



Caching at Scale With Redis

Cloud Caching Techniques
for Enterprise Applications



Lee Atchison

To Beth

My life, my soulmate, my love.

Foreword

By Mike Anand, Chief Marketing Officer, Redis

Nobody likes to wait. That central truth is the impetus behind the technology industry's ongoing efforts to speed up... well, just about everything. And it's the driving force behind the popularity of caching, putting a layer of fast, temporary data storage—holding the most frequently accessed data—in front of the slower, original data store. That speeds responses to users and lets software applications operate at greater scale. Caching is now a mainstream technique to maximize app performance, used just about everywhere speed matters.

But not all caching is created equal.

Redis—an open source, in-memory data structure store commonly used as a database, cache, and message broker—is by far the world's most popular software caching solution. That's no secret. But here at Redis Labs, the home of open source Redis and Redis Enterprise—which provides an enterprise-grade caching platform for globally distributed applications—we felt the need to dig a little deeper into that story and redefine caching from an enterprise perspective.

We wanted to help developers—as well as software architects and other high-level technical decision makers—understand the underpinnings of how caching works. We wanted to go beyond the technical specs to demonstrate when and how caching can provide the most benefit in the enterprise. And we wanted to share best practices on how to get the most out of open source Redis and Redis Enterprise when building caches in real-world enterprise applications.

To tell that story in enough depth to be truly useful, though, we knew a blog post or white paper wasn't going to be nearly enough... we needed a book. A real, full-length book, with chapters on all the key topics and enough room to flesh out all the important details and put everything in perspective.

That's what we wanted this book to be.

But when we sat down to make the book a reality, we quickly realized that we needed an expert, independent voice to be the author. Someone with the expertise to tell the whole story. Someone with the technical credibility to be taken seriously by serious developers. Someone who could connect the technical details to the business impacts. Someone with the wide-ranging background needed to see the larger picture and tell the story from a vendor-neutral, third-party perspective. And someone with the independence to always keep the needs of enterprise readers in mind. Oh, and we needed someone who could explain it all in plain English and straightforward visuals, so it would make sense to as many folks as possible—without dumbing things down and while still bringing real value and insight to experts. A tall order, to be sure.

Lee Atchison was obviously the right person for the job.

Lee spent seven years at Amazon Web Services, where among many other accomplishments, he was the senior manager responsible for creating the AWS Elastic Beanstalk Platform-as-a-Service (PaaS). In his eight years as a Principal Cloud Architect and Senior Director of Strategic Architecture at New Relic, he was instrumental in crafting the monitoring company's cloud product strategy.

That's not all. Lee is a widely quoted thought leader in publications including *diginomica*, *IT Brief*, *ProgrammableWeb*, *CIORReview*, and *DZone*. He has jetted around the world, giving eagerly received technical talks everywhere from London to Sydney, Tokyo to Paris, and all over North America. He's been a regular guest on technology podcasts, articulating deep technical insights with humor and good cheer—and he now hosts his own “Modern Digital Applications” podcast. And perhaps most importantly, he's an accomplished author: his well-received and recently updated book, *Architecting for Scale: High Availability for Your Growing Applications*, attracts long lines of developers queuing up at conferences and trade shows for the chance to receive a signed copy.

That's the story of how this book came to be, and why we think it's so uniquely valuable. We hope you find it useful and informative.

Acknowledgements

I'd like to thank Redis for its help creating this book. Specifically, I would like to thank Fredric Paul. “The Freditor,” as we love to call him, was instrumental in getting this book published, along with many of my past publications. Fred, despite being dual-first-named, you are a great friend, and I love working with you.

I would also like to thank Alec Wagner, who was the editor for this book. I'm sure I will hear the phrases “passive voice vs. active voice” and “sentence case vs. upper case” in my sleep for years to come.

Additionally, thanks to Redis Technical Marketing Managers Tugdual (Tug) Grall and Ajeet Raina for lending their invaluable technical expertise in reviewing the manuscript. And many thanks to Mike Anand, Chief Marketing Officer at Redis, who greenlit the project.

And of course, a big thank you to my lovely wife, Beth, who is always there for me and supports me and my career with such warmth and love. Finally, a shout out to my fur family: Issie, who contributes background noises (snoring) to my conference calls; Abbey, who adds smiles and wags; and Buddha, whose love for keyboard walking is responsible for any typos you may encounter here.

Table of Contents

Chapter 1. Introduction	1
Chapter 2. What Is Caching?	3
Chapter 3. Why Caching?	11
Chapter 4. Basic Caching Strategies	19
Chapter 5. Advanced Caching Architectures and Patterns	37
Chapter 6. Cache Scaling	55
Chapter 7. Cache Consistency	65
Chapter 8. Caching and the Cloud	71
Chapter 9. Cache Performance	79
Glossary. Cache Terminology	89
Epilogue. About the Author, About Redis	95

1.

Introduction

Our modern world demands modern applications. Today's applications must be able to handle large quantities of data, perform complex operations, maintain numerous relationships among data elements, and operate on distinct and disparate states between transactions.

Doing this at the high scale demanded by our modern world is a challenge that requires significant resources—and those demands are constantly growing based on ever-changing needs. Handling these needs while maintaining high availability is simply the cost of entry for any modern application.

While there are many methods, processes, and techniques for making an application retain high availability at scale, caching is a central technique in almost all of them. Effective caching is the hallmark of an effectively scalable application.

This book describes what caching is, why it is a cornerstone of effective large-scale modern applications, and how Redis can help you meet these demanding caching needs.

It then talks about the different types of caching strategies, and what architectural patterns can be implemented with Redis. We then will talk about cache scaling, and cache consistency, and how caching can be utilized in cloud environments. We end with a discussion about measuring cache performance, and conclude with a glossary of important cache terminology.

The audience for this book includes senior-level software engineers and software architects who may already be familiar with Redis and are interested in expanding their uses of Redis. They are building and operating highly scaled, complex applications with large and often unwieldy datasets, and they are interested in how Redis can be used as their primary caching engine for managing data performance.

You will learn how caching, and hence Redis, fits into your complex application architecture and how platforms such as Redis Enterprise can help solve your problems.

You will acquire knowledge needed to discuss with your CTO/CIO/CFO about how Redis and Redis Enterprise can help solve your organization's data scaling problems.

2.

What Is Caching?

*“A cache is a place to hide things...
no, wait—that’s the wrong type
of cache.”*

In software engineering, a cache is a data-storage component that can be accessed faster or more efficiently than the original source of the data. It is a temporary holding location for information so that repeated access to the same information can be acquired faster and with fewer resources. Figure 2-1 illustrates a consumer requesting data from a service, and the data then being returned in response.

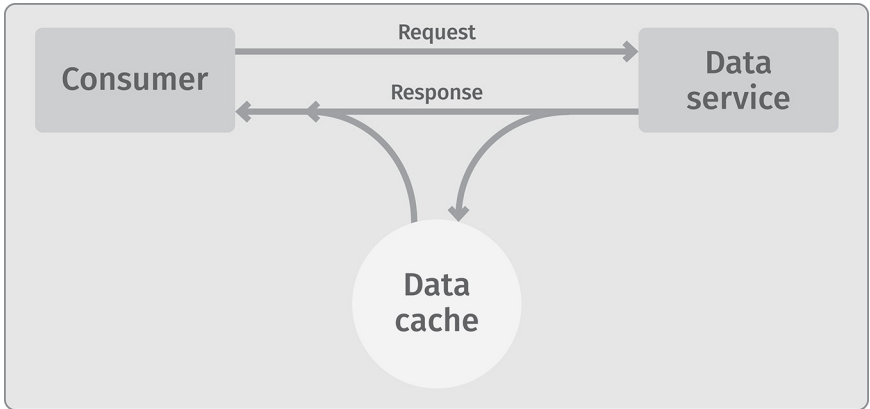


Figure 2-1. A simple cache

But what happens if the data service has to perform some complex operation in order to acquire the data? This complex operation can be resource intensive, time intensive, or both. If the service has to perform the complex operation every time a consumer requests the data, a significant amount of time and/or resources can be spent retrieving the same data over and over again.

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

Caching use cases

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

HTTP application cache: When you access a web page, depending on the website, the page may require a fair amount of processing in order for the page to be created, sent back to you, and rendered in your browser. However, the page—or significant portions of the page—may not change much from one request to the next. A cache is used to store pages and/or portions of pages so they can be returned to a user faster and using fewer resources than if the cache was not used. This results in a more responsive website, and the capability for the website to handle significantly more simultaneous users with a given number of computational resources. An HTTP application cache is illustrated in section 1 of Figure 2-2.

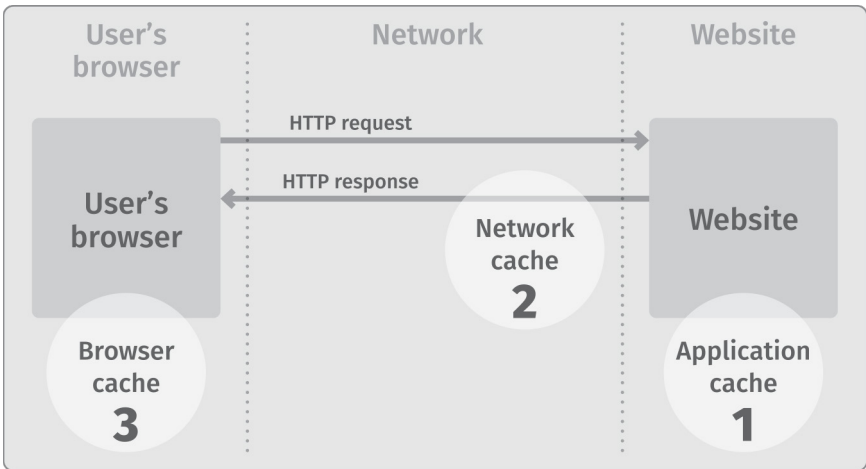


Figure 2-2. Web caching at various locations in a network

HTTP network cache: When a web page is accessed by a user, the site generating the page may be far away, sometimes in another part of the country or another country altogether. It can take a relatively long time for the page to be sent from the website server to your browser before it displays. HTTP protocols provide the capability to store copies of certain static web pages or page elements at intermediate locations along the route between the browser and the website server. Because these caches allow content to be stored closer to the browser, the website can be retrieved faster and the results displayed quicker. This is often the case for static content such as pictures, images, and videos. In Figure 2-2, section 2 is an HTTP network cache.

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

Service cache: Individual services can have internal caches to assist in performing complex and time-consuming operations. These are called service caches. A service cache is very similar to an application cache, but operates and caches in support of the individual service components that make up an application.

Application programming interface (API) cache: When one service calls another using an API, the response to the API call can be stored in a cache and used to return results for equivalent future calls.

Database cache: When a database receives a request to retrieve data, the processing required may be quite extensive. Database caches can be used in several locations within databases in order to speed up queries as well as to conserve compute resources, which aids in database scaling. These caches are typically used to store query results, intermediate results, query parsing results, and query plans. Figure 2-3 illustrates some of the locations where databases can cache the results of a query.

