

Assignment 4: In-Memory Cache

Lee Kai Yang
(23205838 - 100%)
kai.y.lee@ucdconnect.ie

April 12, 2024

[Video Link](#) [GitHub Link](#)

Abstract - This project explores the implementation and impact of an in-memory cache within web service architecture. With the exponential growth of web traffic, the need for high-performance servers becomes paramount. Utilizing cache mechanisms, such as the memcached employed by Facebook, can significantly enhance server efficiency by storing frequently accessed data temporarily in memory. The project investigates the design decisions behind implementing an in-memory cache, including the structure of key classes like Cache, TTLMap, and LinearProbeHashMap. Through benchmarking and examples like Cache Viewer and Benchmark programs, the project demonstrates the effectiveness of caching in improving web service response times. While acknowledging the simplified nature of benchmarks compared to real-world scenarios, the project offers valuable insights into the inner workings of hashmaps and the potential performance gains of caching in production environments.

1 Background

Web traffic nowadays has been drastically increased since the first introduction of the world wide web [1], this also result in the need of high performance web servers that can serve multiple clients at the same time. One crucial technique to achieving such applications is the use of cache [2] wherein similar data that was queried recently can be saved temporarily in memory and subsequent requests can be served quickly by taking the saved query, by doing so during peak times where there might be hundreds if not thousands of requests hitting the server, the server does not need to repeat the same computation or querying the database for similar data for each request. One of the many successful examples of employing caches in production web services is Facebook where they heavily utilised `memcached`, a well-known, simple, in-memory caching solution to scale their backend services during their early days tackling traffic that grew exponentially [3].

2 Introduction

2.1 In-memory cache

Due to the versatility of hashmaps, usually caches are just hashmaps being used as key-value stores. The example below depicts how a cache is being used in the context of a web service. To start off, we take a look at the architecture of the web services in Figure 1.

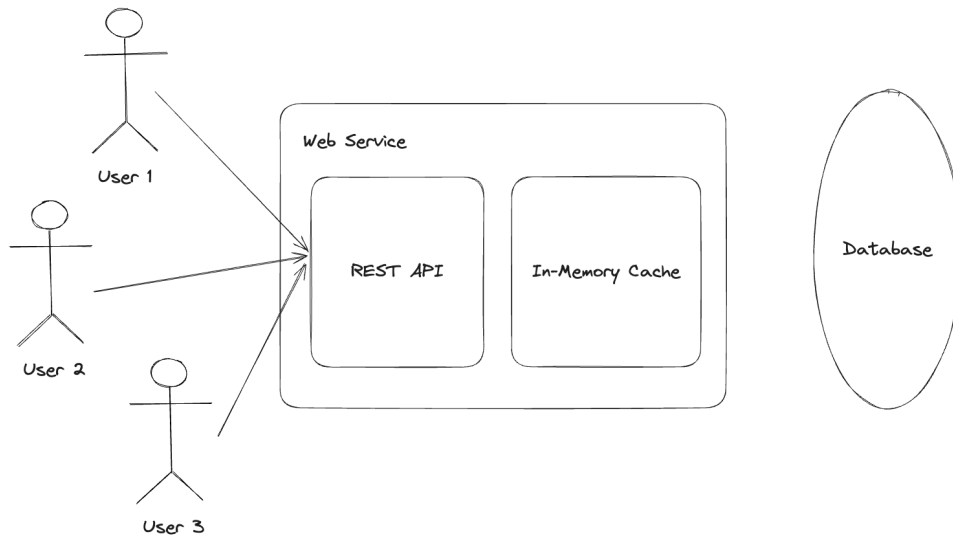


Figure 1: Web service architecture

As can be seen, the in-memory cache is a component within the web service itself that serves the purpose of a middle-man between the database layer and the application layer.

