the results presented in the paper. The framework is very easy to implement and is implemented using the widely used scripting language python. Similar frameworks can be used to model any large scale geo-distributed infrastructure and can be used to produce important insights without having an actual infrastructure.

This report is organized as follows. Section 2 presents the motivation behind this project. Section 3 describes our traffic generation and simulation methodology. Section 4 describes the implementation of (1) traffic generator, (2) cache eviction algorithms, (3) Facebook's cache infrastructure, and (4) the simulator. Section 5 presents the evaluation and results obtained. Section 6 concludes the project and talks about future extensions.

2 Motivation

Facebook is the worlds largest online social network serving billions of photographs to more than a billion users. Caching becomes essential in such system to shield the back-end database from the millions of requests every second. A comprehensive study of Facebook photo caching is provided in the research paper titled An Analysis of Facebook Photo Caching [4]. A small literature survey suggests that a lot of work has been done in studying different caching techniques and cache eviction algorithms [2], [1].However, studying such techniques at the scale of Facebook is not feasible in a lab. This builds up our motivation to simulate such large scale caching mechanism and traffic patterns. This can help us gain insights into dynamics of large scale infrastructure by simulating it on a general computer. Such methods can help researchers to test new techniques on a simulator and get an idea of possible improvements.

3 Methodology

We simulated Facebook's photo-serving infrastructure, fed it by 70 million photo get requests for 1 million unique photographs, stored and analyzed the path of each request through the stack. This section presents our request generation, infrastructure and simulation methodology.

3.1 Scope

Though Facebook hosts billions of photographs, we are taking into account a subset which contains 1 million unique photographs and 70 million requests to these photographs distributed in zipfian fashion. A zipf distributed traffic on small subset is representative of actual traffic to actual deployment with billions of photos [4].

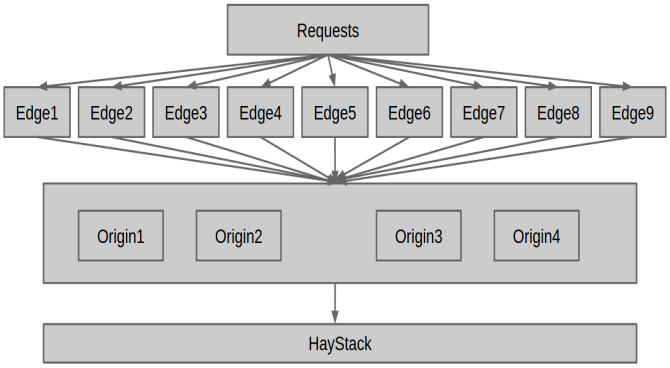


Fig. 1: Block diagram of the overall infrastructure

3.2 Request Generation

Previous work done on analyzing web traffic suggests that popularity of objects on the web follow a Zipfian distribution [6]. The original paper referenced by us also concludes that that Facebook's photo access follows a Zipfian distribution. So the 70 million requests generated for the 1 million unique photographs follow this pattern. This is very intuitive to explain as some photos are very popular (viral photos) and are accessed repeatedly in some time interval, whereas there are other less popular photographs that are accessed not that frequently.

3.3 Infrastructure

3.3.1 Browser Cache

We have not implemented a browser cache in this project because the browser cache resides in the client machine and is different for different clients. We only consider traffic coming to edge caches and beyond.

3.3.2 Edge Cache

Facebook's photo serving infrastructure in the US consists of 9 edge caches spread across the map. The intent is route traffic to the physically nearest edge cache. But this does not happen in reality because different ISP's have different pairing strategies to route their traffic and many a times the requests are served by an edge cache located across the map. So routing the traffic to any one of the edge cache is done in a random fashion which is in line with the findings of the paper [4].