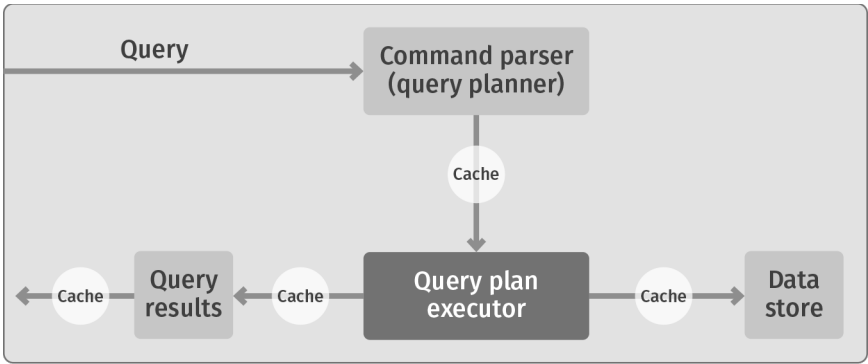


Figure 2-3. Database cache

CPU cache: There are many caches within computers, including in the central processing unit (CPU). The core CPU caches executed commands so that repeated execution of the same or similar instructions can occur considerably faster. In fact, much of the increased speed of computers in recent years is due to improvements in how commands are executed and cached, rather than actual clock speed improvements. Figure 2-4 illustrates this type of cache.

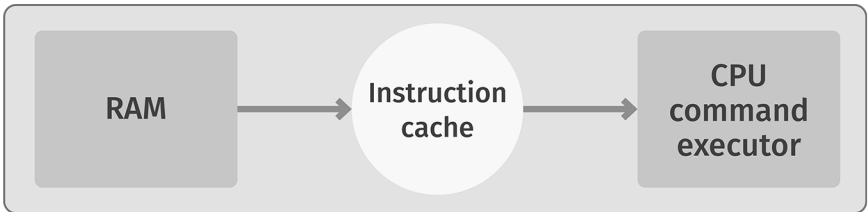


Figure 2-4. CPU instruction cache

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

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

What does a cache need to be useful?

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

1. The operation necessary to calculate or retrieve the requested data must either be slow or require resources to acquire/calculate.
2. The cache must be able to store and properly retrieve the result faster using fewer resources than did the original source.
3. Sometimes, the data involved must be unchanged from one request to another. While it is possible to cache dynamically changing data in certain circumstances, data that doesn't change from request to request is easier to use in caching situations.
4. The operation to calculate or retrieve the requested data must have no side effects (that is, it doesn't store data, it doesn't make changes to other systems, and it doesn't control other software or hardware) on the system operating, other than the consumption of resources. More on this in a moment.
5. The data must be needed more than once. The more times it is needed, the more effective and useful the cache is.

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

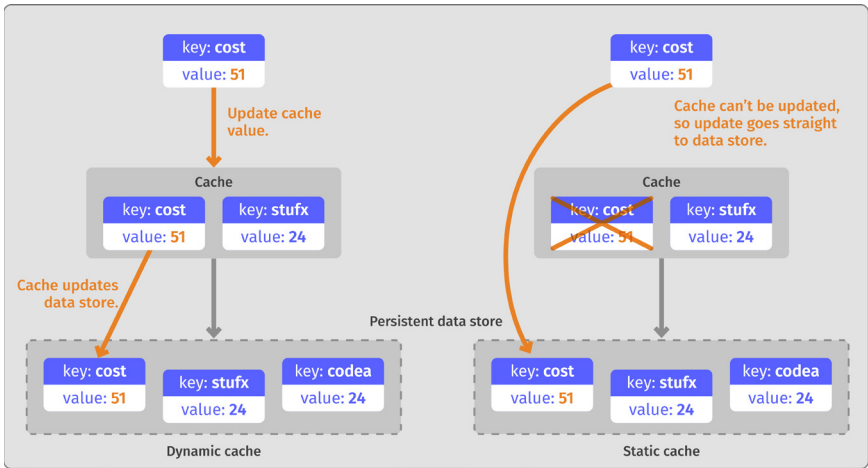


Figure 2-5. Static vs. dynamic cache

Static vs. dynamic caches

Not all caches are accessed in the same manner. While caches are used and accessed in many different ways, there are two fundamental types of patterns for how caches are used. They are **static**, sometimes called **read/only**, caches and **dynamic**, sometimes called **read/write**, caches. The difference is how the caches are populated when data in the underlying data store is changed, and is illustrated in Figure 2-5.

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

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

3.

Why Caching?

“Caching allows you to skip doing important things, and yet still benefit from the results... sometimes.”

Consider a simple service that multiplies two numbers. A call to that service might look like this:

```
MUL 6 7
    result: 42
```

This service could be called repeatedly from multiple sources with many different multiplication requests:

```
MUL 6 7
    result: 42
MUL 3 4
    result: 12
MUL 373 2389
    result: 891,097
MUL 1839 2383
    result: 4,382,337
MUL 16 12
    result: 192
MUL 3 4
    result: 12
```

You see it can take on an entire series of multiplication requests, process them all independently, and return the results.

But did you notice that the last request is the same as the second request? The same multiplication call has been requested a second time. This poor little service, however, doesn't know that it's already been asked to perform the request, and it goes about all the work it needs to do to calculate the result ... again.

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

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

1. Consult the cache, to see if there is an entry representing “3x4”.
2. If there is an entry, return the result from the cache. STOP.
3. If there is not an entry, call the service and get the result of 3 times 4.
4. Store the result of the service call into the cache under the entry “3x4”.
5. Return the result.

What happens next? Well, the first time anyone wants to get the result of 3 times 4, the request has to consult the service directly, because the value has not previously been placed in the cache. However, after the value is calculated, the result is placed in the cache. The cache now stores the operation (3 times 4, written as 3x4) as a key to the result of the operation (12).

The next time someone wants to get the result of 3 times 4, the request sees that the result is already stored in the cache, and it simply returns the stored result. The service itself doesn't need to perform the calculation, saving time and resources.

This cache-aside process is illustrated in Figure 3-1. In the top of this figure (section 1), you can see a consumer making a request to the multiplication service without a cache. Every time the consumer needs to figure out a result, it must ask the multiplication service to process the result.

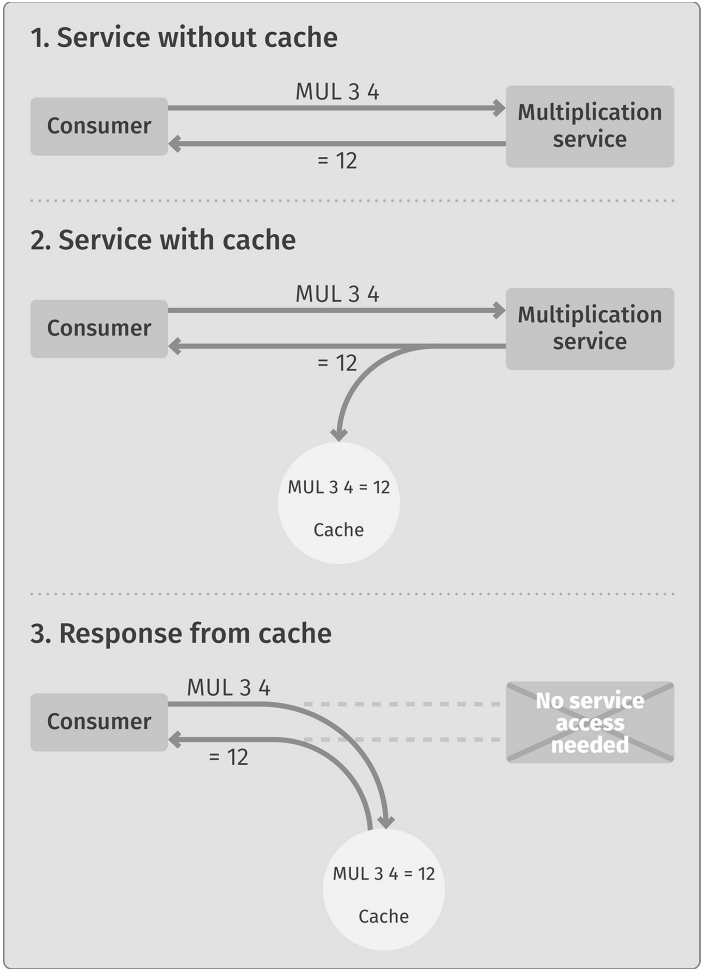


Figure 3-1. Multiplication service with cache

The middle of the figure (section 2) shows that a cache has been hydrated. Now, whenever a service call is made that calculates a result, the request and the corresponding result are stored in the cache. That way, when a repeat request comes in, as shown in the bottom of the figure (section 3), the request can be satisfied by the cache without ever calling the service at all.

OK, what problems does caching solve?

As we discussed in Chapter 2, “What Is Caching?”, there are many use cases for caching, and many types of use cases that can be solved by caching. Here are some typical problems that caching can either solve or help to solve in a modern application:

- **Performance improvement.** Caching improves latency. Latency is the new outage, and if you can avoid having to do a time-intensive calculation by simply using a cached result, you can reduce the latency for all requests that utilize the cache. Over time, this can have a huge performance impact on your application.
- **Scaling.** As an application scales up, resources can become constrained. Caching allows an application to reduce the need to use redundant resources (as shown in our multiplication-service example), which improves the overall scale at which the application can operate without becoming saturated.
- **Resource optimization.** Some resources can be quite expensive in computation usage, memory usage, etc. Multiplying 3 times 4 is not really expensive at all, but running a large data simulation might be very expensive. Caching can reduce the need for some of these operations, which reduces the resources required, and can improve throughput.
- **Convenience and availability.** Sometimes, the resources needed to perform a calculation might not be available. They could be used for other purposes (especially in a high volume, scaled application), or they could be offline or simply unavailable. If the result is already available in a cache, you can return a result without needing the underlying resources, and hence the lack of availability of those resources won't be an issue.

Cache concerns

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

1. Caching can cause the application to not execute desired side effects of targeted operations
2. Inconsistent data in a cache
3. Poor cache performance

Dealing with side effects

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

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

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

- A service that changes the position of a car's steering wheel has an observable impact—it has a side effect.
- Software that stores data in a database has an external impact—it's changing the state of the database.

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

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

Inconsistent cache data

What happens if the cache for the multiplication service doesn’t have the value “12” stored as the result of “3 times 4”, but instead has the value of “13” stored? Then the cache is said to be inconsistent, because the data in the cache doesn’t match the value in the backing service.

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

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

Performance of the cache

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

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

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

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

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

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

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

For more on cache performance, see Chapter 9, “Cache Performance.”

Summary

As we've shown, caching can be a great way to improve the performance, scalability, availability, and reliability of a service. When done properly, caching is highly valuable in applications of almost any size and complexity.

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

4.

Basic Caching Strategies

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

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 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).

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.

