Milestone 4
Final Report

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.
The cache has been widely used in the field of distributed computing with the expansion of local computer systems to distributed systems, and is called distributed cache. Distributed cache can handle a large amount of dynamic data, so it is more suitable for scenarios that require usergenerated content such as social networking sites in the Web 2.0 era. After expanding from local cache to distributed cache, the focus of attention has been extended from the difference in data transmission speed between CPU, memory, and cache to the difference in data transmission speed between business systems, databases, and distributed caches.

2 Related Works

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.

2.1 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 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.

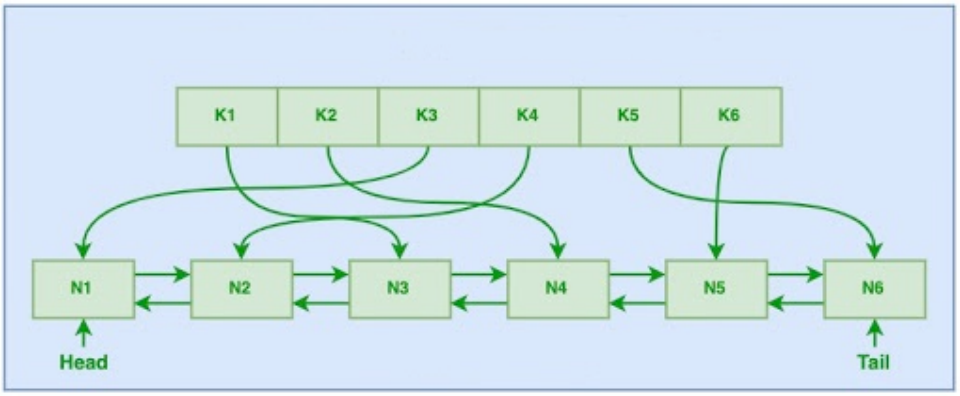
- Cache penetration

  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 Project Design

3.1 LRU Replacement Strategy

The LRU replacement strategy is shown as following:



There are two data structures in the LRU.

1. The rectangles string on the above is a “Map” which stores the mapping relation between the keys and the values. With the storage advantages of map, we can search a specific value as keys and add a record into the map with little time cost.

2. The list below is a “Double Linked List” storing all the values in it. The crucial function of LRU replacement strategy can be achieved with the double linked list. If a specific value has been searched, then we move this block into the end of the list. Therefore, if a new record is entered, then the least recently used block can be removed from the cache, which guarantees the crucial functions of LRU. Moreover, the storage style of double linked list can also save a great amount of time cost if we want to delete or add a new record in this data structure.

The Go programming language has provided the convenient implementation of map and double linked list, so the main object in this part is to achieve the functions in the picture above with Go programming language.

3.2 Single-machine Concurrent Cache

About the concurrent cache in the single machine, we should address several problems as following:

1. Mutex

   In Go programming language, if several goroutines write or read a variable at the same time, the conflict will happen. We should guarantee that there is always one goroutine can access this variable. We use the Mutex to solve this problem. The Go programming has a convenient model about Mutex, and we should achieve this function with it.

2. Concurrent Write and Read

   In order to achieve the concurrency of writing and reading in the single machine, we are supposed to use Mutex to encapsulate some functions of LRU.

3. Intersection

   The distributed cache aims at constructing the reliable intersection between the devices and the clients. After receiving the requests of clients, we should check whether the object has been stored in the cache or not; if there is no avaliable values in the cache, we are suppoed to determine where to search the objective values and return it. In this project, we will construct a structure to implement these functions with Go programming.

3.3 Consistent Hashing

The classic hashing approach used a hash function to generate a pseudo-random number, which is then divided by the size of the memory space to transform the random identifier into a position within the available space. However, the problem in a distributed system with simple rehashing—where the placement of every key moves—is that there is state stored on each node; a small change in the cluster size for example, could result in a huge amount of work to reshuffle all the data around the cluster. As the cluster size grows, this becomes unsustainable as the amount of work required for each hash change grows linearly with cluster size. This is where the concept of consistent hashing comes in.
Consistent Hashing can be described as follows:

1. It represents the resource requestors (which we shall refer to as “requests” from now on, for the purpose of this blog post) and the server nodes in some kind of a virtual ring structure, known as a “hashring.”

