INTERVIEW READY

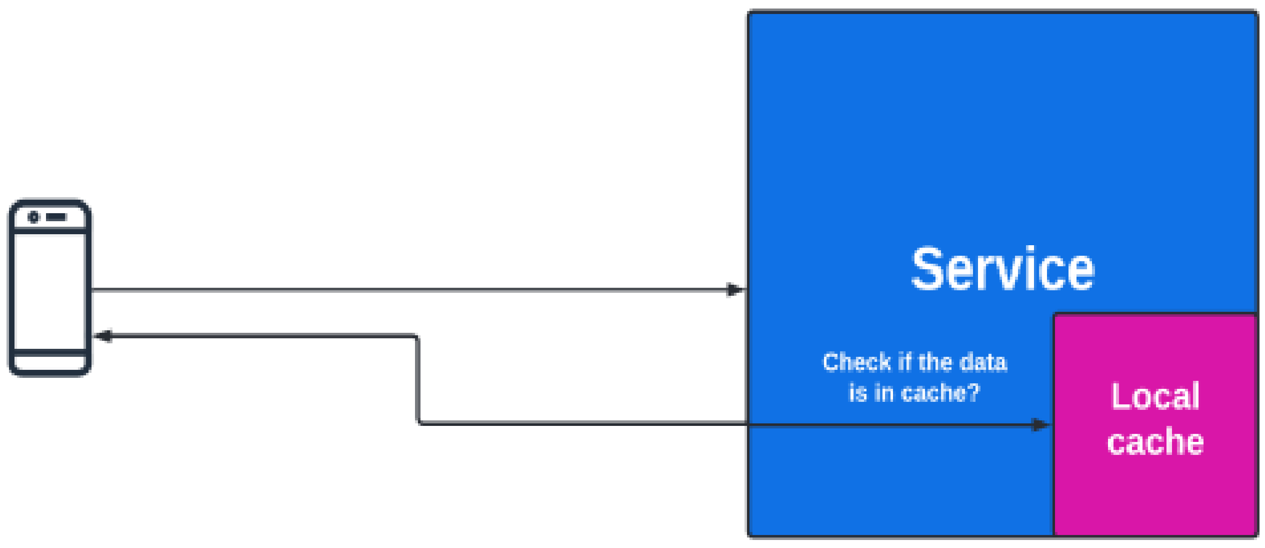
What is a cache?

A cache is temporary storage that stores a part of our database in memory.

It is much faster to get the data from the in-memory storage, so it **reduces the response time**. Also, it **reduces the load on the database** because if the data is present in the cache it does not send requests to the database.

Different types of caches

1. **Local Cache**

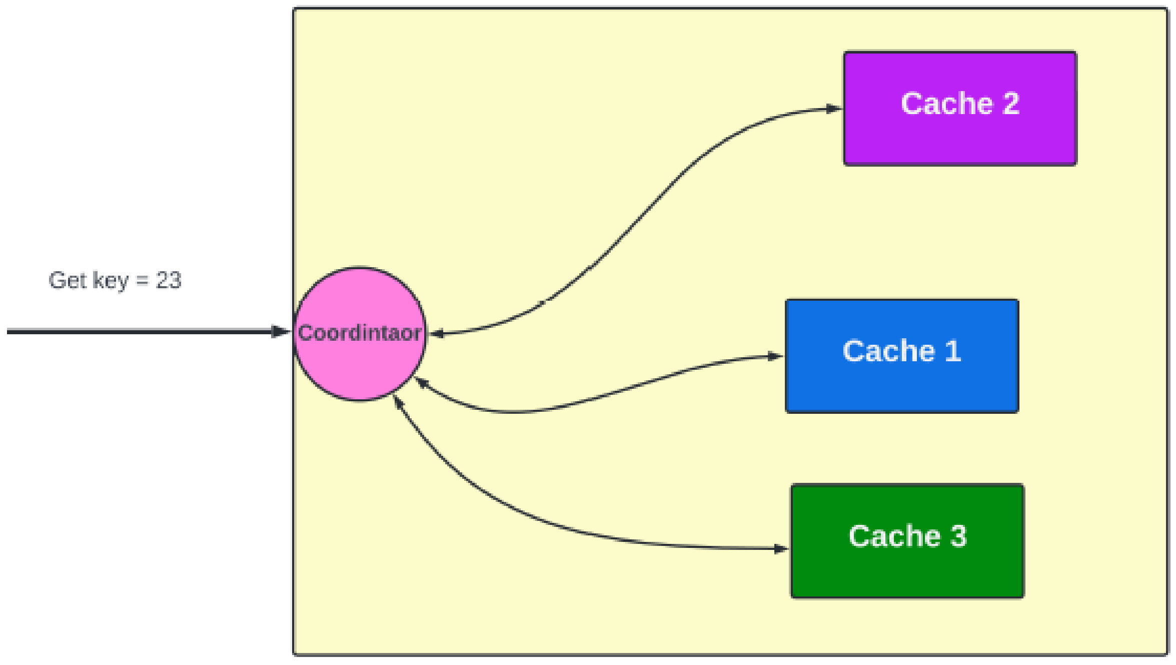


A local cache uses memory inside the service itself.

