Back To Module Home

Basic Building Blocks for Modern System Design
0% completed

Distributed Cache

System Design: The Distributed Cache

Background of Distributed Cache

High-level Design of a Distributed Cache

Detailed Design of a Distributed Cache

Evaluation of a Distributed Cache's Design

Memcached versus Redis

Distributed Messaging Queue

Pub-sub

Rate Limiter

Blob Store

Distributed Search

Distributed Task Scheduler

Sharded Counters

Conclusion

Mark Module as Completed

Background of Distributed Cache

Learn the fundamentals for designing a distributed cache.

We'll cover the following

- Writing policies

- Eviction policies

- Cache invalidation

- Storage mechanism

  - Hash function

  - Linked list

  - Sharding in cache clusters

    - Dedicated cache servers

    - Co-located cache

- Cache client

- Conclusion

The main goal of this chapter is to design a distributed cache. To achieve this goal, we

should have substantial background knowledge, mainly on different reading and writing techniques. This lesson will help us build that background knowledge. Let's look at the structure of this lesson in the table below

Structure of This Lesson

Section	Motivation
Writing policies	Data is written to cache and databases. The order in which data writing happens has performance implications. We'll discuss various writing policies to help decide which writing policy would be suitable for the distributed cache we want to design.
Eviction policies	Since the cache is built on limited storage (RAM), we ideally want to keep the most frequently accessed data in the cache. Therefore, we'll discuss different eviction policies to replace less frequently accessed data with most frequently accessed data.
Cache invalidation	Certain cached data may get outdated. We'll discuss different invalidation methods to remove stale or outdated entries from the cache in this section.
Storage mechanism	A distributed storage has many servers. We'll discuss important design considerations, such as which cache entry should be stored in which server and what data structure to use for storage.
Cache client	A cache server stores cache entries, but a cache client calls the cache server to request data. We'll discuss the details of a cache client library in this section.

Writing policies#

Often, cache stores a copy (or part) of data, which is persistently stored in a data store. When we store data to the data store, some important questions arise:

- Where do we store the data first? Database or cache?
- What will be the implication of each strategy for consistency models?

The short answer is, it depends on the application requirements. Let's look at the details of different writing policies to understand the concept better:
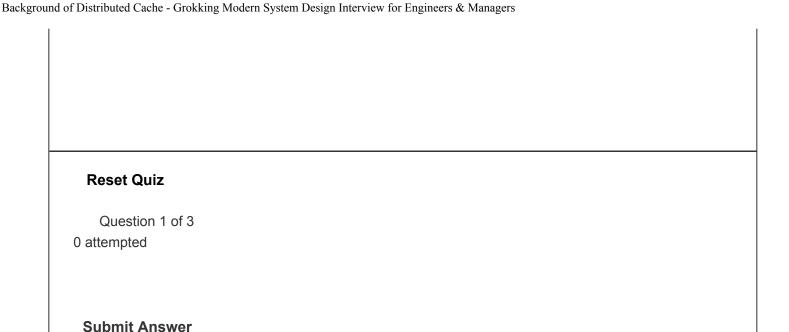
- **Write-through cache**: The write-through mechanism writes on the cache as well as on the database. Writing on both storages can happen concurrently or one after the other. This increases the write latency but ensures strong consistency between the database and the cache.

- **Write-back cache**: In the write-back cache mechanism, the data is first written to the cache and asynchronously written to the database. Although the cache has updated data, inconsistency is inevitable in scenarios where a client reads stale data from the database. However, systems using this strategy will have small writing latency.

- **Write-around cache**: This strategy involves writing data to the database only. Later, when a read is triggered for the data, it's written to cache after a cache miss. The database will have updated data, but such a strategy isn't favorable for reading recently updated data.

Quiz

1

A system wants to write data and promptly read it back. At the same time, we want consistency between the cache and database. Which writing policy is the optimal choice?

Reset Quiz

Question 1 of 3

0 attempted

Submit Answer

Eviction policies#

One of the main reasons caches perform fast is that they're small. Small caches mean limited storage capacity. Therefore, we need an eviction mechanism to remove less frequently accessed data from the cache.

Several well-known strategies are used to evict data from the cache. The most well-known strategies include the following:

- Least recently used (LRU)

- Most recently used (MRU)

- Least frequently used (LFU)

- Most frequently used (MFU)

Other strategies like first in, first out (FIFO) also exist. The choice of each of these algorithms depends on the system the cache is being developed for.