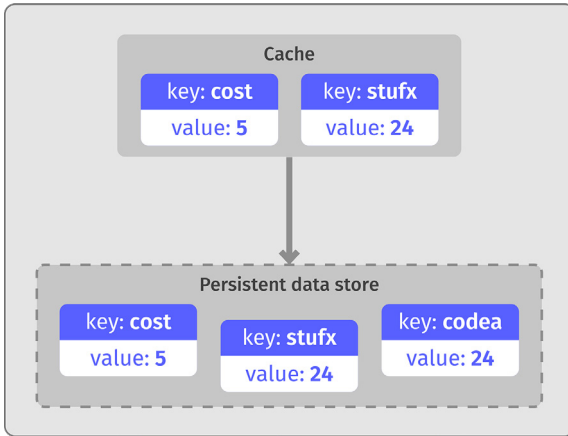


Figure 4-1. Cache in front of a persistent store

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.

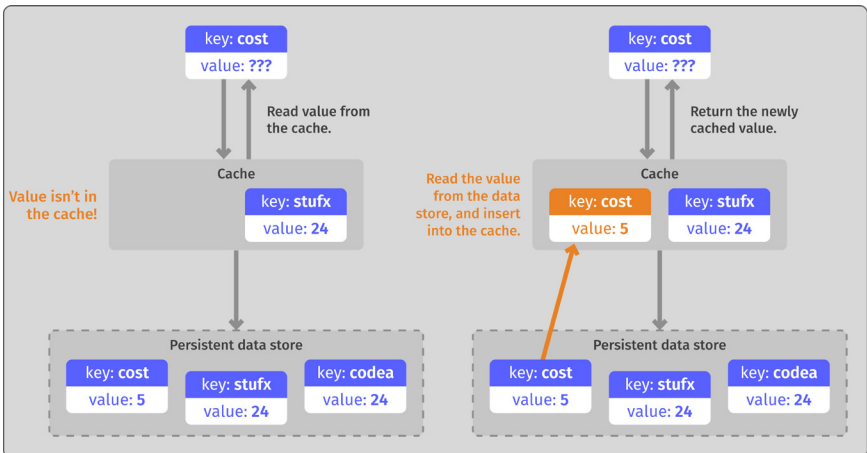


Figure 4-2. Inline cache, in which cache consistency is the responsibility of the cache

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.

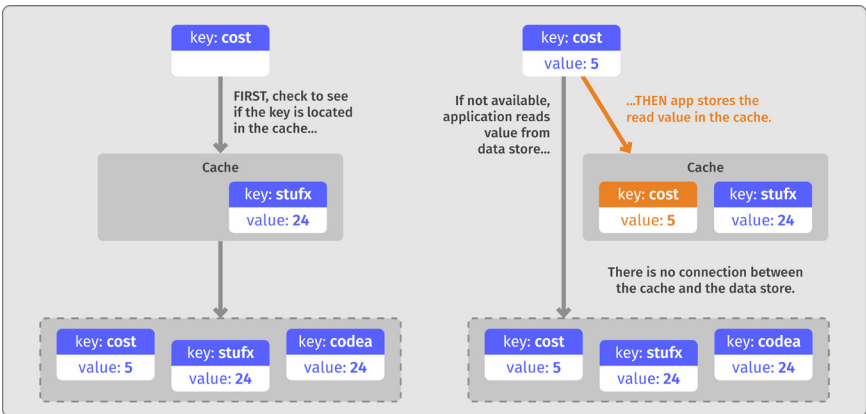


Figure 4-3. Cache-aside, in which cache consistency is the responsibility of the application

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.

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.

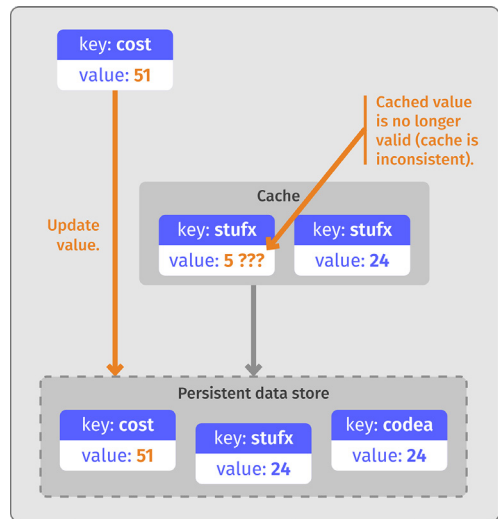


Figure 4-4. Failure in cache consistency

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.

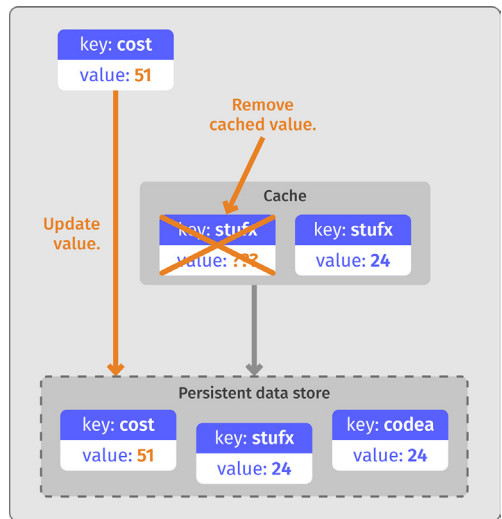


Figure 4-5. Invalidating cache on write

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.

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.

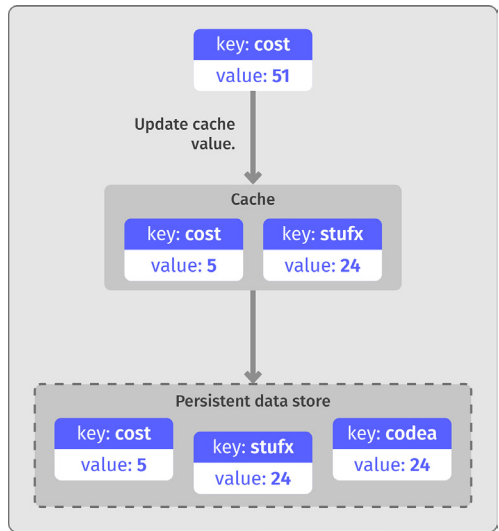


Figure 4-6. An attempt to update a cache value

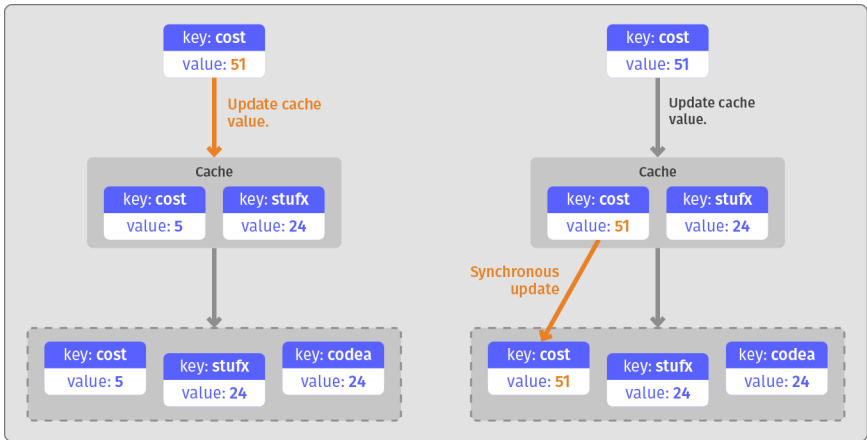


Figure 4-7. Write-through cache

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.

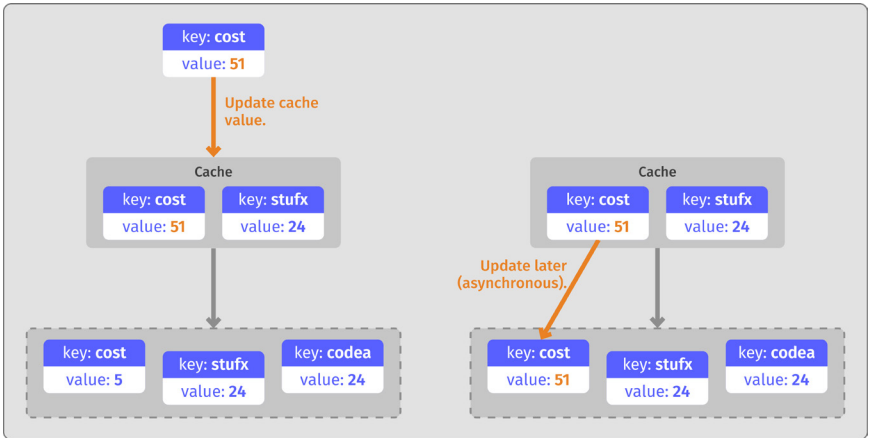


Figure 4-8. Write-behind cache

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.

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.

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.

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	<i>n/a</i>	Data is randomly deleted.
Permanent	<i>n/a</i>	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.