2. The number of locations is no longer fixed, but the ring is considered to have an infinite number of points and the server nodes can be placed at random locations on this ring. Of course, choosing this random number again can be done using a hash function but the second step of dividing it with the number of available locations is skipped as it is no longer a finite number.

3. The requests, ie the users, computers or serverless programs, which are analogous to keys in classic hashing approach, are also placed on the same ring using the same hash function.

We need to implement the following to make it work:

1. A mapping from our hash space to nodes in the cluster allowing us to find the nodes responsible for a given request.

2. A collection those requests to the cluster that resolve to a given node. Moving forward, this will allow us to find out which hashes are affected by the addition or removal of a particular node.

3.4 HTTP Server and Distributed Node

Our system implements management and control of client nodes that store data through a server. The overall system flow is as follows. After the distributed cache system receives the key, it will check whether the key has been cached in the system. If it has been replaced by a walk-through, the cached value will be returned. If not, it should be obtained according to the remote node to determine whether the remote node is There is the cache value. If the remote node cannot obtain the value, the callback function is called and the current value is cached in the system.
Therefore, based on the above system operation process, it can be known that each node in the system needs to realize communication between nodes, obtain node information, and it is a relatively common and simple way to establish a communication mechanism based on HTTP. If a node starts the HTTP service, then this node can be accessed by other nodes. Go language provides the http standard library, which makes it very convenient to build HTTP server and client.
We refine the process mentioned above, and we select nodes through consistent hashing. The key obtained by the hash function is used to judge whether the currently selected node is the target remote node. If it is, we will access

the remote node through the HTTP client, and the client will return the return value after successful operation. If the operation fails, it will be processed at the current node. According to this function, to design a distributed node, the node needs to access the server through HTTP request, and obtain data, and compare and judge the data obtained through HTTP request.

3.5 Prevent Cache Breakdown

For an existing key, when the cache expires, there are a large number of requests at the same time. These requests will penetrate the database, causing a sudden increase in the amount of database requests.This phenomenon is called cache breakdown, and it may occur in hot queries.
The above phenomenon is that multiple threads simultaneously query this data in the database, then we can use a mutex lock on the first request to query the data to lock it.At this point, other threads will wait if they can't get the lock, wait for the first thread to query the data, and then do the cache. Later threads come in and find that there is a cache, so they go directly to the cache.
In our project, we will use the singleflight library officially provided by the go language to solve this problem. singleFlight enables the cache to obtain source data for an invalid key when multiple concurrent requests are made, only one of them is executed, and the remaining blocks wait until the executed request is completed, and then the result is passed to other blocked requests to prevent breakdown effect.

4 Distributed Systems Challenges

1. Heterogeneity

   Distributed Systems are constructed based on different kinds of Internet networks, operating systems, computer hardwares and programming languages, so a common network communication protocol must be considered to shield the differences between heterogeneous systems. In our project, we use middlewares to solve this problem. Middleware is a technical component. It do not need Web framework itself to understand all the business and thus all the functional modules. Therefore, a socket that allows users to define their own functionality will be constructed, which is embedded in the framework as if this functionality were supported by the framework natively. Our project is also based on HTTP protocol to finish the communication between nodes, which is a common solution to the problem of heterogeneity in distributed systems.

2. Openness

   The openness of a computer system is a characteristic that determines whether the system can be extended and re-implemented in different ways. The openness of distributed systems depends largely on the extent to which new resource-sharing services can be added and used by multiple clients. Distributed systems are composed of different components written by different programmers. In order to integrate components into a system, the interfaces published by the components must conform to certain specifications and be understood by each other. The potentials of our project can guarantee the soultion of this problem because the implementation of our project is based on Go programming language platform, which can be extended to many other different programmers and applications.

