CS 654 Project Report

Boxin Lyu 20756975 b5lyu@uwaterloo.ca
Weitian Xing 20757406 weitian.xing@uwaterloo.ca
Jiangqi Zhang 20766023 jiangqi.zhang@uwaterloo.ca

Abstract

Feeding and caching strategies are crucial for the current social network services such as Twitter and Facebook. In this report we describe different approaches for constructing feeds, pull and push based, and approaches for caching the content of tweets, cache-aside and write-back. In order to compare the performance of them, we build a Twitter-like system, Twibo, and use SONETOR to generate synthetic user activities. The activities are then sent to the system for us to collect and analyze the performance statistics. The code and raw test results are available at `https://github.com/distsys-twibo/twibo`.

1 Introduction

Looking at the surface of social network services such as Twitter, they are like Google – fast, and reliable, whereas if we dive deeper into them, it is in fact a surprisingly sophisticated engineering problem to build a system that is able to display one's followee's posts on a user's cell phone in near real-time, not to mention making the system retrieve tweets for thousands of users in every second concurrently.

In this report, we discuss the essential building blocks of a social network service. The first is the construction of feeds, which is the process of retrieving the posts of other users from the database. Two methods for this are included, pull and push based[5] feeders. While the functionality of them are the same, the idea behind them are almost opposite.

The other part to discuss is the algorithms for caching, which is crucial for social networks services due to the fact that an extreme read/write ratio of 30:1 is observed in [9]. Two caching patterns are compared in this report – cache-aside and write-back[2]. Our experiments show that with proper caching mechanisms, the overall performance of the system could be boosted by 40% to 300%.

Finally, the last contribution of this report is a prototype social network system, Twibo. We build this system to get to know the idea of what a real social network service should look like, and what design decisions should be made to make the system easier to use and evolve.

2 Related Work

Caching has been playing an important role in social network services, as the performance of disk-based databases is highly limited and unscalable. In [1], Memached[3] is used as Facebook's in-memory caching solution, and several experiments are conducted for evaluating cache efficiency.

Hit rate is one of the most important metrics for evaluating cache efficiency. Several factors may affect it, including cache locality, user behavior and temporal patterns[9]. In our study, we focus on how memory size affects hit rates. Roselli found even small caches can have a high hit rate, while larger caches have diminishing returns[24].

Due to the difficulty of obtaining real-world user activities, [26] used synthetic workloads to analyze the performance of key-value stores. In this report, SONETOR[8] is used to evaluate the performance of our system; it is a tool for generating artificial activity traces with respect to certain probability distributions that were reported by [27] and [28].

3 Feeding Strategies

In an SNS system, the feeding strategy is the main factor affecting the user experience; it is important for users to get the newest posts from users they are following as soon as possible. Also, when users post updates, SNS system should distribute the updates to all the users' followers. When facing with a large number of users, the performance of the whole system could be affected by feeding strategy. In this section, we will introduce two basic strategies: pull-based and push-based. To some extent, these two strategies are similar to the pull and push strategies in distributed systems for propagating data[19].

The conceptual visualization of the two strategies are shown in figure 1 and 2.

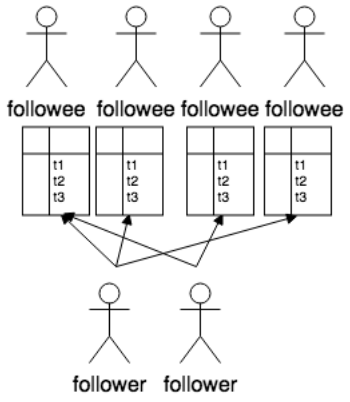


Figure 1: Direction of requests in a pull-based feeder.

3.1 Pull-based Feeder

For the pull-based strategy, the feeder maintains a global collection of posts and a per-user feed list storing tweets posted by each user. In addition, the system also maintains the follow and be-followed relations between users, which forms what is called a social graph. Every time a user sends a request for new feeds, the system finds and returns a set of latest tweets posted by the user's followees.

