Notes - Session 5 - Caches

Table of Contents

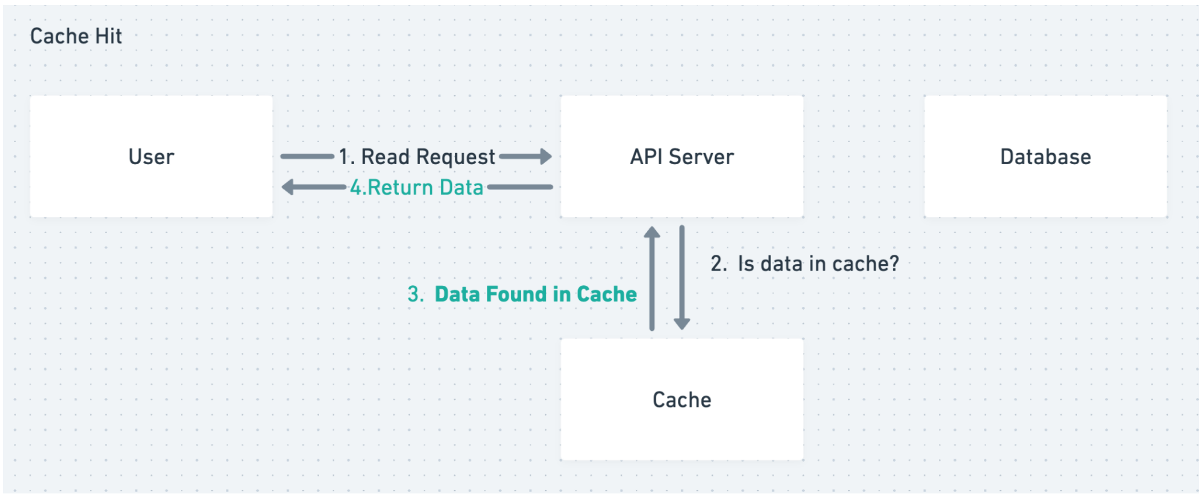
Why should we use cache?

Caches help make things faster by eliminating the following:

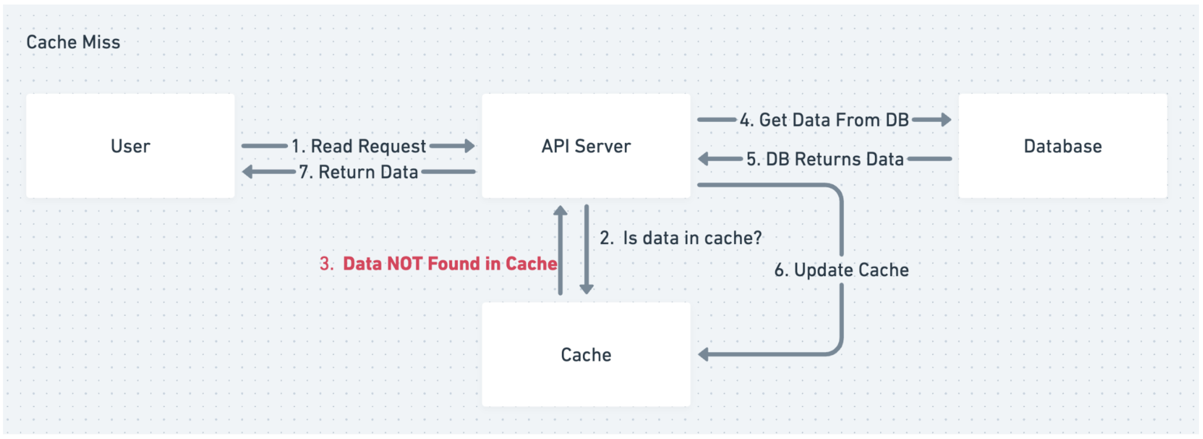
- Network I/O
- Disk I/O
- Costly Computations

How does Cache make things faster?

Cache Hit:



Cache Miss:



Frequently Used Caches:

1. Redis
2. Memcached

How is Cache Updated?