- Since the data is present in the memory, it is much faster to get the data (because we don't need to make any network calls).
- The issue with a local cache is that it can cause a fanout. For example, we have three boxes and each one uses a local cache to store data. If we want to update the profile data of a user, we need to send requests to all the boxes. **This is a fan out.**
- To avoid fanouts we can shard the data and distribute load across the boxes using a load balancing algorithm (like consistent hashing). But what if one of the nodes fails? To avoid this we can replicate the data to multiple nodes. However, this can lead to **data inconsistency**. So using a local cache reduces data consistency while increasing response time.

2. **Global cache**
   - Instead of each node having its data copy, we have a central data copy, which is a single node storing all the key-value pairs in its memory.
   - This **improves the data consistency** but **reduces the response time** (because we have to make network calls).

Global Cache Internals



Global cache

- Internally the architecture of the global cache will be a ring architecture.
- There will be a set of nodes and each node will have a set of key-value pairs.
- When we get a read request we can handle that in two ways
    - The coordinator will look at the request and send the requests to one of the nodes (This works if the data is sharded and we know in which node the data is present).
    - We can also broadcast the request to all nodes. If we want the most consistent data then we send the value that is present in the majority of the nodes. If we want the latest data we will send the value that has the latest timestamp. This is known as a **quorum**.
- When we get a write request:
    - If we have sharded the data, then the coordinator can find out which node contains the key, go to that node and update the key.
    - If we have a distributed consensus protocol, the coordinator will send the request to one of the nodes. The node then broadcasts the changes to all other nodes.

  But what happens if we get a read request while the write requests are in progress?
    - Let's say a write request is in progress. Three are a total of three nodes. Nodes 1 and 2 are updated and have the latest data. Node 3 is not updated.
    - We then get a read request but nodes 1 and the crashes. In this case, the reader gets stale data.
    - To avoid this, we can use the consistency guarantee of quorum i.e, **R+W>N**.
        - R = **Minimum Number of nodes we need to read the data from**
        - W = **Number of nodes, we write the data to**
        - N = **Number of nodes**
    - For our case, N = 3, W = 2 (because we had written data to nodes 1 and 2), so the value of R must be greater than 1. Since there is only one node from which we can read the data, we don't accept the data.

Write Policies in a cache

A write policy is triggered when there is a write operation in the cache. **Keep in mind that it is different from the replacement policy. A replacement policy is triggered when there is no space for a new key and a key is evicted from the cache**

A write request means some entry is **added, updated or deleted** in the cache. But because a cache is a source of truth each of the write requests will impact the entire system.

Write-Back Policy

If the key-value pair that is to be updated is present in the cache then it is updated. However, the key-value pair is not immediately updated in the database. So as long as the cache is alive, users will get consistent data. However, if the cache is not alive, the data will be stale.

- To avoid this problem we can use **Timeout-Based Persistence**. In this mechanism, we have a TTL (Time to Live for each entry in cache). If the timestamp becomes greater than the TTL, the entry is evicted and the data is persisted in the database. This makes our database **eventually consistent**.

- Another approach is to use **Event-Based Write Back**. In this mechanism, instead of keeping a TTL, we are keeping an event-based trigger. For example, we can keep the count of several updates, if it becomes greater than 5 we persist it in the database.

- We can also use **Replacement Write Back**. For each entry in the cache, we have an attribute that tells us if the entry is updated or not. When we want to evict the entry we update the entry in the database if the entry was updated in the cache.

This policy is efficient. It is especially useful when we want efficient querying in our cache and we are fine with persistence not being 100%.

Write-Through Policy

In this policy, when there is a write request, we evict the key that is being updated, while simultaneously updating the database. The next time there is a read request, that is when the cache polls the database for the entry, persists the entry and sends the response to the user.

However, we can run into problems when using this policy.

For example,

- Initially, we have `X = 10`
- There is a write request for `X = 20`

- Then there is a read request for `X`, but the write request is not updated yet. So the read request returns `X = 10`. So it can cause inconsistency.

To avoid such problems, we can lock the data which is to be written and only unlock the data after the update operation is completed.

This policy is useful

- **When we need a high level of consistency**
- **When we need a high level of persistence**

However, it is **less efficient compared to the write-back policy**.

Write-Around Policy

In this policy, when we get a write request, **instead of updating the entry in the cache, we update the entry in the database.** Now when we get a read request, we will send the stale value from the cache. And we will be getting stale value until the entry is evicted from the cache.

