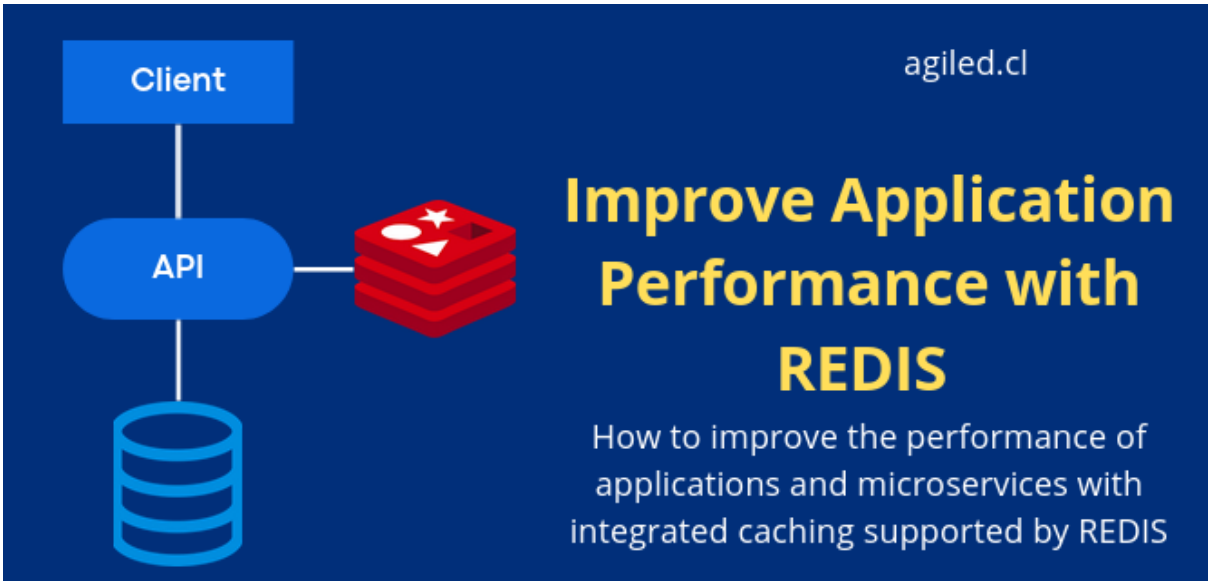


Rodrigo León Nanjarí | 05/01/2024

Improve Application Performance with REDIS

This article shows how to improve the performance of applications and microservices with integrated caching supported by REDIS.

Performance is a big topic in all business applications and web services. Many well-designed, multi-tier enterprise applications have performance and availability issues in production environments, especially in peaks of demand. This is particularly evident with backend systems not fully prepared for online processing or databases poorly tuned for high demanding traffic.

One of the best architectural solutions is to implement a caching system to store in memory the business data frequently accessed with low changing rate. This approach provides important benefits:

- Improve the response time of the application or service, to access data in memory instead of query databases or backend systems.
- Improve the availability of the whole platform, to reduce the amount of resources involved in processing every request (CPU, memory, threads, processes).

- Enable to expose the same application services to other platforms, including mobile apps and business dashboards.

Naturally, not all data is well suited for caching. For data that is changing frequently, like balance account, order status or inventory availability, probably caching is not the best option. But for all the rest, caching is a good option for improving performance and getting a better user experience. In case that your application depends on backend services or legacy systems, you can still adopt integration patterns like **Circuit Breaker** to improve the availability of your application, implementing mechanisms like timeout, retrying or callback.

Online Caching

Online is the most common approach to implement a caching solution. This approach considers to consume the backend services or database and then store the result in cache for a second request. To implement this approach you must consider a Time to Live (TTL) value, in order to define the time the data will be available in the cache system.

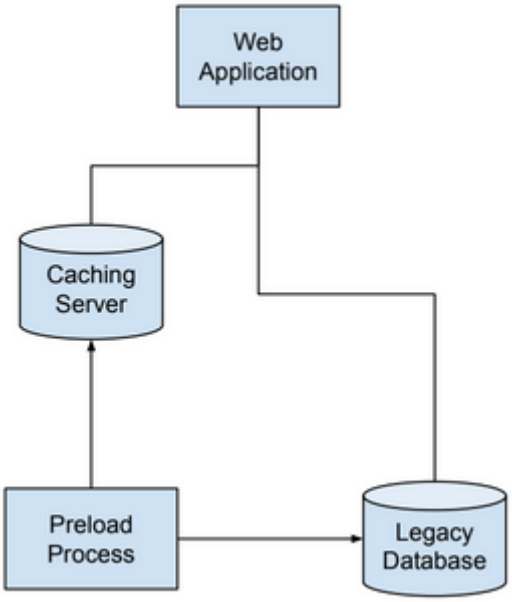
We will see in the next sections how to integrate caching for application and microservices with minimum configuration required.