The advantage of this pull-based strategy is its easiness of implementation. However, a disadvantage is that retrieving new tweets is very costly when using this strategy; the number of database requests

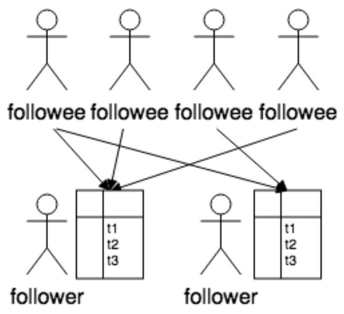


Figure 2: Direction of requests in a push-based feeder.

for each retrieve action could be the same as the number of followees of that user, which can be huge, leaving a bottleneck for the system.

3.2 Push-based Feeder

The idea behind the push-based feeder is completely opposite to the pull-based one; another per-user feed list, or timeline, is maintained by the feeder for the tweets that are posted by a user's followees. In this way, upon the receipt of a retrieve request, the system can simply look at this new feed list and return the latest tweets, avoiding the cost of querying the followees of a user and gathering their tweets from everywhere on the disk of the database.

A potential risk of this kind of feeder is that the cost of posting may be high for some celebrities. For example, Shaquille O'Neal is followed by nearly 15 million people on twitter[21]. When he posts a new tweet, it needs to be pushed to the timeline of all his 15 million followers, which is a tremendous amount of data that could bring a sudden burden to the database.

4 Caching Strategies

Caching is a widely used technique which could improve the performance of systems[22]. By using caches, a large portion of the workload of databases could be reduced, thus shortening the overall response time of requests. In this part, we discuss two common caching algorithms: cache-aside and write-back.

4.1 Cache-aside

Cache-aside is one of the most commonly used caching strategies due to its simplicity[1]. For read operations, the client first tries to retrieve the data from the cache, and if the data is found, the read operation is finished, otherwise the client queries the database for the data and then inserts the data into the cache by itself[13].

For update operations, updating data in the database that is also in the cache is more complex; the order of updating the database and cache could be tricky, since the wrong order may lead to dirty data. In this project, because we make the assumption that tweets created previously would not be modified in the future, only the read operation is implemented. The pseudo code of reading is shown in Algorithm 1.

The primary advantage of using cache-aside is that the system is resilient to cache failures. When cache misses, the data can still be read from the database. On the other hand, a disadvantage is that the system using the cache-aside pattern does not guarantee consistency between the data in the cache and database. To resolve this, setting an expire time for the data in the cache is usually used.

Algorithm 1 Cache-aside: Read

```
Input: $key$
Output: $value$
1: function READ($key$)
2:     $value = cache.get(key)$
3:     if $value \neq None$ then return $value$
4:     else
5:         $value = db.query(key)$
6:         $cache[key] = value$
7:     end if
8:     return $value$
9: end function
```

4.2 Write-back

When strong consistency is not mandated, the write-back caching algorithm is a good choice to speed up the modification of data[20]. While reading data can be implemented the same as the cache-aside pattern, creating and updating data in this pattern is processed directly and solely in the cache, not involving the database at all, and the speed of this is obviously high since there is no hard disk operation. Meanwhile, in order to persist the data in the cache, a background long-running process asynchronously collects all the modified data periodically and transfers the data to the database.

The advantage of this algorithm which modifies data in the cache directly is that the latency can be greatly reduced. However, it must be noted that the cost of this is that there is more inconsistency within the system, which is a trade-off that cannot be avoided.

The pseudo code of write-back strategy is shown in Algorithm 2. Since the read operation of is the same as cache-aside, only new functions are shown. A global queue is maintained to store the name of the keys whose corresponding data is modified and needs to be persisted to the database; the Write function runs forever in the background, retrieves new data from the queue and inserts them into the database.

