DYNAMICCACHE: A Sharded and Replicated Key-Value Cache with Dynamic Shard Replication

James Zhao
Yale University

Abstract

Caching is widely used to increase application throughput and latency. However, changing load conditions (read/write access patterns, query distribution) often hinder a cache from achieving optimal performance. Modifying the cache such that it dynamically alters its sharding and replication in response to varying workload conditions can largely improve cache performance. We present DYNAMICCACHE, a dynamic key-value cache that adapts sharding and replication to optimize for throughput and latency under dynamic access patterns. For evaluations, we run simulated queries with changing load distributions, demonstrating that DYNAMICCACHE is able to adapt shard replication to query patterns and achieves superior performance in query latency and throughput (up to 66% latency and 18% throughput improvements) compared to the baselines.

1 Introduction

Software caches are deployed throughout layers of data center infrastructure. A substantial amount of resources is spent on caching. For example, Twitter reports that caching layer uses TBs of DRAM storage [1], and Netflix reports that 10s of PBs of storage is reserved for caching [2]. The main driving force of cache deployments is cache's ability to serve data with high throughput and low latency. Fetching the data from cache is orders of magnitude faster than retrieving it from a slower backend (e.g., spinning disks), and a cache often serves requests with at least hundreds of times higher throughput than a backend storage.

Cache sharding and replication are crucial in improving the cache performance. Sharding breaks up the data into multiple parts (shards), each of which can be read and written to concurrently, thus increasing read and write throughput. Replication duplicates the data into multiple replicas, each of which can be accessed from concurrently, thus increasing read throughput. However, dynamic sharding introduces the difficulty of shard migration [3,4], where system has to copy data from existing shards to form a new shard. Adopting replication also leads to write amplification, where write requests need to be replicated across multiple shard replicas.

The challenge lies in balancing sharding and replication for a given load condition such that the effective throughput for read and write is maximized. Workload conditions may be dynamic, meaning that the read and write composition and key query distribution may vary with time. For example, internet services often exhibit periodic daily variations in load due to the global distribution of their users [5].

We present DYNAMICCACHE, a key-value cache that adjusts shard replication *dynamically* given the workload. DYNAMICCACHE achieves this by continuously monitoring the read and write latency and throughput and using them as indicators for load on a particular shard. We designed a shard manager that adjusts the shard replication by adding replicas to a busy shard and removing a replica from an idle shard. We report that DYNAMICCACHE is effective in easing the shard overload through a series of evaluations, achieving up to 66% latency reduction and 18% throughput improvements .

2 Related Work

2.1 Software caches

Web applications rely heavily on caching to speed up requests and increase system throughput. Modern caches store computation results [1] using either embedded deployments [6,7] or distributed deployments over the network [8,9].

A cache needs to remain performant (i.e., provide high throughput). Throughput reflects the volume of requests the cache can handle in a given duration. Higher throughput means less CPU resource is needed for the same workload, which translates to reduced expenses.

2.2 Shard management

Various shard management frameworks were introduced that dynamically adapt sharding to query patterns. Vartolomei [10] describes how to optimize sharding and replication given a uniform key query distribution and a dynamic read/write composition. Facebook's ShardManger is another shard management framework that dynamically allocates shards according to user-specified goals and constraints [11]. ShardManager uses a generic constrained optimization solver with local search instead of mixed-integer programming to allocate shards optimally. Twine [4] replaces static-sharding with dynamically balanced sharding strategy via shard migration. Blowfish [12] uses dynamic load balancing and shard management across servers, where shards maintain request queues that are used both to load balance queries as well as to manage shard sizes within and across servers.

Prior works [13, 14] also explored shard selection based on search query patterns and indexing, so that a query can more efficiently skip shards that do not contain relevant data that answers the given query.

DynamicCache differs from prior works by investigating how to adapt to changing query load and patterns by varying the number of shard replicas.

3 System Design

DynamicCache consists of three components: User Client, Distributed Cache, and Shard Manager. The User Client, each node for the Distributed Cache, and shard manager are implemented in `client.go`, `server.go`, and `shardmanager.go` respectively.