Offline Caching

Offline is a most complicated approach to implement a caching solution but equally effective. This approach considers to preload the data in cache in advance before consumption. This can be implemented in two ways:

- Batch process running every hour or every day, then query the business data from a legacy database or backend system and then load the data in the cache system.
- Implement a streaming or queuing mechanism to store or update the data in the cache based in business events like a new order, payment confirmation and so on.

The following figure describes the mechanism of offline caching with a batch process to preload data in cache:



High Availability

All caching solutions should provide a mechanism of high availability, in case of failure. The most common approach is the pattern master - slave, with automatic replication. When a client of the cache system inserts a new value, the platform automatically replicates these values to the slave node. In case of failure, the slave node can take the role of master repository.

Additionally to the replication feature, most caching platforms can store the data on disk, so the system can rapidly recover from a crash loading the memory data from disk.

Caching Solutions

In the market there are many caching products, some open source, some required to be licensed. There are what I know from practical experience:

- REDIS
- Memcached
- Oracle Coherence
- AWS Elastic Cache

In this article, we are going to describe and explain the first one, who is the most flexible and adopted for many use cases. In the case of AWS, Elastic Cache is supported by REDIS or Memcached.

Introduction to REDIS

REDIS is an open source (BSD licensed), initially working only as an in-memory cache system but with more functionality available in newer versions including message broker, and streaming engine. REDIS is written in ANSI C, so is highly performant. At the moment, there is support for Linux, Unix and Mac OS systems.

REDIS provides in-memory storage of different data structures including strings, hashes, lists, sets and sorted sets. Newer versions support range queries, bitmaps, hyperloglogs, geospatial indexes, and streams. The most simple use case is store key-value in memory, for example:

- Key: ID10234 (customer ID)
- Value: 90.0 (credit risk)

You can use REDIS in a command-line scripting tool called **redis-cli**. For storage a simple key-value data, you simply execute:

```
%> PUT ID10234 90.0
```

In order to retrieve the stored value, you simply execute:

```
%> GET ID10234
```

Redis implements a built-in replication and different levels of on-disk persistence, and provides high availability via Redis Sentinel and automatic partitioning with REDIS Cluster. You can run atomic operations on these types, like appending to a string; incrementing the value in a hash; pushing an element to a list; computing set intersection, union and difference; or getting the member with highest ranking in a sorted set.

To achieve top performance, REDIS works with an in-memory dataset. Depending on your use case, Redis can persist your data either by periodically dumping the dataset to disk or by appending each command to a disk-based log. You can also disable persistence if you just need a feature-rich, networked, in-memory cache.