3.3.3 Origin Cache and Haystack

The original paper reveals that there are 4 regional data centers, two on the East Coast (located in Virginia and North Carolina) and two on West Coast (located in Oregon and California) [4]. These data centers host the back-end storage (haystack). Also, each data center comprises of an origin cache. Each request that could not be served by the edge cache is routed to the origin cache. The 1 million photos are sharded across the 4 origin caches and data centers. So whenever there is a miss at the edge cache, it contacts the origin cache based on a consistent hashed value of that photo [4]. In our case we shard the photos across the 4 origin caches by using a mod 4 operator. The photo id number which is an integer is divided by 4 and the remainder is used as the index number of the origin cache to be contacted. When there is a cache miss at the origin cache, the request is sent to the haystack where the data center is located. So, there is a one to one matching from origin cache to haystack instance.

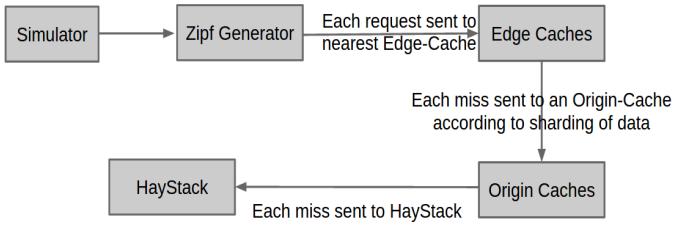


Fig. 2: Flow chart of the simulator

3.3.4 Overall Infrastructure

The overall infrastructure block diagram is shown in Figure 1. On the top there is request generator, which forwards the requests to one of the 9 independent edge caches. On the next level there are 4 Origin Caches which act as a single block, with keys sharded across the four. On the next level there is the storage layer called the Haystack.

3.4 Simulator

The flowchart showing the trace of a request through the stack is shown in Figure 2. The simulator first generates requests through a Zipf generator. Then each request is sent to an edge cache at random. If there is a hit at the edge cache, the request is served by that edge cache and a cache hit is recorded. If there is a cache miss, the request is forwarded to the origin cache that contains the particular key being requested. If there is a hit at the origin cache, the request is sent back through the same path it came through. If there is a miss at the origin cache, the request is sent to the haystack, which returns the key requested back through the forward path. This is repeated for each request and the cache hits and misses through the stack are logged at each cache individually.

After all requests have been simulated, the hit and miss data is aggregated for each level of caches by looking at the records of individual caches. This data can be used to produce different results such as percentage traffic served, hit ratio, etc.

4 Implementation

This section gives the implementation details of the project. The project is implemented using python language. We selected python because more emphasis was on getting the functionality rather than speed and efficient code.

4.1 Work Load

The traffic of requests for photos in Facebook follow a Zipfian distribution [4]. We implement a Zipfian distribution for 70 million requests over 1 million unique keys by using the random.zipf function from the numpy library provided by python language.

```
numpy.random.zipf(1.15, 1000000)
```

The first parameter passed to the function is an alpha value (determines the tail decay rate of the distribution) which is set to 1.15 and the second parameter passed is the number of requests to be generated which is equal to 1 million. The function that implements this Zipfian distribution is called as generateQueries function. The queries are stored in an array and returned to the function caller. The Figure 3 shows the number of requests versus object rank in a log-log scale.

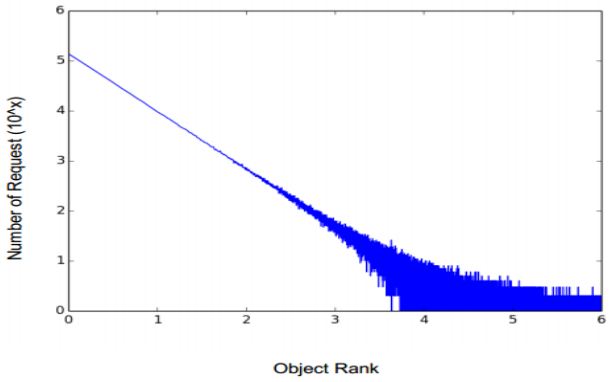


Fig. 3: Number of Requests versus Popularity of Photo

4.2 Cache Eviction Algorithms

As published in the original paper [4], we also implement 4 different cache-eviction algorithms. The implementation here is done keeping in mind the functionality and not the execution speed of the algorithm. We provide two functions in each cache: "set" to set a value for a key and "get" to get the value for a key.