This policy is useful

- **When we need a high level of efficiency**
- **When we need a high level of persistence**

However, it makes our system **eventually consistent**

Replacement Policies

The way we determine which key needs to be evicted is based on the replacement policy.

LRU Policy (Least Recently Used)

In this policy, we **evict the entry that has not been used for the longest.**

To implement this policy, we need to add another data point i.e., `Last Used`.

To understand this policy, let's take an example.

Suppose we have the following data in the cache:

Key	Value	Last Used
2	12	12
1	56	56

We can only store at most two entries in the cache.

If we want to add a new entry to this cache (let's say pair `{3,16}`) at timestamp 80, we need to evict one of the entries. Out of the above two entries, `{2,12}` was not used for the longest **Time interval in which**

an entry was not used = current_timestamp - LastUsed for that entry.

So we will evict the entry `{2,12}` adn add the new entry `{3,16}` to the cache.

Now the cache looks like this:

Key	Value	Last Used
3	16	80
1	56	56

LFU Policy (Least Frequently Used)

In this policy, we **evict the entry that is used least frequently.**

To implement this policy, we need to add another data point i.e., `Frequency` .

Now let's use the same example as above. We have a cache that can store at most 2 values and has the following entries.

Key	Value	Frequency
2	12	2
1	56	7

If we want to evict a key from the cache and add a new entry (let's say `{5,35}`), we will evict the entry `{2,12}` . This is because it is the least frequently used entry. Then we will add the pair `{5,35}` with frequency 1.

If we add another entry (let's say `{6,36}`), we will evict the entry `{5,35}` . If you notice, the hot entry of 5 got kicked out of the cache. So we are continuously loading and evicting from the cache. **When the miss ratio is high and there are a lot of loads and evictions, we call it thrashing**. It happens because the cache does not have enough space (Number of entries cache can store << Number of entries in the database).

In most cases, **LRU Policy works better than LFU**.

Replacement Policy in Memcache

In Memcache we have two different data stores:

- One data store is used to store entries that are less requested. It is also known as cold region
- Another data store is used to store entries that are more requested. It is also known as hot region

To understand how Memcache works, let's take an example.

Suppose we have a cache, where both hot and cold regions can store at most 2 entries. Both hot and cold regions use the least recently used (LRU) policy.

Initially, the cache is empty.

- **We get a read request for key** `1` **at timestamp** `20` There is no entry in the cache for key `1` so it is a miss. We will add the entry `{1,56,20}` to the cold region.

- **We get a read request for key** `1` **again at timestamp** `21` There is no entry in the **hot region** for key `1` but it is present in the **cold region**. So we **promote** `1` to the hot region and add the entry `{1,56,21}` to the hot region and also remove it from the cold region.

- **We get a read request for key** `1` **at timestamp** `22` There is an entry for key `1` in the hot region. So we **hit** the cache.

- **We get a read request for key** `4` **at timestamp** `23` There is no entry in the cache for key `4` so it is a miss. We will add the entry `{4,64,23}` to the cold region.

- **We get a read request for key** `5` **at timestamp** `24` There is no entry in the cache for key `5` so it is a miss. We will add the entry `{5,65,24}` to the cold region.

- **We get a read request for key** `2` **at timestamp** `25` There is no entry in the cold region for key `2` but the cache is full, so we need to evict an entry from the cold region. **Since we are using the LRU policy, we will evict** `4` **(since it has the smallest timestamp).** And add the entry `{2,67,25}`

- **We get a read request for key** `2` **at timestamp** `26` There is an entry for key `2` in the cold region but not in the hot region. So we promote it to the hot region and add the entry `{2,67,26}` to the hot region and also remove it from the cold region.

Hot region

Key	Value	Last Used
1	56	21
2	67	26

Cold region

Key	Value	Last Used
5	65	24

- **We get a read request for key** `5` **at timestamp** `27` There is an entry for key `5` in the cold region. So we need to promote it to the hot region. But the hot region is already full. So we need to evict an entry from the hot region. Since we are using the LRU policy, we will demote `1` to the cold region (we are evicting one from the hot region and adding it to the cold region). After evicting `1` , we add the entry `{5,65,27}` to the hot region and also remove it from the cold region.

Now the regions look something like this:

Hot region

Key	Value	Last Used
2	67	26
5	65	27

Key	Value	Last Used
1	56	27

It is important to note that the frequency of usage also matters. If an entry is used more frequently, it is more likely to be promoted to the hot region.

So whether the entry lives in the hot or cold region is decided by the frequency of usage. And whether the entry needs to be evicted is decided by its LastUsed timestamp.

This process is known as **Segmented LRU**.

Memcache has three regions - **cold, warm and hot**. The hot region does not contain **the Last Used** timestamp. As new entries come in, entries in the hot region are pushed to the warm or cold region.

That's it for now!

You can check out more designs on our video course at InterviewReady.