3. Security

   To achieve the security of Distributed system, we should have encrypt all shared resources to provide appropriate protection. In the consistent hashing component, the selection of hashing function could kindly affect the security of our system.

   Generally speaking, the hash function considers two points: one is the collision rate and the other is performance. Such as CRC, MD5, SHA1. For occasions that require integrity verification, the collision rate is more critical, while performance is less important. Generally use 256-bit SHA1 algorithm, MD5 is no longer recommended. CRC stands for Cyclic Redundancy Check, which has simple coding and high performance, but its security is very poor. As a cache hash algorithm is still very suitable.

   However, if we want to defend attackers‘ eavesdropping, what we have done is not enough. We should use one of standard encryption method on both the request client and the cache node, to encrypt the query plains and the results to avoid being monitered by attackers.

   On the other hand, it is also important to prevent malicious people from exploiting loopholes to maliciously attack Redis services. Attackers can maliciously affect the normal operation of the Redis service, and eventually cause the database to be over-pressured and paralyzed, and the project server crashes, with disastrous

consequences.There are three main causes of unavailability: cache penetration, cache breakdown, and cache avalanche.

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

Cache breakdown: When a key exists, when the cache expires, there are a large number of requests at the same time. 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 every time the DB is requested, if the instantaneous traffic is too large, it penetrates to the DB, causing downtime.

In this project, we have approached the protection of Cache breakdown. Suppose that there are a large number of get(key) requests in a moment, and the key is not cached or cached on the current node. If no protective measures are taken, these requests will be sent to the remote node or read from the local database, which will cause the remote node or Pressure on local databases has soared. Using the prevention mechanism, when the first get(key) request comes, node will record that the current key is being processed, and subsequent requests only need to wait for the first request to be processed and get the return value.By using the method above, we could alleviate the pressure of the cache node and kindly protection our system from malicious attacking.

4. Failure Handling

   In our distributed cache sytem, one of the most common failure is the breakdonw of cache node. If one of our cache node is not accessible, then a large amount of data can not be quickly access, which will then increase the burden of other cache node, the data base and the network quality.

   If we use normal hashing function, then the breakdown of cache node is a really complex problem , since one node could influence all data stored in the system and we have to operate them all. However, the consistent hashing could kindly solve this problem.

   Assuming there are ten nodes, one of the nodes is removed, only 9 are left, then the previous hash(key)% 10 becomes hash(key)% 9, which means that almost all nodes corresponding to the cached value occur Changed. That is, almost all cached values are invalidated. When a node receives the corresponding request, it needs to go to the data source again to obtain the data, which is likely to cause a cache avalanche.

   That is to say, 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 this project, if the remote node is not accessible, what we do is to consider it as a remove operation logically, and assigned the data to other cache node on the hashing ring. To be honest, this is not enough for fault tolerance. In the distributed system, one of the major target of failure handling is to ensure consistency. When a node is down, the master needs to be re-elected to ensure consistency. However, our system is based on memcache, which does not require consistency. Therefore, we still need to learn and to discuss whether what we have done is enough.

   Another failure handling is to prevent cache breakdown. We have mentioned the cache breakdown in the security section. But what we have done is only to prevent. We did not consider the occasion that the system is attacked by cache breakdown. Actually, we have nothing to do but reboot if that happens. This is what we need to achieve in the future.

5. Concurrency

   Fundamental to distributed systems is the concurrency and collaboration among multiple processes. In many cases this means that the process needs to be accessed at the same time. In order to prevent such simultaneous access, it is necessary to provide a solution for mutual rebound visits through the process in order to access or mismatch damaged resources. How to solve the parallel problem is the key to the normal operation of a distributed system. In general, shared resources are locked to protect data security. For a cache system, multiple users or nodes can't change the data at the same time, including adding, deleting and modifying. Therefore, in our system, mutex is used to restrict data modification. However, for the operation of reading data, if the mutex is also used, the efficiency of the system will be reduced. Therefore, the system should support multiple users to read data at the same time. In golang, we can add read-write locks to the data so that multiple users can read the data without modifying the data restrictions.