5 Twibo

5.1 Overview

As its name "Twibo" suggests, this simulation system we build is a minimized prototype of the popular social network service Twitter[10] and its Chinese counterpart Weibo[11]. In this system, the core functionalities defining a social network are implemented, including users' relationship of follow

Algorithm 2 Write-back: Create

```
Input: $key, value$
function CREATE($key, value$)
    $cache[key] = value$
    $queue.append(key)$
end function
function WRITE
    while 1 do
        $key = queue.pop()$
        $value = cache[key]$
        $db.insert(key : value)$
    end while
end function
```

and be-followed, as well as posting tweets and retrieving tweets from the followees that are followed by a user. Thanks to the expressive nature of the programming language Python and the easiness and abundance of its third-party modules, we are able to build the whole system from scratch in less than three weeks.

In the subsequent parts of this section, we will introduce the interfaces of this system and give a detailed explanation of the system's internal modules.

5.2 Interfaces

The interfaces of this system that are exposed to the external world are relatively simple as the system is only a prototype, on the other hand they are also enough for us to evaluate the performance of different feeding and caching strategies. These interfaces are implemented as RESTful APIs[14], and the definitions of them are listed in table 1.

Name	**Method**	**Parameters**	**Response**
/user/create	POST	user id	none
/user/follow	POST	user id, target ids	none
/user/get	GET	user id	user id, followees, followers
/tweet/create	POST	user id, tweet id, content, timestamp	timers
/tweet/get	GET	user id, limit	tweets, timers

Table 1: Public interfaces of Twibo.

The first set of interfaces whose names begin with "/user" are designed to be used for constructing the user relationships, i.e. creating users in the system and establishing their relationships. However, this set of interfaces are not invoked when conducting the load tests; we use them to import the dataset before running the tests as an initialization, and in this way aspects of the system related to creating and retrieving tweets which are more important can be better focused.

The second set of interfaces whose names begin with "/tweet" are those of more importance to the system; when performing the load tests, these two interfaces are the ones that are called and whose performance statistics are measured.

The parameters for the interface of creating a tweet include the creator of the tweet and the tweet itself. These parameters are generated using a tool named SONETOR which will be discussed in a later section. The response of this interface are key-value pairs of runtime statistics of the system when processing this request, e.g. the time used for retrieving the followers of a user and the hit rate of cache.

The parameters of "/tweet/get" are simpler: id of the user who is trying to get new tweets from followees, and how many tweets this user wishes to get in this single request. The response consists of the content of tweets, and again, key-value pairs of timers.

5.3 Modules

The simulation system constitutes of three modules: route handler, feeders and persistence, which form the core of Twibo. In addition to them, load balancers, caching facilities and databases are also required for the system to function. The overview of the system is depicted in figure 3.

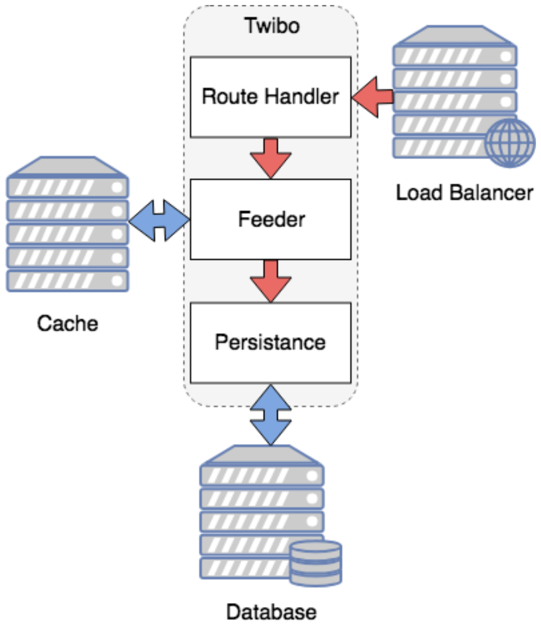


Figure 3: Modules of the simulation system.