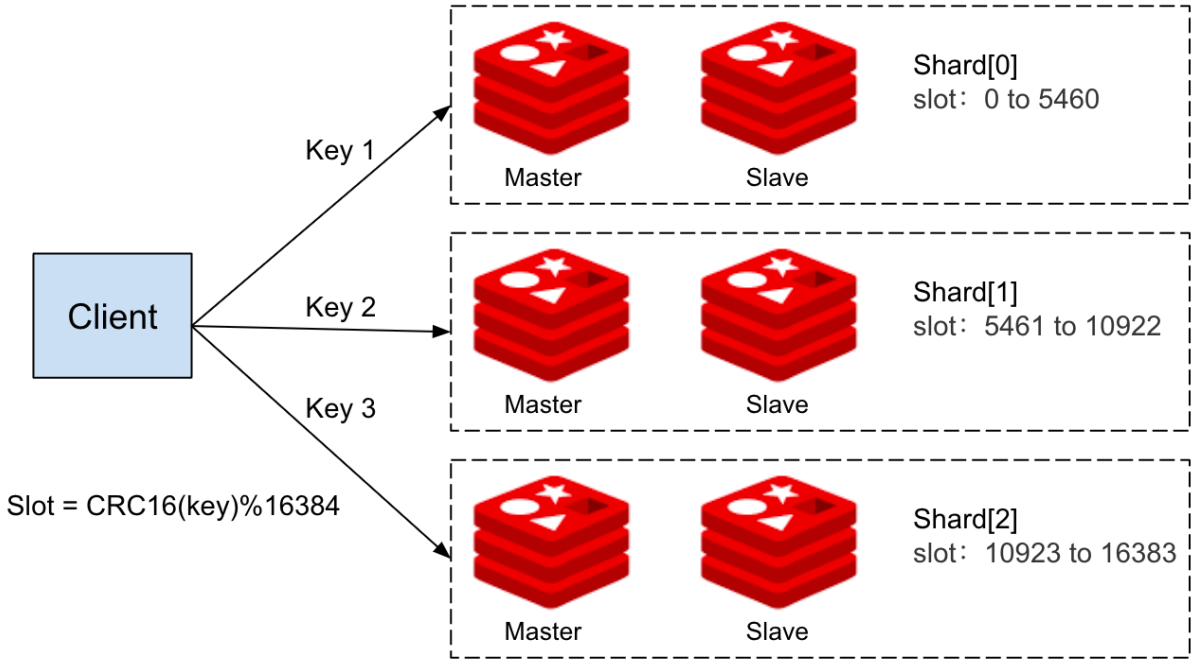
REDIS provides asynchronous replication, with fast non-blocking synchronization and auto-reconnection with partial resynchronization on net split. REDIS has support for many programming languages, including Java, Python, C#, Node.js.

REDIS Cluster

REDIS Cluster provides a way to run an installation where data is automatically shared across multiple Redis nodes. With REDIS Cluster, you get the ability to:

- Automatically split your dataset among multiple nodes.
- Continue operations when a subset of the nodes are experiencing failures or are unable to communicate with the rest of the cluster.

The following figure describes a REDIS Cluster with 6 nodes: 3 masters and 3 slaves. The key space is equally divided in 3 sets:



REDIS with Java SpringBoot

Previously, we mentioned that REDIS has support for different programming languages. In Java there are low level libraries like Lettuce or Jedis and high level abstraction like Spring Data Redis. We are going to explain the last one:

Spring Data Redis.
Spring Data Redis, part of the larger Spring Data portfolio, provides easy configuration and access to Redis from Spring applications. It offers both low-level and high-level abstractions for interacting with the store, freeing users from infrastructural concerns.

The following section describes how to implement a REDIS cache in a Spring Boot application with JPA for database access.

Maven Dependencies

```
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-redis</artifactId>
    <version>3.1.6</version>
</dependency>
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
    <version>4.3.2</version>
  </dependency>
```

Java Configuration

We defined a *RedisTemplate* using the *jedisConnectionFactory*. This can be used for querying data with a custom repository. In this case, we are using the default settings (localhost and port 6379).

```
@Bean
public JedisConnectionFactory redisConnectionFactory() {
   RedisStandaloneConfiguration config =
      new RedisStandaloneConfiguration("server", 6379);
   return new JedisConnectionFactory(config);
}
```

SpringBoot Application

Here we start the SpringBoot application with the annotation *@EnabledCaching*.

```
@SpringBootApplication
@EnableCaching
public class RedisDemoApplication {

  public static void main(String[] args) {
  SpringApplication.run(RedisDemoApplication.class, args);
 }
}
```

SpringBoot Controller

Now we can use the Spring Redis integration in a method of the controller class. The important keyword is *@Cacheable*, which indicates Spring to query REDIS before consuming the database or backend systems.

```
@GetMapping("/product/{id}")
@Cacheable(value = "product", key = "#id")
public Product getProductById(@PathVariable long id) {
   // Query product info from database
}
```

In this case, the cache name is “product” and the key stored in the cache (unique value) is the product identification.

With this configuration, when a client consumes the API **/product/1234**, Spring will check if exists a product stored in cache and return this value. If not, then Spring query the database, fetch the product data and store the object in cache for future consumption.

Summary

This article shows a conceptual introduction to caching and an overview of REDIS, one of the most known cache products available in the market.

We hope that this article will invite you to use REDIS to improve the performance of your applications and microservices.

Bibliography

- REDIS
  https://redis.io/

Rodrigo León Nanjarí
CEO Agiled and Software Architect

Since 2018, Rodrigo has been working with REDIS to improve performance on mission critical applications and services.

rodrigo.leon@agiled.cl
http://www.agiled.cl
https://www.linkedin.com/in/rodrigoleonnanjari/