6. Quality of Service

   Providing high-quality services is a principle that every system needs to guarantee, including distributed systems of course. When the cache system provides services, the main problems and challenges faced include the following aspects: how to ensure data consistency, cache breakdown, cache avalanche, and cache penetration. Cache avalanche refers to the failure of all caches at the same time, causing a large amount of transient database 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 refers to an existing key. At the moment the cache expires, there are a large number of requests at the same time. These requests will break into the database, causing a large amount of transient database requests and a sudden increase in pressure. Cache penetration refers to querying a non-existent data, because it does not exist, it will not be written to the cache, so it will request the database every time. If the instantaneous traffic is too large, it will penetrate the database and cause downtime. In order to ensure data consistency, our distributed cache system uses a consistent hash algorithm. This ensures that every time a node is added or modified, the corresponding node hash value of each data can be guaranteed to remain unchanged. While ensuring data consistency, the number of caches will not cause cache avalanches due to node modification. For cache breakdown and cache penetration, our system is mainly achieved by limiting the number of requests that reach the database through the cache. Limit the number of requests that penetrate the cache to 1, which prevents a large number of data requests from reaching the database after a cache problem.

7. Scalability

   Our project uses a consistent hashing algorithm. The principle is not only to hash the key of the data, but also to hash the node. For example, use the ip value of the node to hash, and then see that the hash value of the key falls on the hash of the node. The range of values determines which node the key is on. In this way, after adding a new node, the impact on the key hit is only the old data between the newly added node and its neighboring clockwise node. Therefore, our distributed cache system can easily increase nodes. So, our distribution has Size Scalability.

   The http mechanism is used for communication between different nodes. This breaks the geographical limitation of nodes, so our distributed cache system has Geographical Scalability.

8. Transparency

   First, the cache itself is a transparent concept. The purpose of the cache is to speed up the query speed so that the data that is frequently used or that has just appeared can be quickly found. The user will not know the existence of the cache.

   Our project uses a consistent hashing algorithm, supports multi-node concurrent caching, and can easily add or delete nodes. In this way, no matter how many nodes there are, the distributed cache system looks like a single node system. This achieves Location Transparency.

   The communication method between our distributed caching systems is http, and different nodes use the same operating system and data storage by default, so in terms of Access Transparency, we still need to improve.

5 Project Outcomes and Reflection

5.1 Measurements

In order to better test the usability and capability of a system, it is also very important to select a good test standard. For our system, normal operation is the core. Therefore, when we test our system, we mainly consider the usability of the main functions. We mainly test the following functions: LRU elimination mechanism, system can execute normally under the condition of concurrency, consistency hash. In addition, our system can run normally in case of abnormal request, such as cache crash.

In the LRU elimination mechanism test, we mainly access the data according to different times to verify whether the system is saved and updated according to the last access time. In concurrent testing, we need to test the read and write operations separately. Read operations need to be able to support multiple hosts at the same time, while writing operations, data will be locked, and can not be concurrent operations. In the conformance hash test, we add and delete real nodes, and then check the corresponding virtual nodes.

In the exception request test, we made multiple requests to the API to see whether a large number of requests triggered the lock mechanism of the system, and limited the number of requests for data, so as to avoid the database crash caused by a large number of requests arriving at the database.

5.2 Potential Benefits and Drawbacks

Redis is one of the most widely used distributed caching systems. Redis is recognized by users because of its stability and fast response speed. So when considering the potential advantages and disadvantages of our system, we compare our caching system with Redis.We mainly compare the data storage mode, data structure, running environment and system function.
Our system stores all the data in the memory, it will hang up after power failure, and the data cannot exceed the memory size. Part of redis is stored on the hard disk, which can ensure data persistence and support data persistence
There are five data structures commonly used in redis, which are much more than our system in data support
At present, redis only supports Linux to go up, thus eliminating the support for other systems. In this way, it can be better used for the optimization of the system environment. In comparison, our system can run in all environments that support golang.
In addition, as a mature cache system, redis has more abundant functions, such as master-slave replication and sentinel mode. These functions can make redis run better and support different application scenarios.

6 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].