Shard replicas are stored on server nodes. Each shard replica is associated with one RWLock such that one node can support concurrent write accesses to different shard replicas stored on that node. The number of concurrent write accesses is bounded by the number of shards housed on that node. Each node will at most store one replica for a shard. Each shard can have multiple replicas. Both the User Client and the Shard Manager have access to the `ShardMap` state, which indicates the state of the Distributed Cache (which nodes host which shards). In distributed setting, `ShardMap` is synchronized through heartbeat messages. Request keys are mapped to shards via a hash-based load balancer.

Users can access DynamicCache through the User Client API by submitting `Get(key)`, `Set(key, value)`, and `Delete(key, value)` requests. For `Get` requests, the User Client hashes the key to determine the relevant shard and reads from one of the r shard replicas at random. r is the replication factor for that shard (Figure 1 illustrates the case where $r = 2$). For `Set` and `Delete` requests, the User Client again hashes the request key to determine the relevant shard and then updates the key and value for all of r shard replicas.

3.1 Performance Metrics

We monitor the performance of DynamicCache with two metrics: per-shard requests per second (*RPS*) and per-shard

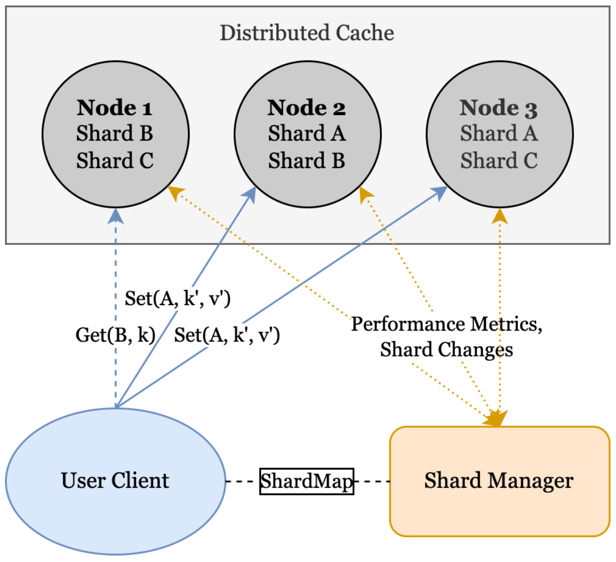

Figure 1: Our distributed cache system. A User Client sends Get and Set requests for a certain key k and the requests are automatically sent to the relevant node(s). Each node contains instrumentation monitoring per-shard-replica latency T and *RPS*. The Shard Manager periodically pulls latency T and *RPS* from each of the nodes, and uses this information to issue add / drop shard replicas commands to the distributed cache.

replica latency (T). Replica latency is measured at 99 percentile using `t-digest` package. Each node calculates these two performance metrics for all of its shard replicas and stores them in memory. The Shard Manager queries each node for its performance metrics via a `getStats` remote procedure call. For each of the two performance metrics, we calculate separate statistics for read requests and write requests latency and throughput for each *shard* by averaging the values over each *shard replica* retrieved from each node.

RPS The per-shard average incoming requests per second (*RPS*), measured over a time interval T equal to the Shard Manager performance metrics update frequency and averaged across all shard replicas per shard.

Latency The per-shard latency, measured from when each request is first received and queued by each node to when the response is finally returned by the node and averaged across all replicas per shard.

After testing both of these performance metrics, we found that latency provides a more efficient sharding strategy compared to requests per second. Therefore, for this project we utilized per-shard latency as our primary performance metric.

