Caching Fundamentals

# What is Caching?

In software engineering, a cache is a data-storage component that can be accessed **faster** or **more efficiently** than the original source of the data.

It is **a temporary holding location** for information **so that repeated access** to the same

information can be **acquired faster and with fewer resources**.

what happens if the data service has to perform some complex operation in order to acquire the data?

This complex operation **can be resource intensive**, **time intensive**, **or both**.

If the service has to perform the **complex operation** **every time** a consumer requests the data, a significant amount of time and/or resources can be spent retrieving **the same data** **over and over again**.

Instead, with a cache, the first time the complex operation is performed, the result is returned to the consumer, and a copy of the result is stored in the cache. The next time the data is needed, rather than performing the complex operation all over again, the result can be pulled directly out of the cache and returned to the consumer faster and using fewer resources. This particular caching strategy is commonly referred to as cache-aside—more on that later.

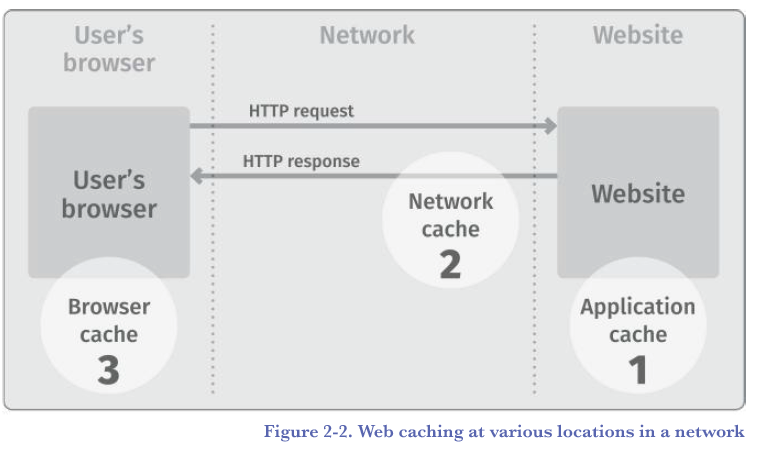
## Caching Use Cases

There are many different types of caches used in computer science, and they are used in almost all aspects of **software** **and hardware** development. Common use cases for caches include:

### HTTP Application Cache

When you access a web page, depending on the website, the page may require a fair amount of processing in order for the page to be created, sent back to you, and rendered in your browser. However, **the page**—**or significant portions of the page**—**may not change much from one request to the next**. A cache is used to store pages and/or portions of pages so they can be returned to a user faster and using fewer resources than if the cache was not used. This results in a **more responsive website**, and the capability for the website to handle **significantly more simultaneous users** **with a given number of computational resources.**

An HTTP application cache is illustrated in section **1** of Figure 2-2.



### HTTP Network Cache

When a web page is accessed by a user, **the site generating the page may be far away**, sometimes in another part of the country or another country altogether. It can take a relatively **long time** for the page **to be**

**sent from the website server to your browser before it displays**.

HTTP protocols provide the capability to **store copies of certain static web pages or page elements at intermediate locations along the route between the browser and the website server**. Because these caches **allow content to be stored closer to the browser, the website can be retrieved faster** and the results displayed quicker. **This is often the case for static content such as pictures, images, and videos.** In Figure 2-2 (in the previous page), section 2 is an HTTP network cache.

### Web Browser Cache

Your web browser itself may also cache some content so that it can be displayed almost instantly rather than waiting for it to be downloaded across a significant distance and potentially over a slower internet link. This **browser cache is particularly effective for images and other static content**. In Figure 2-2, section 3 is a web browser cache.

### Service Cache

Individual services can have **internal caches to assist in performing complex and time-consuming operations**. These are called service caches. A service cache is **very similar to an application cache, but** operates and

caches in **support of the individual service components that make up an application**.

