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.**

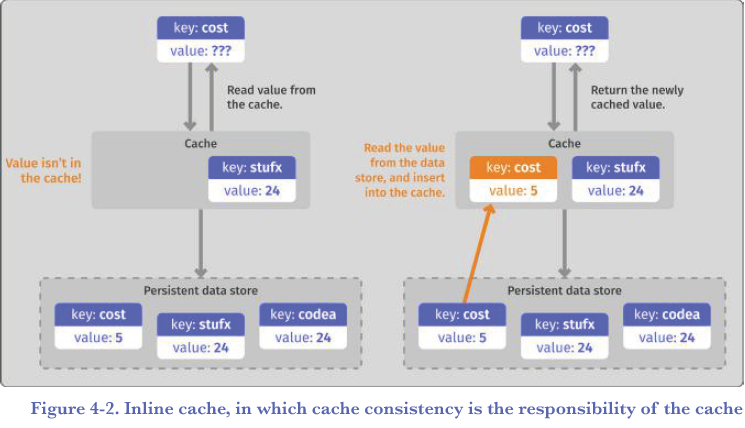
# Inline Cache

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.**

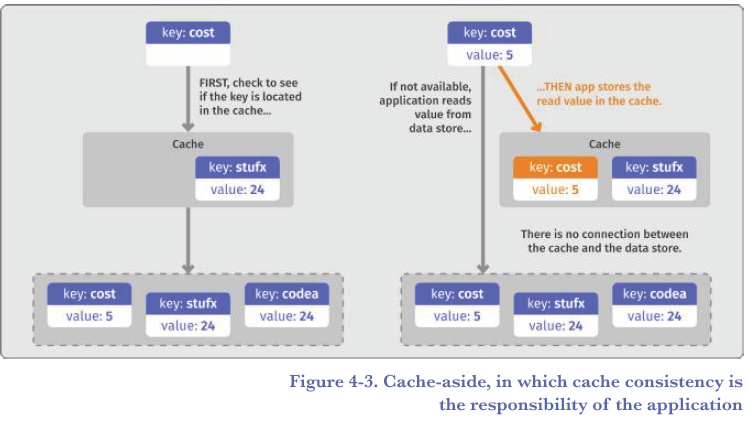


# Cache-aside Patterns

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.



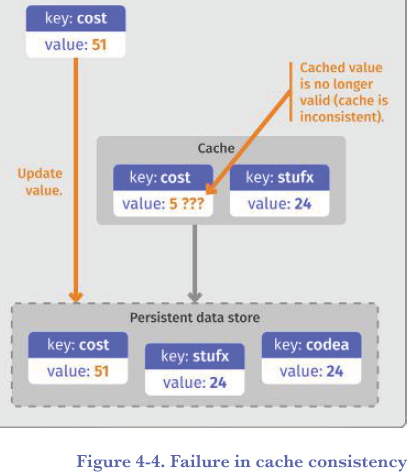
# Cache Consistency

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.

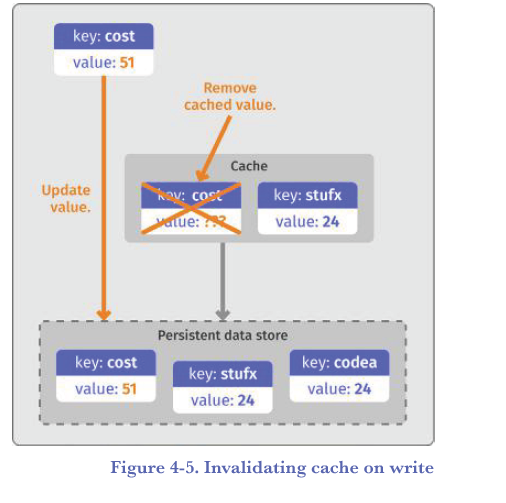
## Maintaining Cache Consistency with Invalidating Caches

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.**



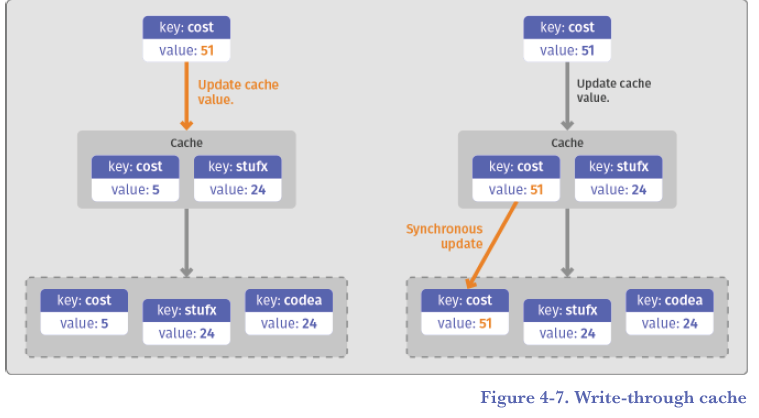
**This strategy must be only used in the cache-aside pattern right?**

