Jahong Liu

System Design Basics 10: Caching

# Caching

Exploring caching's practical uses and its impact on system design

---

## Creating a Tweet

The CPU cache is essentially the computer's speediest memory, allowing the CPU to perform read and write operations significantly faster than on the RAM or disk.

Caching is simply the process of making copies of data.

In a single computer this can mean copying data from RAM or disk into the CPU's cache memory. Even though this data is already present in RAM and Disk, duplicating it in the cache enables us to read and write more data at much quicker rates.

When you have multiple copies of data that are being updated, they are bound to go out of sync. This is not only true of single computers, but database replicas as well. We will discuss this in a later chapter.

Even though the cache outperforms Disk and RAM in terms of speed, it does have a downside: its capacity is restricted, usually to just kilobytes or megabytes. This limitation poses a considerable challenge. As a result, the operating system must make thoughtful decisions regarding what data to store in the cache.

While we are talking about caching as it pertains to a single machine, caching is also widely used in distributed systems.

## Real Life example from Neetcode.io

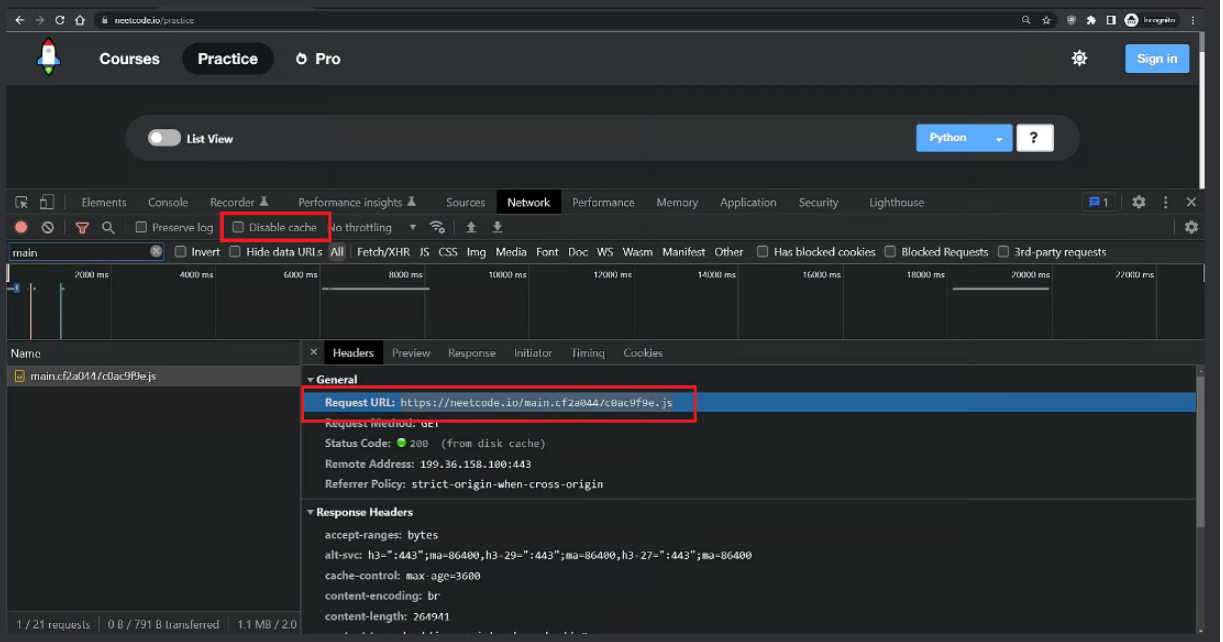
How does a browser decided which files to cache on disk? Usually, it's the static elements that get cached. This might include images or a page whose content remains unchanged irrespective of the user or the number of times it's refreshed.

For instance, on `https://neetcode.io`, if we navigate to the network tab in our developer tools and search for a file named `main`, we'll find a file that looks like `main.cf2a0447c0ac9f9e.js`. It's a static JavaScript file that runs within your browser. This makes it a prime candidate for caching, as it's inefficient to make a network request each time the page loads, particularly when the response stays identical.