5.3.1 Route Handler

The route handler acts as the frontend of Twibo; it accepts requests from clients or the load balancer, and then fulfil those requests by calling corresponding methods provided by the feeder module. It can be viewed as the "view" layer of the MVC model[15].

This module is implemented with aiohttp[16], a high-performance asynchronous web framework of Python. It ensures that the bottleneck of the system is not at the higher, business-logic related module.

5.3.2 Feeder

The feeder module is responsible for connecting the route handler to the underlying database. While different strategies can be applied, this module consistently provides two methods to the route handler module: create a tweet and get tweets. With this design, it is easier to switch between different kinds of feeders.

The implementation of the pull-based feeder is straightforward because this approch does not use cache; it simply manipulates the database directly. On the contrary, push-based feeders are more complex.

First is the per-user feed list; it is implemented with the list data structure of Redis. We create a limited-length list for every user, and append a new tweet to the head of a list when the followee of a user creates a new tweet. At the same time, we insert the new tweet to the database.

Next is the cache-aside version push-based feeder. In this version, creating a tweet still corresponds to inserting it directly into the database, but retrieving tweets now incorporates the cache-aside caching pattern, which is implemented with the normal key-value string pairs of Redis.

Finally, the write-behind version. As described above, the write-behind caching mechanism needs a background process to periodically propagate the updates in cache to the database; this is realized with a long-running coroutine which collects new tweets in the cache and flushes them to the database. While the implementation of retrieving tweets is the same as the cache-aside version, creating a tweet is now different; instead of inserting into the database, the content of new tweets are stored in the cache, and the IDs of them are inserted into the creator's followers' feed lists and also a global list in Redis which acts as a queue of tweets waiting to be persisted.

5.3.3 Persistence

There is no magic in this module; it is a simple abstract of the database and provides methods for retrieving the followees and followers of a user, inserting a tweet into the database and retrieving tweets by tweet IDs or user IDs.

6 User Activity Generation

6.1 The Model

In this section we introduce SONETOR[8], a social network traffic generator which statistically models users interactions. It accepts a user relation graph as input and generates synthetic user activity traces. For every user, SONETOR generates a series of sessions consisting of several activities, and for each activity, two kinds of actions are selected randomly, publish and retrieve, which correspond to creating a tweet and getting tweets from followees respectively.

SONETOR models user activities with two concepts. A session is a chunk of time that a user may actively interact with the system, and an activity is a specific action of either publish or retrieve. In terms of the frequency of sessions and activities, several parameters are introduced: inter-session time, inter-activity time, session length and number of sessions per user in a given period of time. Inter-session time refers to the interval between consecutive sessions, and interactivity time refers to the interval between two consecutive activities. Both of them are subject to log-normal distribution according to the data collected from real social networks[27][28]. Session length denotes the duration from logging in to the system to logging out. Sessions per user refers to the number of sessions for each user and represents the number of times that a user logs in. Both of them are subject to the zipf distribution[12], which states that the frequency of a word and its rank in the frequency table have an inversely proportional relationship.

As for the randomness of actions, publish and retrieve, Markov chain is used to model this. Figure 4 shows the two-state Markov process used by SONETOR, in which different probabilities, p and q, are assigned to the transition of states.

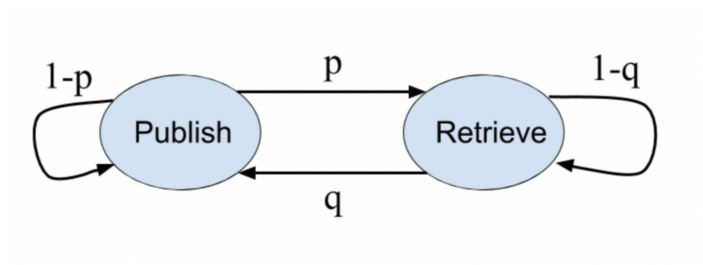
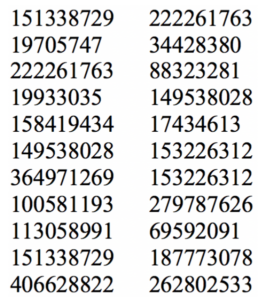


Figure 4: Transition of states.