### Application Programming Interface (API) Cache

When one service calls another using an API, **the response to the API call can be stored in a cache and**

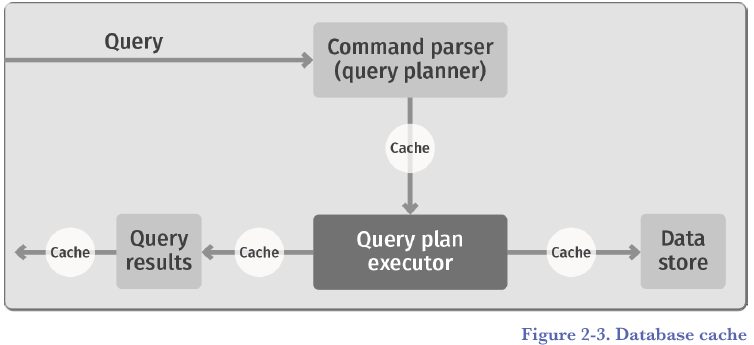
**used to return results for equivalent future calls**.

Come back to this point

### Database Cache

When a database receives a request to retrieve data, the processing required may be quite extensive. Database caches can be used in several locations within databases in order to speed up queries as well as to

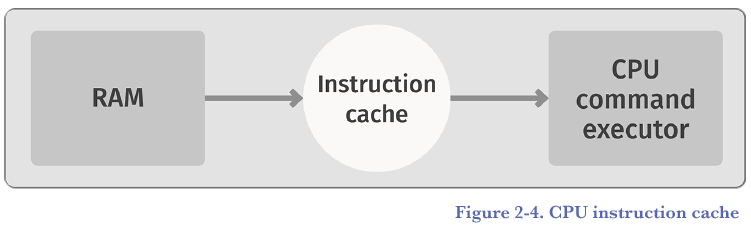
conserve compute resources, which aids in database scaling. These caches are typically used to store query results, intermediate results, query parsing results, and query plans. Figure 2-3 illustrates some of the locations where databases can cache the results of a query.



### CPU Cache

There are many caches within computers, including in the central processing unit (CPU). The core CPU caches executed commands so that repeated execution of the same or similar instructions can occur considerably faster.

In fact, much of the increased speed of computers in recent years is due to improvements in how commands are executed and cached, rather than actual clock speed improvements. Figure 2-4 illustrates this type of cache.



### Memory Cache

RAM is fast, but not fast enough for high-speed CPUs. In order for data to be fetched fast enough from RAM to keep up with high-speed CPUs, RAM is cached into an extremely fast cache and commands are executed

from that cache.

### Disk Cache

Disk drives are relatively slow at retrieving information. RAM, used as the storage medium for a cache, can be put in front of the disk, so that repeated common disk operations (such as retrieving contents of a directory listing), can be read from the cache rather than waiting for the results to be read from the disk.

## What does a Cache Need to be Useful?

In order for a cache to work and provide value, the following must be true:

1. **The operation** necessary to calculate or retrieve the requested data **must either be slow or require resources to acquire/calculate.**
2. **The cache must be able to** store and properly retrieve the result **faster** **using fewer resources than did the original source.**
3. Sometimes, the data involved must be unchanged from one request to another. **While it is possible to cache dynamically changing data in certain circumstances, data that doesn’t change from request to request is easier to use in caching situations.**

4. **The operation to calculate or retrieve the requested data** must **have no side effects** (that is, **it doesn’t store data**, **it doesn’t make changes to other systems**, and it **doesn’t control other software or hardware**) on the system operating, **other than the consumption of resources**. **More on this in a moment.**

5. **The data must be needed more than once. The more times it is needed, the more effective and useful the cache is.**

For a cache to be effective, you need a really good understanding of the statistical distribution of data access from your application or data source. When your data access has a normal (bell-curve) distribution, caching is more likely to be effective compared to a flat data access distribution. There *are* advanced caching strategies that are more effective with other kinds of data access distributions.

