Design patterns for Amazon ElastiCache

Table of Contents

Introduction 03

What is a cache? When should I use one? 04

What is Redis OSS? Why choose Redis OSS as my in-memory cache? 04

Benefits of Amazon ElastiCache and Amazon MemoryDB 05

Redis OSS resources and bottlenecks 06

Core data modeling patterns 07

Read-aside caching pattern 08

Extending the read-aside pattern for changing data 09

Working with TTLs 11

Efficient resource usage in Redis OSS 12

Memory pressure and object sizes 13

CPU conservation and blocking the main thread 14

Optimizing network usage 15

Working with Redis OSS collection types 16

Hashes 17

Lists 17

Sets 21

Working with advanced data types 22

Sorted sets 23

HyperLogLog 24

Working with multiple operations 26

Pipelining 27

Transactions 28

Scripting 29

Advanced patterns 32

Scaling Redis OSS 33

UNLINK instead of DEL 34

Locking with Redis OSS 34

Saving memory with small hashes 36

Negative result caching 37

Conclusion 38

Introduction

Caching is a critical component for most high-traffic applications. Caches can provide data access at high throughput with low latency while relieving load on your systems of record. When used correctly, caches can even provide for use cases that are near impossible to handle with a traditional database.

In this ebook, you will learn how to use Redis OSS as an application cache. We'll cover some background on caches, their use cases, and some unique aspects about Redis OSS. Then, we'll look at how to use Redis OSS well, including basic caching patterns, the core Redis OSS data types, and some advanced tips.

Introduction

What is a cache? When should I use one?

What is Redis OSS? Why choose Redis OSS as my in-memory cache?

Benefits of Amazon ElastiCache and Amazon MemoryDB

Redis OSS resources and bottlenecks

Core data modeling patterns

Read-aside caching pattern

Extending the read-aside pattern for changing data

Working with TTLs

Efficient resource usage in Redis OSS

Memory pressure and object sizes

CPU conservation and blocking the main thread

Optimizing network usage

Working with Redis OSS collection types

Hashes

Lists

Sets

Working with advanced data types

Sorted sets

HyperLogLog

Working with multiple operations

Pipelining

Transactions

Scripting

Advanced patterns

Scaling Redis OSS

UNLINK instead of DEL

Locking with Redis OSS

Saving memory with small hashes

Negative result caching

Conclusion

What is a cache? When should I use one?

The word "cache" applies to a range of concepts in software development, all the way from the CPU cache or page cache on physical hardware to the browser cache in your end user's Chrome window. But if you hear a coworker mention they want to add a cache to your application, chances are they're talking about a centralized, in-memory cache solution like Redis OSS or Memcached.

For these types of caches, the goal is speed. They aim to provide low-latency, high-throughput access to your application data. To do this, these caches strip out features that are common to other centralized databases. For example, caches often use in-memory storage only rather than more persistent disk-based storage. They also eschew replication in most cases or use asynchronous replication to reduce write times. Finally, they typically provide simple, key-based access to your data rather than a flexible query language or secondary indexes.

Because of these differences, caches are rarely used as system-of-record datastores. Rather, they're aimed at more specialized use cases. A review of a variety of production caching use cases at Twitter was able to group caching use cases into three groups:

- **Caching for storage:** The most common use case. With caching for storage, you are looking to provide faster access to your data. It can also be used to reduce load on your downstream database, but beware of becoming over-reliant on your cache.
- **Caching for compute:** Certain results, such as large-scale aggregations or machine learning inference, can be expensive to compute. If a single result is used multiple times, you can save compute by calculating it once and caching it for subsequent usage.
- **Transient data:** While most cached data is stored in a more persistent database elsewhere, there's certain data that doesn't need as much permanence. Caches can be a nice fit for this data. The common examples here are rate limiters or session stores.

In the data modeling sections below, we'll look at modeling patterns that fit into each of these categories.

What is Redis OSS? Why choose Redis OSS as my in-memory cache?

Redis OSS is used in a wide variety of applications and companies, from enormous social media and e-commerce applications to bootstrappers with a side project. Redis OSS has the standard features of in-memory caches mentioned above, from non-persistent storage to key-based data access. However, there are a few aspects of Redis OSS that are unique from other caches.

First, Redis OSS allows for a variety of types of objects to be stored. Many caches provide simple key-value access where values are simple strings. With Redis OSS, your value can be a string, but it can also be a collection like a list, a set, or a hash. It can even be a more advanced type like a sorted set or HyperLogLog. Developers choose Redis OSS not only for its performance, but for the higher-level objects and operations it provides.

Second, Redis OSS allows you to opt-in to database-like features such as replication and persistence. These features allow you to use the performance and functionality of Redis OSS while also receiving better read scalability and availability through replication or durability through persistence.

Introduction

What is a cache? When should I use one?

What is Redis OSS? Why choose Redis OSS as my in-memory cache?

Benefits of Amazon ElastiCache and Amazon MemoryDB

Redis OSS resources and bottlenecks

Core data modeling patterns

Read-aside caching pattern

Extending the read-aside pattern for changing data

Working with TTLs

Efficient resource usage in Redis OSS

Memory pressure and object sizes

CPU conservation and blocking the main thread

Optimizing network usage

Working with Redis OSS collection types

Hashes

Lists

Sets

Working with advanced data types

Sorted sets

HyperLogLog

Working with multiple operations

Pipelining

Transactions

Scripting

Advanced patterns

Scaling Redis OSS

UNLINK instead of DEL

Locking with Redis OSS

Saving memory with small hashes

Negative result caching

Conclusion

Benefits of Amazon ElastiCache and Amazon MemoryDB

Amazon Web Services (AWS) provides two options for using Redis OSS: Amazon ElastiCache and Amazon MemoryDB. The right option for you depends on your needs and circumstances.

ElastiCache provides a standard Redis OSS experience with a fully managed operational model. You'll get all the core features of Redis OSS, including advanced data types, Redis OSS streams, and high performance. Further, the fully managed model means you have fewer operational burdens. The ElastiCache service will handle failovers if your hardware becomes degraded. You can increase your instance sizes with the click of a button, and ElastiCache even provides clustering to spread your data across multiple instances. In a Redis OSS cluster, your overall dataset is partitioned into sections called "shards", each of which hold a subset of your entire dataset. These shards allow for easier horizontal scalability as your application grows. You can trust the experience and automation of the ElastiCache team to handle your Redis OSS cluster.

Additionally, ElastiCache provides some functionality above and beyond the standard Redis OSS installation. ElastiCache supports enhanced I/O multiplexing where it offloads connection management work to the non-primary threads, allowing the engine full use of the main thread. Further, ElastiCache provides the ElastiCache Serverless option for using the Redis OSS API. With ElastiCache Serverless, you get pay-per-use billing with automated scale-up and scale-down to adjust to your application needs.

Some developers like the objects and API of Redis OSS so much that they would like to use it as a more persistent database. MemoryDB is an in-memory database with multi-AZ durability and a Redis OSS-compatible API. You still get the useful object types and blazing fast read performance of Redis OSS while opening Redis OSS up as a primary datastore given the addition of persistence. MemoryDB is a great fit for applications like online gaming where performance is critical but you also can't afford to lose data.

Between ElastiCache and MemoryDB, AWS has the Redis OSS services to fit your application needs.

Redis OSS resources and bottlenecks

In the following sections, we'll learn about specific data modeling patterns you can use with Redis OSS in your application. These patterns are recommended for you to get the best performance out of your Redis OSS usage.

But first, let's think about performance more generally. Whenever you're having a performance issue in your application, there will be a bottleneck in some specific resource that is causing the issue. If you're using a database with a traditional hard disk drive, your bottleneck might be reading the physical bits from disk. Moving to a more modern solid-state drive can reduce this bottleneck and improve performance.

Or, if your users are trying to access your application that's located on a different continent, you have a bottleneck related to the speed of light and the absolute distance between your users and your application. Since the former is hard to change, you'll need to consider ways to reduce that distance, perhaps by replicating your data to additional regions.

In Redis OSS, there are three main resources that are going to affect your performance. They are:

- **Memory.** Redis OSS stores all of its data in memory. This means your Redis OSS instance will need significantly more memory as compared to disk than most data stores. Many of the data modeling tips below focus on how to efficiently use memory in Redis OSS.
- **Compute.** The Redis OSS engine that handles data access is single-threaded. This greatly simplifies data access, but it means you should ensure to avoid compute-heavy operations. Most of the Redis OSS operations are designed to be consistently fast at any scale, but there are some operations that slow down with larger objects. The sections below will often talk about the time complexity of an operation to help optimize your Redis OSS usage.
- **Network.** Redis OSS is a centralized cache that your application will communicate with over a network. The Redis OSS engine is so fast that your request will often spend more time in the network than within Redis OSS itself. You should work to optimize network usage as your application interacts with your Redis OSS instance.

Keep these resources in mind as you read through the data modeling patterns below. The sections often mention these resources specifically to help describe the optimizations you can make. You can also review the 'Efficient resource usage in Redis OSS' section below for a holistic overview of resource optimization in Redis OSS.

Core data modeling patterns

Redis OSS can do a lot of fun, incredible things, and we'll see some of the more advanced patterns later on in this book. However, the most common use case for Redis OSS is as a simple cache that's holding data from a system of record. In this section, we'll look at the read-aside caching pattern, which is the most common pattern for using Redis OSS as a cache. In doing so, we'll be using the simple Redis OSS string data type. In subsequent sections, we'll look at more advanced data types and patterns.

Introduction

What is a cache? When should I use one?

What is Redis OSS? Why choose Redis OSS as my in-memory cache?