6.2 Input and Output

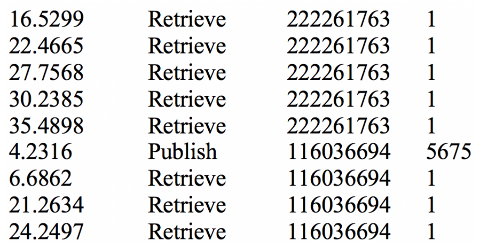
The input of SONETOR is a social network graph representing the follow and be-followed relationship between users. Figure 5 shows a concrete example of this. The data has two columns of user IDs, and each row denotes that the user in the first column follows the user in the second column.



151338729	222261763
19705747	34428380
222261763	88323281
19933035	149538028
158419434	17434613
149538028	153226312
364971269	153226312
100581193	279787626
113058991	69592091
151338729	187773078
406628822	262802533

Figure 5: Example of the input of SONETOR. The first column is the follower's ID, and the second column is followee's id.

Figure 6 shows the output of SONETOR, which has 4 columns: timestamp, action, user ID, and an action attribute. The action attribute is the length of a post for a "publish" action, and 1 for a "retrieve", which does not have any meaning but is for easier parsing of each line when reading the output.



```
16.5299    Retrieve    222261763    1
22.4665    Retrieve    222261763    1
27.7568    Retrieve    222261763    1
30.2385    Retrieve    222261763    1
35.4898    Retrieve    222261763    1
4.2316     Publish     116036694    5675
6.6862     Retrieve    116036694    1
21.2634    Retrieve    116036694    1
24.2497    Retrieve    116036694    1
```

Figure 6: Example of the output of SONETOR. The ratio of publish and retrieve is 1:9. Rows are sorted by the first column at a later step.

7 Experiments

7.1 Dataset

Social circles-Twitter[23] is used as the source of user relations. This dataset includes 81306 users and over 1.7 million relationships. The average number of followers and followees are 29.77 and 15.99; the user with the most followers has 8660 followers, and the highest number of followees for a user is 3373. The distribution of these values are shown in figure 7 and 8.

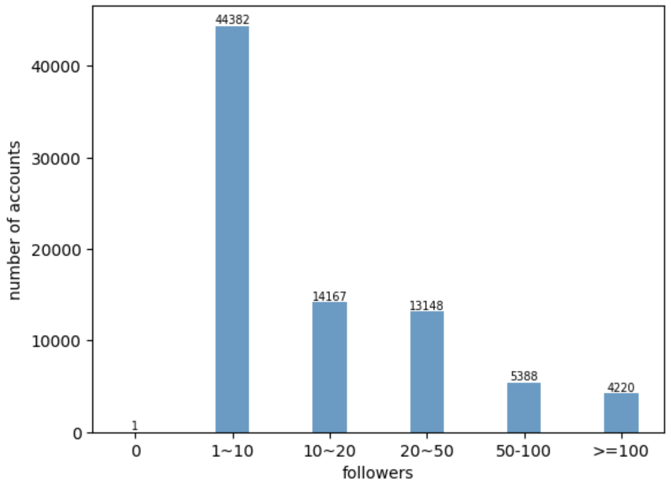


Figure 7: Number of followers in data set.

7.2 Environment Setup

The values of parameters we set for generating user activity traces are listed in table 2. By using this setting of Markov chain, 90% of activities are retrieve and 10% is publish, satisfying the usual workload pattern of a social network service.