4.2.1 FIFO

FIFO stands for First-In-First-Out. A FIFO cache is a traditional queue. We implement this queue using an ordered dictionary provided by the collections library in python. If a key exists in the cache, the get returns the value corresponding to the key, else it returns -1. When the cache reaches its capacity and a new key needs to be set in the cache, we pop an item from the cache in a FIFO order using the last=False parameter for the ordered dictionary.

4.2.2 LFU

LFU stands for Least-Frequently-Used. An LFU cache eviction algorithm evicts a key which is least frequently accessed. We implement this cache using a simple dictionary data structure in python. Every get increases the value corresponding to the key by one and returns the key if the key exists in the cache, else it returns -1. When the cache reaches its capacity and a new key needs to be set in the cache, we find the key with the least value and pop that key to set a new key with value equal to one.

4.2.3 LRU

LRU stands for Least-Recently-Used. An LRU cache eviction algorithm evicts the least recently accessed key. We implement this cache using an ordered dictionary provided by the collections library in python. For every get, if the key exists in the cache, we pop the key from the cache and set it again and return the value corresponding to the key, else we return -1. For every set, if the cache has reached its capacity we pop an item from the cache in FIFO order to set the new key into the cache, else we pop the key from the cache and set it again.

4.2.4 S4LRU

S4LRU stands for Segment-4-LRU. An S4LRU cache is a combination of 4 LRU caches. Each segment has a capacity equal to total capacity divided by 4. A get removes the key from the current segment and puts it in the next higher segment's most recently used end and return the value corresponding to the key if the key exists in the cache, else we return -1. For a set, we set the key in the forth segment of the cache, if the current segment reaches its capacity, we pop an item from the segment in a FIFO order and set that item in the next lower segment and set the new key in the current segment.

4.3 Stack

This section gives details about the implementation of the Facebook's photo serving stack.

4.3.1 Edge Cache and Origin Cache

We implement the edge cache and origin cache as classes. We have a different classes for edge cache and origin cache.

```
class EdgeCache:
  def __init__(self, cache_type):
    self.cache = cache_type
    self.misses = 0
    self.hits = 0
    ...
```

Each object of these classes act as a different cache. The implementation of both the classes is same. We pass the type of cache eviction algorithm as an argument to the constructor of the class. Each set just calls the set method of the corresponding cache-eviction algorithm. On each get, we increase the misses by one if the key does not exist in the cache and return a -1 else increase the hits by one and return the value corresponding to the key.

4.3.2 Haystack

The implementation for haystack is straightforward. We know that haystack consists of all the photos and each request is served successfully by the haystack. Hence, for any get, we just increase the requests-counter by one and return the key.

4.4 Simulator

The simulator simulates the whole Facebook caching infrastructure as well as simulates the photo requests characterized by a Zipfian distribution which are then served by the infrastructure.

4.4.1 Initialization

We initialize 9 edge caches and 4 origin caches with a cache_type (FIFO/LFU/LRU/S4LRU) and the capacity for each cache by making an array of 9 objects of the edge cache class and an array of 4 objects of the origin cache class respectively.

```
if __name__ == "__main__":
  edge=[EdgeCache(S4LRUCache(2000)) for i in range(0,9)]
  origin=[OriginCache(FIFOCache(5000)) for i in range(0,4)]
  hay=HayStack()
  ...
```

Making an array helps us in accessing each cache using its index in the array. We then initialize a haystack by making an object of the haystack class.

4.4.2 Generating Requests

We generate 70 million requests over 1 million unique keys by calling the generateQueries function and store it in an array called requests.