## 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**.

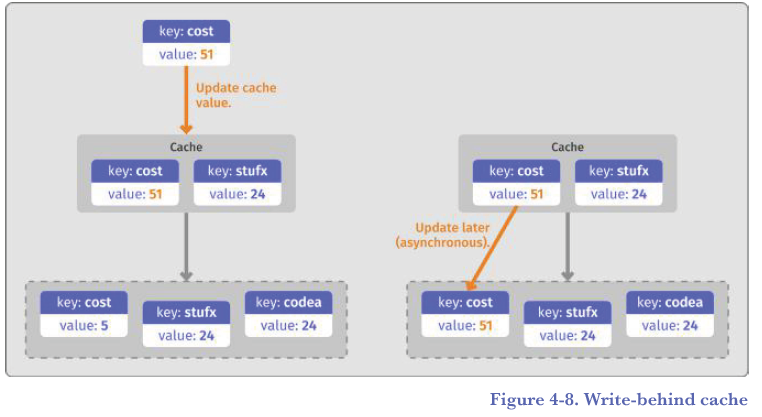
**In order to speed up this write operation, a write-behind cache can be used instead**. With a write-behind cache, the value is updated directly in the cache, just like the write-through approach. However, **the write call then immediately returns(unlike write through that would block right?)**, **without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

# Cache Eviction

Fundamentally, a cache is effective as long as it contains any values that an application requires. Because the purpose of a cache is to provide higher performance and/or higher availability to data than the underlying data store can provide, the cache is useful as long as it contains the necessary values.

However, **a cache is typically smaller than the underlying data store**, so it is necessary **for the cache to contain only a subset of the data held in the underlying data store**.

**In this case, an important job of the cache is to try and determine what data will be needed and make sure that data is available in the cache.**

Initially, this is typically not a problem. In a cache-aside strategy, as values are read, they are simply inserted into the cache. But as time goes on, the cache begins to fill up with data. When a cache is full, and a new piece of data needs to be stored in the cache, how does the cache store the new data? In some cases, the responsibility for cache eviction is relegated to the application. This is commonly referred to as an All-In or no-eviction policy.

I add: *This policy is called all-in or no-eviction policy not because no data will be evicted but because the cache itself doesn’t do any evictions and the application decides on what to be evicted.*

More often, however**, the cache** will remove **older** or **less frequently** **used** data from the cache in order to make room for the newer data. Then, if some application needs that older data in the future, it will need to re-fetch the data from the underlying data store. This process of removing older or less frequently used data **is called cache eviction, because data is evicted, or removed, from the cache.**

Different cache implementations manage the eviction process in different ways for different purposes, depending on the use case’s specific needs. But there are several common methods that are often employed.

I add: *he’s talking about the policies based on which* ***the cache itself****, not the application will attempt to evict data. He called the scenario where the application was responsible for this to be done as no-eviction or all-in policy.*

## Least-Recently-Used (LRU) Eviction

In a cache with a least-recently used (LRU) eviction policy, when the cache is full and new data needs to be stored, the cache makes room for the new data by looking at the data already stored in the cache. It then finds the piece of data that hasn’t been accessed **for the longest period of time**. It then removes that data from the cache and uses the free space to store the new data.

**The premise behind an LRU cache is that data that hasn’t been accessed recently is less likely to be accessed in the future.** Because the goal of the cache is to keep data that will likely be needed in the future, getting rid of data that hasn’t been used recently helps keep commonly used data available in the cache.

## Least-Frequently-Used (LFU) Eviction

In a least-frequently used (LFU) cache, when the cache is full and data needs to be evicted, the cache looks for the data that has **been accessed the fewest** **number of times**, and removes that data to make room for the new data.

**The difference between an LRU and LFU cache is small**. An LRU uses the amount of time since the data was last accessed, while the LFU uses the number of times the data was accessed. **In other words, the LRU bases its decision on an access date, the LFU bases its decision on an access count.**

## Oldest-Stored Eviction

In an oldest-stored cache, when the cache is full, the cache **looks for the data that has been in the cache the longest period of time and removes that data first**.

**This eviction policy is not common in enterprise caches**. This is sometimes called a first-in-first-out (FIFO) cache. The data that was first inserted into the cache is the data that is first evicted.

## Random Eviction

In a random eviction cache, when the cache is full, a randomly selected piece of data is evicted from the cache.