Benefits of Amazon ElastiCache and Amazon MemoryDB

Redis OSS resources and bottlenecks

Core data modeling patterns

Read-aside caching pattern

Extending the read-aside pattern for changing data

Working with TTLs

Efficient resource usage in Redis OSS

Memory pressure and object sizes

CPU conservation and blocking the main thread

Optimizing network usage

Working with Redis OSS collection types

Hashes

Lists

Sets

Working with advanced data types

Sorted sets

HyperLogLog

Working with multiple operations

Pipelining

Transactions

Scripting

Advanced patterns

Scaling Redis OSS

UNLINK instead of DEL

Locking with Redis OSS

Saving memory with small hashes

Negative result caching

Conclusion

Read-aside caching pattern

The read-aside caching pattern is the canonical caching use case. In this pattern, your application code will first try to retrieve an object, such as a user profile or a product, from your cache. If the object does not exist in the cache, your application will fall back to retrieving it from the system of record, such as your database. Once it has retrieved the object, it will then store it in the cache for future use.

Your code to implement this might look something as follows:

```
import redis
import json

client = redis.Redis(host='localhost', port=6379, db=0)

def get_user(user_id):
    # Try to retrieve the user from the cache
    user = json.loads(client.get(f'user:{user_id}'))

    # If the user is not in the cache, retrieve it from the
database
    if user is None:
        user = retrieve_user_from_database(user_id)
        client.set(f'user:{user_id}', json.dumps(user))

    return user
```

The great thing about the read-aside pattern is in its simplicity and ubiquity. You can implement the read-aside functionality with a few lines of code in your service's `get_user` function, and all consumers of the service will benefit from the upgrade.

Further, this pattern can be used broadly across a number of use cases. Almost any object that is frequently accessed and fairly static can be cached -- user profiles, product details, social media posts, and more. Not only will this reduce latency for your end users, but it will also reduce the load on your system of record.

Notice that, even though our User is a complex object, we're storing it as a simple string in Redis OSS. We're using `json.dumps()` to serialize the object to a string before storing it in Redis OSS, and we'll use `json.loads()` to deserialize it back to an object when we retrieve it from Redis OSS. This is a common pattern in Redis OSS, particularly when you'll always be working on the full object rather than a subset of the object. We'll see more advanced patterns for working with complex objects in subsequent sections.

Introduction

What is a cache? When should I use one?

What is Redis OSS? Why choose Redis OSS as my in-memory cache?

Benefits of Amazon ElastiCache and Amazon MemoryDB

Redis OSS resources and bottlenecks

Core data modeling patterns

Read-aside caching pattern

Extending the read-aside pattern for changing data

Working with TTLs

Efficient resource usage in Redis OSS

Memory pressure and object sizes

CPU conservation and blocking the main thread

Optimizing network usage

Working with Redis OSS collection types

Hashes

Lists

Sets

Working with advanced data types

Sorted sets

HyperLogLog

Working with multiple operations

Pipelining

Transactions

Scripting

Advanced patterns

Scaling Redis OSS

UNLINK instead of DEL

Locking with Redis OSS

Saving memory with small hashes

Negative result caching

Conclusion

Extending the read-aside pattern for changing data

The read-aside pattern is great, but the previous example glossed over some complexities. It mentioned that the read-aside pattern works for fairly static data, but that may not describe your data. Maybe it's true for product descriptions, but there are other objects that are more dynamic. These objects are still cacheable, but you need to be more careful about how you handle them.

It's often said that cache invalidation is one of the two hard problems in computer science, but that's what you'll need to do with these dynamic objects. When the object changes, you'll need to make a corresponding change in your cache.

One way you can handle this is by explicitly removing the item from your cache when it's updated in your application. Redis OSS provides the `DEL` command to remove items from your cache. You can add this to your `update_user` function as follows:

```
def update_user(user_id, new_user):
    # Update the user in the database
    update_user_in_database(user_id, new_user)

    # Remove the user from the cache
    client.delete(f'user:{user_id}')
```

Now, the user item will be removed from the cache whenever it's updated in the database. Here we see the benefits of our read-aside pattern -- whether the item does not exist in the cache because it was never there or because it was removed, the application will fall back to the system of record to retrieve the item the next time it's requested.

There are two improvements we can make to this implementation. The first is purely in our code structure to avoid mistakes. Notice that we've been using a key pattern of `user:${userId}`. When that key access is limited to a single place in our code -- the `get_user` function -- it was fine to hardcode the key pattern. However, now that we're using it in multiple places, we should abstract it into a function to avoid mistakes in our code.

You could do this as follows:

```
def get_user_key(user_id):
    return f'user:{user_id}'

def update_user(user_id, new_user):
    # Update the user in the database
    update_user_in_database(user_id, new_user)

    # Remove the user from the cache
    client.delete(get_user_key(user_id))
```

Introduction
What is a cache? When should I use one?
What is Redis OSS? Why choose Redis OSS as my in-memory cache?
Benefits of Amazon ElastiCache and Amazon MemoryDB
Redis OSS resources and bottlenecks
Core data modeling patterns
Read-aside caching pattern
Extending the read-aside pattern for changing data
Working with TTLs
Efficient resource usage in Redis OSS
Memory pressure and object sizes
CPU conservation and blocking the main thread
Optimizing network usage
Working with Redis OSS collection types
Hashes
Lists
Sets
Working with advanced data types
Sorted sets
HyperLogLog
Working with multiple operations
Pipelining
Transactions
Scripting
Advanced patterns
Scaling Redis OSS
UNLINK instead of DEL
Locking with Redis OSS
Saving memory with small hashes
Negative result caching
Conclusion

In our updated example, we have a `get_user_key` function that returns our key pattern. We can use that function within `update_user` and `get_user` when we need to access the key pattern.

A second change we could make is in how we handle the write path. In our `update_user` example, we're removing the user from the cache altogether when it is updated. However, in most cases, a recently updated item will be requested again soon. Rather than deleting our user record from the cache, we could refresh the cache with the new user data. This would allow us to avoid the latency of a cache miss the next time the user is requested.

Our updated code would look as follows:

```python
def update_user(user_id, new_user):
    # Update the user in the database
    update_user_in_database(user_id, new_user)

    # Refresh the user in the cache
    user = json.dumps(new_user)
    client.set(get_user_key(user_id), user)
```

In this updated example, we're setting the user in the cache with the new user data. This will ensure that the next time the user is requested, it will be available in the cache.

This pattern won't work for all workflows. If you're retrieving this record from an API that you don't control rather that from your own system of record, you may not know when all updates occur. Further, if the object can be updated in multiple different places, it can be difficult to know how and when to update. In those situations, you might opt for a simpler solution. You should consider your needs and the tradeoffs of each approach.

Introduction

What is a cache? When should I use one?

What is Redis OSS? Why choose Redis OSS as my in-memory cache?

Benefits of Amazon ElastiCache and Amazon MemoryDB

Redis OSS resources and bottlenecks

Core data modeling patterns

Read-aside caching pattern

Extending the read-aside pattern for changing data

Working with TTLs

Efficient resource usage in Redis OSS

Memory pressure and object sizes

CPU conservation and blocking the main thread

Optimizing network usage

Working with Redis OSS collection types

Hashes

Lists

Sets

Working with advanced data types

Sorted sets

HyperLogLog

Working with multiple operations

Pipelining

Transactions

Scripting

Advanced patterns

Scaling Redis OSS

UNLINK instead of DEL

Locking with Redis OSS

Saving memory with small hashes

Negative result caching

Conclusion

Working with TTLs

In the previous section, we saw explicit deletion or refresh of items as they changed. However, there's another way to handle changing data in your cache: using time-to-live (TTL) values on your keys.

When setting a key in Redis OSS, you can set a TTL on the key. That TTL indicates when Redis OSS should expire that key. Once that time has passed, Redis OSS will automatically remove the key from the cache.

When setting the value of a string key, you can set the TTL at the same time by using the `EX` or `PX` options. The `EX` option sets the TTL in seconds, while the `PX` option sets the TTL in milliseconds. For example, you can set a TTL of 60 seconds on a key as follows:

```
client = redis.Redis(host='localhost', port=6379, db=0)
client.set('my_key', 'my_value', ex=60)
```

For more advanced data types, you'll need to use the `EXPIRE` command to set the TTL after the key has been set. For example, to set a TTL on a daily HyperLogLog to expire after 24 hours, you can use the following:

```
client = redis.Redis(host='localhost', port=6379, db=0)
key = 'users:20240601'
client.pfadd(key, '1234')
client.expire(key, 86400)
```

A TTL is a powerful cache tool to use, and it works in a variety of situations.

For data that is changing somewhat regularly and is out of your control, a TTL can be a nice way to regularly refresh the state of that data. You can prevent overloading your downstream system while maintaining some level of freshness by relying on Redis OSS to expire older data.

For data that is less frequently changing, you can still use a TTL to maintain memory usage of your data. Certain items like social media posts or breaking news articles may be hot for a few hours, but after that, they're rarely accessed. You can set a TTL on these items to ensure that they're removed from the cache after a certain period of time. This can reduce the amount of memory needed for your cache.

Finally, a TTL can be used to reduce load on your database, even if you know the data is likely to be stale. Imagine a high-traffic social network site like Reddit. You can cache the number of likes, the top-performing comments, and other metadata for a post, but this may not help if you're doing explicit cache busts on every update. A popular Reddit post will be updated many times per second. You can alleviate pressure on your database by setting a brief TTL of a few seconds. This will allow you to serve the same, slightly stale, data to multiple users without hitting your database. As the data expires, it can be refreshed to the cache by the next user that requests it.

Efficient resource usage in Redis OSS

Now that we've seen the basic patterns for working with Redis OSS, let's go back and take a deeper look at the resources used in Redis OSS. Optimizing these resources is crucial for performance in Redis OSS. These resources will be discussed regularly in the more advanced data modeling sections that follow.

