LEARNING MADE EASY

Limited Edition

Redis™

for dummies®
A Wiley Brand

Discover Redis data structures

Create applications with Redis

Learn about Redis use cases

Courtesy of

redislabs
HOME OF REDIS

Steve Suehring

About Redis Labs

Modern businesses depend on the power of real-time data. With Redis Labs, organizations deliver instant experiences in a highly reliable and scalable manner. Redis Labs is the home of Redis, the world's most popular in-memory database, and commercial provider of Redis Enterprise that delivers superior performance, matchless reliability and unparalleled flexibility for personalization, machine learning, IoT, search, ecommerce, social and metering solutions worldwide.

Redis Labs, consistently ranked as a leader in top analyst reports on NoSQL, in-memory databases, operational databases, and database-as-a-service, is trusted by over 8,200 enterprise customers, including six Fortune 10 companies, three of the four credit card issuers, three of the top five communication companies, three of the top five healthcare companies, six of the top eight technology companies, and four of the top seven retailers.

Redis has been voted the most loved database, rated the most popular database container, #1 cloud database, and the #1 NoSQL in software stacks.

redislabs.com

@redislabs

https://www.linkedin.com/company/redis-labs-inc/

http://www.youtube.com/c/Redislabs

Learning Redis? Check out Redis University

Redis University was established with the goal to create a destination for all things Redis. Created and delivered by core Redis developers, practitioners and experts, Redis University provides a variety of courses for developers & administrators. Online courses provide an immersive experience, with guided labs and experiments for practitioners to sharpen their skills and deepen their knowledge and insights.

university.redislabs.com

Redis

Redis

Limited Edition

by Steve Suehring

for dummies®
A Wiley Brand

Redis For Dummies®, Limited Edition

Published by
John Wiley & Sons, Inc.
111 River St.
Hoboken, NJ 07030-5774
www.wiley.com

ISBN 978-1-119-52080-1 (pbk); ISBN 978-1-119-52083-2 (ebk)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

Publisher's Acknowledgments

Some of the people who helped bring this book to market include the following:

Development Editor: Elizabeth Kuball
Copy Editor: Elizabeth Kuball
Executive Editor: Steve Hayes
Editorial Manager: Rev Mengle
Business Development Representative: Karen Hattan
Production Editor: Siddique Shaik

Table of Contents

Introduction

NoSQL is a modern data storage paradigm that provides data persistence for environments where high performance is a primary requirement. Within NoSQL, data is stored in such a way that both writing and reading are fast, even under heavy load.

Redis is a market-leading multi-model database that brings NoSQL to organizations both big and small. Redis is open source, and Redis Enterprise adds several enhancements that are important to the enterprise deployments.

About This Book

This book provides a starting point for those new to Redis and those who have heard about Redis but want to see how it can be used in their organizations.

The book serves multiple audiences, with subject matter for managers and developers alike. You certainly can read the book from cover to cover, but I don't assume that you will. Instead, you can comfortably read chapters out of order based on your interest in a particular chapter.

Foolish Assumptions

In writing this book, I assumed that you're familiar with databases, at least at a basic level. If you're a developer, you should have a development environment available on which you can install things. In later chapters, I show examples using Redis that also utilize Docker and Github, so having a development environment available will be helpful.

It's worth noting that Chapters 6 and 7 are written in such a way that you can follow along with what's going on even if you don't run the examples yourself.

Icons Used in This Book

Throughout this book, I use the following icons to call your attention to details that are important:

The Remember icon focuses your attention on an important detail that you may have otherwise missed.

The Technical Stuff icon marks some extended information that may not be of interest to all readers. Only the technical need read the stuff located near these icons.

A helpful little bit of information is all you'll find marked by the Tip icon — possibly something that will make your life a little easier.

Stay away from whatever the Warning icon is warning you against. When you see the Warning icon, you'll know that there might be some danger around.

Where to Go from Here

Redis and Redis Enterprise are quite complex, and this book only covers the tip of the iceberg. For more information, head to `www.redislabs.com`.

IN THIS CHAPTER

» Introducing NoSQL

» Seeing where Redis fits

Chapter 1
What Is Redis?

This chapter gives an overview of NoSQL, including a look at types of NoSQL databases such as key/value, document, column, and graph. The chapter continues with a comparison of NoSQL to methods for traditional data persistence.

The chapter also introduces Redis, a popular multi-model database server. Redis goes beyond NoSQL database to provide several advanced capabilities needed by modern applications.

Introducing NoSQL

The term *NoSQL* is used to describe a set of technologies for data storage. In this section, I explain what NoSQL is, outline the major types of NoSQL databases, and compare NoSQL to relational databases.