Explanation of the above paragraph from **Deep Seek:**

*If your data access follows a normal distribution (a bell curve), caching is likely to be effective. In a normal distribution, a small subset of data is accessed very frequently (the peak of the curve), while the rest is accessed less often. Caching the frequently accessed data can significantly improve performance.*

*In contrast, if data access is evenly distributed (flat distribution), caching is less effective* ***because there’s no clear subset of data that is accessed more frequently than others. This makes it harder to prioritize what to cache****.*

## Dynamic vs Static Caches

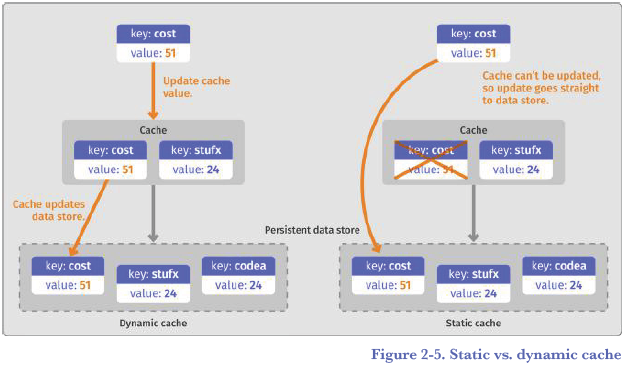
Not all caches are accessed in the same manner. While caches are **used and accessed** in many different ways, there are two fundamental types of patterns for how caches are used.

**They are static, sometimes called read/only, caches and dynamic, sometimes called read/write, caches.**

**The difference is how the caches are populated when data in the underlying data store is changed,** and is illustrated in Figure 2-5:

In a dynamic cache, when a value is changed in the data store, that value is also **changed directly in the cache**—**an old value is overwritten with a new value.** The cache is called a dynamic cache, because **the application** using the cache **can write changes directly into the cache.** **How this occurs can vary** depending on the type of usage pattern employed, but the application using the cache **can fundamentally change data in a dynamic cache.**

In a static cache, when a value is stored in the cache, it cannot be changed by the application. **It is immutable**. **Changes to the data are stored directly into the underlying data store, and the out-of-date values are typically removed from the cache**. **The next time the value is accessed from the cache, it will be missing, hence it will be read from the underlying data store and the new value will be stored in the cache.**



# Why Caching?

consider a simple service that multiplies two numbers. This service could be called repeatedly from multiple sources with many different multiplication requests.

It can take on an entire series of multiplication requests, **process them all** independently, and return the results.

But what if the same multiplication(3x4 for example) call has been requested a second time? This poor little service, however, doesn’t know that it’s already been asked to perform the request, and **it goes about all the work it needs to do to calculate the result** … again.

Multiplying 3 and 4 to get 12 may not seem like a very onerous task. But it’s more complex than you might imagine, and the operations a multiplication service such as this might be asked to perform could be significantly more complex. If the service already has performed the same operation and returned the same result, why should it have to redo the same operation? Although some services can’t skip performing a repeated operation, in a service such as this, **there is no reason to redo a calculation that has already been performed**.

This is where caching comes into play. Rather than sending requests directly to the service, a cache is added to the architecture. Then, to use the service, the cache has to be utilized. Assuming a cache-aside strategy, a request to the service to multiply 3 times 4 could therefore work like this:

**1.** Consult the cache, to see if there is an entry representing “3x4”.

**2.** If there is an entry, return the result from the cache. **STOP**.

**3.** If there is not an entry, call the service and get the result of 3 times 4.

**4. Store the result of the service call into the cache under the entry “3x4”(with 3x4 as the key for example).** 🡪*it says that in this stage the cache gets Hydrated. It’s an interesting term*

**5.** Return the result.

That way, when a repeat request comes in, as shown in the bottom of the figure (section 3), **the request can be satisfied** by the cache **without ever calling the service at all**.

## OK, What Problem Does Caching Solve?

As we discussed in Section 1, “What Is Caching?”, there are many use cases for caching, and many types of use cases that can be solved by caching.

Here are some typical **problems that caching can either solve or help to solve in a modern application:**

**• Performance improvement.** Caching improves latency. **Latency is the new outage**, and if you can avoid having to do a time-intensive calculation by simply using a cached result, you can reduce the latency for all requests that utilize the cache. Over time, this can have a huge performance impact on your application.

**• Scaling.** As an application scales up, **resources can become constrained**. Caching allows an application to **reduce the need to use redundant resources** (as shown in our multiplication-service example), which improves the overall scale at which the application can operate without becoming saturated.