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
```

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.

5.

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.”

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.

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.

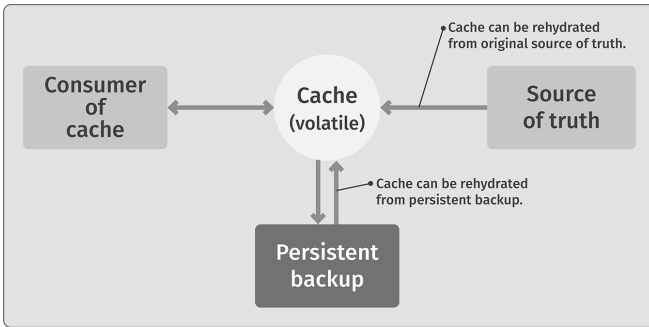


Figure 5-1. Cache rehydration

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.

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.

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.

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.

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.

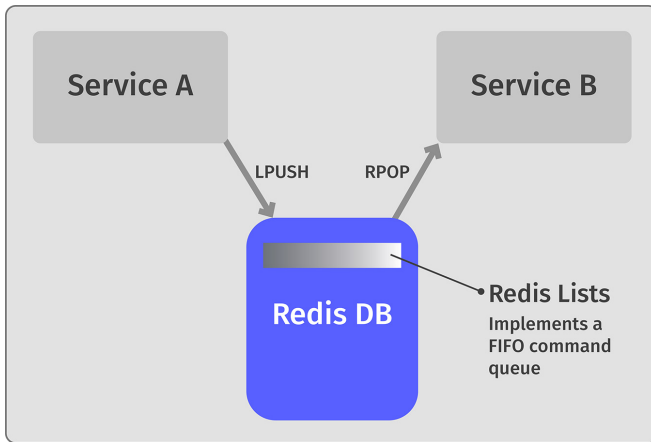


Figure 5-2. Redis List as a FIFO command queue for microservices

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.”

For more information, see “How Redis Simplifies Microservices Design Patterns” on The New Stack (<https://thenewstack.io/how-redis-simplifies-microservices-design-patterns>).

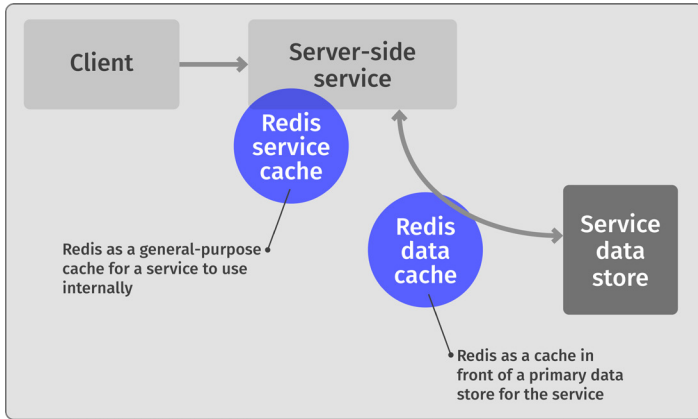


Figure 5-3. Redis as a distributed cache

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.

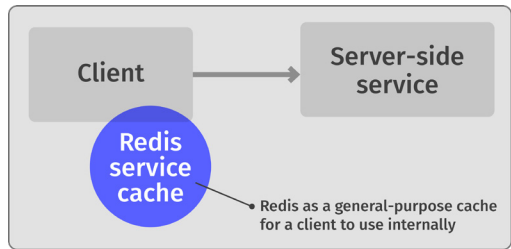


Figure 5-4. Caching interim result

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.

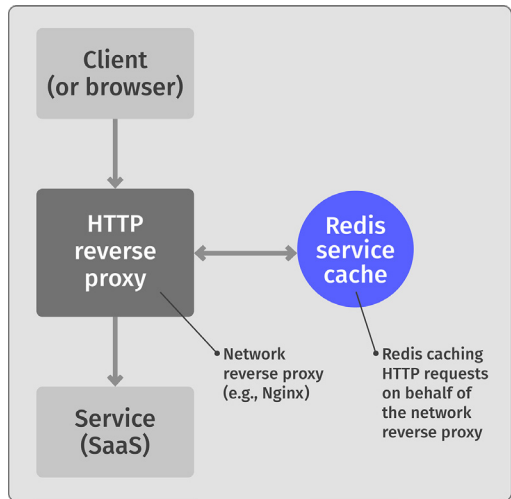


Figure 5-5. Network protocol layer cache

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.

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.

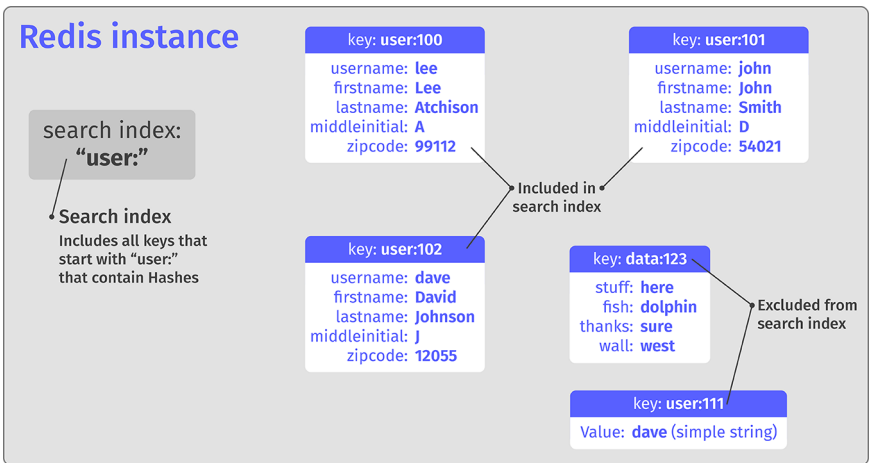


Figure 5-6. Redis instance with Redisearch search index

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"...
```

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.

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
```

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.

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"
```

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

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.

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.

6. Cache Scaling

*“How many seconds does it take to change a lightbulb?
Zero—the lightbulb was cached.”*

Caches are hugely important to building large, highly scalable applications. They improve application performance and reduce resource requirements, thus enabling greater overall application scalability.

But what do you do when the cache itself needs to scale?

Once your application has reached a certain size and scale, even your cache will meet performance limits. There are two types of limits that caches typically run into: **storage limits** and **resource limits**.

Storage limits are limits on the amount of space available to cache data.

Consider a simple service cache, where service results are stored in the cache to prevent extraneous service calls. The cache has room for only a specific number of request results. If that number of unique requests is exceeded, then the cache will fill, and some results will be discarded. The full cache has reached its storage limit, and the cache can become a bottleneck for ongoing application scaling.

Resource limits are limits on the capability of the cache to perform its necessary functions—storing and retrieving cached data. Typically, these resources are either network bandwidth to retrieve the results, or CPU capacity in processing the request. Consider the same simple service cache. If a single request is made repeatedly and the result is cached, you won't run into storage limits because only a single result must be cached. However, the more the same service request is made, the more often the single result will be retrieved from the cache. At some point, the number of requests will be so large that the cache will run out of the resources required to retrieve the value.

Improving cache scalability

In order to improve the scalability of a cache, you must increase storage limits and/or resource limits, as necessary, to avoid either of these limits impacting your cache's performance. Similar to how parts of a computing system are scaled, there are two primary ways to increase the scale of your cache: **vertical scaling** and **horizontal scaling**.

Vertical scaling, or scaling up and scaling down, involves increasing the resources available for the cache to operate. Typically, this involves moving to a more powerful computer running the cache. For a cloud-operated cache, this often means moving to a larger instance.

Vertical scaling can increase the amount of RAM available to the cache, thus reducing the likelihood of the cache reaching a storage limit. But it can also add larger and more powerful processors and more network bandwidth, which can reduce the likelihood of the cache reaching a resource limit.

Horizontal scaling, or scaling out and scaling in, involves adding additional computer nodes to a cluster of instances that are operating the cache, without changing the size of any individual instance. Depending on how it's implemented, horizontal scaling can also improve overall cache reliability, and hence application availability.

In other words, vertical scaling means increasing the size and computing power of a single instance or node, while horizontal scaling involves increasing the number of nodes or instances.

Horizontal scaling techniques

There are many different ways to implement horizontal scaling, each with a distinct set of advantages and disadvantages.

Read replicas

Read replicas are a technique used in open source Redis for improving the read performance of a cache without significantly impacting write performance. In a typical simple cache, the cache is stored on a single server, and both read and write access to the cache occur on that server.

With **read replicas**, a copy of the cache is also stored on auxiliary servers, called read replicas. The replicas receive updates from the primary server. Because each of the auxiliary servers has a complete copy of the cache, a read request for the cache can access any of the auxiliary servers—as they all have the same result. Because they are distributed across multiple servers, a significantly greater number of read requests can be handled, and handled more quickly.

When a write to the cache occurs, the write is performed to the master cache instance. This master instance then sends a message indicating what has changed in the cache to all of the read replicas, so that all instances have a consistent set of cached data.

A large Redis implementation consisting of at least three servers is illustrated in Figure 6-1. All writes to the Redis database are made to the single master. This single master sends updates of the changed data to all of the replicas. Each replica contains a complete copy of the stored Redis database. Then, any read access to the Redis instance can occur on any of the servers in the cluster.

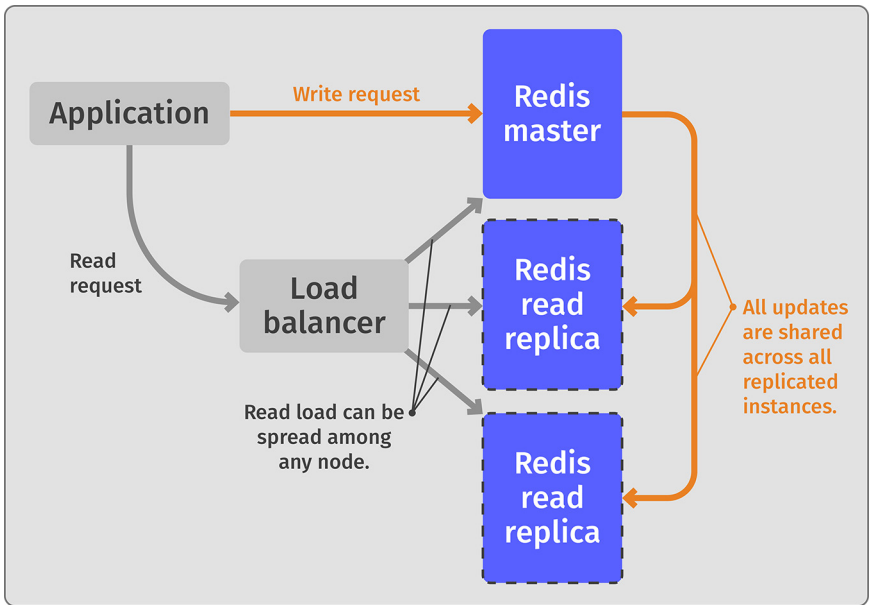


Figure 6-1. Horizontal scalability with read replicas

This model does not improve write performance, but it can increase read performance for large-scale implementation by spreading the read load across multiple servers. Additionally, availability can be improved—if any of the read replicas crash, the load can be shared to any of the other servers in the cluster, so the system remains operational. For increased availability, if the Redis master instance fails, one of the read replicas can take over the master role and assume those responsibilities.

Sharding

Sharding is a technique for improving the overall performance of a cache, along with increasing both its storage limits and resource limits.

With sharding, data is distributed across various partitions, each holding only a portion of the cached information. A request to access the cache (either read or write) is sent to a shard selector (in Redis Enterprise, this is implemented in a proxy), which chooses the appropriate shard to which to send the request. In a generic cache, the shard selector chooses the appropriate shard by looking at the cache key for the request. In Redis, shard selection is implemented by the proxy that oversees forwarding Redis operations to the appropriate database shard. It then uses a deterministic algorithm to specify which shard a particular request should be sent to. The algorithm is deterministic, which means every request for a given cache key will go to the same shard, and only that shard will have information for a given cache key. Sharding is illustrated in Figure 6-2.

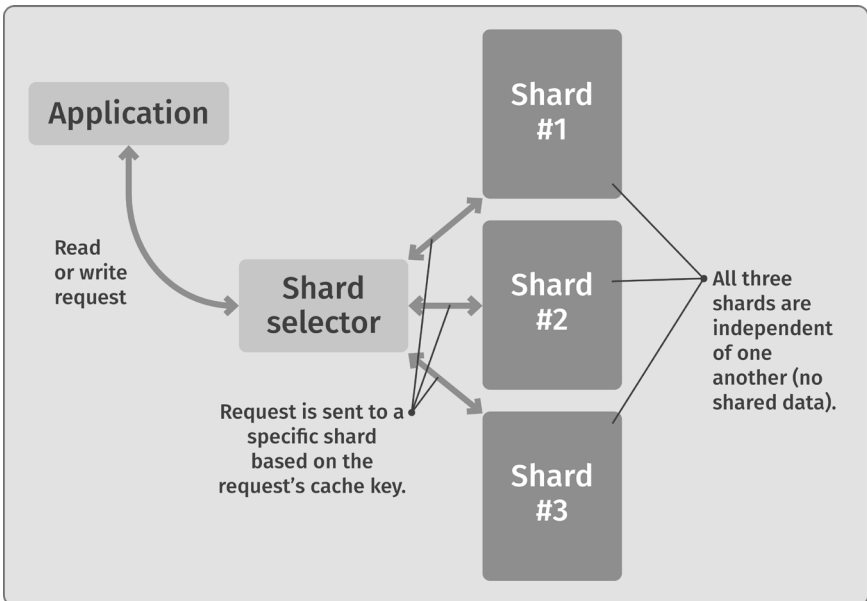


Figure 6-2. Horizontal scaling via sharding

Sharding is a relatively easy way to scale out a cache's capacity considerably. By adding three shards to an open source Redis implementation, for instance, you can nearly triple the performance of the cache, and triple the storage limits.