Defining NoSQL

NoSQL describes technologies for data storage, but what exactly does that mean? Is NoSQL an abbreviation for something? I answer these and other pressing questions in this section.

Depending on whom you ask, *NoSQL* may stand for "not only SQL" or it may not stand for anything at all. Regardless of any

disagreement over what *NoSQL* stands for, everyone agrees that NoSQL is a robust set of technologies that enable data persistence with the high performance necessary for today's Internet-scale applications.

TECHNICAL STUFF

SQL is an abbreviation for Standard Query Language, a standard language for manipulating data within a relational database.

Identifying types of NoSQL databases

There are four major types of NoSQL databases — key/value, column, document, and graph — and each has a particular use case for which it's most suited.

The following sections go into greater detail on the four types of NoSQL.

Key/value

With a key/value storage format, data uses *keys,* which are identifiers that are similar to a primary key in a relational database. The data element itself is then the value that corresponds to the key.

An example of a key/value pair looks like this:

```
"id": 12319054
```

In this example, `"id"` is the key while `12319054` is the value that corresponds to that key.

Column

With a column-oriented data store, data is arranged by column rather than by row. The effect of this architectural design is that it makes aggregate queries over large amounts of data much faster to process.

Document

Document data storage in NoSQL uses a key as the basis for item retrieval. The key then corresponds to a more complex data structure, called a *document,* which contains the data elements for a given collection of data.

Graph

Graph databases use graph theory to store data relations in a series of vertices with edges, making queries that work with data in such a manner much faster.

Comparing NoSQL and relational databases

Regardless of the type of NoSQL database, the patterns and tools that you use to work with data is different from the patterns and tools that you typically find with a relational database. As you just saw, the paradigm for storage and arrangement of the data typically requires a rethink of how applications are created.

Relational databases connect data elements through relations between tables. These relations become quite complex for many applications, and the resulting queries against the data become equally complex. The inherent complexity leads to performance issues for queries.

Many traditional databases include query tools and software to directly manipulate data. With NoSQL, most access will be programmatic only, through applications that you write using the tools and application programming interfaces (APIs) for the NoSQL database.

Relational databases have somewhat less flexibility than a multimodel database such as Redis. Whereas a relational database thrives when data is consistent and well structured, Redis and NoSQL thrive on the unstructured data that is found in today's modern applications while also providing the flexibility to structure data as needed.

Seeing Where Redis Fits

Redis is a NoSQL database and yet much more. Redis is a multimodel database enabling search, messaging, streaming, graph, and other capabilities beyond that of a simple data store.

Multi-model databases

Multi-model databases provide a way to interact with data regardless of its underlying data model.

Redis provides full multi-model functionality through Redis Modules. The use of Redis as a multi-model database enables greater flexibility for application developers within an organization.

Data storage

Redis keeps data in memory for fast access and persists data to storage, as well as replication of in-memory contents for high-availability production scenarios.

When discussing data storage, the concept of durability becomes important. *Durability* is the ability to ensure that data is available in the event of a failure of a database component.

Redis supports multiple modes for ensuring durability, accommodating most data structures and environment-specific requirements

Data structure storage

Redis supports several data structures. In fact, it may be helpful to think of Redis as a data structures store rather than a simple key/value NoSQL store.

Supported data structures include

- Strings
- Lists
- Sets
- Sorted sets
- Hashes
- Bit arrays
- Streams
- HyperLogLogs

Each data structure has a different use case or scenario for which it is best suited. Beyond these data structures, Redis also supports the Publish/Subscribe (PubSub) pattern and additional patterns that make Redis suitable for modern data-intensive applications.

IN THIS CHAPTER

- Understanding the components of Redis
- Identifying how Redis can help you
- Looking at a real-world example

Chapter 2
What Is Redis Used For?

This chapter begins an in-depth examination of Redis. Included in the chapter is a look at the various Redis components and how those components can help you. The chapter concludes with an example of how Redis is used for production applications.

Understanding the Components of Redis

Like other server software, Redis has several components working together to provide robust solutions. Understanding these components and the overall architecture of Redis is the focus of this section.

The server and the command-line interface

Redis runs as server-side software, primarily on the Unix-based operating systems like Linux and macOS and also as a Docker container on Microsoft Windows. Redis server is downloaded and installed in just a few steps and then is ready for use.

TIP

The installation process for Redis is fully documented in the Quick Start guide available at `https://redis.io/topics/quickstart`.

The server listens for connections from clients — either programmatically or through the command-line interface (CLI). Like the CLI for other database servers, the CLI for Redis enables direct interaction with the data on the server.