We run Twibo distributedly on three CS-Teaching servers, each with 6 separate processes. The database, MongoDB, is deployed to an AWS EC2 server with an SSD disk whose IOPS is 1500. As for the caching backend, we launch two Redis instances on the CS-Teaching server, one for storing the feed

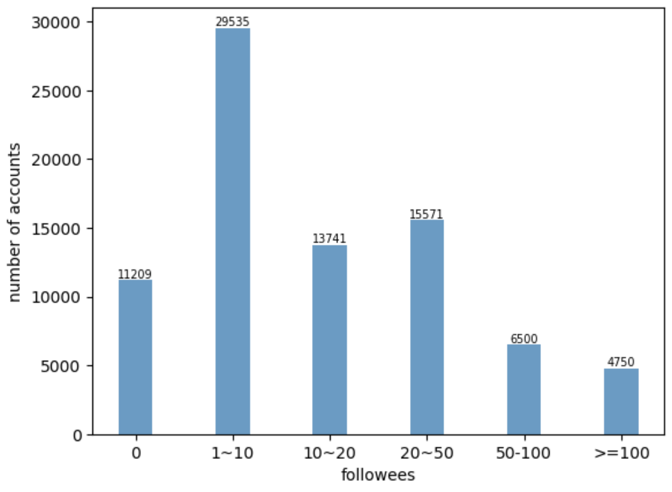


Figure 8: Number of followees in data set.

Parameter name	**Distribution**	**Parameter value**
Intersession time	Lognorm	s=1.789 and loc=2.366
Interactivity time	Lognorm	s=1.789 and loc=2.366
Session length	Zipf	a=1.765 and loc=4.888
Session per user	Zipf	a=1.792
Markov chain	none	p=0.9 and q=0.1

Table 2: Parameter setting in this experiment.

lists and one for the LRU caching of tweet content. By default, we limit the amount of memory for the LRU cache to be 16M. The load-test framework we used is multi-mechanize[17].

Before running each test, we clear the database, feed lists and cache, and reset them to the state when the first one million of publish activities from the output of SONETOR are sent to the system to mimic a more realistic initial environment. Then the activities after the one millionth activity are used as the "test set" and sent to system with different levels of concurrency.

7.3 Feeding Strategies

In order to compare the performance of pull and push based feeders, we load-test the system by launching different number of clients, and then calculate the number of queries processed per second by the system (QPS) and record the response time. Results are shown in table 3 and 4. Note that figures with an italic font means that the system is overloaded under that circumstance and the performance will continue to degrade if the duration of that load-test case is longer.

From the two tables, we can conclude that push based feeder could provide a better performance than pull based feeder; the throughput when using the push based feeder is roughly one time higher.

Feeder \ Clients	8	16	32	64	128	256	512
Pull	115.85	220.80	**333.15**	319.01	197.24	*86.13*	
Push	206.47	404.34	754.61	**964.01**	803.30	671.73	*504.81*
Push+Cache-Aside	206.72	409.87	786.86	1333.45	**2564.51**	2264.65	1811.87
Push+Write-Back	211.75	444.14	890.43	1732.05	**3205.73**	2896.42	2975.98

Table 3: QPS of Twibo under different levels of load when using different feeders. Higher is better.

Feeder \ Clients	8	16	32	64	128	256	512
Pull	76/33	85/35	110/37	408/66	1827/55	*6458/97*	
Push	36/71	39/77	51/97	153/241	386/860	899/942	*2413/3988*
Push+Cache-Aside	36/70	35/71	37/75	44/85	53/151	273/684	790/1040
Push+Write-Back	38/45	35/41	35/41	36/43	38/52	60/113	338/386

Table 4: 95th percentile response time (ms) of retrieving and creating tweets. Lower is better.

In addition, it is also noticeable that there exists a peak for both of these feeders, which is when the number of client is 32 for the pull based feeder and 64 for the push based. When more clients are used to send requests to the system, the performance would degrade significantly. This phenomenon indicates that the system is overloaded, which is verified by inspecting the disk activity on the server hosting the database, and adding more clients would only result in meaningless context switches between the requests rather than a higher throughput.