Data Temperatures

Cache invalidation#

Apart from the eviction of less frequently accessed data, some data residing in the cache may become stale or outdated over time. Such cache entries are invalid and must be marked for deletion.

The situation demands a question: How do we identify stale entries?

Resolution of the problem requires storing metadata corresponding to each cache entry. Specifically, maintaining a time-to-live (TTL) value to deal with outdated cache items.

We can use two different approaches to deal with outdated items using TTL:

- **Active expiration**: This method actively checks the TTL of cache entries through a daemon process or thread.
- **Passive expiration**: This method checks the TTL of a cache entry at the time of access.

Each expired item is removed from the cache upon discovery.

Storage mechanism#

Storing data in the cache isn't as trivial as it seems because the distributed cache has multiple cache servers. When we use multiple cache servers, the following design questions need to be answered:

- Which data should we store in which cache servers?
- What data structure should we use to store the data?

The above two questions are important design issues because they'll decide the performance of our distributed cache, which is the most important requirement for us. We'll use the following techniques to answer the questions above.

Hash function#

It's possible to use hashing in two different scenarios:

- Identify the cache server in a distributed cache to store and retrieve data.

- Locate cache entries inside each cache server.

For the first scenario, we can use different hashing algorithms. However, consistent hashing or its flavors usually perform well in distributed systems because simple hashing won't be ideal in case of crashes or scaling.

In the second scenario, we can use typical hash functions to locate a cache entry to read or write inside a cache server. However, a hash function alone can only locate a cache entry. It doesn't say anything about managing data within the cache server. That is, it doesn't say anything about how to implement a strategy to evict less frequently accessed data from the cache server. It also doesn't say anything about what data structures are used to store the data within the cache servers. This is exactly the second design question of the storage mechanism. Let's take a look at the data structure next.

Linked list#

We'll use a doubly linked list. The main reason is its widespread usage and simplicity. Furthermore, adding and removing data from the doubly linked list in our case will be a constant time operation. This is because we either evict a specific entry from the tail of the linked list or relocate an entry to the head of the doubly linked list. Therefore, no iterations are required.

> **Note:** Bloom filters are an interesting choice for quickly finding if a cache entry doesn't exist in the cache servers. We can use bloom filters to determine that a cache entry is definitely not present in the cache server, but the possibility of its presence is probabilistic. Bloom filters are quite useful in large caching or database systems.

Sharding in cache clusters#

To avoid SPOF and high load on a single cache instance, we introduce sharding. Sharding involves splitting up cache data among multiple cache servers. It can be performed in the following two ways.

Dedicated cache servers#

In the **dedicated cache servers** method, we separate the application and web servers from the cache servers.

The advantages of using dedicated cache servers are the following:

- There's flexibility in terms of hardware choices for each functionality.
- It's possible to scale web/application servers and cache servers separately.

Apart from the advantages above, working as a standalone caching service enables other microservices to benefit from them—for example, Cache as a Service. In that case, the caching system will have to be aware of different applications so that their data doesn't collide.

A depiction of service hosts coordinating with dedicated cache servers

Co-located cache#

The co-located cache embeds cache and service functionality within the same host.

The main advantage of this strategy is the reduction in CAPEX and OPEX of extra hardware. Furthermore, with the scaling of one service, automatic scaling of the other

service is obtained. However, the failure of one machine will result in the loss of both services simultaneously.

Hosting cache and application logic in the same machine

Cache client#

We discussed that the hash functions should be used for the selection of cache servers. But what entity performs these hash calculations?

A cache client is a piece of code residing in hosting servers that do (hash) computations to store and retrieve data in the cache servers. Also, cache clients may coordinate with other system components like monitoring and configuration services. All cache clients are programmed in the same way so that the same PUT, and GET operations from different clients return the same results. Some of the characteristics of cache clients are the following:

- Each cache client will know about all the cache servers.
- All clients can use well-known transport protocols like TCP or UDP to talk to the cache servers.

Point to Ponder

Question

What will be the behavior of cache clients to an access request if one of the cache servers is

dead?

Show Answer

Conclusion#

In this lesson, we learned what distributed caches are and highlighted their significance in distributed systems. We also discussed different storage and eviction mechanisms for caches. Caches are vital for any distributed system and are located at different points within the design of a system. It's important to understand how distributed caches can be designed as part of a large system.

Back

System Design: The Distributed Cache

Next

High-level Design of a Distributed Ca…

Mark as Completed

Report an Issue