3.2 Shard Placement and Load Balancing

After experimenting with several approaches, we decided to determine shard placement via a greedy heuristic-based algorithm that increases or decreases the number of shard replicas if the average shard latency T falls outside a certain range. We decided on this approach because it was simpler and easier to understand than other constrained optimization or learned approaches. It is difficult in practice to support sophisticated shard-placement strategies [11]. The logic for shard placement and load balancing is implemented in the Shard Manager and is straightforward: the shard manager periodically pulls performance metrics from all of the nodes at regular time intervals and calculates the per-shard average total requests per second (RPS) and average per-shard latency (T), adding up reads and writes. For each shard it then compares RPS and T to some thresholds. For example, the shard manager can add a shard if $RPS > RPS_{high}$ is exceeded and remove a shard if $RPS < RPS_{low}$. Similarly, the shard manager compares T to a specific threshold, adding a shard if $T > T_{high}$ and dropping a shard if $T < T_{low}$.

Algorithm 1 summarizes the process using per-shard latency T as an example. For implementation, we considered both RPS and T as indicators of shard load. C_{max} is an array which indicates the maximum number of shard replicas per shard. $[C_1, C_2, ...]$. $C_{replicas}$ is an array that stores the current number of shard replicas for each shard. Shard number is used as the index of the two arrays. $get\_shard\_latency(i)$ obtains per-shard-replica from each node and calculates per-shard latency by averaging per-shard-replica latency over all replicas of the shard.

Algorithm 1 Shard Management Algorithm

```
Max shard replica counts: $C_{max} \leftarrow [C_1, C_2, ...]$
Current shard replica counts: $C_{replicas} \leftarrow [1, 1, ...]$
for every time interval $T$ do
    for every shard $i$ do
        $T_{shard} \leftarrow get\_shard\_latency(i)$
        if $C_{replicas}[i] < C_{max}$ and $T_{shard} > T_{high}$ then
            $add\_shard(i)$
            $C_{replicas}[i] = C_{replicas}[i] + 1$
        else if $C_{replicas}[i] > 1$ and $T_{shard} < T_{low}$ then
            $drop\_shard(i)$
            $C_{replicas}[i] = C_{replicas}[i] - 1$
        else
            pass
        end if
    end for
end for
```

Thus, when a shard experiences high per-shard read or write latency, shard manager will add a shard replica to a node where that particular shard has not been placed. Data from old shard replicas will be copied to the new shard replica to ensure

Time	1	2	3	4	5
T_{read}	330.51	4.89	296.08	4.33	253.33
T_{write}	3848.53	559.46	4599.65	1116.09	4599.65
r_{shard}	2	1	2	1	2

Table 1: An example of dynamic shard replication given a periodic query pattern. T_{read} is the P99 read latency of that shard for the past timestep. T_{write} is the P99 write latency of that shard for the past timestep. r_{shard} is the *new* replication factor for that shard. Latency is measured in microseconds

that all shard replicas share the same content. Similarly, when a shard experiences low write latency but still have redundant shard replicas, shard manager will remove a shard replica from that shard. This is to reduce the resource usage if the shard does not need to handle high workload and do not need many shard replicas. The freed shard replicas are returned back to a *pool* of available shard replicas and can then be used for other hot shards.

Table 1 illustrates an example of this process. T_{write} and T_{read} measures per-shard write or read latency in the *previous* time step. r_{shard} is the *new* replication factor for that shard.

When T_{read} becomes too high, Shard Manager will add one replica for the shard. For example, when time is 1, because for the past time step, T_{read} is high (330.51 us) and T_{write} is high (3848.53us), the shard manager adds one shard replica, increasing the number of replica from 1 to 2. When the time step is 2, because T_{read} decreases to 4.89us and T_{write} decreases to 559.46us, the shard manager removes one shard replica, decreasing the number of replicas from 2 to 1. For time step equals 3, because T_{read} and T_{write} both increase significantly, the shard manager adds one replica to the shard. This example illustrates that DYNAMICCACHE is able to adapt to changing query pattern by dynamically adjusting the number of shard replicas.

Note that we can adjust the interval of changing shard replication so that shard manager does not introduce too much overhead. We can also explore the best latency threshold for adjusting shard replication.

4 Evaluations

Evaluations are done on Apple Macbook with 10 CPU cores, 16 GB of memory.

