Milestone 2

Literature Review

Qian Xuesen Group: Zifan Qu, Zhichao Lin, Zecheng Ding, Jie Hou

1 Introduction

The rapid growth of the world wide web also leads to the increasing demand for network bandwidth. Thus, Proxy caching is proposed to reduce network traffic [Kim1999]. A cache is a component that stores data so future requests for that data can be served faster. This provides high throughput and low-latency access to commonly used application data, by storing the data in memory [OutSystems2020]. It caches popular Internet objects. In the computer system, cache is everywhere. For example, when we visit a web page, web page and referenced static files, they will be cached locally in the browser or CDN according to different strategies Server, when you visit the server for the second time, you will feel that the loading speed of the web page is much faster. For example, the number of likes in social media can not be found from the database every time everyone visits, and then make statistics. The database operation is very time-consuming, and it is difficult to support such a large amount of traffic. Therefore, generally, the like like data is cached in redis In the service cluster. In addition, the cache in the proxy server can improve the response speed and reduce the load of the web server. However, if the object in the cache is different from the original object due to updates on the source server, the value of the cache is reduced.
In this project, we will implement the simplest cache which based on the key value pairs stored in memory. And we will focus on 3 parts: how to increase the hit rate of cache, consistent hashing and cache robustness.

2 Challenges

2.1 Increasing the Hit Rate of Cache

When computers receive a new request which consumes a great amount of time, they can store the result of this request in the cache. Then, if they receive the same request after that, they can return the result stored in the cache directly. It is obvious that the memory capacity of a computer is limited. However, the memory capacity of a computer is limited. If a large project is running in the computer, the system can not store all the results of operations in the cache. Although the design of distributed cache has improves the memory capacity of a computer, we still need to increase the hit rate of cache because the performance and the efficiency of the cache should be guaranteed. Moreover, the cache in a distributed system has different performance from the cache in a single computer [Ji2018]. Increasing the hit rate of cache has a profound effect on the performance of this project.

2.2 Consistent Hashing

The next challenge is to implement consistent hashing.The consistent hashing algorithm is an important link from a single node to a distributed node [Karger1997].
We need a mechanism to select optimal available node to response the requests [Neelakantam2018]. Hashing algorithm could avoid storing same data on multiple nodes, which could waste lots of storage. However, we should consider the size up and down of cache scale, the unexpected failures of specific cache nodes [Neelakantam2018]. A naive hashing algorithm could not handle the hugh amount ofcache operations in a changeable scales [Karger1997].
What's more, we need to consider the load balancing. If too much data is stored in several particular cache nodes, they the overhead of these nodes will be too high and the whole system will be unstable.

2.3 Cache Robustness

In the context of massive data, in order to improve program robustness, the design of distributed cache must consider three possible problems: cache avalanche, cache breakdown, and cache penetration [Ayoub2017].

- Cache avalanche

  All caches fail at the same time, causing a large amount of instantaneous DB requests and a sudden increase in pressure, causing an avalanche. Cache avalanches are usually caused by cache server downtime and cached keys with the same expiration time.

- Cache breakdown

  When an existing key expires, there are a large number of requests at the same time, and these requests will break down to the DB, causing a large amount of instantaneous DB requests and a sudden increase in pressure.

- Cache penetration

  Query a non-existent data, because it does not exist, it will not be written to the cache, so it will request the DB every time. If the instantaneous traffic is too large, it will penetrate the DB and cause downtime.

3 Approaches

3.1 The Implementation of LRU

In order to solve the problem of the capacity limitation of cache, three strategies has been proposed to update the contents in the cache: FIFO, LFU and LRU. FIFO means that "First In First Out", which aims at replacing the oldest record by the new one in the cache. However, during the operations in the computer system, the oldest record may be used frequently, so FIFO is not a good option to increase the hit rate of cache. LFU means that "Least Frequently Used", which aims at replacing the least frequently used record by the new one in the cache. This strategy is an efficient option but it also has remarkable weakness: The memory cost of recording the time of cache blocks may be very high; and if a block has been accessed frequently a long time ago but has not been accessed presently, then it can not be updated. Therefore, LRU (Least Recently Used) is the balanced strategy to increase the hit rate of cache. It will replace the least recently used memory by the new one in the cache. Its memory cost is acceptable and the increasing of hit rate is also remarkable. In addition, for the distributred cache, the LRU algorithm could also be optimized to obtain a better performance [Du2016].