Introduction
What is a cache? When should I use one?
What is Redis OSS? Why choose Redis OSS as my in-memory cache?
Benefits of Amazon ElastiCache and Amazon MemoryDB
Redis OSS resources and bottlenecks
Core data modeling patterns
Read-aside caching pattern
Extending the read-aside pattern for changing data
Working with TTLs
Efficient resource usage in Redis OSS
Memory pressure and object sizes
CPU conservation and blocking the main thread
Optimizing network usage
Working with Redis OSS collection types
Hashes
Lists
Sets
Working with advanced data types
Sorted sets
HyperLogLog
Working with multiple operations
Pipelining
Transactions
Scripting
Advanced patterns
Scaling Redis OSS
UNLINK instead of DEL
Locking with Redis OSS
Saving memory with small hashes
Negative result caching
Conclusion

Memory pressure and object sizes

When you're sizing your ElastiCache cluster, the available memory is likely to be the most important choice as you pick your instance size. As we've seen, Redis OSS stores its data in memory by default, so the available memory has to exceed your desired working set.

The good news is that high memory usage is a straightforward technical problem to solve, as you can increase the size or number of instances in your Redis OSS cluster to access more memory. The largest instances in ElastiCache have hundreds of GiB of memory, and you can have an ElastiCache cluster with up to 500 shards. Few use cases will need more memory than that, and you can even set up multiple clusters if you do.

While the scaling problem of memory is solvable, you don't necessarily want to solve it solely by increasing your memory capacity. After all, increasing the number and size of your instances will increase your bill correspondingly. Instead, think of ways to reduce your memory usage.

The easiest way to reduce your memory usage is to reduce the size of the data you store in Redis OSS. For your object keys, you can abbreviate the key length. It's not uncommon to have Redis OSS clusters with millions of keys and long, descriptive key names. A simple change such as going from `userprofiles:${userId}` with `up:${userId}` will save 10 bytes per key, or 10 MB per million objects.

Shortening your key names will only get you so far, and you don't want to sacrifice readability and debuggability for memory. Another, more effective approach for conserving memory is to reduce the size of your object values. This can be by stripping extraneous data from the values you're storing. If you don't need the extra attributes for your specific caching use case, don't store it in your cache.

Even better, you can compress the value before storing it in Redis OSS. At the cost of a bit of CPU when writing and reading your data, you can significantly reduce the size of data stored in Redis OSS memory.

Remember that compression's benefit depends on your dataset. Do some testing with different compression libraries to see how it affects object size and CPU consumption for your application.

Another way to reduce Redis OSS memory usage is to remove items from your cache when they are unused. You can handle this manually via Redis OSS commands like DEL, but an easier approach is the TTL approach that we saw in the previous section. You can set TTLs on your keys to indicate when Redis OSS should expire them out. Setting a realistic TTL can help with not only ensuring data freshness but also in reducing your required memory footprint.

If you don't handle your memory well and start to push the limits of available memory, Redis OSS will automatically remove items for you. This is done according to the eviction policy you set on your Redis OSS cluster, which can be based on factors such as how often or how recently keys have been accessed or whether the key has a TTL configured. You can even choose not to evict keys when memory limits are reached, which will result in blocking writes to Redis OSS until the necessary memory is available.

When using ElastiCache, you can choose instance types that allow for data tiering. With data tiering, ElastiCache will smartly decide whether to store your item values in memory or on local SSD drives. SSD drives provide slightly slower performance than memory but at a much lower price point. If you have a long tail of less frequently accessed cache data and don't mind the additional latency in reading from SSDs, data tiering can alleviate your memory usage concerns.

Introduction
What is a cache? When should I use one?
What is Redis OSS? Why choose Redis OSS as my in-memory cache?
Benefits of Amazon ElastiCache and Amazon MemoryDB
Redis OSS resources and bottlenecks
Core data modeling patterns
Read-aside caching pattern
Extending the read-aside pattern for changing data
Working with TTLs
Efficient resource usage in Redis OSS
Memory pressure and object sizes
CPU conservation and blocking the main thread
Optimizing network usage
Working with Redis OSS collection types
Hashes
Lists
Sets
Working with advanced data types
Sorted sets
HyperLogLog
Working with multiple operations
Pipelining
Transactions
Scripting
Advanced patterns
Scaling Redis OSS
UNLINK instead of DEL
Locking with Redis OSS
Saving memory with small hashes
Negative result caching
Conclusion

CPU conservation and blocking the main thread

A second, less understood, potential bottleneck in Redis OSS is the CPU.

If you have a larger Redis OSS instance that's experiencing high or variable latency, you might dismiss CPU usage as a source of the problem when you look at the CPU utilization metric. Many larger Redis OSS instances show low single-digit CPU usage. However, this could be hiding a CPU problem. The Redis OSS engine that reads and writes data is single-threaded. If you have a Redis OSS instance with 16 vCPUs, your main thread could be at 100% utilization while your overall CPU utilization would show CPU utilization under 7%!

Thus, CPU usage in Redis OSS is about two things: (1) monitoring the proper metrics to understand when you have a problem, and (2) using Redis OSS in a way to avoid having CPU problems.

To monitor your CPU performance, you should focus primarily on the core that the Redis OSS engine is using. If you are using Amazon ElastiCache, you can use the `EngineCPUUtilization` for this metric. You can set CloudWatch alarms on this metric to warn you of excessive CPU consumption.

Note that, in addition to easily displaying the usage of the engine CPU utilization, ElastiCache also helps conserve CPU usage in other ways. ElastiCache has introduced a number of improvements to use the other CPUs on your ElastiCache instance for other non-core Redis OSS work. This includes not only the background Redis OSS tasks but also TLS termination, network processing, and more. The result is that the Redis OSS core is dedicated to as much pure Redis OSS work as possible without handling other areas of the stack.

Beyond monitoring your CPU usage, you should use Redis OSS in a way that does not cause CPU issues. The Redis OSS documentation includes the Big-O notation of every Redis OSS operation. From that, you can see that the Redis OSS SET command has a time complexity of O(1), indicating that it's a constant-time operation. Many common Redis OSS commands have O(1) time complexity, which make them attractive as your application scales.

Conversely, there are other operations that have less-favorable time complexity. Some operations, like LINDEX to find an element within a list, have O(N) time complexity, where N is the number of elements to traverse before finding the relevant element. Meanwhile others, like the SMEMBERS command to return all members in a set or the dreaded KEYS operation to search all keys in your Redis OSS instance have time complexity of O(N), which can result in wildly different response times depending on the number of elements being acted upon.

You shouldn't let time complexity be the sole consideration in using a command. Redis OSS is still quite fast, even on most O(N) operations, and advanced features like list operations and sorted sets are part of the reason developers love Redis OSS so much. Rather, use this information to guide your usage based on your situation, application, and user needs. If you start to see higher CPU usage, you can use this information to understand how to reduce the strain on your Redis OSS cluster.

Introduction

What is a cache? When should I use one?

What is Redis OSS? Why choose Redis OSS as my in-memory cache?

Benefits of Amazon ElastiCache and Amazon MemoryDB

Redis OSS resources and bottlenecks

Core data modeling patterns

Read-aside caching pattern

Extending the read-aside pattern for changing data

Working with TTLs

Efficient resource usage in Redis OSS

Memory pressure and object sizes

CPU conservation and blocking the main thread

Optimizing network usage

Working with Redis OSS collection types

Hashes

Lists

Sets

Working with advanced data types

Sorted sets

HyperLogLog

Working with multiple operations

Pipelining

Transactions

Scripting

Advanced patterns

Scaling Redis OSS

UNLINK instead of DEL

Locking with Redis OSS

Saving memory with small hashes

Negative result caching

Conclusion

Optimizing network usage

The last resource to consider is network. Redis OSS is a centralized cache, rather than a cache that's local to your application instances. Accordingly, you'll need to initialize a connection over a network and send requests back and forth. Failing to consider the network can significantly increase the latency of your Redis OSS requests.

First, ensure you're optimizing the connection itself. You'll be communicating with Redis OSS over TCP, which involves setting up a connection via the three-way handshake. If you encrypt your communication via TLS, this involves additional round-trip requests to be established. You should reuse your connection across multiple requests if possible to avoid this additional traffic as you communicate with your Redis OSS cluster.

In addition to the additional latency of the TCP connection initialization and TLS configuration, setting up and tearing down connections can also consume CPU on your Redis OSS cluster. Fortunately, if you're using the enhanced I/O features of the new ElastiCache instance types, these operations will be offloaded to the non-core CPU to reduce strain on the Redis OSS engine.

You should also consider the specifics of your infrastructure topology when working with your ElastiCache instances. Keeping your application compute in the same availability zone as your Redis OSS instance will reduce latency by avoiding requests across datacenters. Further, you'll reduce cross-AZ networking costs by staying in the same AZ where possible.

Aside from connection management, you should also consider how to reduce the round trips to your Redis OSS instance. A common pattern is to perform multiple actions in Redis OSS as part of a particular workflow. For example, you may need to store a blob of data, increment a count, and add a value into a set or a HyperLogLog. Rather than making three separate round trips to Redis OSS, you can pipeline these three commands together in a single request to Redis OSS.

With these considerations in mind, let's continue our look at data modeling in Redis OSS.

Working with Redis OSS collection types

So far, we've used the simple string data type in Redis OSS to store our objects. However, much of the joy of using Redis OSS comes in its more advanced data types. In this section, we'll look at the core Redis OSS collections -- hashes, lists, and sets. For each type, we'll look at examples of when you can use them in your application. In later sections, we'll look at even more advanced data types like sorted sets and HyperLogLogs.

Introduction

What is a cache? When should I use one?

What is Redis OSS? Why choose Redis OSS as my in-memory cache?

Benefits of Amazon ElastiCache and Amazon MemoryDB

Redis OSS resources and bottlenecks

Core data modeling patterns

Read-aside caching pattern

Extending the read-aside pattern for changing data

Working with TTLs

Efficient resource usage in Redis OSS

Memory pressure and object sizes

CPU conservation and blocking the main thread

Optimizing network usage

Working with Redis OSS collection types

Hashes

Lists

Sets

Working with advanced data types

Sorted sets

HyperLogLog

Working with multiple operations

Pipelining

Transactions

Scripting

Advanced patterns

Scaling Redis OSS

UNLINK instead of DEL

Locking with Redis OSS

Saving memory with small hashes

Negative result caching

Conclusion

Hashes

The first collection type we'll look at is the hash. A hash is a collection of key-value pairs. It's similar to a Python dictionary or a JavaScript object. You can use hashes to store a collection of related fields, such as a user profile that has name, email address, and profile picture fields, or a social media post that has the post content along with data on likes, views, and shares.

