Caching at the Web Scale

[Tutorial]

Victor Zakhary
University of California Santa Barbara
Santa Barbara, 93106
California, USA
victorzakhary@cs.ucsb.edu

Divyakant Agrawal
University of California Santa Barbara
Santa Barbara, 93106
California, USA
agrawal@cs.ucsb.edu

Amr El Abbadi
University of California Santa Barbara
Santa Barbara, 93106
California, USA
amr@cs.ucsb.edu

ABSTRACT

Today's web applications and social networks are serving billions of users around the globe. These users generate billions of key lookups and millions of data object updates per second. A single user's social network page load requires hundreds of key lookups. This scale creates many design challenges for the underlying storage systems. First, these systems have to serve user requests with low latency. Any increase in the request latency leads to a decrease in user interest. Second, storage systems have to be highly available. Failures should be handled seamlessly without affecting user requests. Third, users consume an order of magnitude more data than they produce. Therefore, storage systems have to be optimized for read-intensive workloads. To address these challenges, distributed in-memory caching services have been widely deployed on top of persistent storage. In this tutorial, we survey the recent developments in distributed caching services. We present the algorithmic and architectural efforts behind these systems focusing on the challenges in addition to open research questions.

Keywords

Distributed caching, Memcached, Replacement policy, Contention

1. INTRODUCTION

During the past decade, social networks have attracted hundreds of millions of users [4, 8]. These users share their relationships, read news [15], and exchange images and videos in a timely personalized experience [10]. To enable this real-time personalized experience, the underlying storage systems have to provide efficient, scalable, highly available access to big data. Social network users consume an order of magnitude more data than they produce [9]. In addition, a single page load requires hundreds of object lookups that should be served in a fraction of a second [10]. Therefore, traditional disk-based storage systems are not suitable to handle requests at this scale due to the high access latency of disks and I/O throughput bounds [22].

To overcome these limitations, distributed caching services have been widely deployed on top of persistent storage in order to efficiently serve user requests at scale. Akamai and other CDNs use distributed caching to bring data closer to the users and to reduce access latency. Memcached [5] and Redis [7] are two distributed open source cache implementations that are widely adopted in the cloud and social networks. The default implementations of memcached and Redis use the Least Recently Used (LRU) cache replacement policy. Although LRU is simple and easy to implement, it might not achieve the highest cache hit rates for some deployments. Increasing the hit rate by 1% can save up to 35% of the average read latency [12]. Therefore, much effort has focused on developing better caching policies that achieve higher cache hit rates [20, 16, 13, 17]. Teams in Facebook [19, 10, 14] and Twitter [3] have focused on the architectural challenges of distributed caching at datacenter scale. Sharding, replication, request batching, load balancing, hierarchical caching, data access skewness, geo-replication, replica consistency, data updates, and cache invalidation are examples of the architectural challenges for distributed caching and current implementations address some of these challenges.

In Section 3.1, we present the data access model. Then, we summarize the recent efforts on cache replacement policies at a single server level in Section 3.2. Finally, we present real deployed systems at a datacenter scale in Section 3.3.

©2017 International World Wide Web Conference Committee (IW3C2), published under Creative Commons CC BY 4.0 License.
WWW 2017, April 3–7, 2017, Perth, Australia.
ACM 978-1-4503-4913-0/17/04.
http://dx.doi.org/10.1145/3041021.3051098

2. TUTORIAL INFORMATION

This is a **half day** tutorial targeting researchers, designers, and practitioners interested in systems and infrastructure research for big data management and processing. The **target audience** with basic background about cache replacement policies, sharding, replication, and consistency would benefit the most from this tutorial. For general audience and newcomers, the tutorial introduces the design challenges that arise when caching services are designed at the web scale. For researchers, algorithmic and architectural efforts in distributed caching are presented together showing the spectrum of recent solutions side by side with their unhandled challenges. Our goal is to enable researchers to develop designs and algorithms that handle these challenges at scale. We prepared some supportive slides covering different design decisions and reasoning behind these decisions. These slides are hosted under the url: `http://cs.ucsb.edu/~victorzakhary/www17/supportive/`

3. TUTORIAL OUTLINE

3.1 Data Access Model

We assume millions of end-users sending streams of page-load and page-update requests to hundreds of stateless application servers as shown in Figure 1a. Application servers hide the storage details from the clients and reduce the number of connections handled by the storage system. Each request is translated to hundreds of key lookups and updates. As traditional disk-based systems cannot efficiently handle user requests at scale, caching services have been widely used to enhance the performance of web applications by alleviating the number of requests sent to the persistent storage. The ultimate objective of caching services is to achieve a high hit-rate because the latency of a cache miss is usually few orders of magnitude more than a cache hit. Therefore, designing a caching service to serve a very large key space using commodity machines introduces many challenges. First, the key space is very large and serving the whole key space using a single machine violates cache locality and increases the miss-rate. To overcome this problem, designers shard the key space into multiple partitions using either range or hash partitioning and use *distributed caching servers* to serve different shards. Second, the key space, even after sharding, is too large to fit in memory of commodity servers. Therefore, a *cache replacement policy* has to be carefully designed to achieve high hit-rates without adding a significant bookkeeping overhead. Also, policies should avoid using shared datastructures between threads to reduce *contentions*. Third, commodity machines can fail and *replication* is needed to distribute the workload and achieve high availability. Figure 1b shows an abstract model for a *distributed caching service* where each caching server is serving a specific shard and each shard is served by multiple replicas.