But sharding isn't always simple. In generic caches, choosing a shard selector that effectively balances traffic across all nodes can require tuning. It can also lower availability by increasing dependency on multiple instances. Failure of a single instance can, if not properly managed, bring down the entire cache.

Redis Clustering addresses these issues and makes sharding simpler and easier to implement. Redis Clustering uses a simple CRC16 on the key in order to select one of up to 1,000 nodes that contain the desired data. A re-sharding protocol allows for rebalancing for both capacity and performance management reasons. Failover protocols improve the overall availability of the cache.

In open source Redis, clustering is implemented client-side in a cluster-aware client library. This works, but requires client-side support of the clustering protocol. Redis Enterprise avoids these issues by implementing a proxy protocol to provide clustering server side, allowing any client to utilize the clustered cache.

Sharding is an effective way to quickly scale an application, and it is used in a number of large, highly scaled applications. While the concept of sharding has inherent advantages and disadvantages, Redis Clustering eliminates much of the complexities of sharding and allows applications to focus on the data management aspects of scaling a large dataset more effectively.

Active-Active (multi-master)

Active-Active, i.e. multi-master, replication is a way to handle higher loads of both the write and the read performance of a cache.

As with read replicas, Active-Active adds multiple nodes to the cache cluster, and a copy of the cache is stored equally on all of the nodes. Because each node contains a complete copy of the cache, this has no impact on the storage limit of a cache. A load balancer is used to distribute the load across each of the nodes. This means that a significantly larger number of requests can be handled, and handled faster, because they are distributed across multiple servers.

When a write to one of the cache nodes occurs, the instance that receives the write sends a message indicating what has changed in the cache to all of the other nodes, so that all instances have a consistent set of cached data.

In the large cache implementation illustrated in Figure 6-3, the cache consists of at least three servers, each running a copy of the cache software and each with a complete copy of the cache database. Any of them can handle any type of data request.

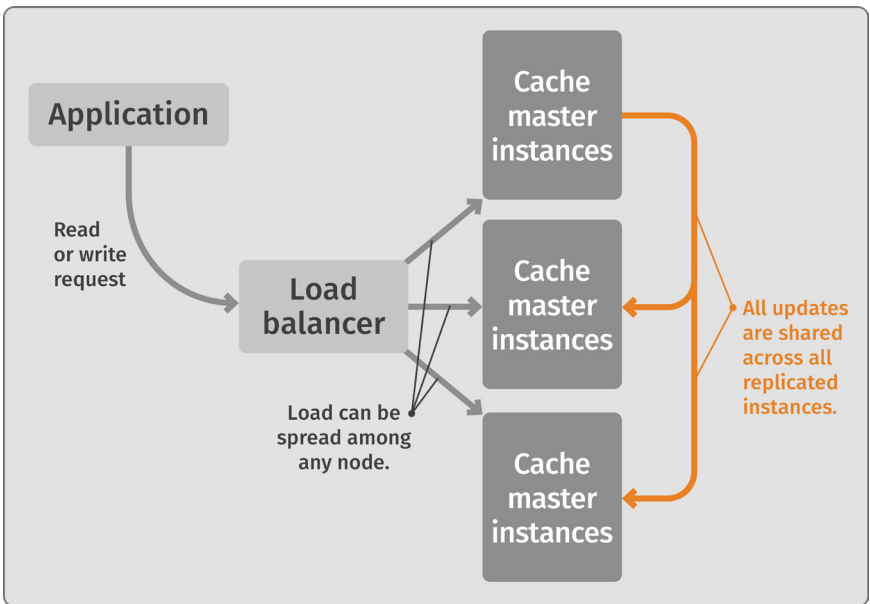


Figure 6-3. Multi-master replication

This model also increases overall cache availability, because if a single node fails, the other nodes can take up the slack.

But what happens when two requests come in to update the same cached data value? In a single-node cache, the requests are serialized and the changes take place in order, with the last change typically overriding previous changes.

In a multi-master model, though, the two requests could come to different masters, and the masters could then send conflicting update messages to the other master servers. This is called a **write conflict**. An algorithm of some sort must be written to resolve these conflicting writes and determine how the multiple requests should be processed (e.g. which one or ones should be processed, and which should be ignored or in what order should the requests be processed). Additionally, **data lag** can occur, meaning that when data is updated in one node, it may take a bit of time before it's updated in all the nodes. Hence, for a period of time, different nodes may contain different data values. This algorithm can be error-prone and result in an inconsistent cache. Care has to be taken that these sorts of problems do not occur, or at least can be successfully repaired when they do.

Redis Enterprise's Active-Active Geo Distribution

Open source Redis does not natively support multi-master redundancy. However, Redis Enterprise does support a form of multi-master redundancy called Active-Active Geo-Distribution.

In this model, multiple master database instances are held in different data centers which can be located across different regions and around the world. Individual consumers connect to the Redis database instance that is nearest to their geographic location. The Active-Active Redis database instances are then synchronized in a multi-master model so that each Redis instance has a complete and up-to-date copy of the cached data at all times. This model is called Active-Active because each of the database instances can accept read and write operations on any key, and the instances are peers in the network.

Redis Enterprise Active-Active Geo-Distribution has sophisticated algorithms for effectively dealing with write conflicts, including implementing conflict-free replicated data types (CRDTs) that guarantee strong eventual consistency and make the process of replication synchronization significantly more reliable. Note that the application must understand the implications of data lag and resulting write conflicts, and must be written so that these issues aren't a problem.





Technique	Storage limits	Resource limits	
Read replica			Easy to configure in open source Redis, but does not improve write performance
Sharding			Improves database capacity as well as performance. Easy to configure with Redis Enterprise Cluster. Requires tuning.
Active-Active			Ensures each Redis master has an up-to-date copy of the cached data at all times. Delivers local latency on read and write operations in multiple regions. Can continue to handle read and write operations even when the majority of geo-replicated regions are down.

Table 6-1. Scaling technique summary

Cache scaling technique summary

Table 6-1 shows a summary of the scaling techniques discussed in this chapter, along with their advantages and disadvantages.

Which technique, if any, you choose to use depends on your application's architecture and requirements, and your organization's goals.

7.

Cache

Consistency

*“Cache consistency isn’t everything.
A cache can be quite wrong,
yet consistently so.”*

Maintaining cache consistency is one of the greatest challenges in managing the operation of a cache, so choosing the right caching pattern is essential.

In Chapter 3, “Why Caching?”, we introduced a multiplication service and demonstrated how caching could be used to improve the performance of this service. Going back to that example, what happens if the multiplication service doesn’t have the value “12” stored as the result of “3 times 4”, but instead has the value of “13” stored?

In that case, when a request comes in to return the result of “3 times 4”, the cached value will be used rather than a calculated value, and the service will return “13”, an obviously incorrect result that would mostly likely never occur in the real world.

This is an example of a cache that is **inconsistent**, because it has stored an up-to-date response to a request. Sometimes, it can be difficult to realize that this is happening, and even more difficult to remove these inconsistent results. This may or may not be a major problem for the application, depending on how the application uses the data.

How does a cache become inconsistent?

There are many ways a cache can become inconsistent, including:

1. When the underlying results change, and the cache is not updated
2. When there is a delay in updating cached results
3. When there is inconsistency across cached nodes

Let’s look at each of these issues independently.

The underlying results change and the cache is not updated

Consider Figure 7-1, which shows a service requesting a result to be read from a presumably slow data source, such as a remote service or database. In order to speed up reading of the data, a cache is used to make access to frequently used results quicker.

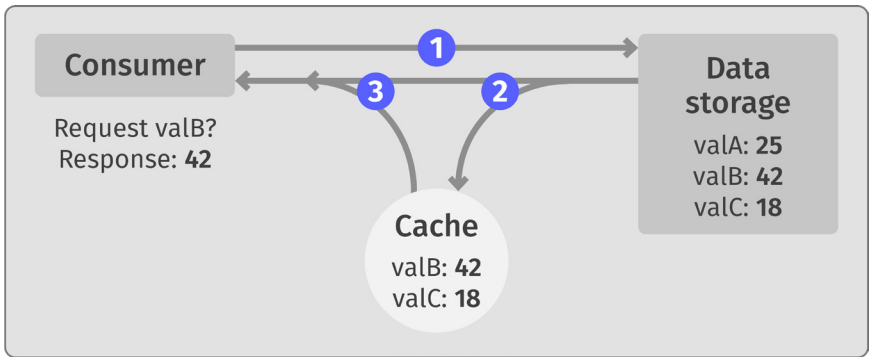


Figure 7-1. A simple cache fronting a slow data source

The user requests a specific value to be read from the slow data source. The cache is consulted. If the value is not in the cache, the slower backing data source is consulted, the result is stored in the cache, and the request returns. However, if the result is stored in the cache, the cached value is returned directly.

But what happens if the value in the underlying data source has changed? In a cache-aside pattern, if the old value is still in the cache, then the old value will be returned from future requests, rather than the new value. The cache is now inconsistent. This is illustrated in Figure 7-2. In order for the cache to become consistent again, either the old value in the cache has to be updated to the new value, or the old value has to be removed from the cache, so future requests will retrieve the correct value from the underlying data store.

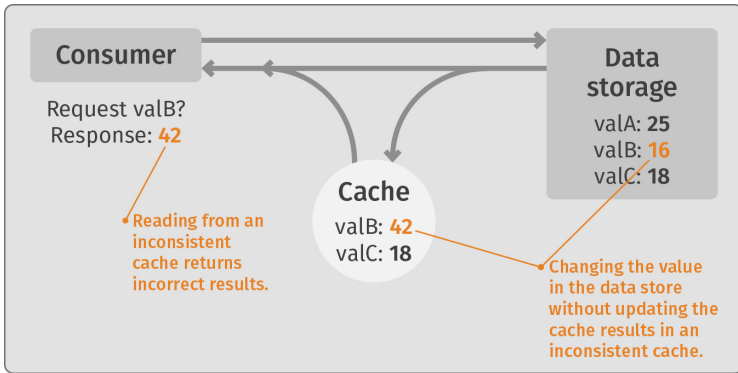


Figure 7-2. Updating the data store without updating cache results in an inconsistent cache.

A delay in updating cached results

When an underlying data store changes a value, and it needs to update the cache about the changed value, often it sends a message to the cache telling it to either remove the old value or update the cached value to the new, correct value. This processing takes some time, during which the cache still has the old, inconsistent value. Any request that comes in after the data value has changed, but before the cache has updated its value, will return the inconsistent, incorrect value.