You'll notice that there is overlap in the examples we've given for hashes and strings. In many cases, you can use either a hash or a string to store your data. The choice between the two comes down to a few factors:

- When *reading* the item, are you using the whole item or just a subset? If you're using the whole item, a string is likely the right choice. If you're using a subset, a hash may be a better fit. This is particularly true when one element is much larger than the others and is not frequently accessed.
- When *updating* the item, are you updating the whole item or just a subset? If you're updating the whole item, a string is likely the right choice as you can simply overwrite the current existing value. If you're updating a subset, a hash may be a better fit. This is particularly true when you have concurrent updates to the item.

Imagine our social media post example. The actual content of the post is unlikely to change very frequently, but you may be often updating the likes and views. If you used a string value, you would need to read the current value, update the likes, and then write the new value back to Redis OSS. If other clients are operating on the value at the same time, you could overwrite each other.

Conversely, if you used a hash, you could use the `HINCRBY` command to increment the likes and views without needing to read the current value. The Redis OSS increment commands are atomic, so you won't have to worry about concurrent updates.

Redis OSS provides a number of helpful operations on hashes, and all of the Redis OSS operations on hashes start with an 'H'. You can use `HSET` to set one or more fields in a hash, `HGET` to retrieve a field from a hash, `HMGET` to read multiple values in a hash, and `HINCRBY` to increment a field in a hash. These operations can build a variety of features in your application.

Hashes are a great fit for a variety of use cases for the same reason that hashes are popular in your application code. They're a great way to store a collection of related fields, and they provide constant time access to each field. In the Advanced Tips section, you can even find a unique trick where using hashes can save significant memory over using individual strings.

Lists

The second Redis OSS collection type we'll look at is the list. A list is an ordered collection of items. Common use cases for lists include queued jobs or a timeline of events.

The important aspect of Redis OSS lists is *ordered* -- you want to be working with items that have some inherent order that affects how you access them. A queue is a great example of this, as you'll be adding elements to one end of a list and processing them on the other end. Likewise, a timeline is great for lists as you can push new events into a list and then read the most recent events from the front of the list.

To understand the importance of ordering in using lists, it helps to know a bit about how Redis OSS implements lists. Redis OSS lists are *linked* lists rather than arrays. Whereas arrays store themselves in a contiguous block of memory, linked lists do not. Rather, each item in a linked list has a pointer to the next item in the list.

Introduction

What is a cache? When should I use one?

What is Redis OSS? Why choose Redis OSS as my in-memory cache?

Benefits of Amazon ElastiCache and Amazon MemoryDB

Redis OSS resources and bottlenecks

Core data modeling patterns

Read-aside caching pattern

Extending the read-aside pattern for changing data

Working with TTLs

Efficient resource usage in Redis OSS

Memory pressure and object sizes

CPU conservation and blocking the main thread

Optimizing network usage

Working with Redis OSS collection types

Hashes

Lists

Sets

Working with advanced data types

Sorted sets

HyperLogLog

Working with multiple operations

Pipelining

Transactions

Scripting

Advanced patterns

Scaling Redis OSS

UNLINK instead of DEL

Locking with Redis OSS

Saving memory with small hashes

Negative result caching

Conclusion

A linked list makes sense for Redis OSS. Redis OSS doesn't need to over-allocate space for an array to allow for growth, and it doesn't need to shift an entire array to a new location if it exceeds the previously allocated memory. Further, adding or removing an item from the front or back of a list is a constant time operation, regardless of the size of the list. For caching use cases, this efficiency is important.

However, it does mean that accessing items in the middle of a list is not as efficient. If you need to access items in the middle of a list, you may want to consider a different data type.

When working with Redis OSS lists, there are two sets of terminology you should know. First, Redis OSS uses the terms 'push' and 'pop' to refer to adding and removing items from the list. Pushing an item onto the list adds the item to the list, such as when you're enqueuing a new job. Then, a client that wants to process a job would pop an item to remove it from the list.

Second, Redis OSS uses the terms 'left' and 'right' to refer to the ends of the list. You should visualize a Redis OSS list as a line, with the left end being the front of the line and the right end being the back of the line. You can push and pop items from either end of the line.

The core Redis OSS list operations use this terminology. You can use `LPUSH` to push an item onto the left end of the list and `RPUSH` to push an item onto the right end of the list. Similarly, you can use `LPOP` to pop an item from the left end of the list, and `RPOP` to pop an item from the right end of the list. Again, because these operations are working at the ends of a list, they are efficient, constant time operations. Redis OSS does have a `LINDEX` command to access items in the middle of a list, but it's an `O(N)` operation, where `N` is the number of items to traverse to reach the desired item.

Let's look at a brief example of how to implement a queue using a Redis OSS list. For most queues, you follow a 'first-in, first-out' (FIFO) pattern. You'll push items onto one side of the list and pop items from the other side.

You can implement this as follows:

```
QUEUE_NAME = ‘jobs’

def enqueue_job(job):
    client.rpush(QUEUE_NAME, json.dumps(job))

def dequeue_job():
    return json.loads(client.lpop(QUEUE_NAME))
```

Redis OSS makes this pretty simple -- we push a serialized job to the right side of the list with `RPUSH` and pop a job from the left side of the list with `LPOP`. This is a simple, efficient way to implement a queue in Redis OSS. Additionally, you can modify this into a 'last-in, first-out' (LIFO) pattern by pushing to and popping from to the same side of the list.

With a queue, you may have a fleet of workers that are processing the queue. At times, the workers available may work through the queue so quickly that there are no elements in the queue. In this case, the workers will get empty results as they try to pop items from the queue. This can be inefficient, as the workers will be making many requests to Redis OSS that result in no work.

Introduction

What is a cache? When should I use one?

What is Redis OSS? Why choose Redis OSS as my in-memory cache?

Benefits of Amazon ElastiCache and Amazon MemoryDB

Redis OSS resources and bottlenecks

Core data modeling patterns

Read-aside caching pattern

Extending the read-aside pattern for changing data

Working with TTLs

Efficient resource usage in Redis OSS

Memory pressure and object sizes

CPU conservation and blocking the main thread

Optimizing network usage

Working with Redis OSS collection types

Hashes

Lists

Sets

Working with advanced data types

Sorted sets

HyperLogLog

Working with multiple operations

Pipelining

Transactions

Scripting

Advanced patterns

Scaling Redis OSS

UNLINK instead of DEL

Locking with Redis OSS

Saving memory with small hashes

Negative result caching

Conclusion

Fortunately, Redis OSS provides a blocking version of the pop operation. You can use `BLPOP` to block until an item is available in the queue. If there are items in the list, Redis OSS will return immediately. If not, it will hold the connection until a new element appears in the list, then return it to the client. You can also configure a timeout for the blocking operation, so that the client will return after a certain period of time if no item is available.

Updating our previous example to use `BLPOP` would look as follows:

```
def dequeue_job_blocking():
    result  = client.blpop(QUEUE_NAME, timeout=30)
    if result:
        _, job = result
        return json.loads(job)
    else:
        return None
```

Note that while this is a 'blocking' operation, it's only blocking for that particular Redis OSS client. It's not blocking the engine thread on the Redis OSS server. Redis OSS will keep processing other requests from other clients while the client is blocked. This is a great way to quickly process your queue without overloading your Redis OSS server.

One other neat thing about the blocking pop operations is that you can provide multiple lists to block on. Redis OSS will pop from the first list that has an item available. This can be a great way to implement a priority queue, where you have multiple queues for different priority levels and you want to process the highest priority items first. Your queue workers could list the queues in order of priority in their blocking pop operation, and Redis OSS will return the first item available from the highest priority queue.

Our updated example would look as follows:

```
HIGH_PRIORITY_QUEUE = ‘high_priority_jobs’
MEDIUM_PRIORITY_QUEUE = ‘medium_priority_jobs’
LOW_PRIORITY_QUEUE = ‘low_priority_jobs’

def dequeue_job_blocking():
    _, job = client.blpop([HIGH_PRIORITY_QUEUE, MEDIUM_
PRIORITY_QUEUE, LOW_PRIORITY_QUEUE], timeout=30)
    return json.loads(job)
```

Introduction

What is a cache? When should I use one?

What is Redis OSS? Why choose Redis OSS as my in-memory cache?

Benefits of Amazon ElastiCache and Amazon MemoryDB

Redis OSS resources and bottlenecks

Core data modeling patterns

Read-aside caching pattern

Extending the read-aside pattern for changing data

Working with TTLs

Efficient resource usage in Redis OSS

Memory pressure and object sizes

CPU conservation and blocking the main thread

Optimizing network usage

Working with Redis OSS collection types

Hashes

Lists

Sets

Working with advanced data types

Sorted sets

HyperLogLog

Working with multiple operations

Pipelining

Transactions

Scripting

Advanced patterns

Scaling Redis OSS

UNLINK instead of DEL

Locking with Redis OSS

Saving memory with small hashes

Negative result caching

Conclusion

Finally, we should note that Redis OSS lists are not just for queues. They're also great for timelines, where you're adding new events to the list and reading the most recent events from the list. You can use the same `LPUSH` operation to add items to the list, and you can use the `LRANGE` command to read a range of items from the list.

We could implement this as follows:

```
def user_timeline_key(user_id):
    return f'timeline:{user_id}'

def add_event(user_id, event):
    client.lpush(user_timeline_key(user_id), json.dumps(event))

def get_recent_events_for_user(user_id, count):
    events = client.lrange(user_timeline_key(user_id), 0, count
- 1)
    return [json.loads(event) for event in events]
```

When a new event comes in, we push it to the left side of the list with `LPUSH`. When we want to read the most recent events, we use `LRANGE` to read a range of items from the list.