3.2 Replacement policy-base solutions

Memcached[5] and Redis[7] are two widely adopted open source implementations for distributed caching services. Memcached provides a simple *Set, Get,* and *Delete* interface for only string keys and values while Redis extends the interface to handle other datatypes. Cloud providers have adopted memcached and Redis and provide customized versions of both as services for their clients [1, 6, 2]. Both memcached and Redis use the LRU cache replacement policy which only tracks the time of access of each key in the cache. Other efforts enhance the performance by introducing more tracking per key access or by sharding the hash table and the tracking datastructures to reduce the contention between threads and avoid global synchronization. Adaptive Replacement Cache ARC[17] tracks the recency and the frequency of access in addition to the recency of key eviction to decide which key should be evicted next. To reduce thread contention, memcached divides the memory into slabs for different object sizes and each slab maintains its tracking information independently. CPHASH[18] introduces a concurrent hash table for multi-core processors. Message passing is used to transfer lookups and inserts between slabs. Request batching and locks avoidance allows CPHASH to achieve 1.6x better throughput than a hash table with fine-grained locks. Sharding the memory between slabs uniformly can lead to under utilization for some slabs and high miss-rate for others. Therefore, Cliffhanger[12] dynamically shards the memory between slabs to achieve the highest overall hit rate. Dynacache[11] dynamically profiles the applications' workload, shards the memory resources, and decides the replacement policy that achieve the highest cache-hit rate for the profiled workload. To avoid slab under-utilization, other solutions suggest mapping keys to multiple slabs. Cuckoo hashing[20] defines two hash functions that map every key to two memory slabs. An insertion might trigger a sequence of insertions or an eviction if the memory slabs of the inserted key are full. Lookups require to check the key in its corresponding two buckets. To optimize the lookup cost, MemC3[13] uses tags for fast lookups optimizing cuckoo hashing for read-dominated workloads and allows multi-reader single-writer concurrent accesses. Li et al.[16] introduces fast concurrent cuckoo hashing to support high throughput multiple writers. Other solutions reduce the miss-rate by increasing the effective cache size. zExpander[21] achieves this by representing data objects in compact and compressed formats.

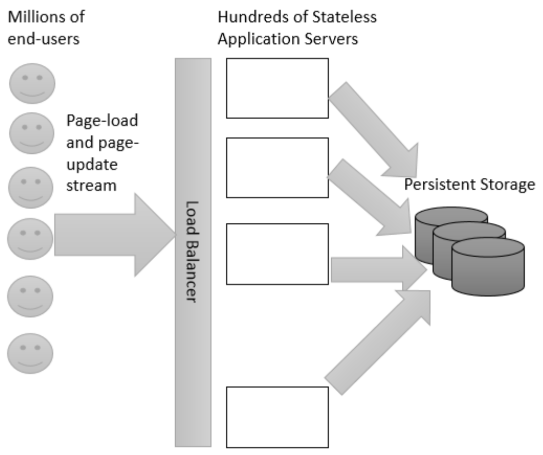
3.3 Real deployed systems

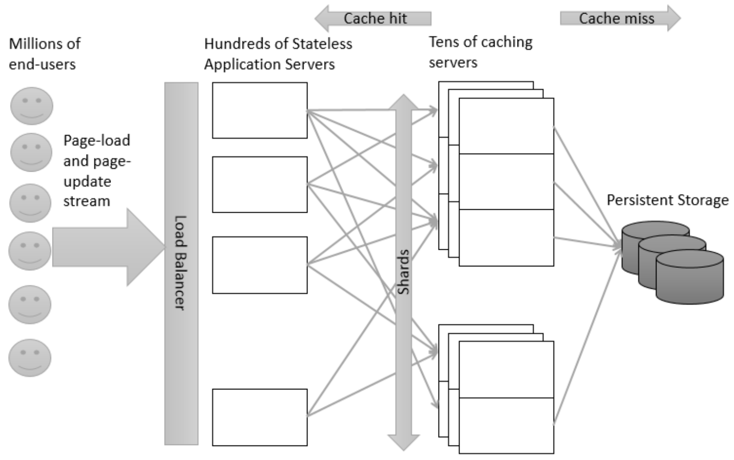
Facebook [19] and Twitter [3] have built their own distributed version of memcached. Facebook scaled memcached [19] by partitioning the key space into different pools. Each pool is served by multiple replicas to tolerate failures and distribute the lookup workload. Both Twitter and Facebook implementations batch requests in a client proxy to reduce the number of requests sent to the server. Invalidation messages are sent from the persistent storage to the cache replica to invalidate the stale values. Facebook and Twitter use memcached as a lookaside cache. However, memcached is not optimized to capture a graph storage model. Therefore, Facebook built Tao [10], a distributed caching service optimized for graph storage models. In Tao, nodes and their associations are served from the same caching server. Tao uses storage and caching *geo-replication* to overcome a datacenter scale outage. Updates go to the master storage replica through a cache leader server which is responsible for all the updates and the invalidation messages for all the data items in its shard. In Tao, heavy hitters (hot data objects) are handled by introducing hierarchical caching where heavy hitters are cached in the upper hierarchy. Both Tao and memcached at Facebook support eventual consistency between the replicas.