This delay could, and should, be quite short—hopefully short enough so that the delay does not cause any serious problems. However, in some cases it can be quite lengthy. Whether or not this delay causes a problem is entirely dependent on the use case, and it is up to the specific application to decide if the delay causes any issues.

Additionally, some caches can be used to cache data from a dynamically changing data store. This is often the case in database caches, for instance, where the underlying data changes occasionally yet regularly.

In these cases, one strategy is to set an expire time on the cache, requiring the cached values to be thrown away and reread from the underlying data store at regular intervals, limiting the amount of time the cached value may be inconsistent. While this strategy can reduce the length of time a cached value is

inconsistent, it doesn't remove the inconsistency entirely. As such, this strategy is used only for caching dynamically changing data, where some amount of variation from returning an accurate result is acceptable. An example of this type of cache might be caching the number of likes on a social media post. In this case, the number changes continuously, but if the cache is set to expire every, say, 15 minutes, you can guarantee the cached value is always accurate to within the most recent 15-minute value. The cache value is still inconsistent, but the inconsistency is minimal, and within the bounds of acceptability for that application use case.

Inconsistency across cached nodes

In highly scaled or highly distributed caches, multiple computer nodes are often used to implement the cache. This was discussed in greater detail in chapter 6, “Cache Scaling.”

In many scenarios, multiple cache nodes will have duplicate copies of all or part of the cache. An algorithm is used to keep the cache values up to date and consistent across all of the cache nodes. This is illustrated in Figure 7-3.

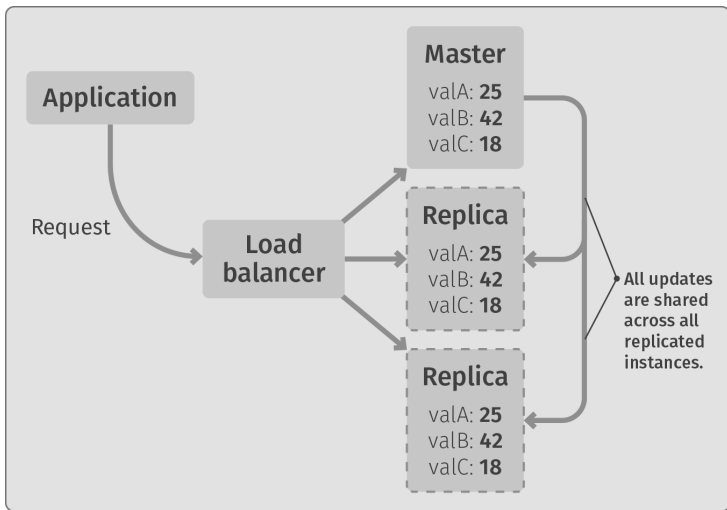


Figure 7-3. Replicated cache with consistent data

However, updating multiple nodes, especially if there are a lot of them or if they are geographically distributed, can take a significant amount of time. During the time the nodes are being updated, different nodes may contain different values. As a result, depending on which cache node receives a request, the result returned can be inconsistent until eventual consistency is reached. This is illustrated in Figure 7-4.

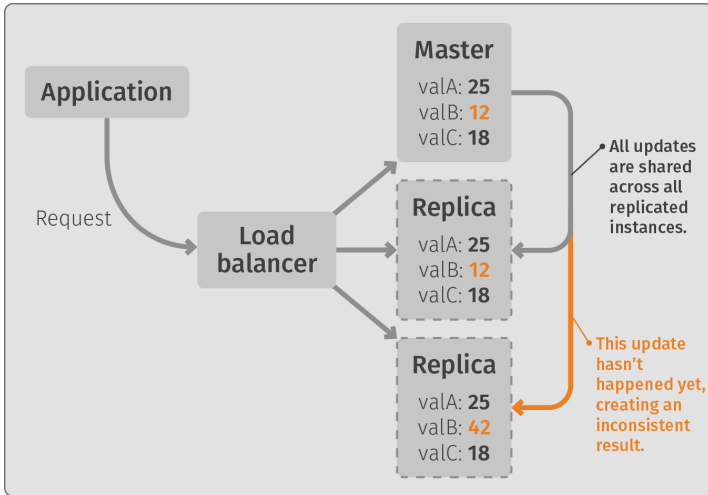


Figure 7-4. Replicated cache with delayed (inconsistent) data

A real-world example of this is when a website is cached at edge locations around the world, in order to speed up access to various portions of a website, such as images, diagrams, and photographs. Often, there are many of these caches around the world, and updating all of them to include updated information, such as an updated diagram, can take a long time. The result is that, for some period of time after an update to a website is made, the “old” website will still be returned for some people in some parts of the world, until all the caches have been updated with the new content. There are various application-specific strategies to address this issue, and how important of a problem this is depends on the application and the application’s needs.

8.

Caching and the Cloud

“Cloudy with a chance of caching.”

Running a single open source Redis instance on premises is rather straightforward. There are a couple options for how to set it up, and none of them are complex.

However, in the cloud, there are many different ways to set up and configure Redis as a cache server. In fact, there are more ways to set up a Redis cache than there are cloud providers. This chapter discusses some of the various cloud options available.

Redis services on major cloud providers

All the major cloud providers offer Redis databases, often configured primarily for caching purposes, that are easy to set up and easy to administrate. They can be utilized just like any of a given provider's other cloud services.

Cloud providers' caches

All major cloud providers, including Amazon Web Services (AWS), Google Cloud, Microsoft Azure, and IBM Cloud offer services that include the open source version of Redis. These are available in a wide variety of sizes. Often, they are available as either single instances or with load-balanced read replicas included. They can be set up in a variety of regions across the globe.

These instances are easy to set up and use, can be turned on/off very quickly, and are typically charged by the hour or the amount of resources consumed. This makes them especially well-suited for development, testing, and autoscaled production environments.

Several other service providers offer preconfigured, cloud hosted versions of Redis instances, including:

- **Redis To Go**
- **Heroku**
- **ScaleGrid**
- **Aiven**
- **Redis Cluster** (specializes in Redis-hosted on Kubernetes)
- **Digital Ocean**
- **cloud.gov** (specializes in governments and government contractors)

Redis Enterprise Cloud

Redis offers a cloud-hosted version of its enterprise-grade, fully supported, enhanced Redis Enterprise software. Redis Enterprise Cloud is the fully managed DBaaS that provides transparent high availability and supports Active-Active Geo-Distribution as well as hybrid and multicloud deployments. Redis Enterprise Cloud is available from Redis on the three major cloud providers, AWS, Azure, and Google Cloud. On Microsoft Azure, for example, Redis Enterprise is offered as a fully supported first-party offering called Azure Cache for Redis Enterprise.

In addition to the usefulness of Redis Enterprise Cloud for high-performance production environments, there is also a free tier that can be used for test drives and development purposes. If you are serious about using Redis for high-end production workloads, Redis Enterprise Cloud could be right for you.

Self-hosted in the cloud

You can, of course, install and run Redis yourself, either the open source or the Redis Enterprise version, on standalone cloud compute instances, just as you can install and run Redis on computers in your own on-premises data centers. Nothing magical is needed. However, it is important to make sure security configurations are created and handled correctly. All the major cloud providers, and Redis Enterprise Cloud, offer standard security capabilities based on their normal cloud-security settings—all set up and pre-configured for you. If you set up and run the instances yourself, however, you have to make sure the security settings are correct, certificates are valid, and security is maintained.

Hybrid and multicloud deployments

As discussed in Chapter 6, “Cache Scaling,” you can set up an open source Redis instance to be composed of one Redis primary instance, and one or more Redis read replicas. This setup is shown in Figure 8-1.

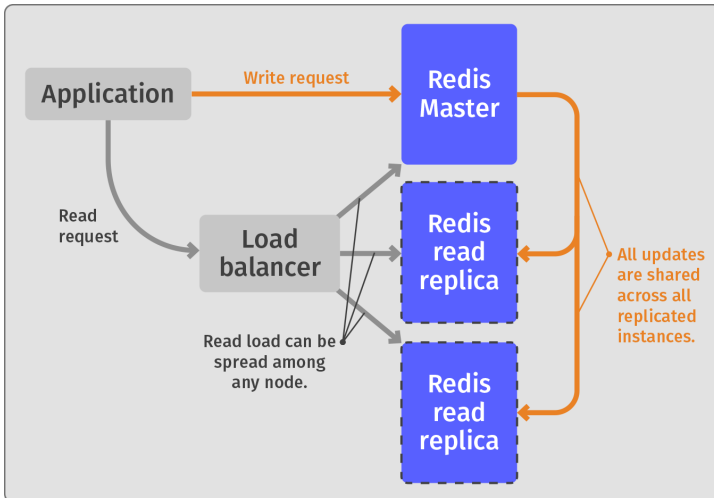


Figure 8-1. Read replicas

In a typical cloud setup, these read replicas are put into separate availability zones for improved system availability. However, it is possible to put the read replicas in entirely different cloud regions in order to provide improved read performance at various geographic locations. This is shown in Figure 8-2.

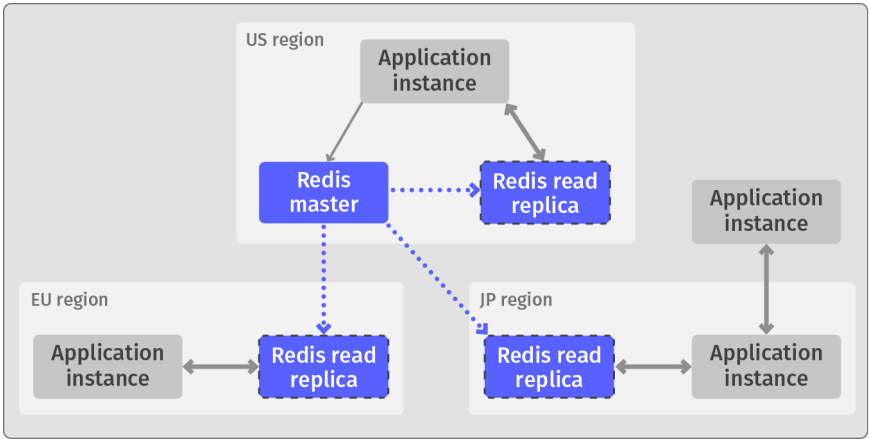


Figure 8-2. Multi-region read replicas

In theory, this model works even in cases in which the different regions are provided by different cloud providers. The only caveat is that the replication setup commands must be available for configuration by the cloud provider, and those commands can be restricted on some levels of service from some providers. Nonetheless, you could set up Redis manually on separate compute instances in multiple cloud providers and then configure the replication so that your read replicas from a given cloud provider are connected to a master in another cloud provider. This is shown in Figure 8-3.