Note that the `LRANGE` operation has time complexity of `O(S+N)`, where `S` is distance that the start of the range is from the head or tail of the list and `N` is the number of elements to return. If you're doing a "most recent" timeline operation, this is a constant-time operation as you'll be reading from the head of the list. However, if you're pulling items from the middle of the list, it can be a more expensive operation.

Lists are a powerful tool in Redis OSS as long as you are working with ordered events that require access to the ends of the list. They're a great fit for queues and timelines, and they provide efficient, constant time operations for adding and removing items from the list.

Introduction

What is a cache? When should I use one?

What is Redis OSS? Why choose Redis OSS as my in-memory cache?

Benefits of Amazon ElastiCache and Amazon MemoryDB

Redis OSS resources and bottlenecks

Core data modeling patterns

Read-aside caching pattern

Extending the read-aside pattern for changing data

Working with TTLs

Efficient resource usage in Redis OSS

Memory pressure and object sizes

CPU conservation and blocking the main thread

Optimizing network usage

Working with Redis OSS collection types

Hashes

Lists

Sets

Working with advanced data types

Sorted sets

HyperLogLog

Working with multiple operations

Pipelining

Transactions

Scripting

Advanced patterns

Scaling Redis OSS

UNLINK instead of DEL

Locking with Redis OSS

Saving memory with small hashes

Negative result caching

Conclusion

Sets

The final core collection data type in Redis OSS is the set. A set is an unordered collection of unique items. The uniqueness is the key attribute here -- you'll want to focus on sets where you either want to ensure uniqueness when adding items or you want to prevent duplicate counts when reading items.

All the set operations start with the letter `S` -- `SADD` to add an element to a set, `SPOP` to remove and return a random element from a set, and `SCARD` to return the number of elements (the cardinality) of a set.

One use case for a set is to handle idempotency to prevent duplicate processing. When performing the `SADD` operation to add an item to a set, Redis OSS will return `1` if the item was added to the set and `0` if the item was already in the set. You can use this to ensure that you only process an item once, even if it's requested multiple times.

You can implement this as follows:

```
def process_item(item_id):
    if client.sadd('processed_items', item_id):
        process_item(item_id)
    else:
        pass
```

You can use sets for a number of use cases, including tracking the count of unique visitors to a page or the users that have voted in a poll today. For the core set operations -- `SADD`, `SPOP`, and `SCARD`, you get constant time complexity, making sets a great fit for high-performance use cases.

There are some advanced set operations that you can use to perform set operations on multiple sets. For example, you can use `SINTER` to find the intersection of multiple sets, `SUNION` to find the union of multiple sets, and `SDIFF` to find the difference of multiple sets. These operations can be useful for a variety of use cases, such as finding the intersection between who is following two different users on a social media platform or finding the union of users that have visited two different pages on a website. However, take note that these operations have less favorable time complexity than the core set operations. If you use them for sets with a large number of elements, you may see a performance hit.

Working with advanced data types

As we move beyond the standard data types, Redis OSS has some more unique data types. These data types aren't included in most other caches or even other database systems. However, they can be incredibly useful for certain use cases. In this section, we'll look at two of these advanced data types -- sorted sets and HyperLogLogs.

Introduction
What is a cache? When should I use one?
What is Redis OSS? Why choose Redis OSS as my in-memory cache?
Benefits of Amazon ElastiCache and Amazon MemoryDB
Redis OSS resources and bottlenecks
Core data modeling patterns
Read-aside caching pattern
Extending the read-aside pattern for changing data
Working with TTLs
Efficient resource usage in Redis OSS
Memory pressure and object sizes
CPU conservation and blocking the main thread
Optimizing network usage
Working with Redis OSS collection types
Hashes
Lists
Sets
Working with advanced data types
Sorted sets
HyperLogLog
Working with multiple operations
Pipelining
Transactions
Scripting
Advanced patterns
Scaling Redis OSS
UNLINK instead of DEL
Locking with Redis OSS
Saving memory with small hashes
Negative result caching
Conclusion

Sorted sets

In the previous section, we looked at list data type, which provides nice ordering semantics, along with the set data type, which provides uniqueness. But what if you want both -- a unique collection of items that are ordered along some metric? Further, what if you could update the ordering of individual items in the collection?

Enter the sorted set. The sorted set provides a unique collection of items that are ordered along a client-specified 'score'. Sorted sets are one of the most loved parts about Redis OSS as they vastly simplify certain types of problems.

The most common example use case for a sorted set is a leaderboard. Imagine you have a multiplayer game where users can earn points. As the users play, they continue to rack up points. You want to show what place they're currently in, as well as the users that are immediately above and below them. All of this can be done with a sorted set.

Continuing with Redis OSS's theme of using a prefix to denote the type of data, sorted sets use the prefix `Z`. The `ZADD` command is used to add an item to a sorted set with the given score. If that item is already in the set, the score will be updated. This is the command you would use to update a user's score in the leaderboard as they gain points.

To get the user's current place in the leaderboard, you can use the `ZRANK` command. This will return the 0-based index of the user in the sorted set.

Then, if you want to retrieve the users that are immediately above and below the user in the leaderboard, you can use the `ZRANGE` command. This will return a range of items from the sorted set, which you can use to find the users that are immediately above and below the user.

The basic implementation of a leaderboard would look as follows:

```python
def set_points_for_user(user_id, points):
    client.zadd('leaderboard', {user_id: points})

def get_user_place_in_leaderboard(user_id):
    return client.zrank('leaderboard', user_id)

def get_users_around_user_in_leaderboard(user_id):
    place = get_user_place_in_leaderboard(user_id)
    return client.zrange('leaderboard', place - 1, place + 1)
```

One of the interesting things about sorted sets is their performance profile. We saw that lists have nice, constant time characteristics for operating on the ends of lists but slower, `O(N)` time characteristics for operating on elements in the middle of the list. Sorted sets, on the other hand, are able to efficiently operate on any item in the list. The `ZADD` operation is `O(log(N))`. Further, finding the rank of a given element with `ZRANK` is `O(log(N))` as well. Even the `ZRANGE` operation is `O(log(N) + M)`, where `M` is the number of elements to return. This makes sorted sets a great fit even for large, fast-moving leaderboards.

Like with sets, there are some operations with less favorable time complexity. All the set-like operations that compare two or more sets -- operations like `ZINTER` and `ZUNION` -- are going to get slower as the size of your sets increase. Use these sparingly and with smaller sorted sets.

Introduction
What is a cache? When should I use one?
What is Redis OSS? Why choose Redis OSS as my in-memory cache?
Benefits of Amazon ElastiCache and Amazon MemoryDB
Redis OSS resources and bottlenecks
Core data modeling patterns
Read-aside caching pattern
Extending the read-aside pattern for changing data
Working with TTLs
Efficient resource usage in Redis OSS
Memory pressure and object sizes
CPU conservation and blocking the main thread
Optimizing network usage
Working with Redis OSS collection types
Hashes
Lists
Sets
Working with advanced data types
Sorted sets
HyperLogLog
Working with multiple operations
Pipelining
Transactions
Scripting
Advanced patterns
Scaling Redis OSS
UNLINK instead of DEL
Locking with Redis OSS
Saving memory with small hashes
Negative result caching
Conclusion

While leaderboards are the canonical use case for sorted sets, they can be used for a variety of other use cases. We can even use them to augment some of the examples in previous sections.

For example, in the section on Redis OSS lists, we talked about using multiple lists and the `BLPOP` command to implement a priority queue. This works for simple situations where the priority is coarse-grained and items are not updated once they are in a queue. However, sorted sets can give you a more fine-grained priority queue as you explore the entire space of a score range. Further, you can update the score of a job in the queue to change its priority.

Even simple operations like cancelling an item in a queue are easier with sorted sets. With a list, you'd need to iterate through the entire list to find the job to cancel. With a sorted set, the `ZREM` command allows you to quickly remove one or more elements.

We also saw Redis OSS lists used for timelines. Again, this works great for a static timeline of operations. But what if you have a timeline of events that can be updated? If you're showing a social media feed of recent items, you may need to reorder an item if the post is updated. With a sorted set, changing the timestamp score of an element is a simple, fast operation.

HyperLogLog

The second advanced data type we'll look at is the HyperLogLog. The HyperLogLog is unlike most data structures you've seen before.

In some ways, a HyperLogLog overlaps with a set, as it provides a way to store a collection of unique items. However, storing enormous sets can be memory-intensive. A HyperLogLog provides a way to store a collection of unique items with a much smaller memory footprint. To do this, the HyperLogLog is a *probabilistic* data structure. Rather than giving you an exact count of unique items, it gives you an estimate of the number of unique items. This estimate is very close to the actual count, with a given level of error, but it's not exact.

The HyperLogLog is a great fit for use cases where you need to estimate the number of unique items in a large set. For example, you could use a HyperLogLog to estimate the number of unique visitors to a website in a given time. Each time a user visits the website, you would add the user's ID to the HyperLogLog using the `PFADD` command. Then, you can use the `PFCOUNT` command to estimate the number of unique visitors to the website.

If you were doing this with a Redis OSS string and using the `SETBIT` and `BITCOUNT` commands for these counts, you would need to use a single bit for each visitor. This means one million users would require a 125 KB string, and it would scale linearly from there -- two million users would require a 250 KB string, etc. With a HyperLogLog, the maximum size is only 12 KB and can be used to count over *18 quintillion* (2^64) unique items.

Introduction

What is a cache? When should I use one?

What is Redis OSS? Why choose Redis OSS as my in-memory cache?

