Hot Key Migration in Memcached

Introduction

Memcached is an in-memory cache value system used to store data objects in RAM and service client requests by cache lookup instead of a database read. This can reduce the read request time from 20,000 µs to 250 µs [1]. However this scheme also suffers from cache misses whenever a new server is added or removed from the cluster.

Motivation

The primary goal of this project is to investigate and implement techniques to reduce cache miss rate whenever the distributed cache system is scaled up or down.
When a memcached server's cache is filled up with keys, it has to evict old keys to store new keys. The eviction policy can result in thrashing. This can be alleviated by adding another memcached server to the configuration cluster. However a cold server adds a spike in cache misses because the keys remapped to the new server are not present in it's cache. The keys are read from the database and then added to the new server. Our project investigates this behavior by establishing baseline results with 2 experiments and then improves upon the performance of adding a new cold server to the cluster by moving the hot keys preemptively [2].

Setup and Experiment Workload

The memcached server setup includes 3 instances of the memcached server all running on the same machine but using different port numbers. The client application written in Golang uses an open source client API library to access memcached servers. All memcached servers are allocated 64 MB of memory for key/value data store. The setup is shown in *Figure 1*.
We chose the Zipfian distribution since it better models the real world distribution of how frequently the data is accessed. Not all data is accessed at the same rate. This results in some keys accessed more frequently than the others. Zipfian distribution is used to model hot and cold keys. The setup generates 229 MB of data. Each experiment involves sending 475,000 requests for keys, of which 68,000 are unique keys. We simulated the database reads by adding a latency of 8 ms for all database read requests and 0.3 ms for a memcached server read.

Consistent Hashing

The client library was updated with consistent hashing to avoid remapping nearly all keys when a cold server is added to the configuration. We thought it was important to swap out standard hashing with consistent hashing to avoid the cache misses from standard hashing. This consistent hashing guarantees that that the keys are on existing servers are not shuffled around when a new server is added.8

Client Prototype

We used open source libraries and modified them for our experiments. The backend memcached servers is written in C [3]. The frontend client library is in Golang [4]. The client library was updated to replace standard hashing with consistent hashing [5].

We wrote all of our experiments to utilize the frontend client library in Golang. The Golang client library would make requests against the memcached servers to simulate a production environment. We used this structure when creating our following four experiments.

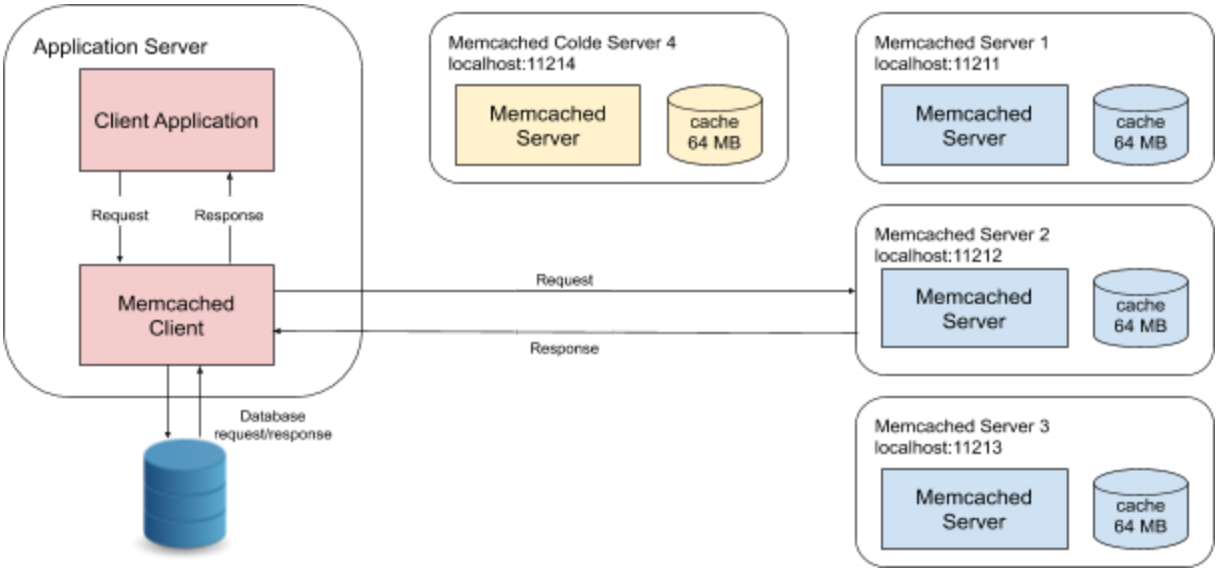

Fig 1 - Memcached client and server setup.

Experiments

1. Baseline experiment

The baseline experiment involves 3 memcached servers servicing 475K client requests for 68K unique keys. With a capacity of 64 MB per server, the workload of 229 MB exceeds the total capacity of the three servers. Figure 2 shows the mean response time of the baseline experiment.

2. Addition of cold server to the configuration

This setup adds a cold server to the memcached server configuration halfway through the experiment. This results in a spike of cache misses and the effect can be seen in figure 3. When the cold server is added, all cache misses cause a database read of the key/value pair.

3. Improvement I : Copy over cache misses for the cold server

We intended for this experiment to be a performance optimization to reduce the cache miss rate. In this experiment, we used the old configuration of the servers to fetch the key/value pair from the memcached servers instead of the database read. Whenever a request was issued, we would first check to see if the value could be obtained from the original key “owner” memcache server before adding it to the recently new server. If not, we would have to go to the database to fetch the value. Figure 4 shows the results of this experiment.

