Basic Caching Strategies

A typical use case for a cache is as a temporary data store in front of the system of record. This temporary memory store **typically provides faster access to data** than the more-permanent memory store.

This is either because:

* the cache medium used **is itself physically faster** (e.g., RAM for the cache compared with hard disk storage for the permanent store)
* or because the cache is **physically or logically(What do you mean by logically? could It mean that it bypasses some calculations? Or maybe like when you have slave databases in a cluster?)** located nearer the consumer of the data (such as at an **edge location** or on a local client computer, rather than in a backend data center).

Note: *Edge locations are data centers that are owned by cloud service providers and are located all over the world. They are designed to deliver services with the lowest latency possible. These data centers are closer to users, often in major cities, so responses can be fast and snappy.*

**At the most basic level,** this type of cache simply **holds duplicate copies of data** **that is** **also** **stored in the permanent memory** store.

When an application needs to access data, it typically first checks to see if the data is stored in the cache. If it is, the data is read directly from the cache. This is usually the fastest and most reliable way of getting the data. However, if the data is not in the cache, then the data needs to be fetched from the underlying data store. After the data is fetched from the primary data store, it is typically stored in the cache so future uses of the data will benefit by having the data available in the cache.

There are many different ways that caches can be accessed, and there are many different ways the data in the cache can be stored and consumed. There are also a number of standard cache strategies, architectures, and usage patterns that make use of the cache in different ways.

Here, though, we’re going to separate caching into **inline** **and** **cache-aside strategies**, based **on how data flows through the cache.**

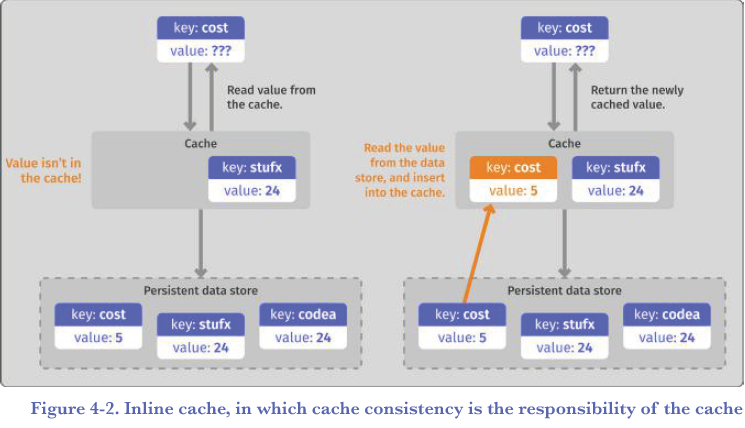
# Inline Cache

An inline cache—which can include **read-through**, **write-through**, and **read/write-through** caches—is a **cache that sits in front of a data store**, and **the data store is accessed through the cache.**

This is where the **application treats cache as the main data store** and [**reads data from it** and **writes data to it**](https://www.alachisoft.com/resources/docs/ncache/prog-guide/data-source-provider.html). **The cache is responsible for reading and writing this data to the database**, thereby **relieving the application of this responsibility.**

Take a look at Figure 4-2. If an **application** wants to read a value from the data store, it **attempts to read** the value **from the cache**. If the cache has the value, it is simply returned. **If the cache does not have the value**, then **the cache reads the value from the underlying data store.** The cache then remembers this value and returns it to the calling application. The next time the value is needed, it can be read directly from the cache.

**So in this scheme, cache consistency is the responsibility of the cache itself.**

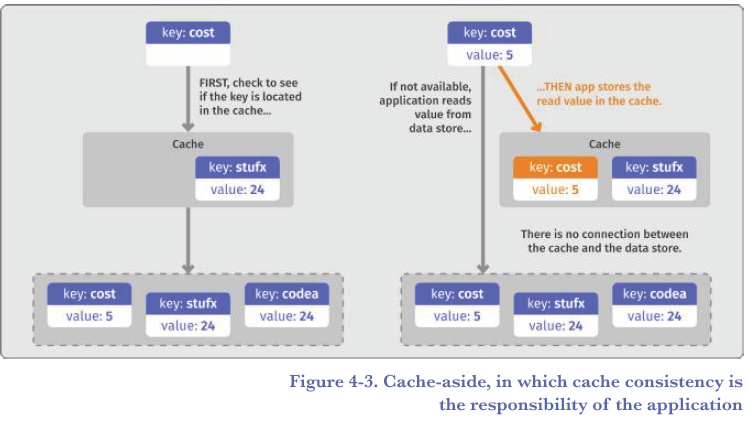


# Cache-aside Patterns

In a cache-aside pattern, the cache is accessed **independently of the data store**. **In a cache-aside pattern, cache consistency is the responsibility of the application.**

When an **application** needs to read a value, it first checks to see if the value is in the cache. If it is not in the cache, then **the application accesses the data store directly** to read the desired value. **Then, the application stores the value in the cache for later use**. The **next time** the value is needed, **it is read directly from the cache**.