This isn’t a commonly used eviction technique, usually one of the algorithmic approaches is chosen instead. However, **this technique is easy for the cache to implement, and therefore fast for it to execute**. **But** these types of caches **are more likely to evict the wrong data**, which means **they tend to create a larger number of cache misses later**, when still-needed evicted data is accessed once again. **That’s why random eviction is not typically used in production.**

**I add:** *It’s probably useful when you have an even/uniform distribution in access to your data.*

## Time-to-Live Eviction

In a time-to-live (TTL) eviction, data values are given a period of time—potentially seconds, minutes, hours, days, years—that they are to be stored in the cache. After that period has elapsed, **the value is removed from the cache, whether or not the cache is full**.

* **In Redis, this is done at the key level, not at the cache-eviction policy level.(Explore this more)**

**Session management is a common use case for TTL eviction**. A session object stored in a cache can have a **TTL set to represent the amount of time the system waits before an idle user is logged off.** Then, **every time the user interacts with the session, the TTL value is updated and postponed**. If the user fails to interact by the end of the TTL period, the session is evicted from the cache and the user is effectively logged out.

## Cache Persistence

I add: *based on the upcoming sections, you can call this strategy a permanent cache where data is never evicted form the cache by the cache itself. We talked about no-eviction policy where the eviction was the responsibility of the application and I think in this permanent approach even the application won’t evict anything but I don’t think it’s all that important.*

**In a persistent cache(***I think he means permanent cache because in the glossary section, persistent cache is defined as a cache whose data is stored somewhere to be retrieved even after a shutdown***), data is never evicted**. If a cache fills with data, then **no new data is stored in the cache until data has been removed** from the cache using some other mechanism, **sometimes via a manual method**. **That means the newest data, the most recently accessed data, is the data that is effectively “evicted,”** because it’s never allowed to be stored in a full cache in the first place.

This is simpler and more efficient for the cache to implement, because it doesn’t require any eviction algorithm.

This method can be used in cases where the cache is at least as large as the underlying data store. (Of course, **that’s not common, as most caches do not store all of the data from the underlying data store**. Typically, **only certain datasets, such as sessions, are cached.**) **If the cache has enough capacity to hold all the data, then there is no chance that the cache will ever fill, and hence eviction is never required**.

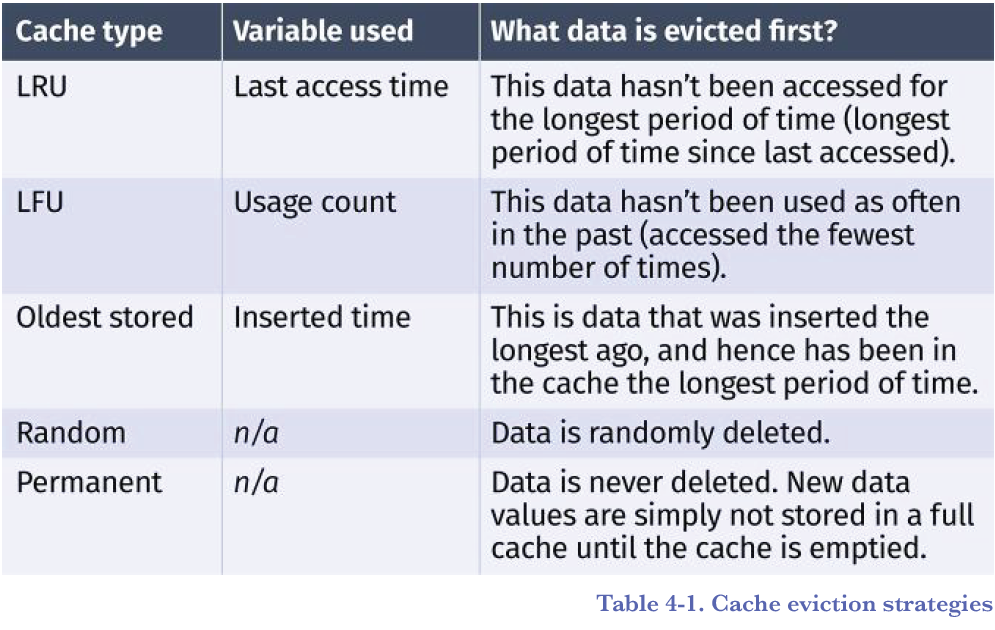
## Cache Thrashing

Sometimes a value is removed from the cache, but is then requested again soon afterwards and thus need to be re-fetched. This can cause other values to be removed from the cache, which in turn requires them to be re-fetched later when requested.