Pull-Based Approach:

Also known as ‘Lazy Population’

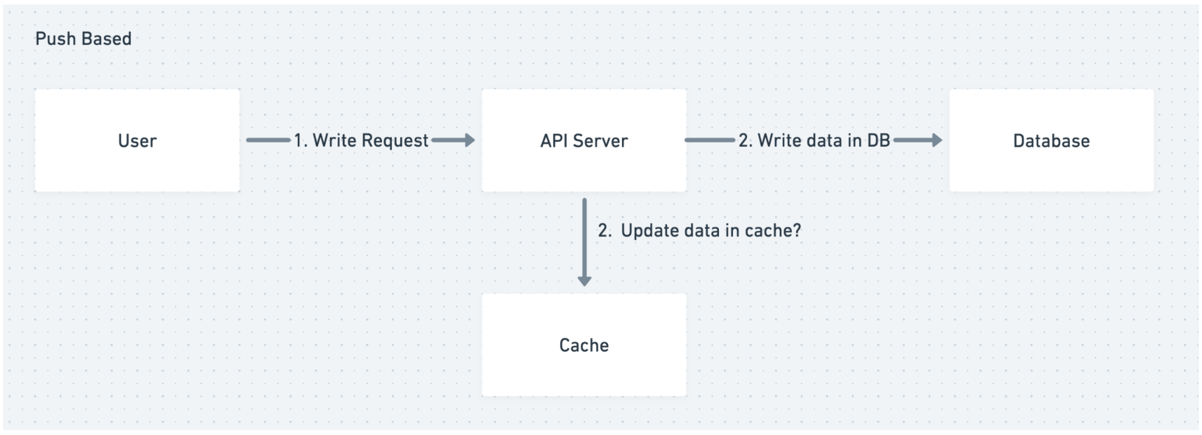
Pull based approach is similar to ‘Cache Miss’ approach described above

Use cases:

1. Almost all standard cache-uses follow pull-based approach

Push-Based Approach:

Also known as ‘Eager Population’



Use cases:

1. Live Cricket Scores
2. Celebrity posts on social media

Why can’t We Cache Everything

Because Caches are very costly. Using Caches in your design is a trade-off between cost and latency

Cache Writing Policies

When designing systems, it’s important to consider, Where do we store the data first? In DB or in Cache?

1. 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.

2. 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.

3. 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 favourable for reading recently updated data.

Questions-about Write-through policy?

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?

2. With reference to performance, what should a write-heavy system avoid?

3. Outdated or stale data entry is a typical issue in which writing policy?

Eviction Policies

Caches are small, and thus fast. Since we can't store everything in the cache, therefore, we need an eviction mechanism to remove less frequently accessed data from the cache.

The most frequently used cache eviction policies are:

1. Least Recently Used (LRU)
2. Least Frequently Used (LFU)
3. First In First Out (FIFO)

Data Temperatures:

Data can be classified into three temperature regions depending on the access frequency:

Hot: This is highly accessed data.
Warm: This is less frequently accessed data.
Cold: This is rarely accessed data.

Cold data frequently gets evicted from the cache and gets replaced with hot or warm data.

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.

How do we identify stale data?

Caches maintain metadata about each cache entry. The metadata includes a value called 'Time To Live (TTL)'.

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

1. Active expiration:

   This method actively checks the TTL of cache entries through a daemon process or thread.

2. 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

In case of distributed cache, it's important to understand the following to improve the performance of distributed cache:

- Which data should we store in which cache server?

- What data structure should we use to store the data?

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.
    - Simple hashing won't work in case of server failures or while scaling
    - Consistent hashing is best suited for this

- Locate cache entries inside each cache server
    - Typical hash functions to locate a cache entry to read or write inside a cache server
    - Cons
        - Doesn't say anything about how to implement a strategy to evict less frequently accessed data from the cache server
        - Doesn't say anything about what data structures are used to store the data within the cache servers

Linked List

Doubly Linked List is more suited for this use-case.

Why so?
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.

Scaling Caches

Caches scale the same as Databases i.e.
1. Horizontal Scaling
2. Vertical Scaling