Benefits of Amazon ElastiCache and Amazon MemoryDB

Redis OSS resources and bottlenecks

Core data modeling patterns

Read-aside caching pattern

Extending the read-aside pattern for changing data

Working with TTLs

Efficient resource usage in Redis OSS

Memory pressure and object sizes

CPU conservation and blocking the main thread

Optimizing network usage

Working with Redis OSS collection types

Hashes

Lists

Sets

Working with advanced data types

Sorted sets

HyperLogLog

Working with multiple operations

Pipelining

Transactions

Scripting

Advanced patterns

Scaling Redis OSS

UNLINK instead of DEL

Locking with Redis OSS

Saving memory with small hashes

Negative result caching

Conclusion

The implementation of this would look as follows:

```
def get_unique_visitor_key_for_date(date):
    return f'unique_visitors:{date}'

def add_user_to_unique_visitors(user_id):
  date = datetime.datetime.now().strftime('%Y-%m-%d')
  altered = client.pfadd(get_unique_visitor_key_for_date(date),
  user_id)
  return altered

def get_unique_visitor_count_for_date(date):
  return client.pfcount(get_unique_visitor_key_for_date(date))
```

There are two main tradeoffs with the HyperLogLog. First, it's a probabilistic data structure, so the count is not exact. With how Redis OSS has implemented the HyperLogLog, the standard error is 0.81%. Second, the HyperLogLog is not as flexible as a set. You can't remove items from a HyperLogLog.

Even when adding an element to a HyperLogLog, you won't always be sure if the element was present previously or not. The `PFADD` operation will return `1` if the HyperLogLog was altered and `0` if it was not altered. The fact that a HyperLogLog was altered by a `PFADD` means that the element definitely did not exist before, but the fact that it was not altered does not mean that the element did exist before. The reason behind this is beyond the scope of this ebook and has to do with how the HyperLogLog is implemented.

The HyperLogLog is powerful, but use it with caution. It's a fast, space-efficient way to estimate the number of unique items, but it's not a perfect fit for all use cases. Consider when you need more accuracy than what the HyperLogLog can provide.

Working with multiple operations

To this point, we've learned the core Redis OSS data types -- what they are, how to use them, and when they're useful. All of this is useful as you consider which data type is right for your use case. In the examples we show, you're usually running a single Redis OSS command.

Yet in many scenarios, you'll need to perform multiple Redis OSS operations in a single workflow to achieve your desired functionality. For example, you may need to both update the score for a user in a sorted set while also updating the score for that same user in the user's cached record. Or, you may need to create a new list item while also setting the TTL for that item.

In this section, we'll look at a few patterns for working with multiple Redis OSS operations in a single workflow. There are three ways to handle this -- pipelining, transactions, and scripting -- and we'll look at each in turn.

Introduction

What is a cache? When should I use one?

What is Redis OSS? Why choose Redis OSS as my in-memory cache?

Benefits of Amazon ElastiCache and Amazon MemoryDB

Redis OSS resources and bottlenecks

Core data modeling patterns

Read-aside caching pattern

Extending the read-aside pattern for changing data

Working with TTLs

Efficient resource usage in Redis OSS

Memory pressure and object sizes

CPU conservation and blocking the main thread

Optimizing network usage

Working with Redis OSS collection types

Hashes

Lists

Sets

Working with advanced data types

Sorted sets

HyperLogLog

Working with multiple operations

Pipelining

Transactions

Scripting

Advanced patterns

Scaling Redis OSS

UNLINK instead of DEL

Locking with Redis OSS

Saving memory with small hashes

Negative result caching

Conclusion

Pipelining

The first way to handle multiple operations in a single workflow is via pipelining (also known as ‘batching’ in some Redis OSS client implementations).

We’ve seen that working with Redis OSS is very fast. Unless you’re working with huge objects, you’re going to be able to handle hundreds of thousands of Redis OSS operations per second. If you’re running multiple Redis OSS commands in a single workflow, the bottleneck isn’t going to be the speed of the Redis OSS server. Rather, it’s going to be the network latency between your application and the Redis OSS server.

Pipelining is a way to reduce the impact of network latency on your Redis OSS operations. With pipelining, you can send multiple commands to Redis OSS in a single request. Redis OSS will process the commands in the order they were received and return the results in the same order. There’s a single round trip for the entire set of commands, rather than a round trip for each command.

Let’s go to our example of updating a user’s score in a sorted set and updating the user’s score in the user’s cached record. We can use pipelining to send both commands in a single request. We can use the pipeline command in your Redis OSS SDK to start the pipeline and the execute command to execute the pipeline.

The implementation of this would look as follows:

```
def update_user_score(user_id, new_score):
    pipeline = client.pipeline()
    pipeline.zadd(‘leaderboard’, user_id, new_score)
    pipeline.hset(f’user:{user_id}’, ‘score’, new_score)
    pipeline.execute()
```

Our two commands -- `ZADD` and `HSET` -- are sent in a single request.

Pipelining is great when you have multiple, independent commands. However, you can’t use pipelining when you have multiple commands *that rely on the results of previous commands*. You aren’t able to, for example, read the result of one command and use it in a subsequent command during a pipeline. For those situations, you’ll need to look at transactions or scripting.

Further, don’t assume that pipelining will give you guarantees like a SQL transaction. The only guarantees you get from pipelining is that the commands you send will be executed in the order you send them. However, it is possible for commands from other clients to be executed between the commands in your pipeline.

Introduction

What is a cache? When should I use one?

What is Redis OSS? Why choose Redis OSS as my in-memory cache?

Benefits of Amazon ElastiCache and Amazon MemoryDB

Redis OSS resources and bottlenecks

Core data modeling patterns

Read-aside caching pattern

Extending the read-aside pattern for changing data

Working with TTLs

Efficient resource usage in Redis OSS

Memory pressure and object sizes

CPU conservation and blocking the main thread

Optimizing network usage

Working with Redis OSS collection types

Hashes

Lists

Sets

Working with advanced data types

Sorted sets

HyperLogLog

Working with multiple operations

Pipelining

Transactions

Scripting

Advanced patterns

Scaling Redis OSS

UNLINK instead of DEL

Locking with Redis OSS

Saving memory with small hashes

Negative result caching

Conclusion

Another good use case for pipelines is to maintain capped collections. In our section on Redis OSS lists, we showed how to maintain a list of recent events for user timelines. However, if we're continually pushing new items to the front of our list, our list will grow without bound. You can use the `LTRIM` command to trim the list to a certain length.

We can combine this together so that when we add a new event to the list, we also trim the list to a certain length. We can use pipelining to send both commands in a single request.

The implementation of this would look as follows:

```
def add_event_to_user_timeline(user_id, event):
    pipeline = client.pipeline()
    pipeline.lpush(user_timeline_key(user_id), json.
dumps(event))
    pipeline.ltrim(user_timeline_key(user_id), 0, 100)
    pipeline.execute()
```

If our list is less than 100 items, the `LTRIM` command will have no effect. If our list is more than 100 items, the `LTRIM` command will trim the list to 100 items and prevent us from using too much memory.

A final common use case of pipelines is to set TTLs for items. In our section on TTLs, we showed how to set a TTL on a key when you set the value of a string key. However, you can't set a TTL when manipulating a collection type. You can use pipelining to add elements to a collection and then set the TTL for the collection in a single request.

Transactions

The second way to handle multiple operations in a single workflow is via transactions. Redis OSS transactions have some similarities to database transactions, but they're not quite the same. By itself, the main functionality that transactions give you over pipelining is the guarantee that multiple commands will be executed atomically. They won't interleave commands from other clients in the middle of your transaction.

This is a nice feature, but it's not *that* helpful for most situations. It is more useful when it's possible that the interleaving commands may conflict with your pipeline/batch. In general, it's a good protection against inconsistency, as it's also possible that one command in the batch would be executed but the other may be dropped if there is a failure. In our pipelining examples above, none of them benefit from the atomicity of a transaction.

The real power of Redis OSS transactions comes with combining them with the `WATCH` command. The `WATCH` command allows you to monitor one or more keys for changes. If any of the keys change before the transaction is executed, the transaction will be aborted. This is powerful -- we now get the powerful 'compare and set' semantics that we're used to from databases.

One good use case for Redis OSS transactions is to help with distributed locking, as discussed in the Advanced Tips section below.

Introduction

What is a cache? When should I use one?

What is Redis OSS? Why choose Redis OSS as my in-memory cache?

Benefits of Amazon ElastiCache and Amazon MemoryDB

Redis OSS resources and bottlenecks

Core data modeling patterns

Read-aside caching pattern

Extending the read-aside pattern for changing data

Working with TTLs

Efficient resource usage in Redis OSS

Memory pressure and object sizes

CPU conservation and blocking the main thread

Optimizing network usage

Working with Redis OSS collection types

Hashes

Lists

Sets

Working with advanced data types

Sorted sets

HyperLogLog

Working with multiple operations

Pipelining

Transactions

Scripting

Advanced patterns

Scaling Redis OSS

UNLINK instead of DEL

Locking with Redis OSS

Saving memory with small hashes

Negative result caching

Conclusion

Scripting

The previous examples have been *client-side* approaches to handling multiple operations in a single workflow. However, client-side approaches come with limitations due to the network latency between the client and the Redis OSS server. You either have the limitation of independent commands in the pipelining scenario, or the separation of read and write commands in the transaction scenario.

This leads us to the third way to handle multiple operations in Redis OSS -- scripting. With scripting, you are writing a script in Lua that is executed on the Redis OSS server. This script is executed atomically, meaning that, once the script starts, no other commands will be executed until the script is finished. This allows you to perform multiple operations in a single workflow without the limitations of the client-side approaches.

Lua scripting avoids some of the complicated workarounds of the other approaches, particularly when you have complex logic in your workflow.

For example, think of a token bucket rate limiting algorithm. You are attempting to limit the number of requests by a particular client in a given time period by giving them a certain number of tokens. These tokens are replenished at a certain rate. Rather than proactively replenishing the token bucket, you can calculate the number of tokens a client should have when they make a request by storing both the current number of tokens and the last time they were replenished. When checking to see if the current request should be allowed, you can check the previous refill time to see if additional tokens should be added to the bucket. Then, you can see if any tokens exist in the bucket for the current request.

