redis

Enterprise Caching: Strategies for Optimizing App Performance

Part 1 of the Enterprise Caching Series

Based on *Caching at Scale with Redis*

by Lee Atchison

Contents

1. What is Enterprise Caching and Why Do You Need It?

Modern businesses need to compete in a cutthroat digital landscape. Users of applications have uncompromising expectations, and apps have evolved to become tremendously complex, requiring databases to have the horsepower, agility, and consistency to process mountains of data in real time, all the time.

What does real time mean?
Research into human response indicates applications have roughly 100 milliseconds (ms)—one third of the time it takes to blink—before users feel like they're waiting for a response. If digital interactions are going to be seamless, then they need to happen in real time. That means every request needs to be sent, processed, and responded to in less than 100ms.

This meteoric shift has magnified the importance of caches, transforming them from bonus performance enhancers to indispensable assets with a direct impact on engagement. Customers are brutal in their assessments of digital experiences and won't tolerate lags or downtime. If a business doesn't produce flawless application experiences, customers will simply move to a competitor that does.

Caches turbocharge application response times by keeping copies of the most frequently accessed data in-memory in front of a slower, original database. Caches take over a lot of the heavy lifting done in the main database, freeing up additional resources to process other incoming queries, and delivering the most important and frequently accessed data in real time.

This makes apps highly responsive to user demands, enabling a seamless experience that eliminates lag, boosts engagement, and enhances brand perception—all of which leads to an increase in revenue and market share.

Businesses looking to create global applications that meet modern expectations need to leverage an enterprise-grade cache. Enterprise caching provides and maintains sub-millisecond response times under extreme workloads with high volumes of globally distributed data. Because enterprise-grade caches are an integral part of mission-critical application architectures, they have a higher degree of consistency, availability, performance, scalability, and geo-distribution capabilities than standard caches.

Although the caching principles remain the same, the technical requirements to effectively provide enterprise-grade caching are far more complex. Using Lee Atchison's book, *Caching at Scale With Redis*, we'll show you the most powerful strategies and techniques for you to optimize cache performance at any scale.

1. What is Enterprise Caching and Why Do You Need It?

The risks of not having an enterprise-grade cache

Digital applications have become so deeply intertwined with everyday life that people turn to them for a multitude of everyday tasks. With this prominence, users have come to expect flawless digital experiences with apps that are always available and perform in real-time.

Enterprise-proven caching is mandatory to provide seamless application experiences on a global level.

A business may have the most exciting and innovative application that perfectly addresses customers' pain points, but without a flawless digital experience, users will switch off and migrate to an app that meets their expectations.

The importance of response time

Salesforce gathered insights from 15,000+ global consumers and business buyers to gain insights on modern customer engagement. In the study, 83% of consumers claimed that the experience a business provides is just as important as the product/services they offer. Also, 80% of customers place the same value on seamless engagement as they do on product quality.

On top of that, slow response times are enough to disengage users, making them more likely to switch to a competitor's app. This is a well-established fact, as many additional studies similarly highlight the correlation between high engagement levels and the customers' perceived value of a product or service.

According to research by Unbounce, 70% of customers surveyed said that poor page loading times directly impact their willingness to make purchases from online retailers. This carries over to mobile browsing, too. A Deloitte Digital study of mobile sites found that a mere 100-millisecond improvement in loading time is enough to boost conversions by 8% for retail sites and 10% for travel sites. In the same study, Deloitte Digital found that application responses to user commands that exceeded 100ms were not perceived as immediate by the user.

This demonstrates that any application that can't send, process, and retrieve data in 100ms doesn't meet user expectations for real-time.

The outcome of disengaged users only heads in one direction, and that's south: a drop in DAUs, MAUs, a stained brand image, dwindling brand value, decimated revenue, and diminished profits.

2. Basic Caching Strategies

"There are three hard things in building software: maintaining cache consistency and off-by-one errors."

Scaling is essential to maximizing cache performance. And to scale it's imperative to understand the basic enterprise caching strategies that are vital to elevating performance, which is what we'll cover in this chapter. Atchison first introduces two basic caching strategies that are determined by how data flows through the cache.

This is followed by an overview of the importance of cache consistency and how failure in cache consistency leads to failure in performance. Atchison dives into the heart of this issue by providing a range of powerful techniques that you can use to maintain cache consistency.

Additional topics include cache eviction, cache persistence, and cache thrashing, each of which are dissected to highlight the role they play in achieving enterprise-level caching. The chapter then ends with an analysis of how Redis provides high performance caching through its approach to cache eviction and approximation.

Reading this chapter, you'll understand the ins and outs of the basic yet significant caching strategies that will take you one step closer to enterprise level.

"Chapter 4: Basic Caching Strategies" from *Caching at Scale with Redis* by Lee Atchison

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 used is itself physically faster (e.g RAM for the cache compared with the hard disk storage for the permanent store), or because the cache is physically or logically located near the consumer of the data (such as at an edge location or on a local client computer, rather than in a backend data center).

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

Figure 4-1 shows this fundamental view of a cache.

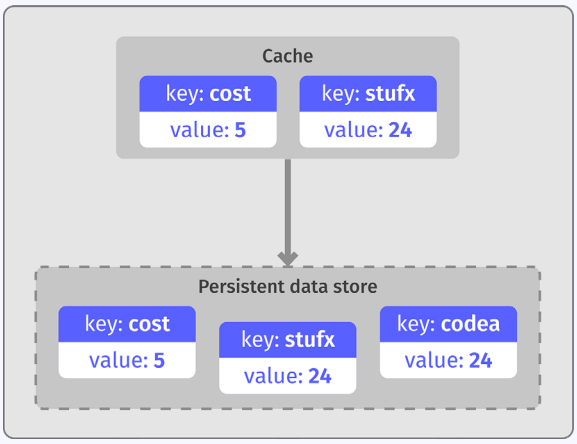

Figure 4-1. Cache in front of a persistent store

2. Basic Caching Strategies

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.

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.