The current systems have addressed many of the distributed caching challenges. However, challenges like 1) data access skewness, 2) dynamical changes in access pattern, 3) providing stronger guarantees of replica consistency, and 4) providing consistency between multiple data representations are still open research questions that require innovative algorithmic and architectural solutions to provide these guarantees at scale.

4. BIOGRAPHICAL SKETCHES

Victor Zakhary is a PhD student at the University of California at Santa Barbara. His current research work is in the areas of data placement for geo-replicated databases and for distributed caching services to achieve low access latency and dynamically handle data access skewness.


Millions of end-users
Hundreds of Stateless Application Servers
Page-load and page-update stream
Load Balancer
Persistent Storage
(a) System model without caching



(b) System model with caching

Figure 1: System model

Divyakant Agrawal is a Professor of Computer Science at the University of California at Santa Barbara. His current interests are in the area of scalable data management and data analysis in cloud computing environments, security and privacy of data in the cloud, and scalable analytics over big data. Prof. Agrawal is an ACM Distinguished Scientist (2010), an ACM Fellow (2012), and an IEEE Fellow (2012).

Amr El Abbadi is a Professor of Computer Science at the University of California, Santa Barbara. Prof. El Abbadi is an ACM Fellow, AAAS Fellow, and IEEE Fellow. He was Chair of the Computer Science Department at UCSB from 2007 to 2011. He has served as a journal editor for several database journals and has been Program Chair for multiple database and distributed systems conferences. Most recently Prof. El Abbadi was the co-recipient of the Test of Time Award at EDBT/ICDT 2015. He has published over 300 articles in databases and distributed systems and has supervised over 30 PhD students.

5. ACKNOWLEDGEMENTS

This research was supported by gifts from Oracle Research and Huawei Research.

6. REFERENCES

[1] Amazon elasticache in-memory data store and cache. `https://aws.amazon.com/elasticache/`.
[2] Azure redis cache. `https://azure.microsoft.com/en-us/services/cache/`.
[3] Caching with twemcache. `https://blog.twitter.com/2012/caching-with-twemcache/`.
[4] Facebook company info. `http://newsroom.fb.com/company-info/`.
[5] Memcached. a distributed memory object caching system. `https://memcached.org/`.
[6] Memcachier. `https://www.memcachier.com/`.
[7] Redis. `http://redis.io/`.
[8] Twitter: number of active users 2010-2016. `https://www.statista.com/statistics/282087/number-of-monthly-active-twitter-users/`.
[9] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.
[10] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, et al. Tao: FacebookâĂŹs distributed data store for the social graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, 2013.
[11] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti. Dynacache: Dynamic cloud caching. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, 2015.
[12] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 379–392, Santa Clara, CA, Mar. 2016. USENIX Association.
[13] B. Fan, D. G. Andersen, and M. Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 371–384, 2013.
[14] 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*, pages 167–181. ACM, 2013.
[15] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web*, pages 591–600. ACM, 2010.
[16] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the Ninth European Conference on Computer Systems*, page 27. ACM, 2014.

[17] N. Megiddo and D. S. Modha. Arc: A self-tuning, low overhead replacement cache. In *FAST*, volume 3, pages 115–130, 2003.

[18] Z. Metreveli, N. Zeldovich, and M. F. Kaashoek. Cphash: A cache-partitioned hash table. In *ACM SIGPLAN Notices*, volume 47, pages 319–320. ACM, 2012.

[19] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, 2013.

[20] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[21] X. Wu, L. Zhang, Y. Wang, Y. Ren, M. Hack, and S. Jiang. zexpander: a key-value cache with both high performance and fewer misses. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 14. ACM, 2016.

[22] H. Zhang, G. Chen, B. C. Ooi, K.-L. Tan, and M. Zhang. In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 27(7):1920–1948, 2015.