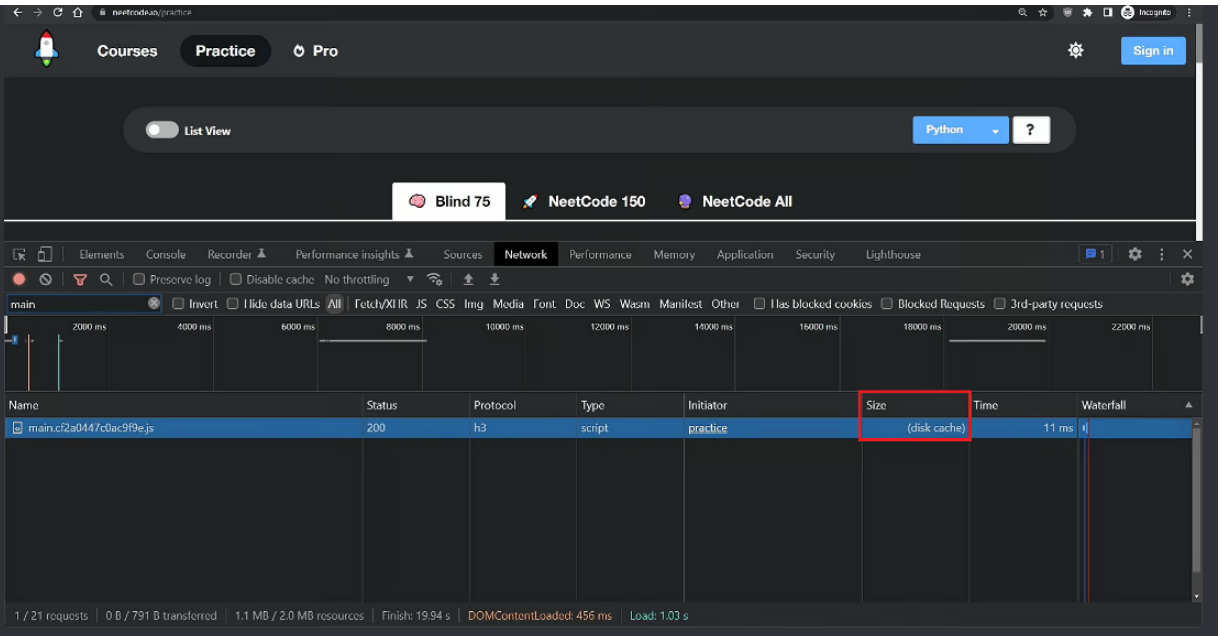
By caching this page, we sidestep additional HTTP requests when we need to pull up this JavaScript page—it's readily available in our cache.

Where is this cache located? In this case, your browser is using the disk as a cache, since while it *is* slower than memory, it's still much faster than a network request (in most cases).