Sharding In Caches

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

1. Dedicated Cache Servers

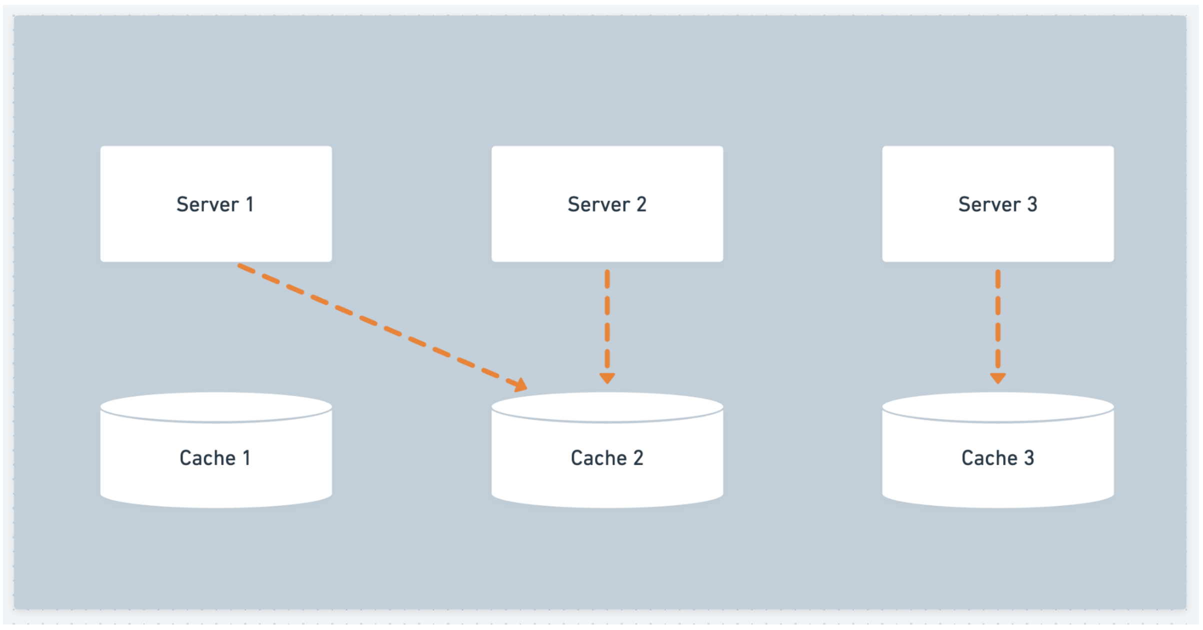
Cache is separated from the App/Web/API servers

Pros:

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

Cons:

- Need to ensure that data from different applications don't collide



2. Co-located Cache

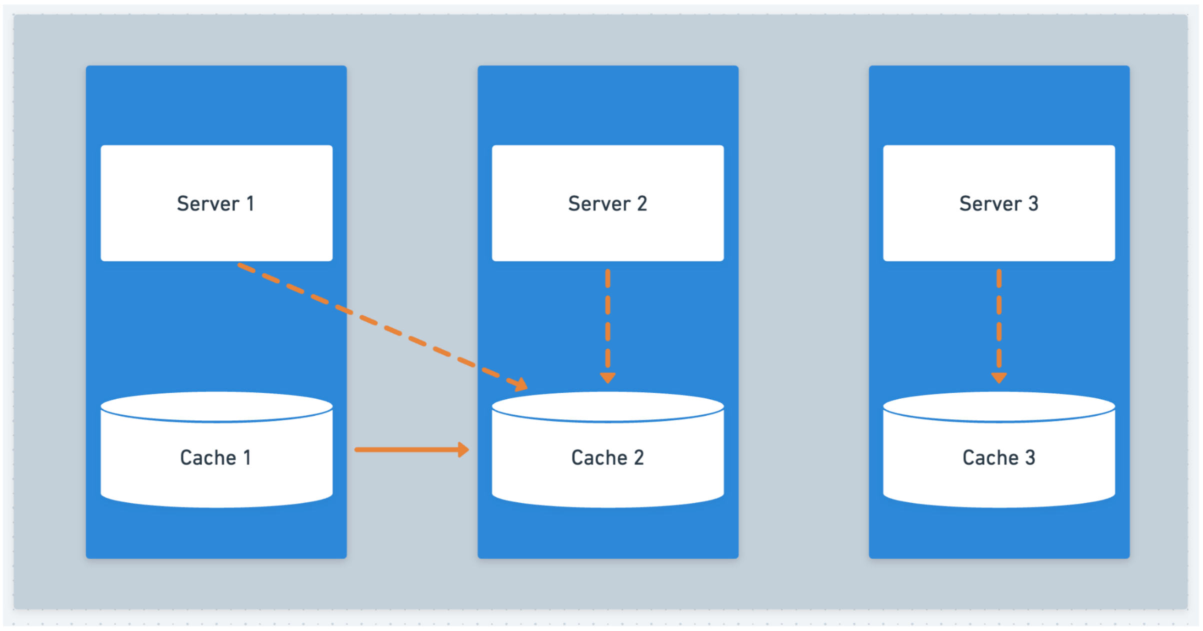
The co-located cache embeds cache and service functionality within the same host.

Pros:

- The main advantage of this strategy is the reduction in CAPEX and OPEX of extra hardware
- With the scaling of one service, automatic scaling of the other service is obtained

Cons:

- The failure of one machine will result in the loss of both services simultaneously


Server 1
Server 2
Server 3
Cache 1
Cache 2
Cache 3