Unlike an inline cache, in the cache-aside pattern **there is no direct connection between the cache and the underlying data store.** **All data operations** to either the cache or the underlying data store **are handled by the application**. This is shown in Figure 4-3.



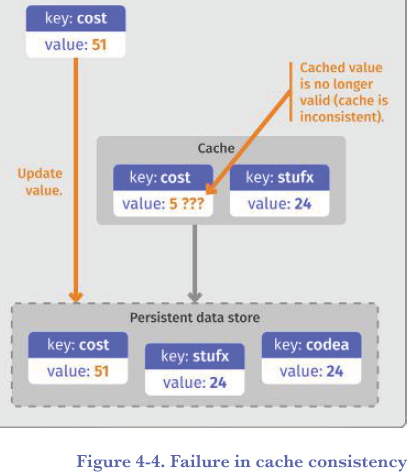
# Cache Consistency

As stated earlier, a cache simply stores a copy of data held in an underlying data store. Because it is a copy of the data that is stored in the cache, when things go wrong or someone makes a mistake, it is possible that the value stored in the cache may differ from the value stored in the underlying data store. **This can happen, for example, when the underlying data changes and the cache is not updated with the new value in a timely manner. When this happens, a cache is considered inconsistent.**

**Cache consistency** is the measure of whether data stored in the cache has the same value as the source data that is stored in the underlying data store. Maintaining cache consistency is essential for successfully utilizing a cache.

This problem is illustrated in Figure 4-4. In this diagram, an application changes a data value in the underlying data store (changing the key “cost” from the value “5” to the

value “51”). Meanwhile, the cache keeps the older value (the value “5”). **Because the cache has a value that is different from the underlying data store, the cache is considered inconsistent.**



How do you maintain cache consistency between a cache and the underlying data store? There are many caching techniques for successfully maintaining cache consistency.

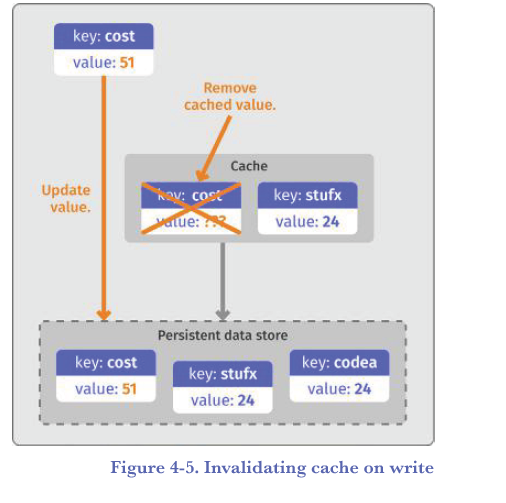
## Maintaining Cache Consistency with Invalidating Caches

The most basic way to maintain cache consistency is to use **cache invalidation**. Cache invalidation is, quite simply, **removing a value from a cache once it has been determined that the value is no longer up to date.**

Take a look at Figure 4-5. In this diagram, the value of key “cost” is being updated to the value “51”. This update is written by the application directly into the data store. In order to maintain cache consistency, once the value has been updated in the data store, the value in the cache is simply removed from the cache either by the application **or the data store itself (HOW??)**.