We compare DYNAMICCACHE with Baseline-1 (lab4, without shard manager, and with shard replication set to 1) and Baseline-5 with shard replica count set to 5. We also set the maximum number of shard replica DYNAMICCACHE can have to 5, meaning that the shard manager will not allocate more replicas than this amount.

We compare DYNAMICCACHE with both Baseline-1 and Baseline-5 as we want to investigate whether the potential improvement is brought by *dynamically adjusting shard replicas*

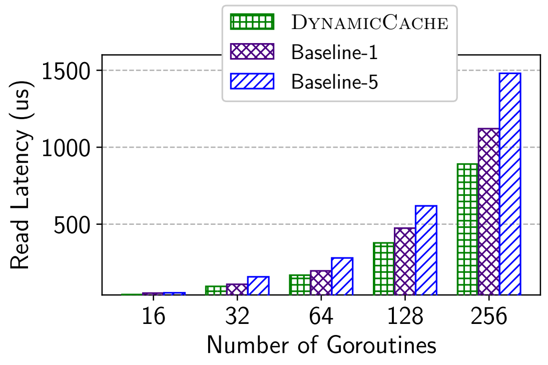

Figure 2: Average per-shard read latency comparison for DynamicCache, Baseline-1, Baseline-5. The Y-axis records the average per-shard read latency, measured in microseconds.

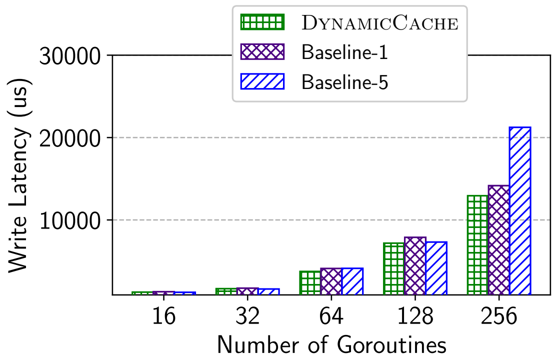

Figure 3: Average shard write latency comparison for DynamicCache, Baseline-1, and Baseline-5. The Y-axis records the average per-shard read latency, measured in microseconds.

or *adding more shards*.

In this evaluation, we control the number of goroutines to vary the amount of contention on our system. We measure the average per-shard read latency, average per-shard write latency, and throughput of the compared systems.

We first evaluate how dynamically adjusting shard replication performs in terms of read latency. Fig. 2 compares the average per-shard read latency between the three setups (DynamicCache, Baseline-1, and Baseline-5). We observe that using shard manager improves the average shard read latency. This is because adding more shard replica spreads out the read requests, alleviating lock contention under high load scenario. DynamicCache consistently outperforms Baseline-1 and Baseline-5 across all number of goroutines, and the improvement brought by the shard manager increases as the number of goroutines increases. With 256 number of goroutines, Baseline-5 with the maximum number of shard replicas (5) suffers from up to 66% times compared to DynamicCache. This suggests that in the high-contention scenario (larger number of goroutines) always having a high number of replicas may not be a good strategy due to replicated write requests for multiple shard replicas. On the flip side, we observe that always using a low shard replication is not optimal either, Baseline-1 is up to 26% slower than DynamicCache in read latency. This evaluation illustrates the importance of dynamically adjusting shard replication factor.

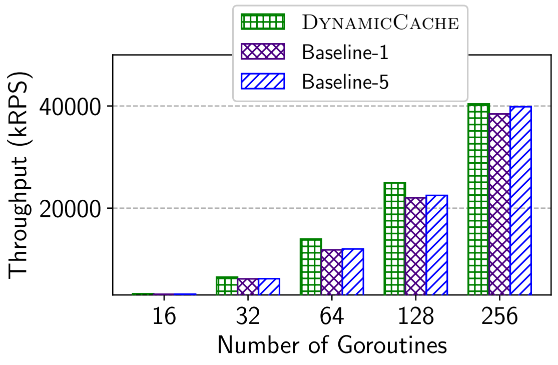

Figure 4: System throughput comparison for DynamicCache, Baseline-1, and Baseline-5.