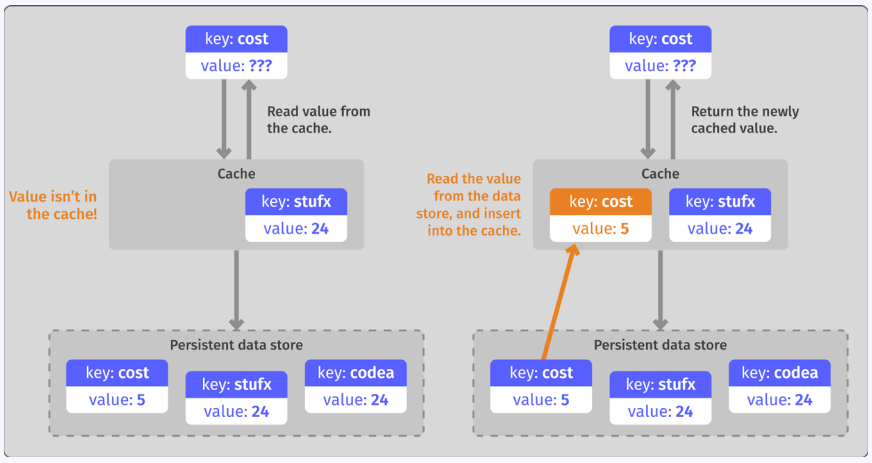

Figure 4-2. Inline cache, in which cache consistency is the responsibility of the cache

2. Basic Caching Strategies

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.

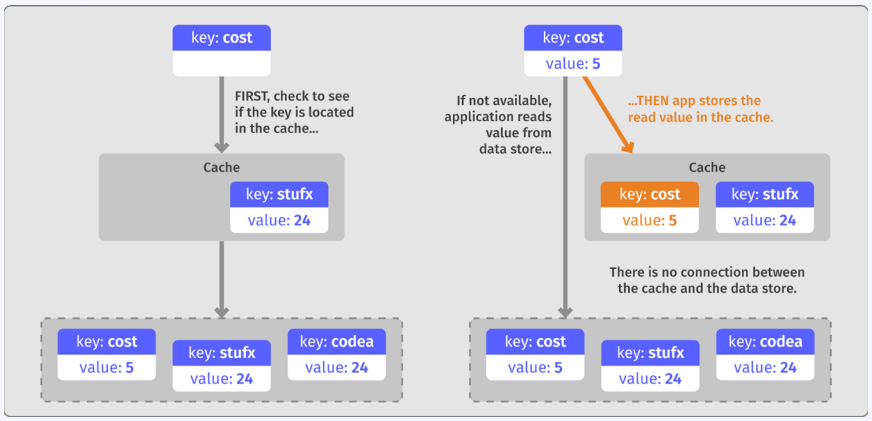

Figure 4-3. Cache-aside, in which cache consistency is the responsibility of the application

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.

2. Basic Caching Strategies

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.

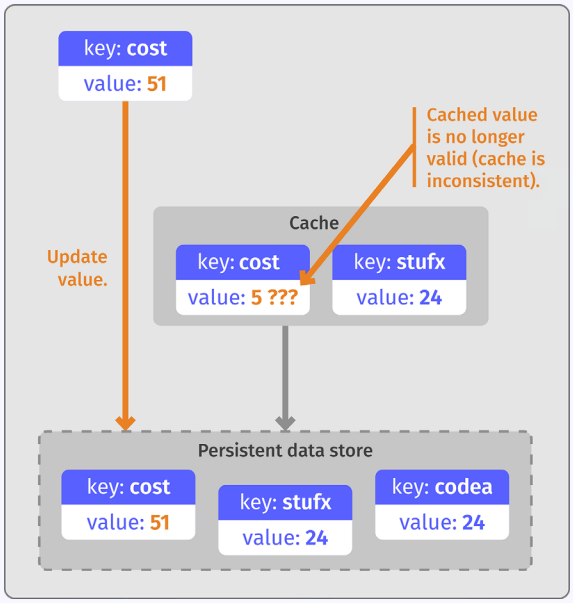

Figure 4-4. Failure in cache consistency

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. 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.

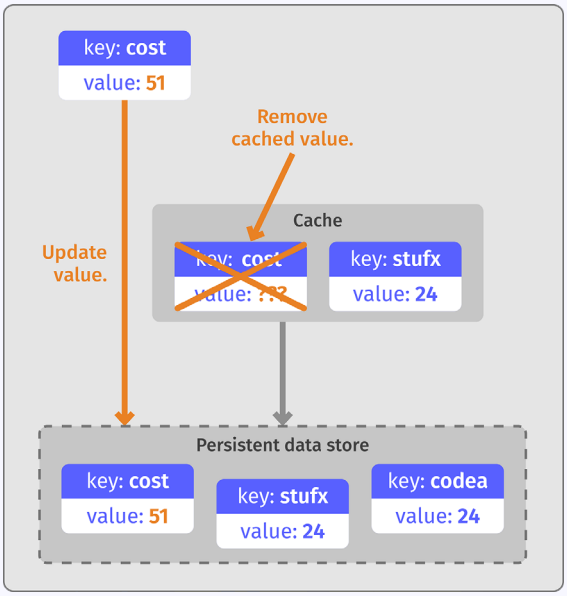

Figure 4-5. Invalidating cache on write

2. Basic Caching Strategies

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. The cache then updates the value in the data store, as shown in Figure 4-7.

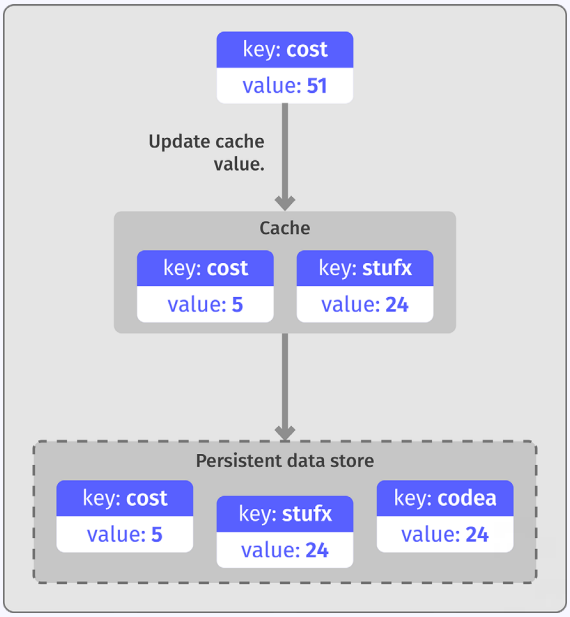

Figure 4-6. An attempt to update a cache value

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.