For a high-traffic workflow, the race conditions by issuing multiple sequential commands can be problematic. Further, using transactions with the `WATCH` command can result in a lot of contention. You can push this compute logic to the Redis OSS server via Redis OSS scripting.

Introduction

What is a cache? When should I use one?

What is Redis OSS? Why choose Redis OSS as my in-memory cache?

Benefits of Amazon ElastiCache and Amazon MemoryDB

Redis OSS resources and bottlenecks

Core data modeling patterns

Read-aside caching pattern

Extending the read-aside pattern for changing data

Working with TTLs

Efficient resource usage in Redis OSS

Memory pressure and object sizes

CPU conservation and blocking the main thread

Optimizing network usage

Working with Redis OSS collection types

Hashes

Lists

Sets

Working with advanced data types

Sorted sets

HyperLogLog

Working with multiple operations

Pipelining

Transactions

Scripting

Advanced patterns

Scaling Redis OSS

UNLINK instead of DEL

Locking with Redis OSS

Saving memory with small hashes

Negative result caching

Conclusion

The Lua script below shows how this can be implemented:

```
-- Key name for the bucket data
local bucketKey = KEYS[1]
-- Current timestamp
local now = redis.call('TIME')[1] -- Get current time in
seconds
local refillRate = 60 -- Time (in seconds) after which tokens
are refilled
local tokensToAdd = 5 -- Number of tokens added per refill
local maxTokens = 100 -- Max number of tokens in the bucket

-- Fetch current state
local bucket = redis.call('HMGET', bucketKey, 'tokens',
'lastRefill')
local tokens = tonumber(bucket[1])
local lastRefill = tonumber(bucket[2])

-- Refill tokens if necessary
if lastRefill then
    local elapsed = now - lastRefill
    local refills = math.floor(elapsed / refillRate)
    if refills > 0 then
        tokens = math.min(tokens + refills * tokensToAdd,
maxTokens)
        lastRefill = lastRefill + refills * refillRate
        redis.call('HMSET', bucketKey, 'tokens', tokens,
'lastRefill', lastRefill)
    end
else
    -- Initialize if not set
    tokens = maxTokens
    lastRefill = now
    redis.call('HMSET', bucketKey, 'tokens', tokens,
'lastRefill', lastRefill)
end

-- Attempt to take a token
local tokenTaken
if tokens > 0 then
    tokens = tokens - 1
    redis.call('HSET', bucketKey, 'tokens', tokens)
    tokenTaken = true
else
    tokenTaken = false
end

-- Return whether a token was taken and the remaining tokens
return {tokenTaken, tokens}
```

Introduction

What is a cache? When should I use one?

What is Redis OSS? Why choose Redis OSS as my in-memory cache?

Benefits of Amazon ElastiCache and Amazon MemoryDB

Redis OSS resources and bottlenecks

Core data modeling patterns

Read-aside caching pattern

Extending the read-aside pattern for changing data

Working with TTLs

Efficient resource usage in Redis OSS

Memory pressure and object sizes

CPU conservation and blocking the main thread

Optimizing network usage

Working with Redis OSS collection types

Hashes

Lists

Sets

Working with advanced data types

Sorted sets

HyperLogLog

Working with multiple operations

Pipelining

Transactions

Scripting

Advanced patterns

Scaling Redis OSS

UNLINK instead of DEL

Locking with Redis OSS

Saving memory with small hashes

Negative result caching

Conclusion

This is somewhat complex, but let's break it down. First, we take the name of the key given from the command to check. This key has two properties: (1) `tokens` and (2) `lastRefill`. We calculate the time since the last refill. If the time since the last refill is greater than the refill rate, we add tokens to the bucket. Then, we check if there are any tokens in the bucket. If there are, we take one and return `true`. If there are not, we return `false`. We also return the number of tokens remaining in the bucket.

This is an operation that would be difficult to handle without Redis OSS scripting. But be careful with the power provided by Redis OSS scripts. Remember that the Redis OSS server will block as your script executes, so you want to ensure the script is efficient. Use the same principles we learned above to make efficient use of Redis OSS data types and commands.

Advanced patterns

With the basics out of the way, let's look at some advanced tips and patterns in Redis OSS. These tips and patterns are not necessary for every use case, but they can be incredibly useful in the right situation. Further, they help you to build an understanding of how Redis OSS works and how to use it effectively.

Introduction
What is a cache? When should I use one?
What is Redis OSS? Why choose Redis OSS as my in-memory cache?
Benefits of Amazon ElastiCache and Amazon MemoryDB
Redis OSS resources and bottlenecks
Core data modeling patterns
Read-aside caching pattern
Extending the read-aside pattern for changing data
Working with TTLs
Efficient resource usage in Redis OSS
Memory pressure and object sizes
CPU conservation and blocking the main thread
Optimizing network usage
Working with Redis OSS collection types
Hashes
Lists
Sets
Working with advanced data types
Sorted sets
HyperLogLog
Working with multiple operations
Pipelining
Transactions
Scripting
Advanced patterns
Scaling Redis OSS
UNLINK instead of DEL
Locking with Redis OSS
Saving memory with small hashes
Negative result caching
Conclusion

Scaling Redis OSS

As your Redis OSS needs grow, you'll need to think about how to scale your Redis OSS deployment. There are a number of ways to scale Redis OSS, and the right way for you will depend on your specific use case.

The first, and simplest, way to scale Redis OSS is to increase the size of your Redis OSS instance. This is the typical approach if memory is your bottleneck. As your application usage grows, your memory needs will grow accordingly. Increasing the size of your Redis OSS instance is the easiest way to handle this. When using a managed provider like Amazon ElastiCache, scaling your instance size is a simple operation with minimal downtime.

For other situations, increasing your instance size may not work for your needs. It may be because you've reached the maximum size of a Redis OSS instance. A more likely reason is that you're hitting a compute or network bottleneck, rather than a memory bottleneck. In these cases, you'll need to look at more advanced scaling options.

To determine the best pattern for scaling in this situation, it's helpful to understand your application. Specifically, you'll want to know if you have a read-heavy workload or a write-heavy workload. Read-heavy workloads can be scaled by using read replicas, while write-heavy workloads can be scaled by using sharding.

If you don't know your workload, you can look to the CloudWatch metrics for your ElastiCache cluster. The `SetTypeCmds` shows the number of write commands in your cluster, while the `GetTypeCmds` shows the number of read commands. Once you've identified your workload, you can look to the appropriate scaling pattern.

Read replicas are a great way to scale a read-heavy workload. A read replica is a copy of your primary Redis OSS instance that is kept in sync with the primary instance. Data is copied to replicas asynchronously from the primary instance. You can use the read replica to offload read operations from your primary instance. This can be a great way to reduce the load on your primary instance and improve the performance of your read operations.

In ElastiCache, you have "cluster mode disabled" if you are using read replicas with no sharding of your cluster. Your primary instance can have up to five read replicas. When you add replicas, your cluster will have a primary endpoint that you can use for both read and write operations, as well as a single reader endpoint that load balances among the read replicas.

For write-heavy workloads, sharding your cluster is a better fit. When sharding your Redis OSS cluster, you are splitting your dataset across multiple Redis OSS instances. Each instance is responsible for a different portion of the dataset. This alleviates write pressure as different shards can handle writes for different parts of the dataset. You can also use sharding to increase the size of your dataset beyond the maximum size of a single Redis OSS instance.

When you have sharded your ElastiCache cluster, you have 'cluster mode enabled'. You can have up to 500 shards in a cluster, and each shard can have its own read replicas. When working with an ElastiCache cluster with cluster mode enabled, you'll have a single configuration endpoint to connect to. The cluster will handle routing your requests to the appropriate shard, so you won't need to devise your own sharding strategy. To enable this, you must use a client that supports cluster mode.

ElastiCache provides a number of options for scaling your Redis OSS deployment, and you should use the right one for your needs. Think carefully about your Redis OSS scaling strategy to ensure that you're solving the scaling problem you actually have.

Introduction
What is a cache? When should I use one?
What is Redis OSS? Why choose Redis OSS as my in-memory cache?
Benefits of Amazon ElastiCache and Amazon MemoryDB
Redis OSS resources and bottlenecks
Core data modeling patterns
Read-aside caching pattern
Extending the read-aside pattern for changing data
Working with TTLs
Efficient resource usage in Redis OSS
Memory pressure and object sizes
CPU conservation and blocking the main thread
Optimizing network usage
Working with Redis OSS collection types
Hashes
Lists
Sets
Working with advanced data types
Sorted sets
HyperLogLog
Working with multiple operations
Pipelining
Transactions
Scripting
Advanced patterns
Scaling Redis OSS
UNLINK instead of DEL
Locking with Redis OSS
Saving memory with small hashes
Negative result caching
Conclusion

UNLINK instead of DEL

One surprising source of performance issues with Redis OSS is when dealing with large collections. Operations that were fast when dealing with smaller collections can become slow when dealing with larger collections. This is why we've emphasized understanding your data types and their intended operations, with an eye to time complexity of the relevant commands.

Even with that knowledge in hand, the behavior of the `DEL` command can still be surprising. The `DEL` command removes a key from the keyspace. If you have a collection like a list, set, or hash, the `DEL` command has to traverse the entire collection to remove the key. With a large collection, this can be a slow operation.

The `UNLINK` command is a newer command in Redis OSS that provides a more efficient way to remove keys from the keyspace. The `UNLINK` command marks the key for deletion, and the values won't be read by other clients. However, the actual work of evicting the key and its elements from memory is done in a different thread. Recall that Redis OSS is single-threaded, so this allows the lightly used background threads to do the work of removing the key from memory.