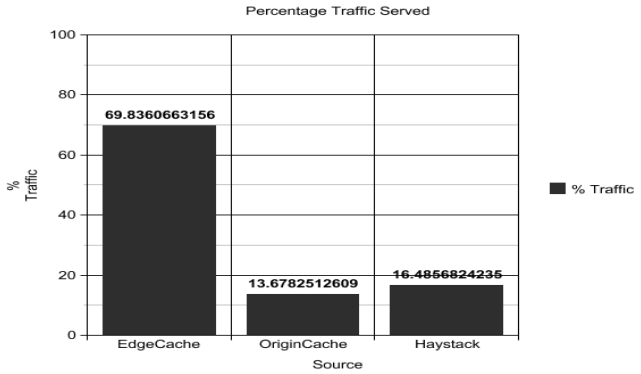


Fig. 4: Percentage traffic served by each layer

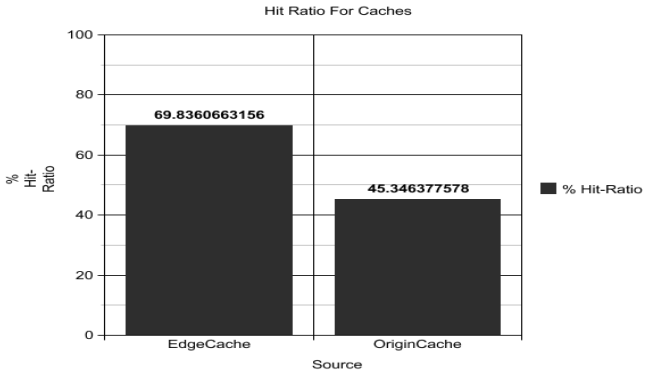


Fig. 5: Percentage hit-ratio for each layer

4.4.3 Fetch

For each request in the requests array generated in *4.4.2* we fetch the request across the caching infrastructure. If there is a miss on any cache we append that cache to an array to back-populate the cache while traversing back the caching infrastructure. The request should get routed to a geographically nearer edge cache and we implement this by randomly selecting an edge cache out of 9 edge caches and consider it to be the nearest edge cache for that particular request. If there is a miss from the selected edge cache we find out the origin cache which should have the corresponding request by taking a modulo 4 of the request. If there is a miss from the origin cache we fetch the request directly from the haystack. Finally the key gets set in each cache object in the back-populate array.

4.4.4 Output

At the end of each simulation an output is generated which gives the percentage traffic served by each layer and the hit-ratio for each layer in the caching infrastructure.

5 Evaluation

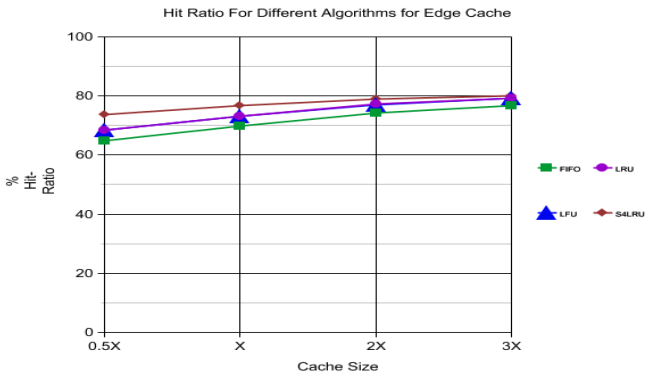
We used 70 million requests over 1 million unique keys under a Zipfian distribution to evaluate the performance of the simulation of the Facebook's caching infrastructure. We also did experiments with different cache sizes and different cache-eviction algorithms. No warming of caches was involved in any simulation.

5.1 Percentage Traffic Served

Figure 4 shows the percentage traffic served by different layers in the stack under the default cache-eviction algorithm being a FIFO algorithm. We observed that with the edge cache size of 18,000 unique keys and origin-cache size of 20,000 unique keys, the edge cache layer served nearly 70%, the origin-cache layer served nearly 13% and the haystack served nearly 16% of the traffic.