This back-and-forth motion can lead to a condition known as “cache thrashing,” which reduces cache efficiency. **Cache thrashing typically happens when a cache is full and not using the most appropriate eviction type for the particular use case.**

Often, **simply adjusting the eviction algorithm** **or changing the cache size can reduce thrashing.**

## Comparing Eviction Types



* There is no right or wrong eviction strategy, the correct choice **depends on your application needs and expectations.**
* **Most often, the LRU or LFU is the best choice**, but which of those two depends on specific usage patterns.
* **Analyzing data access patterns** **and distribution** is usually required to determine the proper eviction type for a particular application, but **sometimes trial and error is the best strategy to figure out which algorithm to select**.
* The **oldest-stored eviction** strategy is also an option that can be tried and measured against LRU and LFU. He said earlier though that this strategy is not common in enterprise applications
* The **random-eviction** option is not used very often. You can test it in your application, but most situations will find one of the other strategies work better.
* Some applications require **maximum performance** **across an even distribution of data access**, **so evictions cannot be tolerated**. But when using this option, **space management becomes a concern** that must be managed appropriately.

I add: *I think that this option means that you have to cache everything without evicting anything. That is the permanent.*

# Warm vs Cold Caches

In many cases, **especially for session caches**, cache-aside caches are empty when the cache first starts up. But **static data caches(**Why only statics? Maybe because you can have a reliable snapshot of the current data that will not be possible in the case of dynamic data where data is changing regularly**)** are often “**seeded**” to avoid poor performance on startup and rehydration.

I add: *what is the difference between cache rehydration and seeding a cache? Hydration is done to restore data that has previously been in the cache. For example, after a system reboot.*

*Seeding the cache on the other hand is done to reduce the cache misses in the beginning where the cache is empty. Let’s say you want to scale out the cache, you’re not rehydrating the new instance but you pre-load it with data to warm it up.*

In this case (he most likely means when you haven’t seeded the cache), all requests for data from the cache will fail (**a cache miss**), because no data has been stored in the cache yet. This is often called a **cold cache**.

As applications request data, data is read from the permanent data store and stored in the cache. As time goes on, more and more data is stored in the cache and available for use by the consumer applications. Over time, this results in **fewer cache misses and more cache hits**. The performance of the cache improves as time goes on. This is called a **warm cache**.

The process of initially seeding data into a cache is called **warming the cache**, or simply **cache warmup**.

When the data is added continuously over time, the process is referred to as **pre-fetching.**

# Redis as a Cache

* Redis makes an ideal application data cache.
* It runs in memory (RAM), which means it is fast.
* Redis is often used as a cache frontend for some other, slower but more permanent data store, such as an SQL database.
* Redis can also persist its data, which can be used for a variety of purposes, including automatically **warming the cache** during recovery.
* As a cache, **open source Redis is typically used in a cache-aside strategy**, and as such the **programming logic to manage the cache is typically within the application that is using it**.
* **READ ABOUT REDIS GEARS FOR USING IT AS INLINE CHACHE AS WELL**

## Cache Eviction with Redis

**Typically, a Redis cache stores less data than the underlying data store**, so cache eviction is something that must be considered.

By default, when a Redis database fills up, future writes to the database will simply fail, preventing new data from being inserted into the database. This mode can be used to implement **the permanent cache eviction policy**, described earlier.

However, Redis can be configured to operate differently when it fills up, by setting the following option:

**“maxmemory-policy allkeys-lru”**

When the database is filled, old data will be automatically evicted from the database in a least-recently used first eviction policy. This mode can be used to implement an LRU cache.

Redis can also be configured as an LFU cache, using the LFU eviction option:

**“maxmemory-policy allkeys-lfu”**

Redis can implement other eviction algorithms as well. For example, it can implement a random-eviction cache:

**“maxmemory-policy allkeys-random”**

Other, more complex configurations can also be used.

For instance, you can assign expiration times to **individual cached values**, and have **different times allocated to different values**. These values will automatically be evicted at their expiration time.

Additionally, if the database fills before the eviction time has been set, you can use an LRU or random-eviction strategy that prioritizes the values that have an expiration time set, leaving non-expirable values in the database longer:

**“maxmemory-policy volatile-lru”**

**“maxmemory-policy volatile-lfu”**

**“maxmemory-policy volatile-random"**

This gives you plenty of options to implement complex and application-specific eviction strategies.

I add: *these policies indicate that only the values with expiration time will be evicted, and among those an LRU, LFU, or Random policy will be applied. How about when you want to have an LRU for example on the non-expirable ones but you want to prioritize the expirable ones to be evicted first?* ***it needs more investigation.***

Question: How to update TTLs? I think it can be used to have the LRU and LFU at the same time.

## Approximation Algorithms

**It should be noted that the Redis LRU and LFU eviction policies are approximations**. When using the LRU expiration option, Redis does not always delete the true least-recently used value when a value needs to be deleted. Instead, it samples several keys and deletes the least-recently used key among the sampled set.

These approximation algorithms, in practice, hew(hew means cut :) ) very close to the statistical algorithm expectations. **But they require significantly less memory to implement**, **making more memory available for storing data values and keys**. **It is also possible to fine-tune the approximation algorithms to better suit your needs.**