Because the value is no longer available in the cache, the application has to get the value from the underlying data store. By removing the newly invalid value from the cache, **in**

**a cache-aside pattern, the next usage of the value will force it to be read from the underlying data store, guaranteeing that the new value (“51”) will be returned.**



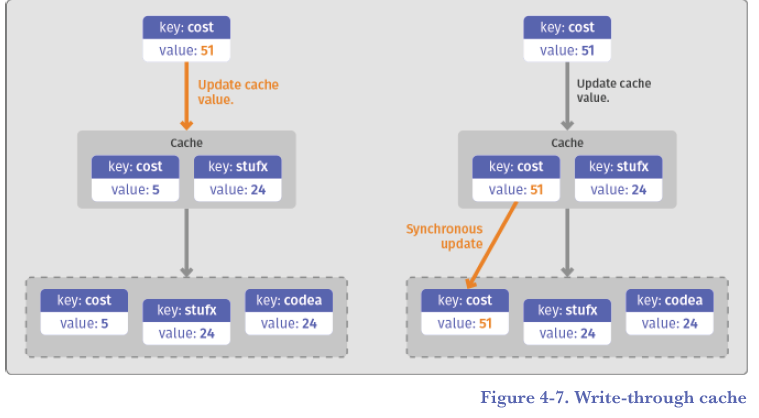
**This strategy must be only used in the cache-aside pattern right?**

## Maintaining Cache Consistency with Write-through Caches

In a write-through cache, rather than having the application update the data store directly and invalidating the cache, the application **updates the cache** with the new value, **and the** **cache updates the data store synchronously**. This means that the cache maintains an up-to-date value and can still be used, yet the data store also has the newly updated value. **The cache is responsible for maintaining its own cache consistency.**

In Figure 4-6, you can see the key “cost” stored in the data store has a value of “5” and that value is also stored in the cache. If an application now wants to update that value to “51,” in a write-through cache, **that value is written to the cache directly**.

**As soon as the write is complete,** both the cache and the data store have the same value (the new value, “51”), and so the cache remains consistent. Anyone else accessing the value from either the cache or the data store will get the new value, consistently and correctly.



I add:

*I think by synchronously they mean that an application’s attempt to update a value will block until the data has been updated in both the cache and the data store.*

***Think about scenarios in which one of the update operations fails? What’ll happen then?***

## Write-behind/Write-back cache strategies

One downside of the write-through strategy is that the actual write is relatively slow, because the write call has to update both the cache and the underlying data store. Hence, two writes are required, **and one of them is to the slow backend data store**.

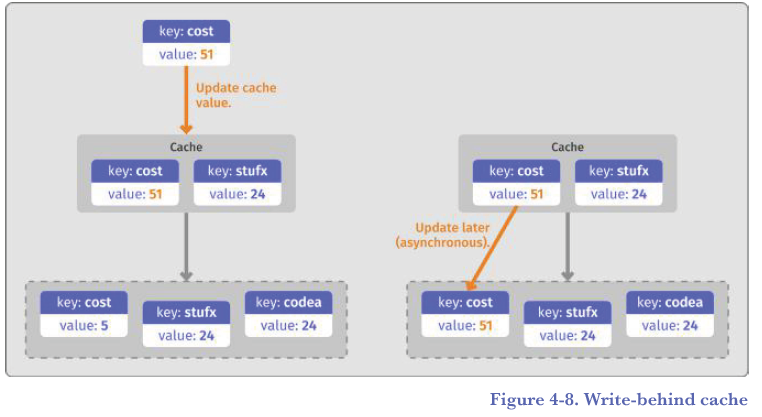
**In order to speed up this write operation, a write-behind cache can be used instead**. With a write-behind cache, the value is updated directly in the cache, just like the write-through approach. However, **the write call then immediately returns(unlike write through that would block right?)**, **without updating the underlying data store**.

**From the application perspective, the write was fast, because only the cache had to be updated.**

At this point in time, the cache has the newer value, and the data store has an older value. **To maintain cache consistency**, **the cache then updates** **the underlying data store** with the new value, **at a later point in time**. This is typically a **background**, **asynchronous** activity performed by the cache.

Although this process results in a faster application write operation, there is a tradeoff. **Until the cache updates the data store** with the new value, **the cache and data store hold different values.** The cache has the correct value, and the underlying data store has an incorrect, or stale, value. This gets remedied when the write-behind operation in the cache updates the data store—**but until then, the cache and data store are out of sync**. **The cache is considered inconsistent.**

**This would not be a problem if all access to the key was performed through this cache.** **However, if there is a mistake or error of some kind and the data is accessed directly from the underlying data store, or through some other means, it is possible that the old value will be returned for some period of time**. Whether or not this is a problem depends on your application requirements. See Figure 4-8.



I’ll add a point about the write-back strategy:

*I think write back is a more generic term for when the datasource is updated later in time, in the case of write-behind (a type of write-back), the data will always be written to the data store some time after the cache gets updated, but the more generic write-back strategy just means later in time due to a trigger of some kind.*

# Cache Eviction