5.2 Percentage Hit Ratio

Hit-ratio is the percentage of hits as compared to the hits and the misses. The Figure 5 shows the hit-ratio for edge cache and the origin-cache layers. The hit-ratio for haystack would be a 100% because it serves all the requests successfully. We see that edge cache layer has a hit-ratio of nearly 70% and the origin-cache layer has a hit-ratio of around 45%, these numbers are for the edge cache size of 18,000 unique keys and origin-cache size of 20,000 unique keys and FIFO as the cache-eviction algorithm.



	0.5X	X	2X	3X
FIFO	64.83 (21%)	69.84 (22%)	74.32 (24%)	76.75 (24%)
LRU	68.47 (22%)	73.09 (23%)	77.19 (24%)	79.3 (25%)
LFU	68.51 (22%)	73.13 (23%)	77.08 (24%)	79.31 (25%)
S4LRU	73.56 (23%)	76.72 (24%)	79.06 (25%)	80.09 (25%)

Fig. 6: Simulation of edge caches with different cache algorithms and sizes. The size x here corresponds to 18000 unique keys

5.3 Cache Eviction Algorithms and Cache Size

We tested 4 different cache eviction algorithms namely FIFO, LFU, LRU and S4LRU for the edge cache layer keeping the cache-eviction algorithm for the origin-cache layer as FIFO. We also ran the simulation for 4 cache sizes - 9000, 18000, 36000, 54000 unique keys.

We measured the percentage hit-ratio for edge cache layer for the combination of cache sizes and the cache-eviction algorithms. Figure 6 shows the change in percentage hit-ratio with the change in cache size for different cache-eviction algorithms.

6 Conclusion and Future Work

We simulated the Facebook photo-serving stack, logging request path for 70 million requests on 1 million unique photos representative of Facebook's full work-load. The results obtained by us are inline with the results published in the original paper [4]. This serves as a validation for the technique used by us to simulate a complex, geo-distributed infrastructure. Our project can serves as a framework to test different cache eviction techniques at different layers and see where there are chances of improvements without actually deploying any code in a production environment. Frameworks similar to this can be beneficial to researchers trying to improve performance of

huge, geo-distributed clusters by simulating the whole infrastructure on a commodity machine.

Our project has a number of possible future extensions which can bring even deeper insights into the mechanics of Facebook caching infrastructure. Different cache eviction algorithms can be plugged in at different layers and we can see which algorithm is better suited for which layer. Delays across networks can be modeled and plugged in into the simulator and we can get a delay analysis of different requests through the stack. Such analysis can help in determining any scope of improvement before doing any real deployment.

The source code of the project is available for viewing and downloading on github [3]

Acknowledgments: We are thankful to Daniel A. Jimnez for helping us setting the scope and goals of the project. We also thank him for providing the right direction towards simulating the infrastructure rather than building a similar infrastructure.

REFERENCES

[1] L. Cardenas, J. A. Gil, J. Domenech, J. Sahuquillo, and A. Pont, "Performance comparison of a web cache simulation framework," in *Advanced Information Networking and Applications, 2005. AINA 2005. 19th International Conference on*, vol. 2. IEEE, 2005, pp. 281–284.

[2] Z. Chen, Y. Zhou, and K. Li, "Eviction-based cache placement for storage caches." in *USENIX Annual Technical Conference, General Track*, 2003, pp. 269–281.

[3] Github, "Source code of the project hosted on github," https://github.com/deepd/facebook-photo-caching, 2015.

[4] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li, "An analysis of facebook photo caching," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 167–181.

[5] B. Insider, "News article on Facebook white paper," http://www.businessinsider.com/facebook-350-million-photos-each-day-2013-9, 2013.

[6] Z. Liu, N. Niclausse, and C. Jalpa-Villanueva, "Traffic model and performance evaluation of web servers," *Performance Evaluation*, vol. 46, no. 2, pp. 77–100, 2001.