If you'd like to do this for all operations, you can use the `lazyfree-lazy-user-del` parameter in your ElastiCache parameter group. This will cause all DEL operations to be handled by the `UNLINK` command. For more on this, see the ElastiCache parameter group documentation.

Locking with Redis OSS

A common use case for Redis OSS is as a lock manager. Perhaps you have a distributed system where you need to ensure that only one client can perform a certain operation at a time. Redis OSS provides a nice, centralized primitive to provide this in your application.

For a simple implementation of locking, you can use the `SET` command with the `NX` option. This will set a key in Redis OSS if it does not already exist. If the key does exist, the command will return `None`. Then, when you are finished performing the work, you can remove the key with the `DEL` command.

Our basic lock implementation looks as follows:

```
def acquire_lock(lock_name, lock_timeout):
    return client.set(lock_name, ‘locked’, ex=lock_timeout,
nx=True)

def release_lock(lock_name):
    client.delete(lock_name)
```

Introduction
What is a cache? When should I use one?
What is Redis OSS? Why choose Redis OSS as my in-memory cache?
Benefits of Amazon ElastiCache and Amazon MemoryDB
Redis OSS resources and bottlenecks
Core data modeling patterns
Read-aside caching pattern
Extending the read-aside pattern for changing data
Working with TTLs
Efficient resource usage in Redis OSS
Memory pressure and object sizes
CPU conservation and blocking the main thread
Optimizing network usage
Working with Redis OSS collection types
Hashes
Lists
Sets
Working with advanced data types
Sorted sets
HyperLogLog
Working with multiple operations
Pipelining
Transactions
Scripting
Advanced patterns
Scaling Redis OSS
UNLINK instead of DEL
Locking with Redis OSS
Saving memory with small hashes
Negative result caching
Conclusion

We can rely on Redis OSS built-in expiration to help with deadlocks or failed clients, which greatly simplifies lock implementation. While this setup may work for some situations, you may need to customize it further.

One thing to note is that, when releasing the lock, we aren't checking to see if the lock is currently held by us. This can work when the lock name itself is unique -- such as a job ID for a queue processor -- but it's not great if the lock is more contentious, such as a singleton lock for making calls to downstream systems. As our client goes to release the lock, it's possible the original lock will have already expired while a new lock has been acquired by another client. In this case, our original client will be releasing the lock of the new client.

To handle this, we can ensure the lock value contains a value known only by our client. This can be a client identifier or a random string of bytes generated by the lock requester. Then, when releasing the lock, we can check to see if the lock value is the same as the value we expect. If it is, we can release the lock. If it's not, we can assume the lock has already been released and we can move on.

Our updated lock implementation would look as follows:

```
def acquire_lock(lock_name, lock_timeout, lock_value):
    return client.set(lock_name, lock_value, ex=lock_timeout,
nx=True)

def release_lock(lock_name, lock_value):
    if client.get(lock_name) == lock_value:
        client.delete(lock_name)
```

While this helps a bit, notice that we could hit a race condition when releasing a lock. It's possible that a new lock could be set between the time we check the lock value and the time we release the lock. Again, this could cause us to release the new lock instead of the old lock. Here, you could look to scripting or transactions, as discussed in the previous section.

Another potential issue is around fault tolerance. If this is an important workflow, we might be nervous about having our locking mechanism be a single point of failure. To handle this, we could add a read replica to our Redis OSS cluster for when the primary instance fails. However, in strict situations, you need to worry about the primary instance failing before it replicates lock information to the read replica.

This is a great situation to use MemoryDB. With MemoryDB, your writes are durably committed across multiple AZs before being acknowledged. This will ensure the status of your lock is not lost during an instance failure. If you need this kind of consistency in managing your distributed locks, look at using MemoryDB to avoid complex failure modes.

Introduction

What is a cache? When should I use one?

What is Redis OSS? Why choose Redis OSS as my in-memory cache?

Benefits of Amazon ElastiCache and Amazon MemoryDB

Redis OSS resources and bottlenecks

Core data modeling patterns

Read-aside caching pattern

Extending the read-aside pattern for changing data

Working with TTLs

Efficient resource usage in Redis OSS

Memory pressure and object sizes

CPU conservation and blocking the main thread

Optimizing network usage

Working with Redis OSS collection types

Hashes

Lists

Sets

Working with advanced data types

Sorted sets

HyperLogLog

Working with multiple operations

Pipelining

Transactions

Scripting

Advanced patterns

Scaling Redis OSS

UNLINK instead of DEL

Locking with Redis OSS

Saving memory with small hashes

Negative result caching

Conclusion

Saving memory with small hashes

For most Redis OSS deployments, memory is the most expensive resource. As your Redis OSS usage grows, you'll look at ways to save memory, whether through faster expiration of keys, compression of values, or other techniques. When that fails, you may increase your Redis OSS instance size or add more instances to your cluster.

One trick to reducing your Redis OSS data usage involves grouping a number of unrelated string keys into a single hash key. Due to the internals of how Redis OSS is structuring hashes under the hood, a single hash key with 100 fields will use significantly less memory than 100 separate string keys. You can find more information about the Redis OSS internals here.

Let's see how this would work in practice. Recall in our section on core modeling patterns that we were caching user records for our application. In doing so, we used a key pattern of `user:${userId}` to store the user record. This meant every user record was stored as a separate string key in Redis OSS.

Let's switch that to group user records together in a hash by the prefix of their user ID. The key name will be `user:${userIdPrefix}`, where `userIdPrefix` is all but the last two characters of the user ID. Then, the user record will be stored in the hash as a field with the last two characters of the user ID as the field name. For a user with the ID of `1234`, the key name will be `user:12` and the field name will be `34`.

The implementation of this would look as follows:

```
def get_user_key(user_id):
    return f'user:{user_id[:-2]}'

def get_user_field(user_id):
    return user_id[-2:]

def get_user(user_id):
    user_key = get_user_key(user_id)
    user_field = get_user_field(user_id)
    user = client.hget(user_key, user_field)
    if user is None:
        user = get_user_from_database(user_id)
        client.hset(user_key, user_field, user)
    return user
```

This small change can save 40% or more of the memory used by your user records. Note that there is a small tradeoff in additional CPU usage, as Redis OSS will need to operate on the entire hash when getting or setting a single field. If your application is more constrained on memory than on CPU, this tradeoff is likely worth it.

Note that you will lose some functionality with this approach. Most specifically, you can't expire fields within a hash individually, you can only expire the entire hash. If this is a problem for your application, you should stick with the string-based approach to storing your data.

Finally, when designing the number of elements in your hash, pay attention to the underlying configuration of your Redis OSS cluster. For Redis OSS versions before 7.0, the `hash-max-ziplist-entries` parameter controls the maximum number of elements in a hash before it is converted into a true hash under the hood. For Redis OSS 7.0 and later, the `hash-max-listpack-entries` parameter controls this behavior. You should ensure the number of elements in your constructed hashes is below this threshold to ensure you're getting the memory savings you expect.

Introduction

What is a cache? When should I use one?

What is Redis OSS? Why choose Redis OSS as my in-memory cache?

Benefits of Amazon ElastiCache and Amazon MemoryDB

Redis OSS resources and bottlenecks

Core data modeling patterns

Read-aside caching pattern

Extending the read-aside pattern for changing data

Working with TTLs

Efficient resource usage in Redis OSS

Memory pressure and object sizes

CPU conservation and blocking the main thread

Optimizing network usage

Working with Redis OSS collection types

Hashes

Lists

Sets

Working with advanced data types

Sorted sets

HyperLogLog

Working with multiple operations

Pipelining

Transactions

Scripting

Advanced patterns

Scaling Redis OSS

UNLINK instead of DEL

Locking with Redis OSS

Saving memory with small hashes

Negative result caching

Conclusion

Negative result caching

The last advanced tip is around negative result caching. This is a pattern that is often overlooked but can be a great way to protect your downstream systems.

As noted, one of the reasons to use caching is to reduce the load on your downstream systems. This can be a system-of-record database that you control, or it can be a different service that you're calling over the network. In either case, you want to reduce the load on that system as much as possible.

In the read-aside cache pattern, we showed how to store the results of an operation to the cache. However, in certain cases, the operation might not return a successful result. This could be due to the absence of the required data, a client-side error, or a server-side error. When that error occurs, you may be tempted to return to your client without storing the result in the cache. However, if a subsequent request is made that will have the same negative result, you'll be hitting your downstream system again.

To help with these, you can store the negative result in the cache itself. To implement this, we can update our read-aside cache function as follows:

```
def get_user(user_id):
    user = client.get(f'user:{user_id}')
    if user is None:
        user = get_user_from_database(user_id)
        if user is not None:
            client.set(f'user:{user_id}', user, ex=3600)
        else:
            client.set(f'user:{user_id}', 'null', ex=60)
    return user
```

In this case, we'll store the null user record in our cache.

Notice that you may want to set a shorter expiration time for the negative result than for the positive result. This is because the negative result is likely to be less stable than the positive result. You may even want to vary the expiration time based on the type of error that occurred. For example, if the error was a client-side error based on data given to you by the requester, you may cache it longer as this is unlikely to be fixed. However, if there is a server-side error, you may want to cache it for a shorter period of time -- long enough to give the system time to recover, but not so long that you're serving stale data.

In any case, be sure to consider results other than the happy path when working with a cache. You can save your downstream systems a lot of work by caching negative results.

Conclusion

In this ebook, we've looked at basic data modeling patterns, like the read-aside pattern, and at core data types in Redis OSS. We saw how to use them in your application and when they're useful. We also looked at more advanced data types, like sorted sets and HyperLogLogs, to see when they can be used in your application. Finally, we reviewed some advanced patterns about scaling your cluster, implementing distributed locking, and saving memory.

Redis OSS is a powerful, fast, and flexible key-value store that can be used in a variety of ways. Used correctly, it can be an elegant way to delight your end users as you can reduce latencies or provide advanced features like leaderboards. We hope this ebook has given you a good understanding of how to use Redis OSS effectively in your application.