3.2 The Design of Hashing Ring

To solve the problem of cache unavilible caused by the change of nodes, we need to implement consistent hashing. Based on consistent hashing algorithm, after adding or subtracting slots, a hash table with K keywords and n slots (nodes in a distributed system) only need to remap K/n keywords on average [Karger1997].
The steps of consistent hashing are as follows: The consistent hash algorithm maps the key to the 2^{32} space, and connects this number end to end to form a ring.

1. Calculate the hash value of the node/machine (usually using the name, number and IP address of the node) and place it on the ring.

2. Calculate the hash value of the key and place it on the ring. The first node found clockwise is the node/machine that should be selected.

In other words, the consistent hash algorithm only needs to relocate a small part of the data near the node when adding/deleting a node, instead of relocating all nodes, which solves the above problem.
In addition, we need to avoid uneven load. If the server has too few nodes, it is easy to cause key tilt. In order to solve this problem, the concept of virtual nodes is introduced. One real node corresponds to multiple virtual nodes. Assuming that one real node corresponds to three virtual nodes, then the virtual nodes corresponding to peer1 are peer1-1, peer1-2, and peer1-3 (usually implemented by adding numbers), and the other nodes also operate in the same way.

1. Calculate the hash value of the virtual node and place it on the ring.

2. Calculate the hash value of the key, and find the virtual node that should be selected clockwise on the ring, for example, peer2-1, then it corresponds to the real node peer2.

The virtual node expands the number of nodes and solves the problem that data is easy to tilt when there are fewer nodes. And the cost is very small, only need to add a dictionary (map) to maintain the mapping relationship between real nodes and virtual nodes.

3.3 Mutex and Bloom Filter

Here are two ways to improve the cache robustness:

1. Mutex

   When multiple concurrent requests obtain source data for an invalid key, only one of them is executed, and the remaining blocked waiting until the executed request is completed, and the result is passed to other blocked requests to prevent breakdown.

2. Bloom filter

   The data structure of the Bloom filter is a set of hash functions and a bitmap. In order to add data items to the Bloom filter, we will provide K different hash functions, generate multiple index values and set the value of the corresponding bit in the bitmap to "1". For subsequent data, you only need to call this group of hash functions here to determine whether the corresponding positions are all 1s, and then you can determine whether the data may already exist or not.

   Through the Bloom filter, it can be judged in advance whether the key value passed in is in the overall database, which can avoid the cache penetration caused by a large number of malicious and false Key values [Haswell2015].

4 Project Proposal

Design a distributed cache system, considering issues including cache update strategy, concurrency mechanism, distributed node communication, cache robustness.

5 References

1 Du, Shumeng, et al. "The Optimization of LRU algorithm based on pre-selection and cache prefetching of files in hybrid cloud." *2016 17th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*. IEEE, 2016.

2 Haswell, Jonathan M. "Cache system optimized for cache miss detection." U.S. Patent No. 8,938,603. 20 Jan. 2015.

3 Ji, Kaiyi, Guocong Quan, and Jian Tan. "Asymptotic miss ratio of LRU caching with consistent hashing." *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018.

4 Karger, David, et al. "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web." *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. 1997.

5 Kim, Joo-Yong, Kyoung-Woon Cho, and Kern Koh. "A proxy server structure and its cache consistency mechanism at the network bottleneck." *Proceedings. Twenty-Third Annual International Computer Software and Applications Conference (Cat. No. 99CB37032)*. IEEE, 1999.

6 Mina Ayoub 2020. 3 Major Problems And Solutions In The Cache World. [online] Available at: <https://medium.com/@mena.meseha/3-major-problems-and-solutions-in-the-cache-world-155ecae41d4f> [Accessed 29 October 2020].

7 Neelakantam, Srushtika. "How We Implemented Efficient Consistent Hashing." *Ably Blog: Data in Motion*, Srushtika Neelakantam, 19 June 2018, www.ably.io/blog/implementing-efficient-consistent-hashing/.

8 OutSystems. 2020. Improving Performance With Distributed Caching. [online] Available at: <https://success.outsystems.com/Documentation/Best_Practices/Performance/Improving_Performance_With_Distributed_Caching> [Accessed 29 October 2020].