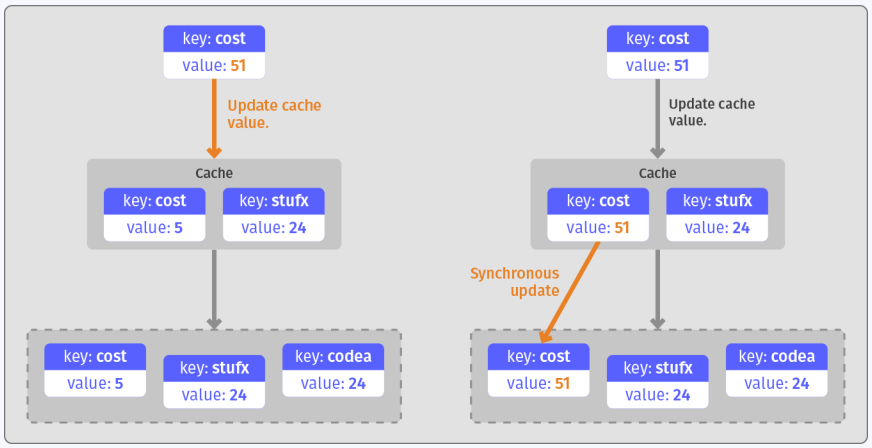

Figure 4-7. Write-through cache

2. Basic Caching Strategies

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, 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.

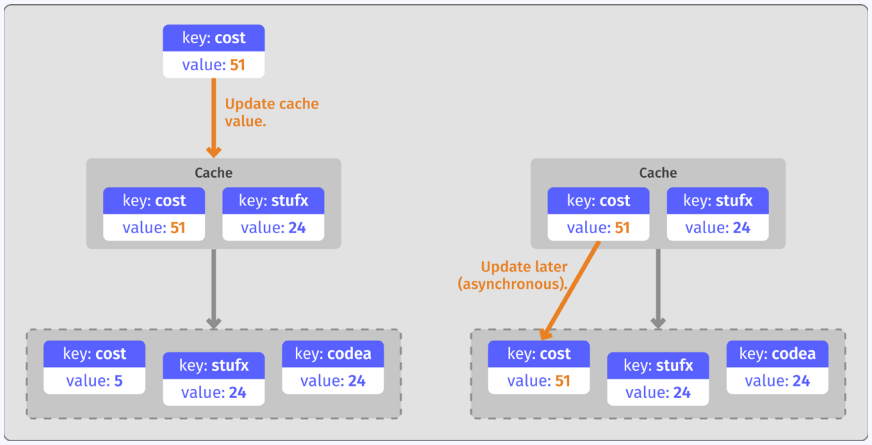

Figure 4-8. Write-behind cache

2. Basic Caching Strategies

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.

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.

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.

2. Basic Caching Strategies

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.

2. Basic Caching Strategies

Time-to-live (TTL) 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.

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

In a persistent cache, 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.

2. Basic Caching Strategies

Comparing eviction types

Table 4-1 compares the various eviction techniques.

Cache type	Variable used	What data is evicted first?
LRU	Last access time	This data hasn't been accessed for the longest period of time (longest period of time since last accessed).
LFU	Usage count	This data hasn't been used as often in the past (accessed the fewest number of times).
Oldest stored	Inserted time	This is data that was inserted the longest ago, and hence has been in the cache the longest period of time.
Random	*n/a*	Data is randomly deleted.
Permanent	*n/a*	Data is never deleted. New data values are simply not stored in a full cache until the cache is emptied.

Table 4-1. Cache eviction strategies

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. 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.

2. Basic Caching Strategies

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 are often "seeded" to avoid poor performance on startup and rehydration. In this case, 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.