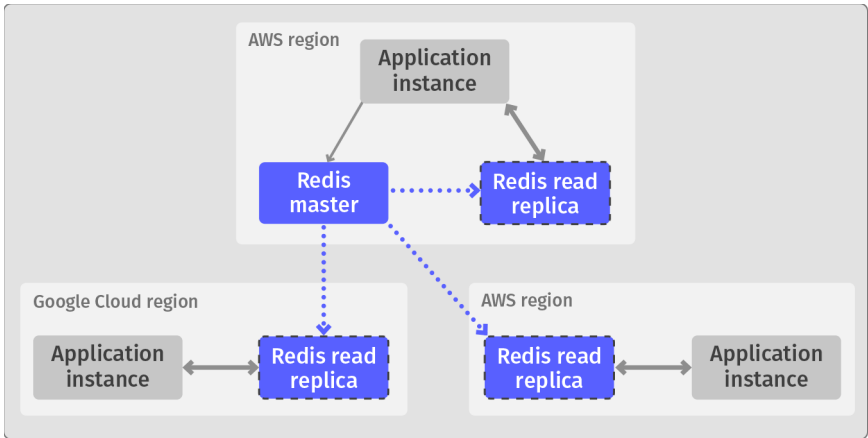


Figure 8-3. Multicloud read replicas

Redis Enterprise cluster deployments

Using basic open source Redis, you are limited to a single master and any number of read replicas. All writes must be sent to the single Redis master. This limits the usefulness for multi-region deployments, and multi-provider, multicloud deployments. This is because when an application is spread across multiple regions and/or multiple providers, only the read performance can be improved by specifying a local read replica, as shown in Figures 8-2 and 8-3. Write requests must still go back to the single master instance. So, in the Figure 8-2 example, if the application in the JP region wants to write to the Redis database, it must send that write to the Redis master instance in the US region. This can have a significant impact on write performance.

In order to improve both write and read performance in a multi-region or multicloud deployment, you must use a different replication architecture than the simple master-replica architecture described here. Instead, you must create a multi-cluster topology as shown in Figure 8-4. This requires multi-master capability, which is not available out of the box in open source Redis, but in Redis Enterprise, multiple masters can be deployed across multiple regions using an Active-Active deployment.

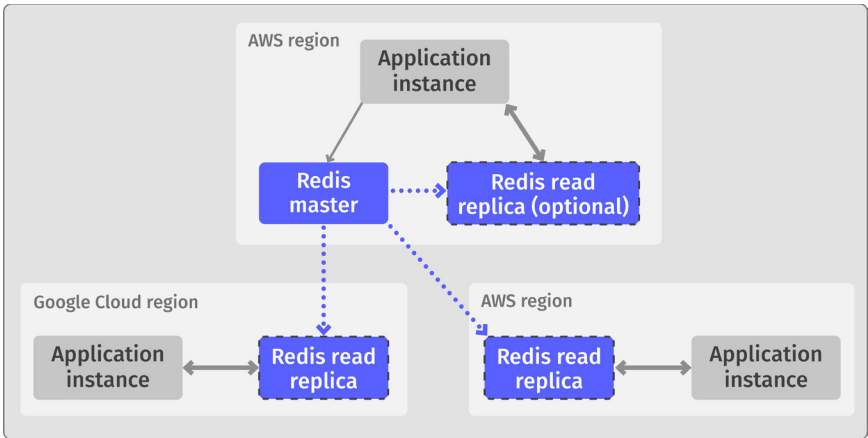


Figure 8-4. Multi-region Active-Active replication with Redis Enterprise

In this model, both reads and writes can be processed from any of the Redis master instances in any region or with any provider. This improves application performance dramatically. After a write occurs in a given region, it is automatically replicated to all the other masters in the cluster.

To take advantage of this type of cluster replication, you must use Redis Enterprise. For cloud-hosted databases, that means you must either use Redis Enterprise Cloud instances, or you must roll your own self-hosted Redis instances on cloud compute instances or container images. None of the major cloud providers offer multi-region Active-Active deployments natively, nor do they offer deployments across cloud providers. For this type of large-scale, highly available distributed architecture, you must use Redis Enterprise.

9. Cache Performance

*“A cache can improve performance...
or ruin it.”*

Caches are useful only if they are effective. What do we mean by effective? Well, it depends on why you are using a cache. A cache is most commonly used to decrease the number of long or expensive operations that need to be performed, increasing the speed or efficiency of request processing.

Consider the multiplication example described in Chapter 2, “Why Caching?” In that section, we described a simple multiplication service with a cache in front of it. Whenever a request is made to perform a multiplication, first the cache is consulted to see if the request has already been processed. If not, the multiplication service is called, and the result is returned and stored in the cache for future use. That way, the next time the same request is made, the result is already stored in the cache.

How does the cache improve performance? The diagram in Figure 9-1 shows the same cached multiplication service introduced in Chapter 2. This version, though, shows how much time it takes to retrieve a value from the cache (hypothetically, 1 millisecond), compared with having the multiplication service calculate the result (25ms). Put another way, the first time the request is made, the multiplication service has to be consulted, so the entire operation takes approximately 25ms. But each subsequent equivalent operation can be performed by retrieving the value from the cache, which takes only 1ms, in our example. The result is improved performance for cached operations.

This is how a cache improves performance.

Notice that talking to and manipulating the cache also takes time. So requests that must call the service also have to first check the cache and add a result to the cache. This additional effort is called the **cache overhead**.

In a cache-aside strategy, the cache overhead includes the **cache check** time (to see if the result was previously in the cache) and the **cache write** time (to store the newly calculated result in the cache). Requests that end up having to call the service anyway take more effort than simply calling the service. Put in cache terms, a **cache miss** (that is, a request that cannot be satisfied by results stored in the cache) incur additional overhead compared to just calling the service. By corollary, a **cache hit** (that is, a request that can be satisfied by results stored in the cache) are satisfied significantly faster using only the results stored in the cache.

The total time a request takes to process as a result of a cache miss is:

```
Request_Time = Cache_Check + Service_Call_Time + Cache_Write
```

In our multiplication service, let's make the following assumptions:

```
Cache_Check = 1ms # How long to get a value from the cache
```

```
Cache_Write = 1ms # How long to write a new value into the cache
```

```
Service_Call_Time = 25ms # How long service takes to process multiplication request.
```

Therefore, the `Request_Time` for our multiplication service, when we have a cache miss is:

```
Request_Time = 1ms + 25ms + 1ms
```

```
Request_Time = 27ms # For a cache miss
```

Notice that this total time is greater than the time it takes for the service to process the request if there was no cache (25ms). The additional 2ms is the cache overhead.

But requests that can be fulfilled by simply reading the cache take less effort, because they do not have to make any calls to the service. Put in cache terms, requests that cache hit take significantly less time by avoiding the service call time.

The total time a request takes to process as a result of a cache hit is:

$$\text{Request_Time} = \text{Cache_Check}$$

Therefore, the `Request_Time` for our multiplication service, when we have a cache hit is:

$$\text{Request_Time} = 1\text{ms}$$

So, some requests take significantly less time (1ms in our example), while other requests incur additional overhead (2ms in our example). Without a cache, all requests would take about the same amount of time (25ms in our example).

In order for a cache to be effective, the overall time for all requests must be less than the overall time if the cache didn't exist.

This means, essentially, that there needs to be more cache hits than cache misses overall. How many more depends on the amount of time spent processing the cache (the **cache overhead**) and the amount of time it takes to process a request to the service (**service call time**).

The greater the number of cache hits compared with the number of cache misses, the more effective the cache. Additionally, the greater the service call time compared with the cache overhead, the more effective the cache.

Let's look at this in more detail. First, we need to introduce two more terms. The **cache miss rate** is the percentage of requests that generate a cache miss. Conversely, the **cache hit rate** is the percent of requests that generate a cache hit. Because each request must either be a cache hit or cache miss, that means:

$$\text{Cache_Miss_Rate} + \text{Cache_Hit_Rate} = 1 \text{ (100\%)}$$

Now, let's use these rates to determine the efficiency of our service's cache.

When using our multiplication service without a cache, each request takes 25ms. With a cache, the time is either 1ms or 27ms, depending on whether there was a cache hit or cache miss. In order for the cache to be effective, the 2ms overhead of accessing the cache during a cache miss must be offset by some number of cache hits. Put another way, the total request time without a cache must be greater than the total request time with a cache for the cache to be considered effective. Therefore, in order for the cache to be effective:

$$\text{Request_Time_No_Cache} \geq \text{Request_Time_With_Cache}$$

$$\begin{aligned} \text{Request_Time_With_Cache} = \\ & (\text{Cache_Miss_Rate} * \text{Request_Time_Cache_Miss}) + \\ & (\text{Cache_Hit_Rate} * \text{Request_Time_Cache_Hit}) \end{aligned}$$

And since:

$$\text{Cache_Hit_Rate} = 1 - \text{Cache_Miss_Rate}$$

You can rewrite this as:

$$\begin{aligned} \text{Request_Time_With_Cache} = \\ & (\text{Cache_Miss_Rate} * \text{Request_Time_Cache_Miss}) + \\ & (\mathbf{1 - \text{Cache_Miss_Rate}}) * \text{Request_Time_Cache_Hit}) \end{aligned}$$

Therefore:

$$\text{Request_Time_No_Cache} \geq \text{Cache_Miss_Rate} * \text{Request_Time_Cache_Miss} + (1 - \text{Cache_Miss_Rate}) * \text{Request_Time_Cache_Hit}$$

For our multiplication example, that means:

$$25\text{ms} \geq \text{Cache_Miss_Rate} * 27\text{ms} + (1 - \text{Cache_Miss_Rate}) * 1\text{ms}$$

$$25\text{ms} \geq (\text{Cache_Miss_Rate} * 27\text{ms}) + 1\text{ms} - (\text{Cache_Miss_Rate} * 1\text{ms})$$

$$25\text{ms} \geq 1\text{ms} + \text{Cache_Miss_Rate} * 26\text{ms}$$

$$24\text{ms} \geq \text{Cache_Miss_Rate} * 26\text{ms}$$

$$\text{Cache_Miss_Rate} \leq 24/26$$

$$\text{Cache_Miss_Rate} \leq 92.3\%$$

Given that cache hit rate + cache miss rate = 1, we can do the same calculation using the cache hit rate rather than the cache miss rate:

$$25\text{ms} \geq (1 - \text{Cache_Hit_Rate}) * 27\text{ms} + \text{Cache_Hit_Rate} * 1\text{ms}$$

$$25\text{ms} \geq 27\text{ms} - \text{Cache_Hit_Rate} * 27\text{ms} + \text{Cache_Hit_Rate} * 1\text{ms}$$

$$25\text{ms} \geq 27\text{ms} - \text{Cache_Hit_Rate} * 26\text{ms}$$

$$2\text{ms} \leq \text{Cache_Hit_Rate} * 26\text{ms}$$

$$\text{Cache_Hit_Rate} \geq 2/26$$

$$\text{Cache_Hit_Rate} \geq 7.7\%$$

In other words, in this example, as long as a request can be satisfied by the cache (cache hit rate) at least 7.7% of the time, then having the cache is more efficient than not having the cache.

Doing the math the other way, you could ask a different question. If the average request time is 25ms without a cache, what would be the average request time if the cache hit rate was 25%? 50%? 75%? 90%?