4. Improvement II: Copy over hot keys preemptively to the cold server

This experiment involves copying over the 20% hottest keys from each server preemptively, destined to be remapped to the new cold server by consistent hashing. When the new server was added, the keys that had been remapped according to the consistent hashing scheme would be identified. These keys would then be set on the new server in order of their relative “hotness” (i.e. the hottest keys would be set, then the next hottest, etc.). Only the keys that made up 20% of the heat would be set to the new server. According to the Zipfian distribution, this should make up approximately 98% of the accesses.

Results

The results shown from the graph in Figure 2 show the baseline for the experiment, without scaling any servers up. Each dot in these graphs show a mean response time over a period of 1000 requests. As discussed, the workload is intentionally too large to be held in just three memcached servers alone. We use this to show the effects of introducing a cold server into the mix with Figure 3. Compared to Figure 2, it is easy to pinpoint the exact time at which the new server is introduced, as there is a large spike in the mean response time. This occurrence is due to the fact that 1/4th of the key space will now result in a cache miss the next time that they are asked for, even if they were once cached on another server. This emphasizes the key issue that we attempt to alleviate in experiments 3 and 4.

Figure 4 shows the results of attempting to use the old mapping configuration in order to mitigate some of these cold cache misses in experiment 3. However, when comparing the two graphs it does not appear that this noticeably improves the spike in performance. Our working theory is that, since the keys no longer map to the old servers, the less frequent keys are evicted fairly quickly and are no longer cached.

Figure 5 shows the results of experiment 4, where we move 20% of the remapped hot keys to the new server preemptively. As can be seen on the graph, this experiment was much more fruitful. Rather than having a large spike halfway through the graph, there is only a slight bump. This leads us to believe that hot key migration is indeed a potential avenue for improving the miss rate caused by a cold start.

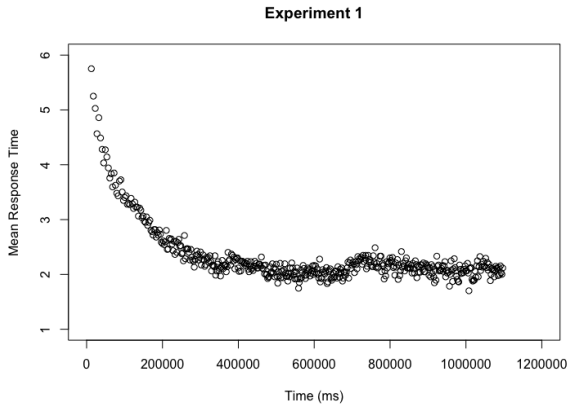

Figure 2

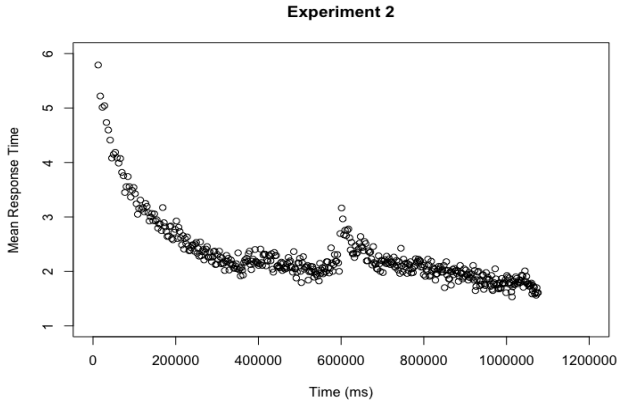

Figure 3

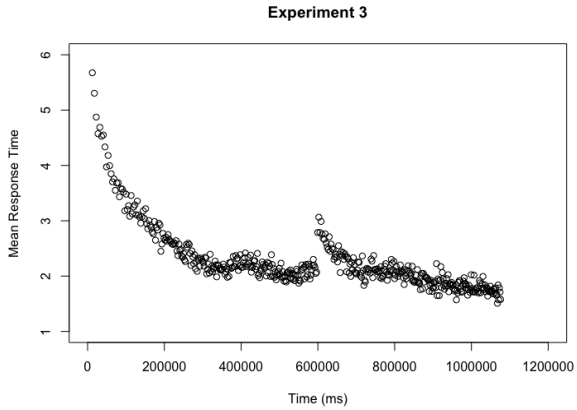

Figure 4

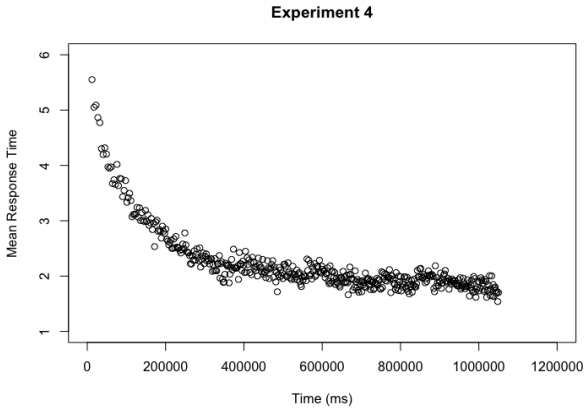

Figure 5

References

1. "Latency Numbers Every Programmer Should Know", https://gist.github.com/jboner/2841832
2. "Saving Cash by Using Less Cache", Timothy Zhu, Anshul Gandhi, Mor Harchol-Balter, Michael A. Kozuch
3. Memcached server library, https://github.com/memcached/memcached
4. Memcache client in Go, https://github.com/rainycape/memcache
5. Consistent hashing, https://github.com/stathat/consistent