**Caching**

Cache is defined as hardware/software component which helps in serving data which is either **frequently requested** or it is **expensive to compute** on, so cache stores the computed response and helps in **saving expensive operations**.

Ex: Suppose a client requests an image to server, which in turn fetches from database. Fetching image from database might be an expensive operation performed by server (time consuming), hence, when the same client demands the same image again, we can store the previously fetched image in cache and respond with same image again, without need of going to db.

Caching can be done on server side (in reverse proxy) or on client side (forward proxy or browser cache).

Req### Fetch

DB

Server

Client

Res###

|  |  |
| --- | --- |
| Key | Value |
| Req123 | Res569 |
| Req209 | Res908 |
| Req786 | Res008 |

`

Cache memory

Server/ application layer checks in cache for the request sent by user, whether it was fetched previously and do we have response cached for it.

If server is able to get the response of request from Cache memory, we call it **Cache Hit**.

If server is unable to fetch response from cache (we call it **cache miss**), it means, the request is new and was not demanded before, hence it needs to fetch from DB, perform computation and send response to client and also store in cache for next time.

**Cache Invalidation**

Data stored in cache is volatile and it needs to be invalidated or removed after some time.

Cache Invalidation is needed because, the data which you store will get stale at some point of time. Process of removing the old value of cache is called cache invalidation.

Ex: Suppose, we request the server for fetching current active users. Server response with answer as 5000 and stores in cache as well. Now after 15-20 mins, if we again ask for current active users, it will directly fetch from cache and respond as 5,000.

But if user asks the same question after 2-3hrs, then 5,000 will obviously be a wrong response**.** Hence, response stored in cache is wrong and needs to be removed and filled with new value when fetched from database. This is called warming up the cache.

So how to decide on invalidating a cache?

Using expiry time. (TTL)

Against every entry in cache, you mention a time to live (TTL), like few secs, mins, hrs, days.

Hence, whenever a request comes after the set time limit, the cache will be invalidated and new response will be fetched from DB and stored in cache again.

It’s actually very difficult to determine what should we keep the TTL and it depends on the purpose of app.

**Cache Eviction**

Cache memory is limited. Suppose we can store only 1000 key-value pairs in cache memory. Then when a new 1001st request comes, and needs to be stored in cache, then any one of the key-value pair has to be deleted to make room for new request. This way of deleting a cached item is called **cache eviction.**

**Cache Eviction Strategies:**

1. First in First Out (**FIFO**): Oldest value is removed first.
2. Least Frequently Used (**LFU**): Value which is least fetched from cache by server for responding to client, is removed. Depends on **number of times** accessed.
3. Least Recently Used (**LRU**): The cache item which was not fetched by server recently, is removed. Depends on last **time** when it was accessed.

**Cache Patterns**

1. **Cache Aside Pattern.**

Req### Fetch

DB

Server

Client

2. Res###

Cache

When client requests from server, it checks cache and responds with data available in cache. If cache does not contain the answer/response for that request, Server application communicates with DB, gets the value, does processing, update cache and then respond to client.

Here, cache is kept aside of database, it does not directly talk to database, server app does the communication.

Suppose, due to a request from client, server updates a value in database

(a = ~~10~~ -> 15) but, when client demands the same value, it fetches from cache memory, since its expiry is not met. So while asking value of “a”, server gets it from cache (a = 10) which is an old value.

Thus, this problem can be avoided by adding a logic that, whenever an update happens in database, for any value, similar update must be done in cache memory too.

Hence, combination of TTL and logic is used.

1. **Read Through Pattern**

Req###

Server

DB

Cache

Client

2. Res###

In this model, the server application never talks to Database, it only talks to Cache memory.

When first request comes, there will always be a cache miss. Then cache will fetch from DB, populate cache memory and then send data to server app.

**Advantage**:

Useful in use cases where there are **Read-Heavy** requests, where user frequently reads data from server.

**Disadvantage**:

When request is new, there will always be **a cache miss**, where the cache needs to perform a trip to DB.

This problem can be solved by **pre-heating** the cache. If it’s known in advance, that what kind of data can be requested, then cache is filled with that data, before client can make any requests.

Generally in read through pattern, the cache memory is implemented by third party providers or library and the modelling between cache and DB has to be similar.

Whereas, in cache-aside pattern, modelling of cache and DB can be different.

1. **Write through pattern**

Req###

Server

DB

Cache

Client

2. Res###

Similar to read through, here writes are done to cache memory first and then to DB.

Only disadvantage is added latency, of writing to cache and then again to DB, where in cache aside, we directly wrote to DB.

Read and write through are generally used together.

1. **Write Around Pattern**

Req### Read

Server

Cache

Client

1. Update

DB

1. Res###

Write

Here, in case of reading, we read from cache, saving time.

In case of writing, we write to DB directly and avoid the extra trip of updating cache first hence, an advantage over write through pattern.

This pattern is used when our application is **write heavy**.

1. **Write Back Pattern**

Req###

Server

DB

Cache

Client

2. Res###

In this pattern, reads are done from Cache, whereas writes are also done to cache, but are accumulated in cache for some time.

After a while, all the **write** requests are **sent in Bulk** to Database.

This pattern is useful to **handle DB Failure**. Suppose DB goes down for some time, all write requests are kept with cache. When DB is up and running, all writes are performed.

Disadvantage: If **cache** goes **down**, all writes are lost.

**Summary**

* **Read aside pattern:**

1. Useful when application is read-heavy.
2. Works even when cache goes down.
3. Need to implement update logic + decide on TTL to keep cache ad DB consistent.

* **Read/Write Through pattern**

1. Great alternative to read-heavy workloads.
2. Data modeling of cache and DB needs to be same.
3. Cache failure results in system failure.
4. Cache layer leads to extra latency while writing, which can be solved by **write around pattern.**

* **Write Back Pattern**
  1. Can handle database failure by accumulating all writes and then updating in Bulk at once.
  2. Useful for write heavy workloads
  3. If cache goes down, system goes down.
  4. Used by various DBs as internal implementation.

**Cache can sit anywhere, browser, forward / reverse proxy, server app, between server and DB, our DB can have its own cache.**

**Depending on your need.**