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
```

2. Basic Caching Strategies

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.

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 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.

3. Advanced Caching Architectures and Patterns

"Caches are effective when they reduce redundant repeated requests that generate the same result that is equivalent and equal...repeatedly."

Enterprise caches are unique. They need to be flexible enough to enable seamless integration, agile enough to respond efficiently to sporadic surges in traffic and have the horsepower to simultaneously process millions of queries from different users around the world. Caches that can meet these requirements require unique architectures.

This chapter takes on these complexities by analyzing the different architectures and patterns that exist in enterprise caching. Atchison breaks everything down into granular detail and simplifies the intricacies behind advanced caching architectures.

As you'll soon discover, a volatile cache is a destructive cach,e and one that's susceptible to memory erasure. Deleted user data decimates the user experience and poses a threat to the effectiveness of any application.

In this chapter, Redis' architectural makeup is unraveled to reveal the unique components that are responsible for its ability to scale and provide an enterprise-proven cache that skyrockets application performance. Snippets of code are peppered throughout this section to demonstrate how different Redis commands shape its architecture to be that of an enterprise level.

Reading this chapter, you'll understand what components are required to achieve enterprise level-caching and how this contributes to creating an asset that's mandatory for elite application performance.

"Chapter 5: Advanced Caching Architectures and Patterns" from *Caching at Scale with Redis* by Lee Atchison

Caches have lots of capabilities, features, and use cases that go beyond simply storing key-value pairs. This chapter discusses some of these more advanced aspects of caching.

Cache persistence and rehydration

Cache persistence is the capability of storing a cache's contents in persistent storage, so that a power failure or other outage does not lose the contents of the cache.

3. Advanced Caching Architectures and Patterns

A **volatile cache** is a cache that exists in memory and is subject to erasure when a power outage or system restart occurs. In case of failure or other issue, the contents of a volatile cache cannot be assumed to be available—the system must always be functional, even if the cache itself is removed. The performance of the system may be impacted, but the capabilities and functionality cannot be affected.

With **cache persistence**, contents persist even during power outages and system reboots. An application might rely on an object being stored in the cache forever. A persistent cache may be chosen for performance reasons, as long as it is acceptable for an application to fail and not perform if a value is removed inappropriately.

Typically, a volatile cache is implemented in RAM, or some other high performance, non-permanent memory store. A persistent cache typically relies on a hard disk, SSD, or other long-term persistent storage.

It is possible to implement a persistent cache in volatile memory, as long as the cache mechanism makes appropriate redundant backups of the cached contents into persistent memory. In this situation, if the cache fails (such as via a power outage or system reboot) such that the volatile memory is wiped clean, then the redundant cache copy in the persistent memory is used to re-create the contents of the cache in the volatile memory, before the cache comes back online. This process is called cache rehydration.

A volatile cache can be rehydrated from either a persistent backup storage medium, or from the backing store that provides the reference copy of the required data, as shown in Figure 5-1.

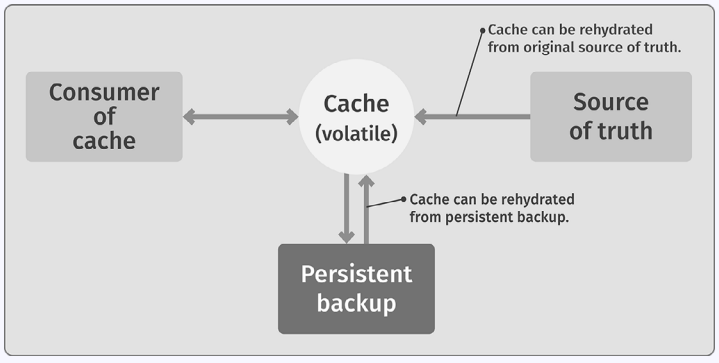

Figure 5-1. Cache rehydration

Redis can operate as either a volatile or persistent cache. It uses RAM for its primary memory storage, making it act like a volatile cache, yet permanent storage can be used to provide the persistent backup and rehydration, so that Redis can be used as a persistent cache.

3. Advanced Caching Architectures and Patterns

Redis persistence

Redis offers a range of persistent options, specifically:

1. Append-only files (AOF)
2. Point-in-time backups (RDB)
3. A combination of both

Together, these provide a variety of options for making data persistent as needed while maintaining the performance advantages of being RAM-based.

Append-only files (AOF)

Redis uses a file called the append-only file (AOF), in order to create a persistent backup of the primary volatile cache in persistent storage. The AOF file stores a real-time log of updates to the cache. This file is updated continuously, so it represents an accurate, persistent view of the state of the cache when the cache is shut down, depending on configuration and failure scenarios. When the cache is restarted and cleared, the commands recorded in the AOF log file can be replayed to re-create the state of the Redis cache at the time of the shutdown. The result? The cache, while implemented primarily in volatile memory, can be used as a reliable, persistent data cache.

The option `APPENDONLY yes` enables the AOF log file. All changes to the cache will result in an entry being written to this log file, but the log file itself isn't necessarily stored in persistent storage immediately. For performance reasons, you can delay the write to persistent storage for a period of time to improve overall system performance. This is controlled via `APPENDFSYNC`. The following options are available:

- `APPENDFSYNC no`: This allows the operating system to cache the log file and wait to persist it to permanent storage when it deems necessary.
- `APPENDFSYNC everysec`: This forces a write of the AOF log file to persistent storage once every second.
- `APPENDFSYNC always`: This forces the AOF file to be written to persistent storage immediately after every log entry is created.

To be completely safe and to guarantee that your cache operates as a persistent cache correctly, you should use the `APPENDFSYNC always` command, as this is the only way to guarantee that a system crash will not cause data loss. However, if your business can cope with some amount of cache loss during a system crash, then the `everysec` and `no` options can be used to improve performance.

3. Advanced Caching Architectures and Patterns

The result of the `APPENDONLY` command is a continuously growing log file. Redis can, in the background, rebuild the log file by removing no longer necessary entries in the log file. For example, if you:

1. Added an entry to the cache
2. Changed the entry's value
3. Changed the entry's value again
4. Deleted the entry

Then there would be four entries in the log file. Ultimately, all four of these entries are no longer needed to rehydrate the cache, since the entry is now deleted. The following command will clean up the log file:

```
BGREWRITEAOF
```

The result is a log file with the shortest sequence of commands needed to rehydrate the current state of the cache in memory. You can force this command to be executed automatically, which is the recommended best practice, rather than manually.

Point-in-time backups with Redis

Sometimes it is useful to create a backup copy of the current contents in the cache. This is what the RDB backup is for. This command creates a highly efficient, smallest possible, point-in-time backup file of the current contents of the cache. It can be executed at any point in time as follows:

```
SAVE
```

This command creates a dump.rdb file that contains a complete current snapshot of the Redis cache. Alternatively, you can issue the following command:

```
BGSAVE
```

This command returns immediately and creates a background job that creates the snapshot.

3. Advanced Caching Architectures and Patterns

Comparing RDB to AOF persistence

If your goal is to create a reliable, persistent cache that can survive process crashes, system crashes, and other system failures, then the only reliable way to do that is to use AOF persistence with APPENDFSYNC set to always. No other method guarantees that the entire state of the cache will be properly stored in persistent storage at all times. If your goal is to maintain a series of point-in-time backups for historical and system-recovery purposes (such as saving one backup per day for an entire month), then the RDB backup is the proper method to create these backups. This is because the RDB is a single file providing an accurate snapshot of the database at a given point in time. This snapshot is guaranteed to be consistent. However, RDB cannot be used to survive system failures, because any changes made to the system between RDB snapshots will be lost during a system failure.

So, depending on your requirements, both RDB and AOF can be used to solve your persistent needs. Used together, they can provide both a system-tolerant persistent cache, along with historical point-in-time snapshot backups.

Mixed RAM/SSD caching with Redis Enterprise

Open source Redis requires the entire cache, both the keys and the value of the keys, to be stored only in RAM. However, in Redis Enterprise, you can configure Redis to store the value of keys in either RAM or SSD flash memory. This allows significantly larger cache implementations. While caching is not its main use case, this feature, called Redis on Flash (RoF), is part of Redis Enterprise and can be useful in caching environments.

In RoF, all the data keys are still stored in RAM, but the value of those keys is intelligently stored in a mixture of RAM and SSD flash storage. The value is stored based on a least-recently used (LRU) eviction policy. More actively used values are stored in RAM and lesser used values are stored in SSD.

Given that SSD storage is significantly larger and less expensive than (if not quite as fast as) RAM, using RoF can allow you to build significantly larger caches more cost effectively.

Note that the use of persistent SSD flash memory does not automatically convert your cache into a persistent cache. This is because the keys are still stored in RAM, regardless of where your data values are stored, RAM or SSD. Therefore, using RoF with SSD storage does not remove the requirement of creating AOF and/or RDB backup files to create a true persistent cache.

3. Advanced Caching Architectures and Patterns

In-cache function execution

Redis allows you to execute arbitrary functions within the cache database itself. This is useful for a number of in-database application execution processes. Essentially, you can execute full-fledged Python scripts (with more languages coming) within the execution environment of a Redis instance.

As a simple example, imagine you have in a Redis database a few Hash maps that represent user-related information, such as first name, last name, and age. Then you can use the `RG.PYEXECUTE` command to execute a Python script to perform data cleanup on this information. Here is a sample script that deletes all users who are younger than 35 years old:

```
> RG.PYEXECUTE "GearsBuilder().filter(lambda x:
int(x['value']['age']) > 35).foreach(lambda x:
execute('del', x['key'])).run('user:*')"
```

The RedisGears module—a serverless engine for transaction, batch, and event-driven data processing—creates a powerful execution environment that allows you to build complex caching mechanisms. For instance, you could implement inline or aside caches talking to other backend databases from within RedisGears. It can be used to implement the write-through and write-behind caching patterns.

Microservices architectures

Redis has many uses in building microservices-based architectures. The most common use case is as an asynchronous communications channel. Redis can implement a high-speed queue for sending commands, responses, and other data asynchronously between neighboring services. This is shown in Figure 5-2, where Service A is set up to send messages to Service B using a Redis Lists object within a Redis instance. This use case is described in the Redis Lists data type later in this chapter.

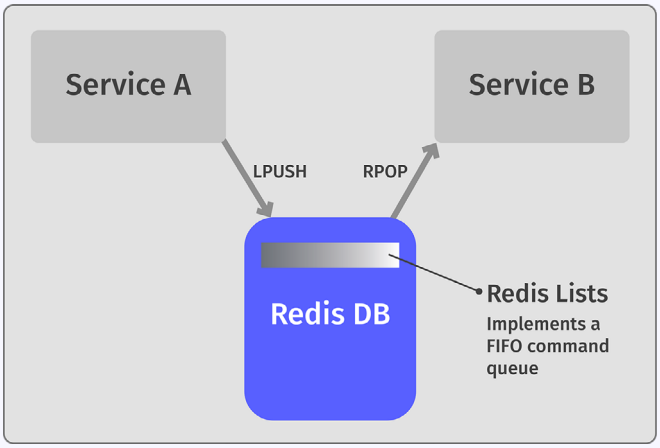

Figure 5-2. Redis List as a FIFO command queue for microservices

3. Advanced Caching Architectures and Patterns

Microservices can also take advantage of Redis as a classic cache. This can be as an internal, server-side cache storing interim data used internally by a service. More specifically, a Redis instance can be used as a cache-aside cache fronting a slower data store, as shown in the Redis data cache example in Figure 5-3. Cache-aside caches are described in more detail in Chapter 4, "Basic Caching Strategies."

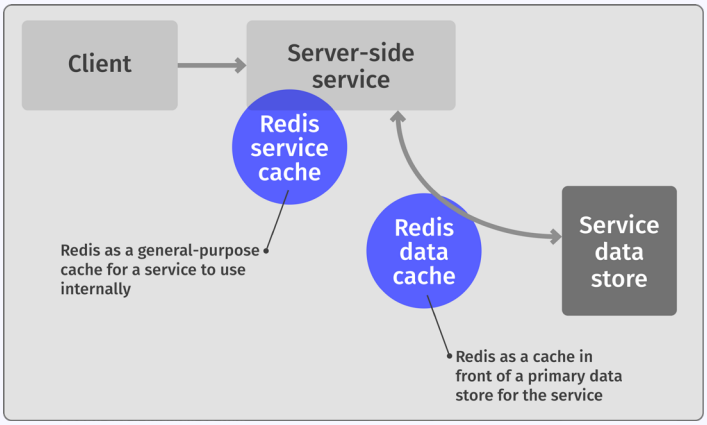

Figure 5-3. Redis as a distributed cache

On the client side, Redis can be used to cache interim results, fronting calls to backend services, and reducing the need to call backend services. This is shown in Figure 5-4.

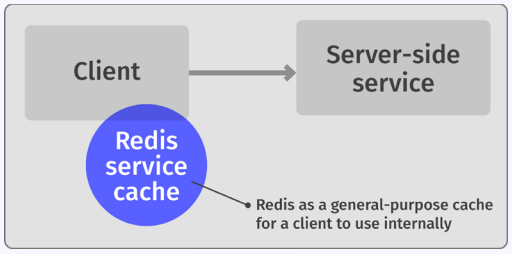

Figure 5-4. Caching interim result

Finally, a cache can be inserted between two services and used at the network level, providing HTTP-level caching capabilities.

Caching at the HTTP level is managed via HTTP headers, such as `Cache-Control`, `Expires`, and `Last-Modified`.

These are typically processed by intermediaries such as reverse proxies (e.g., Nginx). This is shown in Figure 5-5.

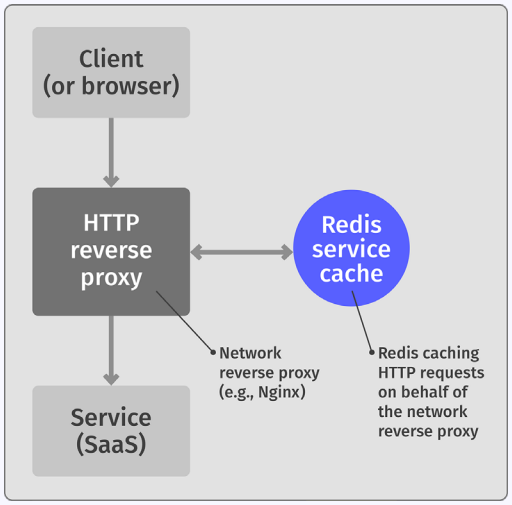

Figure 5-5. Network protocol layer cache

Cache search

RediSearch, a module available in Redis Enterprise, is a source-available secondary index, query, and full-text search engine over Redis. It provides direct, search engine–like capabilities in a Redis instance. While it is possible to implement a search engine using a standard Redis instance, the RediSearch module simplifies much of the complexity in creating a search engine. More importantly, RediSearch lets you perform SQL-like queries on a Redis database, benefiting from the improved performance on secondary indexes.

3. Advanced Caching Architectures and Patterns

The RediSearch module allows you to create an index of keys that contain Hash data types. The index represents the attributes that you plan to query within all Redis keys that are included in the index. Once the index is created, search terms can be applied against the index to determine which Hash keys contains data that match the search terms. This is illustrated in Figure 5-6.

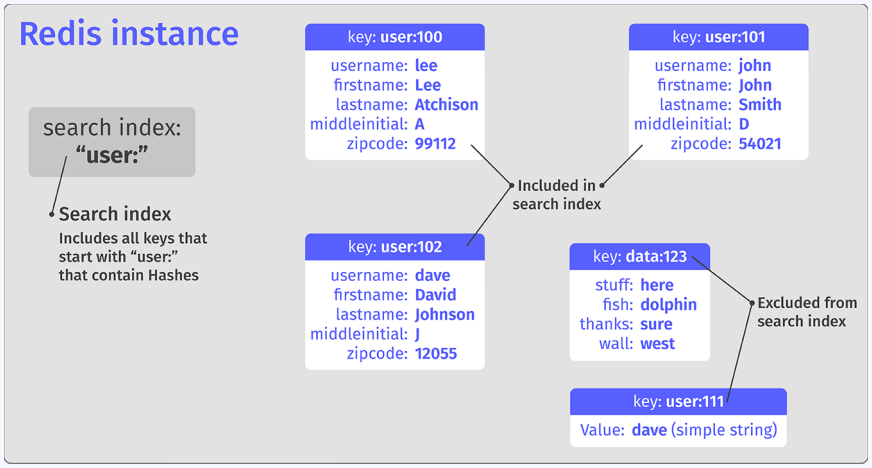

Figure 5-6. Redis instance with RediSearch search index

In code, let's assume you have the following hash keys setup. These are straight from Figure 5-6:

```
redis> HSET user:100 username lee firstname Lee lastname Atchison
middleinitial A zipcode 99112
redis> HSET user:101 username john firstname John lastname Smith
middleinitial D zipcode 54021
redis> HSET user:102 username dave firstname David lastname Johnson
middleinitial J zipcode 12055
```

Next, create a search index:

```
redis> FT.CREATE user_idx PREFIX 1 "user:" SCHEMA username TEXT
firstname TEXT lastname TEXT middleinitial TEXT zipcode
```

This creates a search index using all Hash entries with a key that starts with "user:".

Now, you can execute a search on this index:

```
redis> FT.SEARCH user_idx "john" LIMIT 0 10 ...returns data from key
"user:101" and "user:102"...
```

3. Advanced Caching Architectures and Patterns

Enhanced data handling

A typical cache uses a key-value system. A key represents the identifier for the value stored in the cache. But what is the data type of the value? In a typical cache, this is usually either just a numeric value or a string representation of some other data type. In Redis, the basic datatype is a String. However, Redis supports many other data types in the value field.

Lists

A single Redis key can have a value that represents a list of strings. There are specialized commands for manipulating this list that allow the list to serve a variety of purposes. For example, you can implement a simple first-in-first-out (FIFO) queue using the `LPUSH` (left push) and `RPOP` (right pop) commands. These commands will add (push) a string to the left side of the list and remove (pop) the string off the right side of the list. This allows strings to be sent in a simple FIFO model. For example:

```
redis> LPUSH thelist "AAA"
1
redis> LPUSH thelist "BBB"
2
redis> LPUSH thelist "CCC"
3
redis> LRANGE thelist 0 -1
1) "CCC"
2) "BBB"
3) "AAA"
```

The three `LPUSH` commands pushed three elements on the list. The `LRANGE` command prints the contents of the list from left to right. So, after executing the three `LPUSH` commands above, the list contains the values ["CCC", "BBB", "AAA"], in that order.

3. Advanced Caching Architectures and Patterns

You can then use the RPOP command to pull elements off the list. Assuming you have the above list, then the following commands will return the following results:

```
redis> LPUSH thelist "AAA"
1
redis> LPUSH thelist "BBB"
2
redis> LPUSH thelist "CCC"
3
redis> LRANGE thelist 0 -1
1) "CCC"
2) "BBB"
3) "AAA"
...

redis> RPOP thelist
"AAA"
redis> LRANGE thelist 0 -1
1) "CCC"
2) "BBB"
...

redis> RPOP thelist
"BBB"
redis> LRANGE thelist 0 -1
1) "CCC"
...

redis> RPOP thelist
"CCC"
redis> LRANGE thelist 0 -1
nil
```

3. Advanced Caching Architectures and Patterns

Used together, the `LPUSH` and `RPOP` commands implement a FIFO queue:

```
redis> LPUSH thelist "AAA"
1
redis> LPUSH thelist "BBB"
2
redis> LPUSH thelist "CCC"
3
redis> RPOP thelist
"AAA"
redis> RPOP thelist
"BBB"
redis> RPOP thelist
"CCC"
redis> RPOP thelist
nil
```

The Lists data type is most commonly used for queues and scheduling purposes, and it can be used for these purposes in some cache scenarios. Some caches utilize queues, such as Redis Lists, to order or prioritize data that is stored in the cache.

Sets

A single Redis key can contain a set of strings. A Redis Set is an unordered list of strings. Unlike the Redis Lists data type, the Redis Sets data type does not dictate an order of insertion or removal. Additionally, in Redis Sets, a given data value (a given string) must be unique. So, if you try to insert the same string value twice into the same set, the value is only inserted once.

3. Advanced Caching Architectures and Patterns

Sets are very useful for operations such as determining existence, and as such are quite useful in caches. The primary insertion command is the SADD command. The primary removal is the SREM command, and the primary query command is the SISMEMBER command. The entire set can be examined with the SMEMBERS command. The following is an example of how they work together:

```
redis> SADD theset "AAA"
1
redis> SADD theset "BBB"
1
redis> SMEMBERS theset
1) "AAA"
2) "BBB"
redis> SADD theset "CCC"
1
redis> SADD theset "BBB"
0
redis> SMEMBERS theset
1) "AAA"
2) "BBB"
3) "CCC"
redis> SISMEMBER theset "AAA"
1
redis> SISMEMBER theset "BBB"
1
redis> SISMEMBER theset "DDD"
0
redis> SREM theset "BBB"
1
redis> SMEMBERS theset
1) "AAA"
2) "CCC"
```

3. Advanced Caching Architectures and Patterns

In a cache application, sets can be used to test for repeated operations. For example, you can use a set to determine if a given command has been requested from the same IP address recently:

```
redis> SADD command1:ipaddr "10.3.1.12"
redis> SADD command1:ipaddr "10.21.23.43"
redis> SADD command1:ipaddr "22.101.15.31"
redis> SADD command1:ipaddr "10.3.1.12"
redis> SMEMBERS command1:ipaddr
1) "10.3.1.12"
2) "10.21.23.43"
3) "22.101.15.31"
redis> SMEMBERS command1:ipaddr "10.21.23.43" 1
redis> SMEMBERS command1:ipaddr "15.3.2.11" 0
```

Hashes

A single Redis key itself can represent a set of key-value pairs in the form of a Hash. A Hash allows for the creation of custom structured data types and stores them in a single Redis key entry.

The classic use case for a Redis Hash is to store properties for an object, such as a user:

```
redis> HSET user:100 username lee
redis> HSET user:100 password 123456
redis> HSET user:100 first Lee last Atchison middleinitial A
redis hmgetall user:100
1) "username"
2) "lee"
3) "password"
4) "123456"
5) "first"
6) "Lee"
7) "last"
8) "Atchison"
9) "middleinitial"
10) "A"
```

3. Advanced Caching Architectures and Patterns

You can get individual properties from a single Redis entry, using the same data as before:

```
redis> hmget user:100 username
1) "lee"
redis> hmget user:100 first last
1) "Lee"
2) "Atchison"
```

You can also change individual properties. Again, using the same data as before:

```
redis hmgetall user:100
1) "username"
2) "lee"
3) "password"
4) "123456"
5) "first"
6) "Lee"
7) "last"
8) "Atchison"
9) "middleinitial"
10) "A"
redis> HSET user:100 password "456789"
redis hmgetall user:100
1) "username"
2) "lee"
3) "password"
4) "456789"
5) "first"
6) "Lee"
7) "last"
8) "Atchison"
9) "middleinitial"
10) "A"
```

In a cache, you can use Hashes to store more complex properties-based data.

3. Advanced Caching Architectures and Patterns

RedisJSON

Redis can also store arbitrary JSON data into a single Redis key. This is accomplished using a Redis module called RedisJSON that lets you store virtually any type of data, including custom data types, in Redis in a format that can be easily reconstituted into usable data.

Here is a sample creation and manipulation of a JSON document within a single Redis key named "testkey":

```
redis> JSON.SET testkey . '[ 123, { "life": 42 },{"fish","please"} ]'
OK
redis> JSON.GET testkey
"[123,{\"life\":42},{\"fish\",\"please\"}]"
```

Once created, you can manipulate individual elements of a JSON document:

```
redis> redis> JSON.SET testkey . '[ 123, { "life": 42
},{"fish","please"} ]'
OK
redis> JSON.GET testkey [1].life
"42"
redis> JSON.GET testkey [1].fish
"please"
```

Database vs. caching use cases

There are many more data types, including Sorted Sets, Streams, Lists, Strings, and more. Each data type has a significant number of commands and options for how to use them. For more information on Redis data types, take a look at the online documentation:

https://redis.io/topics/data-types-intro

Many of these data types will be more useful in traditional database use cases, rather than traditional cache use cases. One of the benefits of Redis is that it works great as a cache, but you can also use it as a primary NoSQL database. Redis' versatility is one of its greatest strengths.

4. Making the Transition to Enterprise Caching

Caching at scale with optimized performance requires an enterprise-grade cache. Having one at the core of your data strategy will take digital experiences levels to a whole new level.

Your cache needs to meet the criteria below for it to be considered enterprise-grade:

Availability

Low availability leads to catastrophic results. If your cache isn't available when needed, your most important and frequently accessed data will be slowed down. Your main database could also be swamped with a vast amount of requests to process and will be working on overdrive to try and keep up with the user demands.

This degrades performance by allowing lag to create friction between the user and an app. Additionally, SLAs may not be met, which will have a huge impact on your business. Users expect the same seamless experience every time they use an application, making it absolutely mandatory that a cache and the data it provides are always available.

Performance

Users expect high throughput and sub-millisecond latency. They won't tolerate anything less. Not only do you have to meet these expectations, you also need to hit the bullseye with every other service across the board. Elite application performance hinges on a cache's ability to constantly produce data in real-time.

Scalability

A high performance caching layer should scale globally with ease and have the agility to respond to intermittent spikes in traffic with hyper-efficiency. Sudden surges may be caused by events such as Valentine's Day and Black Friday, but such events are predictable.

Caches operating on an enterprise level need to be able to respond efficiently to unexpected shifts in user behavior that arise from both unforeseen events: natural disasters or pandemics for example and more predictable ones like the steady growth of an app's user base. Scaling should be easy and done without impacting performance or requiring downtime or migrations.

Geo-distribution

Modern expectations require applications to be available across multiple regions 24/7. This is non-negotiable, but is complicated by the fact that your customers may not be stationary. People are always on the move and the world is more globalized than it's ever been before, meaning that your customers could be banking whilst shopping or even playing an online game whilst crossing countries. An enterprise-grade cache requires the same low local latency regardless of user location, providing a seamless real-time global experience.

4. Making the Transition to Enterprise Caching

Do I Need Enterprise Caching?

At this point you may be wondering whether or not you need enterprise caching. It's an important question that will determine the success of your application. Many developers find out the hard way by using localized caching to support global apps, only to experience a complete failure in meeting customer expectations due to the drastic inadequacies of data infrastructure.

Once this happens it can be even more difficult to recover and get a foothold in the market again. It's important to remember that caches are no longer luxury items. Times have changed and the digital marketplace is faster, more competitive, and more cutthroat: the smallest of edges can bring out the biggest results.

As a result, enterprise caching is mission-critical because if you're not leveraging one, your competitors will be. And of course, if you're unable to provide real-time responsiveness without a cache, then caching automatically becomes indispensable.

There are several questions you should ask yourself if you're unsure about enterprise caching:

- Can your database handle the extra workload that will be added when your cache is not there to run interference?
- Can your network handle unexpected surges in traffic and still provide a high throughput?
- Does your database have the capacity to handle the added pressure of rehydrating the cache once it recovers?
- Does your cache need to be replicated in multiple regions to ensure availability?

If you've answered no to any of these questions, then the prospect of enterprise caching is one that you should consider.

4. Making the Transition to Enterprise Caching

Criteria for Evaluating Enterprise Cache

Optimizing cache performance at enterprise scale is no small feat. Enterprise-grade caches alleviate much of the risk and operational burden of performing at scale. But choosing the best enterprise-grade cache for your application can be tricky. It requires evaluating different caching services and their capabilities, features, and components. How do you find the right solution for your applications?

Below is a checklist you can use to discover the most optimal enterprise-ready caches available on the market:

- Provides consistent high throughput and sub-millisecond latency
- Highly available (offers 5-9s availability)
- Geographically distributed and available across time zones with local low-latency
- Has automatic failover and failback
- Linearly scales to optimize infrastructure resources
- Scales with no performance degradation
- Backup with no performance degradation
- Provide intelligent tiering to manage large data sets economically
- Multi-tenancy to enable efficient infrastructure utilization
- Run on-premises, in a hybrid environment, and multiple clouds without data transfer issues
- Enterprise-grade support
- Offers the innovation of an open source foundation and community

4. Making the Transition to Enterprise Caching

Basic Caching vs. Enterprise Caching

Below is a chart that will help you differentiate between attributes of a basic and an enterprise-grade cache. As you'll discover, there's a range of features you need to take into consideration when looking for an enterprise-proven cache.

	Basic Caching	Enterprise-Grade Caching
High throughput	✕	✓
Low latency	✓	✓
Cloud DBaaS	✓	✓
Hybrid deployment	✕	✓
Multi-cloud deployment	✕	✓
5-9s' High availability	✕	✓
Geo-distribution	✕	✓
Guaranteed local low-latency	✕	✓
Data consistency with no performance degradation	✕	✓
Linear scaling	✓	✓
Infinite scaling	✕	✓
Basic clustering	✓	✓
Advanced clustering	✕	✓
Intelligent tiering	✕*	✓
Multi-tenancy	✕	✓
Geo-distribution	✕	✓
Eventual consistency	✕	✓
RBAC support	✕	✓
Support for Kubernetes as a native service (AKS, GKE, EKS, OpenShift)	✕	✓
Upgrade with no downtime	✕	✓
Auto failure detection	✕	✓
Enterprise-grade support	✕	✓

** Partial for AWS*

4. Making the Transition to Enterprise Caching

Real-time Enterprise Caching with Redis

Any app requires real-time data to stay afloat. Caches must be able to transmit, process, and retrieve data within 100ms for it to be deemed to be real time. But ramping this up to a global level requires an enterprise-proven cache that meets a long checklist of specific criteria.

As a result, companies are turning to Redis Enterprise to maximize application performance, keep users engaged, and boost revenue. Redis Enterprise is the world's favorite database for a reason: it's powerful, it's consistent, and it can be run anywhere.

Here's what you can also expect with Redis Enterprise:

- **High availability:** Redis Enterprise provides 5-9s SLAs across multiple geographies or clouds. This includes a number of features, including backups and automated cluster recovery—both of which are crucial for business critical apps.
- **Global Distribution:** Active-Active Geo Replication enables globally distributed applications that guarantee sub-millisecond local latency across the globe with data consistency.
- **High performance and auto-scaling:** Redis Enterprise easily handles usage spikes with automatic dynamic scaling that works behind the scenes to perform consistently and without error.
- **Storage cost:** Redis on Flash provides intelligent tiering, keeping the most frequently used data in lightning-fast RAM, while storing less-frequently accessed data in less-expensive SSD.This tiering provides cost-efficiency for r large datasets while maintaining application performance.
- **Multi-cloud & hybrid:** The flexibility that Redis Enterprise provides allows you to choose between a range of different deployment options clouds to ensure that you find the right fit for your business and applications.

Final thoughts

Up to now we've explored Lee Atchison's excerpts on enterprise caching, highlighting the most important caching concepts and exploring how to achieve them on an enterprise level.

If there's one thing to remember it's this: Now that demand for real-time experiences is at an all time high, caching has never been more crucial to enterprise applications.

A few short years ago, a cache was a "nice-to-have" now it is a mission-critical requirement. Very few databases now have the horsepower, agility, and consistency to process mountains of data on a global level at any given time, and very few users have the patience to endure flawed or slow digital experiences. As a result, enterprise caching has evolved to become a mandatory performance enhancer.

But as we've highlighted, the technical requirements to provide enterprise-grade caching are complex. And it's not easy to identify which one is the right fit for you.

So what's next?

From the above you'll have gained an insight into the advanced caching capabilities of Redis and its capacity to provide businesses with elite caching.

If you want to understand more about Redis Enterprise and how it can transform your company's digital experiences with enterprise caching, then make sure to download *The Buyer's Guide to Enterprise Caching*.

You can also discover more about Redis Enterprise at https://redis.com/.

About Lee Atchison

Lee Atchison is an established thought leader in cloud computing and application. Having dedicated over thirty years to working in product development, architecting, modernizing, and scaling, Lee has worked for some of the world's biggest heavyweights, including Amazon and Amazon Web Services.

To discover more about Lee and some of his other recent books, visit his website at leeatchison.com and make sure to follow him on Twitter.

Atchison is widely referenced in many publications and is invited to share his insights at public speaking events all over the world.

About Redis

Data is the lifeline of every business, and Redis helps organizations reimagine how fast they can process, analyze, make predictions, and take action on the data they generate. Redis provides a competitive edge to any business by delivering open source and enterprise-grade data platforms to power applications that drive real-time experiences at any scale.

Developers rely on Redis to build performance, scalability, reliability, and security into their applications. Born in the cloud-native era, Redis uniquely enables users to unify data across multi-cloud, hybrid, and global applications to maximize business potential. Learn how Redis can give you this edge at redis.com.