**• Resource optimization.** Some resources can be quite expensive in computation usage, memory usage, etc. Multiplying 3 times 4 is not really expensive at all, but running a large data simulation might be very expensive. Caching can reduce the need for some of these operations, which reduces the resources required, and **can improve throughput.**

**• Convenience and availability.** Sometimes, the resources needed to perform a calculation might not be available. They could be used for other purposes (especially in a high volume, scaled application), or they could be offline or simply unavailable. **If the result is already available in a cache, you can return a result without needing the underlying resources, and hence the lack of availability of those resources won’t be an issue.**

# Cache Concerns

Of course, like any other technology, caching also involves trade-offs. **Not all situations are appropriate use cases for caching**. In many cases, caching may not add value, and in a few cases, caching can actually degrade performance. Specifically, there **are three things to be careful of when deciding if and how to cache:**

**1.** Caching can cause the application **to not execute desired side effects** **of targeted operations**

**2. Inconsistent data in a cache**

**3. Poor cache performance**

## Dealing with Side Effects

In our multiplication example, the service doesn’t do anything but calculate a result. It doesn’t store data, it doesn’t make changes to other systems, and it doesn’t control other software or hardware. It only calculates a result. The service is said to have no side effects.

Compare this to a service that might perform some physical action (such as turning a car’s steering wheel) or might change data in some other system (such as updating data in a user’s profile record). These types of services have side effects, because simply calling the service causes changes to the application, system, or the external world.

**An application or service is said to have a side effect if it modifies some state—virtual or physical—outside of the local environment of the application or service**. Often, if you can observe that an application has done something, then what you observed is a side effect.

• A service that changes the position of a car’s steering wheel has an observable impact—it has a side effect.

• Software that stores data in a database has an external impact—it’s changing the state of the database.

Often, these “side effects” are desired, and may even be critical to the application doing its job. Although it is possible to cache services that have side effects, **care must be taken that the implied side-effect actions are handled correctly.**

Improperly caching services with side effects is the cause of many software failures and system outages. **It is easy to introduce bugs into a system when adding caching if side effects aren’t properly taken into account.**

## Inconsistent Cache Data

What happens if the cache for the multiplication service doesn’t have the value “12” stored as the result of “3 times 4”, but instead has the value of “13” stored?

Then **the cache is said to be inconsistent, because the data in the cache doesn’t match the value in the backing service.**

**I add:** *I think he doesn’t mean that a cached data has been calculated wrong but it says that the cached data doesn’t represent what the actual service would have produced if it had been accessed directly. Like when you have stale data in your cache that must be updated or any other reason.*

Although it might not seem likely in our multiplication service example, depending on the chosen caching pattern, **maintaining cache consistency can be difficult in some situations.** **It’s very easy for cached results to get out of sync with the resource they are caching.**

Cache consistency is such an important topic that we will dedicate most of an entire chapter to it—see Chapter 7 of the Redis document, “Cache Consistency.”

## Performance of a Cache

Caches are useful because typically they improve your application performance in some way—whether in terms of latency, resource usage, throughout, or some other measure.

But this isn’t always the case. **Caches are most effective when the following two criteria are true:**

**1.** In a cache-aside pattern, the resources it takes to check and fill the cache are significantly smaller than the resources it takes to perform the backing operation in the first place.

**2.** The number of times that the cache has the correct response (**and hence the backend operation can be avoided**) **is significantly greater than the number of times that the cache does *not* have the up-to-date response** (and hence the backend operation must execute normally).

The more both of these two statements are true, the more effective the cache is. The less either or both of these are true, the less effective a cache is. In extreme cases, a cache can actually make performance worse. In those cases, there is no reason to implement the cache in production.

If, for example, the resources it takes to manipulate the cache is greater than the resources it takes to perform the backing operation in the first place, then having a cache can make your performance worse.

Additionally, when you check to see if the correct response is in the cache, and it is unavailable more often than not, then the cache isn’t really helping that much, and the overhead of checking the cache can actually make performance worse. Situations like this are not good use cases for caching.

For more on cache performance, see Chapter 9 of the Redis document, “Cache Performance.”

# An Important note from the First 3 Introductory Sections

* be aware that improper caching can actually **make performance worse**.
* Even more important, improper caching **can introduce bugs and failures into your system.**

**GET BACK TO CACHE USE CASES**