By inspecting the breakdown of the response time of retrieving tweets, it can be found that the time reduced by using the push based feeder is because it does not need to know which users are followed by the user who is retrieving, as opposed to the pull based feeder which needs to get all followees all the time. In the push based feeder, this operation is moved to the creation of a tweet, and this is ideal for a social network system because the read/write ratio is extremely high.

7.4 Caching Strategies

In table 3, the tests indicates that using cache could drastically boost the performance of the system. When there are 64 clients, the throughput is 38% and 80% higher when using the cache-aside and write-back strategy for the push based feeder, and the response time of retrieving tweets is also reduced by 71% and 76% respectively. The number of client required for these two approaches to reach the peak is also higher, implying that the pressure put on the disk of the database server is lower than the original approach, which is expected.

In terms of the response time of creating tweets in table 5, we observe that the write-back strategy does reduces it by approximately a half, and we can also see that the response time of retrieving is reduced as well. It might be worthwhile to explore in the future why the performance of retrieval is improved; we speculate that this is because the new tweets that are randomly coming are batched and sent to the database together, for which the database may be able to process more efficiently than to process them one by one.

7.5 Effect of Available Memory

To further explore the factors that affect the effectiveness of the cache, we test the system by limiting the memory for the cache with different values and fix the number of clients to be 64. The result of this experiment is shown in figure 9 and table 5.

There is a clear trend that with more memory available, the cache hit rate goes higher, and the number of database requests decreases as more tweets requested can be found in the cache directly, and consequently the throughput of the system is higher.

Meanwhile, we observe that the hit rate would increase linearly as more memory is used initially, and would then saturate, which is about 16M when there are 64 clients, and it is at this point that the throughput of the system stops increasing. This saturation point is similar to the peak in previous experiments, but it is not likely that the system's performance would degrade if even more memory is added.

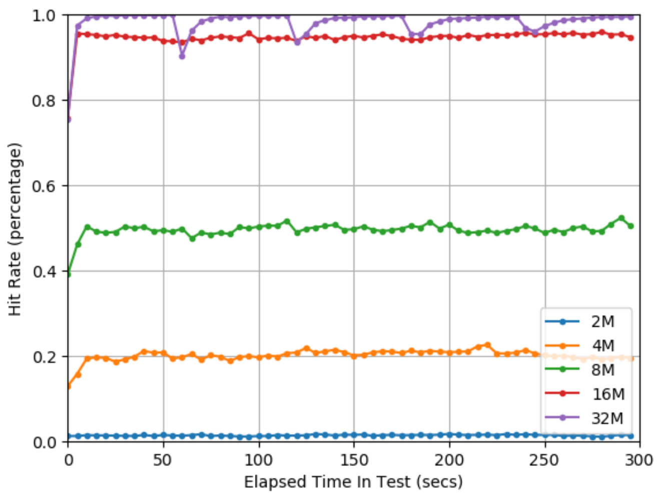


Figure 9: Cache hit rates and their relation to the amount of available memory.

Memory / Feeder	2M	4M	8M	16M	32M
Push+Cache-Aside	997.89/31.74	1068.60/25.21	1305.18/15.70	1578.08/1.64	1582.04/0.49
Push+Write-Back	1145.78/26.50	1306.39/16.66	1577.31/9.02	1656.14/0.88	1711.29/0.42

Table 5: QPS and average number of DB requests per "retrieve" request.

8 Conclusion

In this report, the essentials of a social network service are discussed, that is the feeding and caching strategies. Our experiments show that the push based feeder has a better performance than the pull based feeder, and using caching facilities is crucial for a system whose read request dominates. It is also shown that more memory available for the cache could be beneficial for the system's performance, but in reality system designers should notice that there exists a saturation point; adding more memory after this point would not yield much gains.

References

[1] Nishtala, Rajesh, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy et al. "Scaling memcache at facebook." In Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13), pp. 385-398. 2013.

[2] Architecture of sina weibo for multi-billion requests per day, `https://cloud.tencent.com/info/94286ab1a431cbb5961c538b2d517e3f.html`.