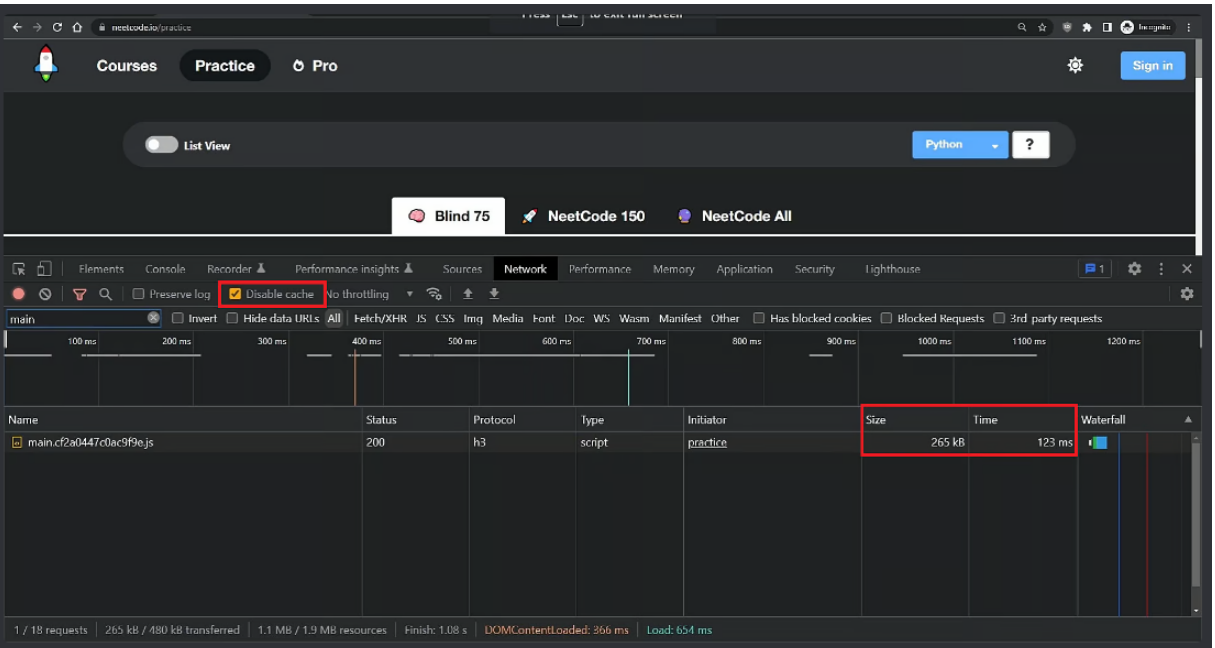
The visuals below demonstrate the performance differences between when the file is cached and when it is not cached.




The visual above shows the cached JS file. Notice that the cache is enabled.




> Looking under the size column, we can see that the file is in disk cache.




> The visual above demonstrates the impact of disabling cache on the time taken to fetch the file.

Notice how it only `123 ms` to fetch the resource via a network request, where as it took `11 ms` via disk. This is because caching is disabled and we are forced to make a network request.

> **Question**: The difference between `123 ms` and `11 ms` is rather miniscule so why even bother enabling the cache? **Answer**: Because we are dealing with only one file. If we have a 100 resources and each takes an additional `100 ms` to load the cost starts to add up. This negatively impacts the user experience and even small delays in page load time can lead to users abandoning a website.

# Diving deeper into caching

## The client's perspective

## Diving deeper into caching

### The client's perspective

When a browser needs to load a resource, such as an image file, it follows a sequence of steps to determine where to get the file:

1. **Check the Memory Cache**: The browser first checks its memory cache. This is used for resources downloaded in the current browsing sessions (since memory is non-persistent).
2. **Check the Disk Cache**: If the resource isn't in the memory cache, the browser checks the disk cache, a more persistent cache that contains resources from sites visited in the past.
3. **Network Request**: If the resource isn't in either the memory or disk, the browser makes a network request to the server hosting the resource.

When a cached file is found, this is known as a **cache hit**. When a cached file is not found, this is known as a **cache miss**.

> It should also be noted that data could be cached and we could still have a cache miss. This data might be stale and the cache might have expired.

## Cache Ratio

A cache hit ratio refers to how many content requests a cache can fulfill successfully, relative to the total number of requests it receives. This can be easily calculated by:

$$\frac{no.\ of\ cache\ hits}{no.\ of\ cache\ hits + no.\ of\ cache\ misses}.$$

To convert this into a percentage, we can multiply this ratio by $100$. As you can guess, a higher percentage is preferable. This ratio is particularly important for CDNs (Content Delivery Networks), which we will discuss in the subsequent chapter.

## The server’s perspective

The server’s perspective is indded more complex than that of the client’s. We can illustrate this with example.