For our multiplication service, we use this equation:

$$(1 - \text{Cache_Hit_Rate}) * 27\text{ms} + \text{Cache_Hit_Rate} * 1\text{ms}$$

For cache hit rate of 25%:

$$(1 - 0.25) * 27\text{ms} + 0.25 * 1\text{ms}$$

$$0.75 * 27 + 0.25 * 1$$

$$20.25 + 0.25$$

$$20.5\text{ms}$$

The average request time assuming a cache hit rate of 25% is 20.5ms. Much faster than the 25ms for no cache!

But it gets better, using our other cache hit rate assumptions:

Cache_Hit_Rate = 50%:

$$(1 - 0.5) * 27\text{ms} + 0.5 * 1\text{ms}$$

$$0.5 * 27 + 0.5 * 1$$

$$13.5 + 0.5$$

$$= 14\text{ms}$$

Cache_Hit_Rate = 75%:

$$(1 - 0.75) * 27\text{ms} + 0.75 * 1\text{ms}$$

$$0.25 * 27 + 0.75 * 1$$

$$6.75 + 0.75$$

$$= 7.5\text{ms}$$

Cache_Hit_Rate = 90%:

$$(1 - 0.9) * 27\text{ms} + 0.9 * 1\text{ms}$$

$$0.1 * 27 + 0.9 * 1$$

$$2.7 + .9$$

$$= 3.6\text{ms}$$

As the cache hit rate increases, the average request time improves dramatically. So if the cache hit rate increases from 25% to 90%, the average request time drops from 20.5ms to 3.6ms—86% lower than without a cache (3.6ms compared with 25ms)!

In other words, the higher the *cache hit rate*, the more effective the cache.

These calculations are all based on the amount of time it takes for the request to be processed by the multiplication service without the cache (25ms in our example). But this value is just an assumption. What happens if that value is larger, say 500ms?

```
Cache_Hit_Rate = 25%:  
  (1 - 0.25) * 502ms + 0.25 * 1ms  
  0.75 * 502 + 0.25 * 1  
  376.5 + 0.25  
  = 376.75ms
```

```
Cache_Hit_Rate = 50%:  
  (1 - 0.5) * 502ms + 0.5 * 1ms  
  0.5 * 502 + 0.5 * 1  
  251 + 0.5  
  = 251.5ms
```

```
Cache_Hit_Rate = 75%:  
  (1 - 0.75) * 502ms + 0.75 * 1ms  
  0.25 * 502 + 0.75 * 1  
  125.5 + 0.75  
  = 126.25ms
```

```
Cache_Hit_Rate = 90%:  
  (1 - 0.9) * 502ms + 0.9 * 1ms  
  0.1 * 502 + 0.9 * 1  
  50.2 + .9  
  = 51.1ms
```

We can see that for a service that takes more resources and has a larger request time without a cache, the impact of the cache on the request time becomes much greater. In particular, at a cache hit rate of 90%, the average request time is 89.8% better than without a cache (51.1ms compared with 500ms).

In other words, the greater the cost of calling the un-cached service, the greater the effectiveness of the cache for a given cache hit rate.

These calculations can and should be performed on each caching opportunity to determine whether or not—and to what extent—the application can effectively utilize a cache.

Glossary.

Cache

Terminology

The following are definitions in general use when talking about caches and cache architectures:

Cache consistency. A measure of whether the contents of the cache matches the underlying data store or service. A cache is considered *consistent* if the data it returns is correct. A cache is considered *inconsistent* if the data it returns does not match the underlying data store or service.

Cache eviction. The act of removing (often older or less frequently used) data from a volatile cache to make room for new data when it is determined the data is either no longer needed, or is less likely to be needed than other data in the cache.

Cache hit. A request for data that can be satisfied using data stored in the cache. If there is a cache hit, it means the cache was successful in handling the request without engaging the backend data store or service.

Cache locality. The set of data available within a cache. Often, depending on the application, the data that will be needed in the future can be inferred by data used in the past, and that inferred data can be loaded in advance into the cache. This inferred data is the cache locality. This practice is common for caches such as CPU or memory caches, where reading one memory location often results in nearby memory locations also being accessed.

Cache miss. A request for data that cannot be satisfied by existing data in the cache and must be sent to the backend data store or service in order for it to be processed. In the case of a cache miss, the cache is not successful in satisfying the request.

Cache overhead. The amount of time in excess of the time it would normally take without a cache to return the desired data result from the data store or service. Cache overhead is used in calculating the effectiveness of a cache in order to improve performance.

Cache warmup. The process of taking a cold cache and converting it into a warm cache. This process may involve artificially storing data, at startup, into the cache to make it a warm cache. But, more often than not, a cache starts out cold and warms up gradually over time as more requests are made. The result is a cache that goes from mostly cache misses to mostly cache hits.

Cold cache. A cache that is empty or does not yet have sufficient data stored in it to result in requests being satisfied from the cache. A cold cache results in a high number of cache misses.

Data lag. The time it takes for a changed value in a database or cache made to one server to be propagated to all other servers in the system.

Flush (cache flush). The action of emptying a cache of all data. When a cache has been flushed, it no longer contains any data and subsequent requests will result in cache misses.

Horizontal scaling. The act of scaling a system to support larger requests and more requests by adding additional servers and other components to the system, rather than making those servers and other components bigger.

Invalidation (cache invalidation). The act of marking data in the cache as incorrect and invalid. When a request arrives at a cache that corresponds to data that has been marked as invalid, it is treated as if the data is not in the cache and a cache miss is generated, forcing the request to be passed on to the underlying data store or service.

Least Frequently Used (LFU). A cache-eviction strategy used by full caches to remove less-likely-to-be-used data from the cache to make room for more-likely-to-be-used data. The LFU strategy removes data that has been used the fewest number of times.

Least Recently Used (LRU). A cache eviction strategy used by full caches to remove less-likely-to-be-used data from the cache to make room for more-likely-to-be-used data. The LRU strategy removes data that hasn't been used for the longest period of time.

Multi-master. A database or cache in which requests, including update requests, can be sent to more than one server, and the database itself coordinates between the servers to make sure the changes are properly updated.

Permanent cache. A cache where data is never evicted or removed. If a permanent cache fills with data, then no new data can be stored in the cache until data has been removed from the cache using some other, often manual, method.

Persistent cache. A persistent cache’s content survives even after power outages and system reboots. An application might rely on an object being stored in the cache forever. It can depend on the value existing for performance reasons, and it is acceptable for an application to fail and not perform if the value is removed inappropriately.

Read replica. A database or cache server that provides a read-only version of the database or cache. It is used to improve scalability for read operations in an easy and typically conflict-free manner.

Sharding. A technique for improving the overall performance of a cache, along with increasing both its storage limits and resource limits. It involves splitting a database or cache into distinct and isolated components that each handle parts of an application’s total data needs. A shard key is used to determine which parts of the data go to one component versus another component. For example, the customer ID could be used to put customers whose names start with letters “a” to “e” on one server, and those starting with “f” to “m” on another server, etc.

Vertical scaling. The act of scaling a system to support larger requests and more requests, by making individual servers and other components of the system bigger and more powerful, rather than adding more servers or components in parallel.

Volatile cache. A cache in which data is evicted, or removed, from the cache if there is no longer any room in the cache to store new data. Typically, data that is old, stagnant, or less likely to be needed by the user is removed from the cache. Different eviction algorithms (LRU, LFU, etc.) are employed to decide what data to remove from the cache.

Warm cache. A cache that has sufficient data stored in it so that there is a reasonably high chance of a request being satisfied from data in the cache, resulting in a high number of cache hits.

Write conflict. When two writes of different values occur from different sources to the same location. To resolve the conflict, the system must determine which value to use.

A note about consistency vs. coherency

In this book, we use the term **cache consistency** to refer to whether a cache is returning a correct result, in comparison to the source data. This is a valid use of the term, and is how most people think about consistency. It's also colloquially correct.

However, cache purists—especially those working on advanced CPU caching as discussed briefly in Chapter 2, “What is Caching”—may use a related term: **cache coherency**. Accurate definitions of these two terms are much debated in the field, but here are simplistic definitions of the two terms:

- **Cache consistency** defines how requests to a given data index (key) from outside the system are ordered to ensure a stored value is correct.
- **Cache coherency** defines the behavior of reading and writing the same data index (key), such that the system behaves as if there is no cache in the system. In other words, reading a given data value will give the same result whether it is read through a cache or not.

It might seem that, technically, what we are referring to in this book is closer to cache coherency than to cache consistency. However, for most people and in most conversations, if you simply use the phrase “cache consistency,” you will be well understood. Either way, if you don't understand these distinctions, don't worry too much. The difference is important mostly for those who deal with the likes of CPU caches.

Epilogue.

About the Author,
About Redis

About the Author

Lee Atchison is a recognized industry thought leader in cloud computing, and the author of multiple books including the best-seller *Architecting For Scale*, published by O'Reilly Media, currently in its second edition.

Lee has 34 years of industry experience. Lee spent seven years at Amazon Web Services, where he led the creation of the company's first software download store, created the AWS Elastic Beanstalk service, and managed the migration of Amazon's retail platform to a new service-based architecture. Additionally, Lee spent eight years at New Relic, where he led the construction of a solid service-based system architecture and system processes, thus allowing New Relic to scale from a startup to a high-traffic public enterprise.

Lee has consulted with leading organizations about modernizing their application architectures and transforming their organizations at scale. Lee is an industry expert and is widely quoted in publications such as *InfoWorld*, *diginomica*, *IT Brief*, *ProgrammableWeb*, *CIORview*, and *DZone*. He has been a featured speaker at events across the globe, from London to Sydney, Tokyo to Paris, and all over North America.

About Redis



Modern businesses depend on the power of real-time data. With Redis, organizations deliver instant experiences in a highly reliable and scalable manner.

Redis is the world's most popular in-memory database, and commercial provider of Redis Enterprise, which delivers superior performance, matchless reliability, and unparalleled flexibility for personalization, machine learning, IoT, search, e-commerce, social, and metering solutions worldwide.

Redis, consistently ranked as a leader in top analyst reports on NoSQL, in-memory databases, operational databases, and Database-as-a-Service (DBaaS), is trusted by more than 7,400 enterprise customers, including five Fortune 10 companies, three of the four credit card issuers, three of the top five communication companies, three of the top five healthcare companies, six of the top eight technology companies, and four of the top seven retailers.

Redis Enterprise, available as a service in public and private clouds, as downloadable software, in containers, and for hybrid cloud/on-premises deployments, powers popular Redis use cases such as high-speed transactions, job and queue management, user session stores, real time data ingest, notifications, content caching, and time-series data.



ON TRACK? OFF TRACK?
UNLOCK THE PROVEN STRATEGY...

MODERNIZE YOUR ENTERPRISE

Thoughtful advice and expertise from an external point of view on modernizing your enterprise applications

Lee Atchison advises leaders facing stagnant revenue and unhappy staff on how to modernize their environment to make customers happier and staff energized for growth and success.

Interested in talking to Lee about your application modernization process? Set up a free 30-minute consultation at **atchisontechnology.com**.

Interested in reading more articles and books by Lee Atchison? **Go to leeatchison.com**.

Prefer to listen rather than read? Check out Lee's podcast, Modern Digital Applications, at **mdacast.com**.