[3] Memcached, `https://memcached.org`.

[4] Redis, `https://redis.io`.

[5] Van Houdt, Benny. "Performance comparison of aggressive push and traditional pull strategies in large distributed systems." In 2011 Eighth International Conference on Quantitative Evaluation of SysTems, pp. 265-274. IEEE, 2011.

[6] Leskovec, Jure, and Julian J. Mcauley. "Learning to discover social circles in ego networks." In Advances in neural information processing systems, pp. 539-547. 2012.

[7] De Domenico, Manlio, Antonio Lima, Paul Mougel, and Mirco Musolesi. "The anatomy of a scientific rumor." Scientific reports 3 (2013): 2980.

[8] Bernardini, Csar, Thomas Silverston, and Olivier Festor. "SONETOR: A social network traffic generator." In 2014 IEEE International Conference on Communications (ICC), pp. 3734-3739. IEEE, 2014.

[9] Atikoglu, Berk, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. "Workload analysis of a large-scale key-value store." In ACM SIGMETRICS Performance Evaluation Review, vol. 40, no. 1, pp. 53-64. ACM, 2012.

[10] Twitter, `https://twitter.com`.

[11] Weibo, `https://www.weibo.com`.

[12] Adamic, Lada A., and Bernardo A. Huberman. "Zipf's law and the Internet." Glottometrics 3, no. 1 (2002): 143-150.

[13] Pamula, Narendra Babu, K. Jairam, and B. Rajesh. "Cache-aside approach for cloud design pattern." Int. J. Comput. Sci. Inf. Technol 5, no. 2 (2014): 1423-1426.

[14] Richardson, Leonard, and Sam Ruby. RESTful web services. " O'Reilly Media, Inc.", 2008.

[15] Krasner, Glenn E., and Stephen T. Pope. "A description of the model-view-controller user interface paradigm in the smalltalk-80 system." Journal of object oriented programming 1, no. 3 (1988): 26-49.

[16] aiohttp, `https://github.com/aio-libs/aiohttp`.

[17] multi-mechanize, `https://github.com/cgoldberg/multi-mechanize`.

[18] Denning, Peter J. “The locality principle.” In Communication Networks And Computer Systems: A Tribute to Professor Erol Gelenbe, pp. 43-67. 2006.

[19] Van Houdt, Benny. “Performance comparison of aggressive push and traditional pull strategies in large distributed systems.” In 2011 Eighth International Conference on Quantitative Evaluation of SysTems, pp. 265-274. IEEE, 2011.

[20] Jouppi, Norman P. Cache write policies and performance. Vol. 21, no. 2. ACM, 1993.

[21] SHAQ, `https://twitter.com/SHAQ`

[22] Tanenbaum, Andrew S., and Maarten Van Steen. Distributed systems: principles and paradigms. Prentice-Hall, 2007.

[23] J. McAuley and J. Leskovec. Learning to Discover Social Circles in Ego Networks. NIPS, 2012.

[24] Roselli, Drew S., Jacob R. Lorch, and Thomas E. Anderson. “A Comparison of File System Workloads.” In USENIX annual technical conference, general track, pp. 41-54. 2000.

[25] Lublin, Uri, and Dror G. Feitelson.“The workload on parallel supercomputers: modeling the characteristics of rigid jobs.” Journal of Parallel and Distributed Computing 63, no. 11 (2003): 1105-1122.

[26] Vasudevan, Vijay R. Energy-efficient data-intensive computing with a fast array of wimpy nodes. No. CMU-CS-11-131. CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE, 2011.

[27] Benevenuto, Fabrcio, Tiago Rodrigues, Meeyoung Cha, and Virglio Almeida. “Characterizing user behavior in online social networks.” In Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement, pp. 49-62. ACM, 2009.

[28] Gyarmati, Lszl, and Tuan Anh Trinh. “Measuring user behavior in online social networks.” IEEE network 24, no. 5 (2010): 26-31.