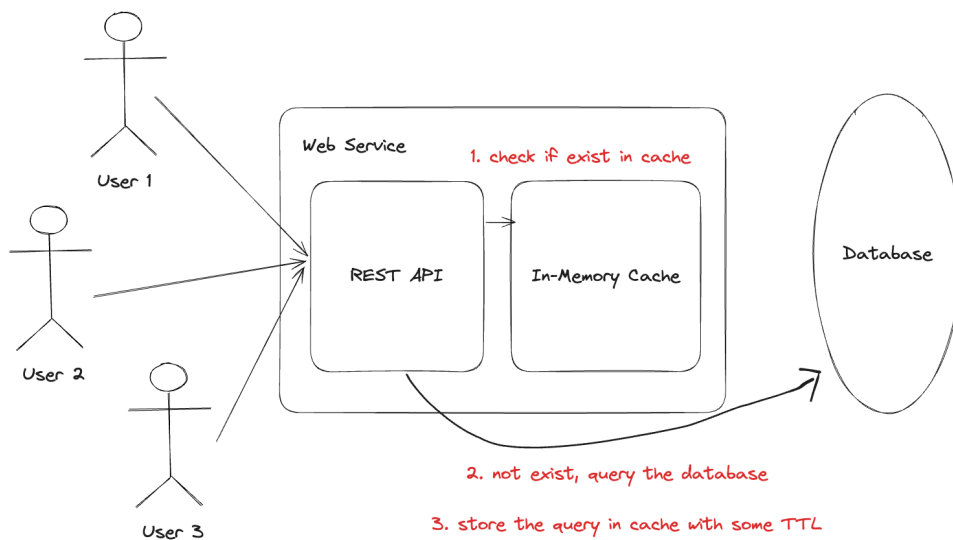


Figure 2: Web service flow

Figure 2 shows the typical flow when the server receive requests from clients. It first queries the cache and see if there is existing data being cached, if yes then it takes that and serve the client directly with that data otherwise it queries the database and store the results into the cache with a certain time-to-live (TTL). Specifying TTL is important here because doing so helps the cache to stay small, avoiding it to takes up a lot of unnecessary system memory when not in use. When the TTL has elapsed, the data will be evicted (removed) automatically.

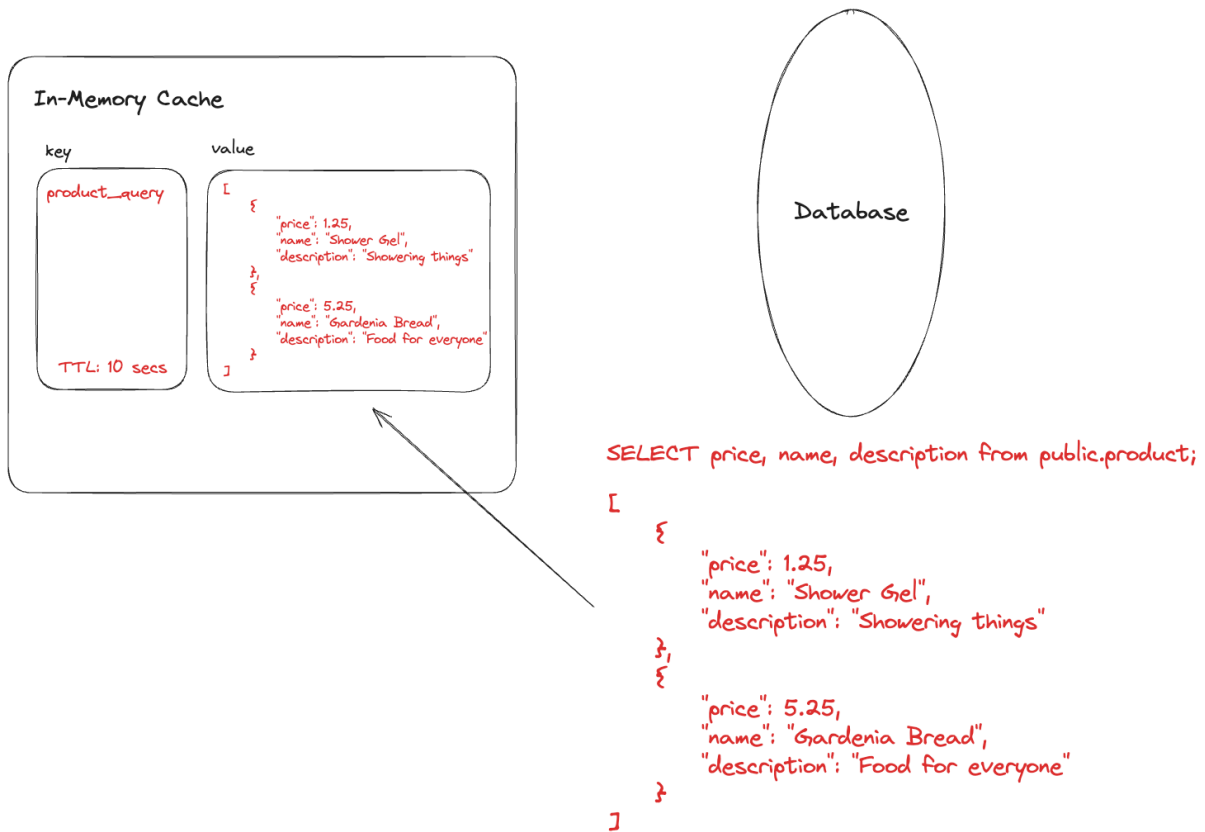


Figure 3: Caching SQL queries

In a typical web service caching scenario as shown in Figure 3, it is common for a service to cache popular queries and since the cache serves as a key-value store [4], these queries can be temporarily cached as long as the key given is unique for each query.