Revisiting the Twitter scenario, when a client issues a getTweet( ) request, how should the server determine which tweets to cache and which to retrieve from the disk? One way could be gauging this by the frequency with which certain tweets are accessed. To illustrate, only a minor proportion of tweets, made by a limited number of individuals, really go viral. These are typically posted by public figures or celecrities. In contrast, the bulk of content creators won’t see much engagement on their posts. Hence, the popularity of

tweets could be a viable criterion to decide which tweets make it into the cache. Following this, we delve into different caching modes.

### Write-around cache

About 6, 000 tweets are being created every second. Out of all of these tweets, it only makes sense to add them to memory if they are being accessed. Under the write-around cache mode, when a new tweet is posted, it is written directly to the main storage (disk) rather than the cache. By writing aroud the cache, Twitter will save cache space for more popular content that is accessed more frequently.

However, if all of a sudden, a tweet becomes popular and gain traction, it would be added to the cache after the first access. So, when the tweet is accessed again, subsequent reads would result in cache hits.

### Write-through cache

Another cache mode is the write-through cache. The write-through cache will write to both the memory and the disk, regardless of whether the tweet is accessed or not. Applying this to our Twitter example, each time a new tweet is posted, it would be written t both the main storage and the cache and is readily available in both locations. With this stratefy, we avoid stale data and data inconsistency. Both the disk and the cache memory will have the same version of the tweet. However, this might put more load on the memory bus, filling the cache up with dta that might not be accessed again.

## Eviction Policies

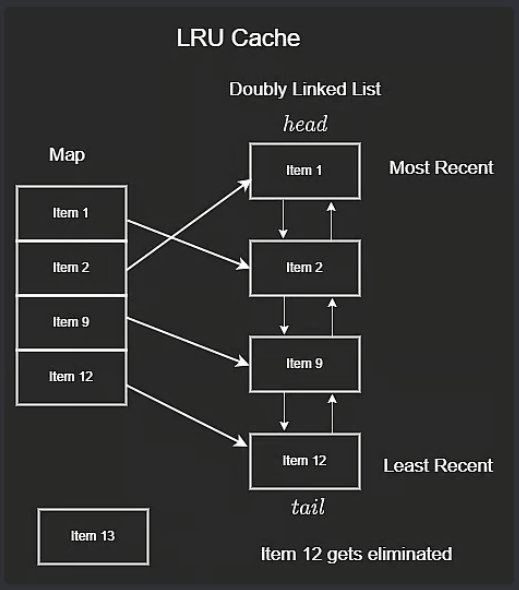
An eviction policy is a system that determines which items get removed from the cache when the cache is full and new items need to be added. Since the cache size is limited, the system has to decide what to evict from the cache. There are a couple of eviction policies that are important to discuss:

### FIFO (First In First Out)

One example of an eviction policy is First In First Out (FIFO). The FIFO policy is similar to the queue interface. When the cache becomes full, the first piece of data to be cached is evicted first.

### LRU (Least Recently Used)

Imagine doing spring cleaning, where your aim is to get rid of the least recently used items such as clothing and office supplies. The concept of the Least Recently Used (LRU) cache is based on the same idea. The principle behind LRU caching is that if an item has not been accessed for a long time, it is less likely to be accessed in the future as well. Therefore, the item should be evicted from the cache. LRU caching would be particularly useful if there were a single person with a really popular tweet, as we wouldn't want that tweet to be removed from our cache.




> The visual above shows that Item 12 would get eliminated when deciding eviction upon adding Item 13.

### LFU (Least Frequently Used)

This one might make more sense now that we understand LRU. Least Frequently Used (LFU) eviction policy evicts the items that are used least frequently. It assumes that if an item is not accessed frequently, it is unlikely to be accessed frequently in the future as well.

In terms of implementation, LFU can be implemented using key-value pairs, where the key represents the item and the value represents the frequency of its usage. When the cache space runs out, the item with the smallest frequency is evicted.

While this approach seems reasonable, it has limitations when applied to Twitter. For example, tweets from 2013 that have a large number of views might never be evicted due to their frequency, which would prevent new tweets from being stored. Consequently, LRU is a better model for Twitter.