We observe similar trend with average shard write latency, as shown in Fig. 3, showing that DynamicCache improves the write performance even with more write requests. We note that the write latency for baseline with 5 shard replicas is much higher than DynamicCache and Baseline-1, indicating that over-provisioning the shard replicas (by picking the maximum replication factor) degrades the write performance, as expected. Notably, DynamicCache achieves 39% latency improvement compared to Baseline-5. By monitoring when the shard becomes hotter or cooler, DynamicCache is able to remove unnecessary replicas from a less-requested shard thereby minimizing the unnecessary read requests.

The throughput comparison is presented in Fig. 4. DynamicCache outperforms both the Baseline-1 and Baseline-5, indicating that the dynamic balancing of shard replicas improves throughput performance, up to 18%. We believe that by dynamically adjusting the shard replication, DynamicCache is able to adapt to changing workloads and ensure that there is no hot key or unnecessarily replicated shards, thereby achieving high throughput. We conduct experiments with both uniform and hot key query patterns. For uniform query patterns we generate 100 keys evenly spread out across the five shards, while for hot key we generate keys that will be hashed to the same shard.

5 Conclusion

We have presented DynamicCache, a sharded and replicated key-value cache that dynamically allocates shard replicas. Our evaluation shows that DynamicCache is able to outperforms the baseline with one replica and the baseline with 5 replica (upper bound of number of shard replica) in terms of per-shard read and write latency, while achieving comparable throughput compared to the baselines.

Acknowledgments

We would like to thank Professor Richard Yang, Xiao Shi, and Scott Pruett for teaching CS 426 Distributed Systems at Yale and providing us with help, feedback and inspiration and for designing the original sharded key-value cache assignment (Lab 4) on which this project expands.

References

[1] Juncheng Yang, Yao Yue, and KV Rashmi. A large scale analysis of hundreds of in-memory caching clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 191–208, 2020.

[2] Sailesh Mukil, Prudhviraj Karumanchi, Tharanga Gamaethige, and Shashi Madappa. Cache warming: Leveraging ebs for moving petabytes of data. https://netflixtechblog.medium.com/cache-warming-leveraging-ebs-for-moving-petabytes-of-data-adcf7a4a78c3.

[3] Yimeng Liu, Yizhi Wang, and Yi Jin. Research on the improvement of mongodb auto-sharding in cloud environment. In *2012 7th international conference on Computer science & education (ICCSE)*, pages 851–854. IEEE, 2012.

[4] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, et al. Twine: A unified cluster management system for shared infrastructure. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 787–803, 2020.

[5] Kiryong; Boeve, Daniel; Ha and Anca Agape. Throughput autoscaling: Dynamic sizing for facebook.com. Engineering at Meta, Sept. 14 2020 [Online].

[6] Benjamin Berg, Daniel S Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, et al. The {CacheLib} caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 753–768, 2020.

[7] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. Tao: Facebook's 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.

[8] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[9] pelikan. https://github.com/twitter/pelikan.

[10] Nicolae Vartolomei. A model for choosing the number of shards and replicas for optimal efficiency. https://nvartolomei.com/a-model-for-choosing-the-number-of-shards-and-replicas-for-optimal-efficiency, August. 20 2020.

[11] Sangmin Lee, Zhenhua Guo, Omer Sunercan, Jun Ying, Thawan Kooburat, Suryadeep Biswal, Jun Chen, Kun Huang, Yatpang Cheung, Yiding Zhou, et al. Shard manager: A generic shard management framework for geo-distributed applications. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 553–569, 2021.

[12] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. Blowfish: Dynamic storage-performance tradeoff in data stores. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 485–500, 2016.

[13] Praveen M Dhulavvagol, Vijayakumar H Bhajantri, and SG Totad. Performance analysis of distributed processing system using shard selection techniques on elasticsearch. *Procedia Computer Science*, 167:1626–1635, 2020.

[14] Mon Shih Chuang and Anagha Kulkarni. Improving shard selection for selective search. In *Asia Information Retrieval Symposium*, pages 29–41. Springer, 2017.