3 Design decisions

Since the cache is basically just a key-value store with added features such as TTL hence it is important to have an efficient hashmap data structure. In the section below, designs and approaches taken for each class will be discussed.

3.1 Classes overview

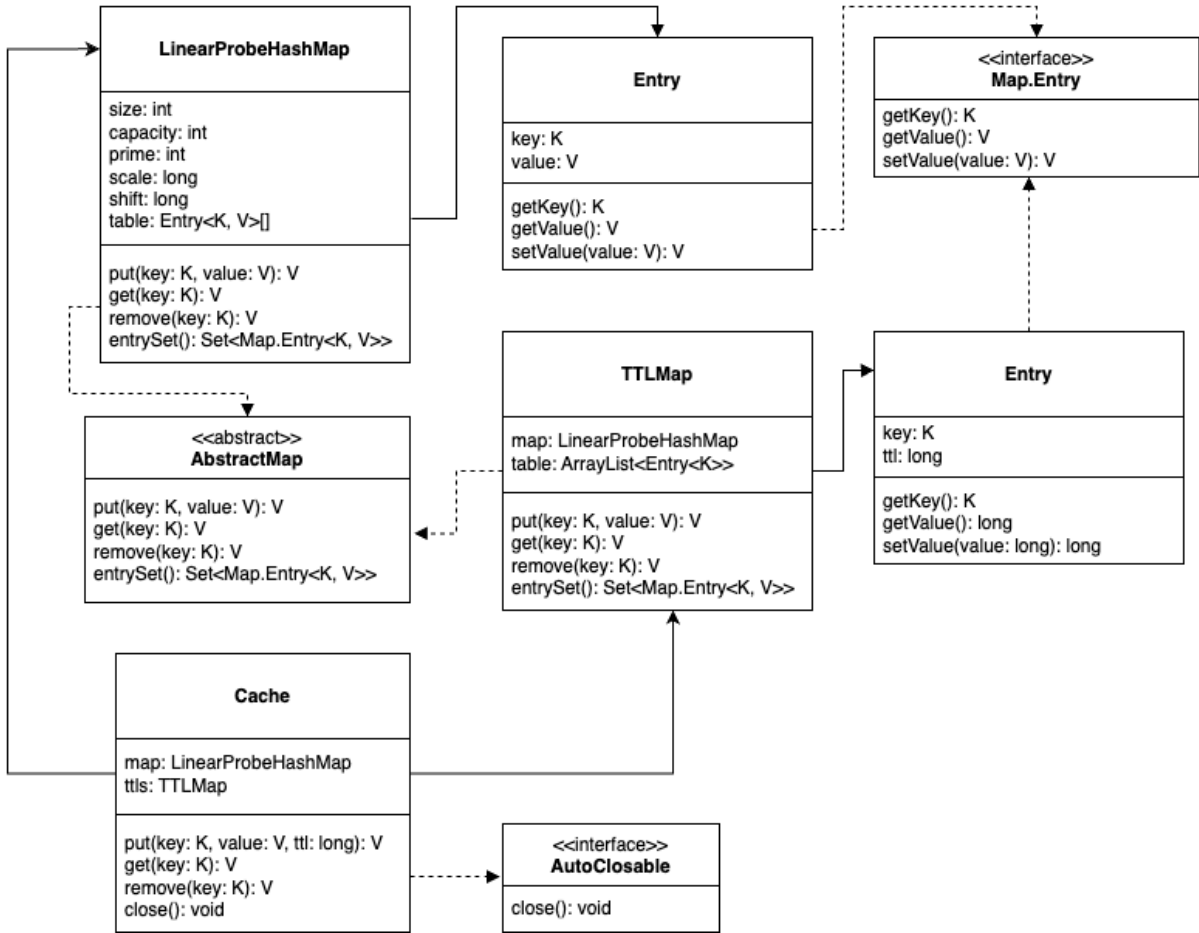


Figure 4: UML Diagram

As can be seen from Figure 4, there are three main classes:

- **Cache** - In-memory cache that supports cache eviction based on time-to-live (TTL).
- **TTLMap** - A type of map that helps stores the time-to-live (TTL) for an associated key.
- **LinearProbeHashMap** - HashMap that uses linear probing to resolve hash conflicts.

3.2 Cache

This class can be treated as a wrapper of both **LinearProbeHashMap** and **TTLMap** where from the API we can see that the main difference is that it allows an additional `ttl` to be specified during `put`. What this class does is that during an insert operation, it inserts the data into the underlying **LinearProbeHashMap** at the same time create a record on the **TTLMap** and within the class itself it runs a task repeating every one second to check if the time to live has been exceeded, if so then it evicts the item from the data structure.

3.3 TTLMap

This class is almost identical to any other hashmaps except that it can only take `long` as the value type and that the underlying hash table is arranged in a sorted manner according to the value. To achieve this, we run a binary search to find the suitable index to house the new item every time the `put` operation is invoked.

3.4 LinearProbeHashMap

This class provides a generic HashMap implementation that is suited for most use cases and since it uses linear probing as a hash collision resolution, it is more memory efficient and have better cache performance over separate chaining resolution. This is mainly due to the fact that linear probing uses arrays instead of linked list but this also makes the hashmap performs better with low load factor since the lesser the load factor, the lesser the probe time is required. For your information, load factor is the ratio of the current size of the map against its current capacity and probe time is the time taken to find an element in the table circularly.

Due to these properties of the hashmap, we add a check after every `put` operation to make sure the load factor is $\leq 50\%$ and if it exceeds this threshold then we resize the map so it has twice the capacity hence giving it more space until load factor nears to 50% again.

On another note, this hashmap uses the default `Object.hashCode()` from Java as the hash function and it uses a combination of the Multiply-Add-and-Divide (MAD) and division method for compression. The reason for using them in combination is due to the fact that MAD requires its N to be a prime number but the hashmap's capacity is not necessarily a prime number hence we use an internally defined prime number instead of the capacity. Therefore, to ensure the hash it gives is in the range of our capacity, we further compress it down with the division method.

4 Test cases

The test cases for the aforementioned classes can be found in `src/test/java/cache` and can be ran by executing `./gradlew test` in the terminal.

5 Examples

5.1 Cache Viewer

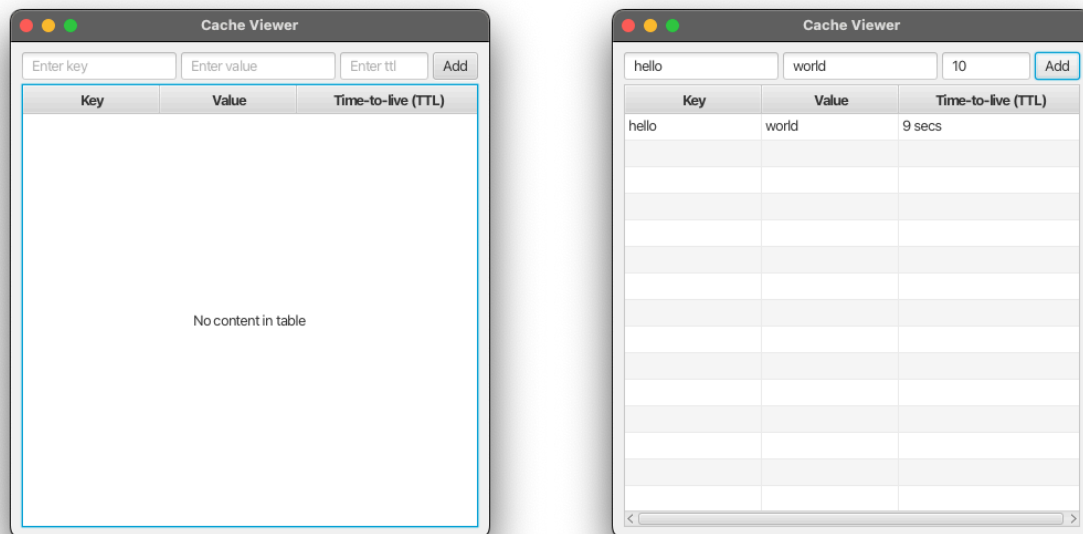
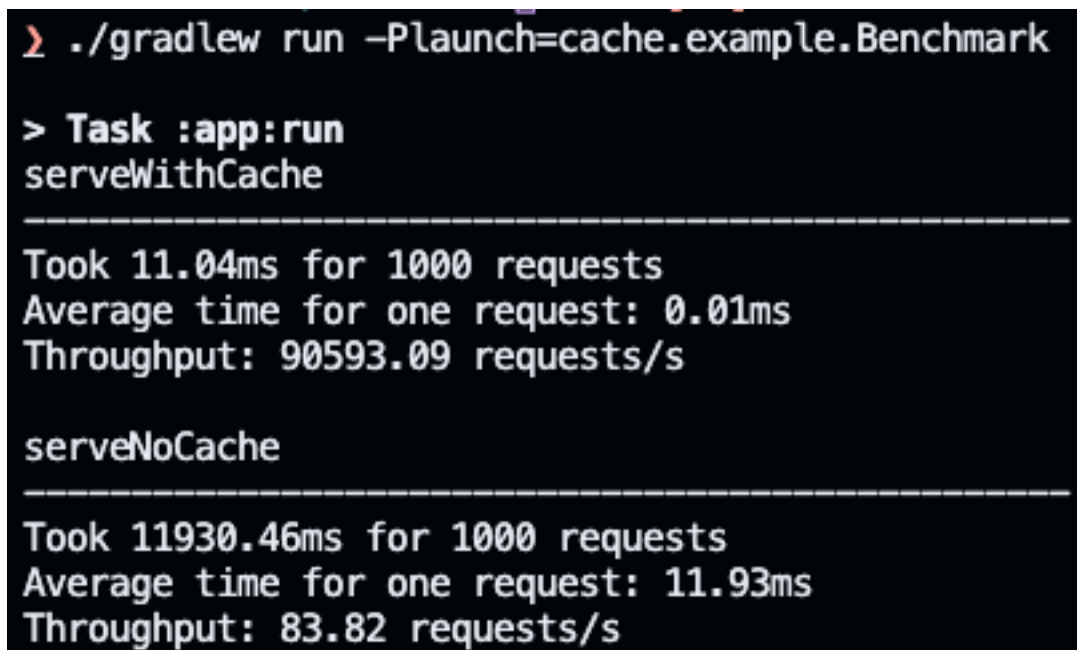


Figure 5: Cache viewer

To visualise the cache I built a simple GUI as shown in Figure 5 that allow users to insert a new record onto the cache with specified TTL and it also displays the state of the cache in realtime where users are able to see when records are added and when records are evicted. To run the visualizer, run `./gradlew run -Plaunch=cache.example.App` in the terminal.

5.2 Benchmark

The motivation of building such in-memory cache is to speed up the web service and I devised a mock environment to test the impact of having a web service with and without cache. In the benchmark program, it run two scenarios and measure the time taken for each and print the results to the console as shown in Figure 6. The benchmark can be ran by the command `./gradlew run -Plaunch=cache.example.Benchmark`.



```
> ./gradlew run -Plaunch=cache.example.Benchmark

> Task :app:run
serveWithCache
-----
Took 11.04ms for 1000 requests
Average time for one request: 0.01ms
Throughput: 90593.09 requests/s

serveNoCache
-----
Took 11930.46ms for 1000 requests
Average time for one request: 11.93ms
Throughput: 83.82 requests/s
```

Figure 6: Benchmark result

As can be seen that when using a cache, it significantly reduces the time to service one request, the speed up is about **1080**. This is because when not using a cache, every requests has to query the database and in this case we do a `Thread.sleep(10)` to simulate the time needed to query.

6 Conclusion

In conclusion, I am satisfied with what this project had achieved and I learned how `HashMap` works under the hood, a data structure that I use almost every day from the standard library but never dig into it to see what it is doing. As a side note, the benchmark that was conducted was not necessarily accurate because there are a lot more moving parts in an actual production system that cannot be simulated easily however it does provide a solid baseline for expectation in performance gains.

Bibliography

- [1] Ben Newton, Kevin Jeffay, and Jay Aikat, "The Continued Evolution of Web Traffic." Aug. 01, 2013.

- [2] Jian Yin, Lorenzo Alvisi, Mike Dahlin, and Arun Iyengar, “Engineering Web Cache Consistency.” pp. 224–259, Aug. 01, 2002. doi: 10.1145/572326.572329.
- [3] Rajesh Nishtala *et al.*, “Scaling Memcache at Facebook.” Apr. 03, 2013.
- [4] Kefei Wang, Jian Liu, and Feng Chen, “Put an elephant into a fridge: optimizing cache efficiency for in-memory key-value stores.” pp. 1540–1554, May 01, 2020. doi: 10.14778/3397230.3397247.