The client and drivers

Numerous client libraries are available supporting many programming languages. It is through these clients and drivers that you interact programmatically with data found on a Redis server.

For example, if your organization uses Python for its programming language of choice, you'll probably integrate with Redis through the `redis-py` package, though you have the opportunity to use more than a dozen other Python-related packages for Redis integration, too.

The clients and drivers are typically shared under an open-source license, though the license varies by project. See `https://redis.io/clients` for more information.

I won't list all 200 or so supported languages, but featured client libraries for several popular languages include the following:

- **Java:** Three popular clients include Jedis, Lettuce, and Redisson, all of which serve slightly different needs.
- **Node.js:** The recommended client for Node.js is node_redis.
- **C#:** Two popular clients include ServiceStack.Redis and StackExchange.Redis.
- **PHP:** PHP has several clients with PHP, but Predis is recommended.
- **C:** hiredis is the official Redis client for the C language. Also see hiredis-vip for cluster-related C language support.

Databases, memory, and persistence

There is no formal database creation step with Redis. Like database creation, there isn't a formal table creation step necessary with Redis either. The `SET` command is used to create data within the current database.

CREATING AND QUERYING DATA

The SET command adds a key to the database in Redis. For example, to create a key for various pieces of furniture in your living room, you might do this:

```
SET furniture:couch:color green

SET furniture:recliner:color brown

SET furniture:chair:color: tan
```

Alternatively, you could retrieve all keys with the KEYS command:

```
KEYS furniture*
```

Note: The KEYS command used in the preceding example is not typically recommended for production usage. Use it for debugging only.

Those familiar with formalized database creation and definition may be uncomfortable with the seemingly informal process of database creation and data handling. However, it's through this flexibility that the true power of Redis is found.

Data is stored in random access memory (RAM) on the Redis server. This means that as data is added, additional RAM is used. Redis on Flash (see Chapter 4) provides a method for supplementing RAM with flash-based memory. Redis writes the contents of the database to disk at varying (and configurable) intervals depending on the amount of data that changes during the interval. Persisting data to disk ensures durability in the event of a software or hardware failure that renders the server unavailable. Other means for providing durability, such as clustering for high availability, are common with Redis in a production environment.

Identifying How Redis Can Help You

This section examines a few popular use cases for Redis. Redis has the necessary capabilities to meet user expectations for performance and features. For example, benchmarks show that Redis Enterprise in an ACID configuration is able to perform more than

500,000 operations per second with sub-millisecond latency and can also achieve 50 million operations per second with the same performance on only 26 compute nodes. The performance of Redis coupled with search features like autocomplete and result highlighting makes the entire user experience better.

Personalization with session management

A session is loaded when a user logs in or when he's using the application in order to track his activity. By nature, session-related data needs to be readily available, with low latency to meet performance requirements that users expect.

Redis is a great fit for such applications because data is available in-memory and data is structured based on its use in the application.

Social apps

End users expect real-time or near-real-time performance from social apps. From chat to follows to comments to games, social apps present a challenge for disk-based data stores. An in-memory data store gives the performance necessary for these applications.

Several features of Redis make implementation of social app features possible:

- Intelligent caching
- Pub/sub pattern for incoming data
- Job and queue management
- Built-in analytics
- Native JSON-handling

JavaScript Object Notation (JSON) is a structured data format. By being native JavaScript, JSON-formatted data can be used directly in an app without needing to be transformed into another format.

Search

Allowing users to search data is challenging. Allowing users to search data while providing high performance is even more difficult. With other, slower data stores, secondary indexes frequently need to be added in order to provide adequate performance.

Redis in the Real World

This section looks at some common use cases for Redis related to e-commerce. For example, many e-commerce sites provide search capabilities and need to do so in a high-performance environment using autocomplete. Although this certainly isn't an exhaustive list, it does highlight several popular ways to use Redis.

Caching

Providing fast response time is more important than ever. However, responding quickly, even under high demand, can be resource intensive. This is often solved with caching.

Redis can be used as a means to cache data between the application and the backend data store, such as another relational or NoSQL database. Doing so frees up the database for other operations while also enabling user-friendly fast response.

Large datasets

REMEMBER

Redis handles caching well because of its native data types and its efficient use of memory.

The performance of Redis means that recommendations and customer analytics can be done in real time.

The use of Redis on Flash makes large dataset analysis cost-effective. In this use case, Redis Enterprise Flash is used to extend RAM.

Full-text search

The RediSearch module is used to extend the capabilities of Redis. RediSearch can work up to 500 percent faster than stand-alone search engine products and includes features like scoring, filtering, and query expansion.

Automatic suggestions based on the search are provided with RediSearch, too. All of this is done with the performance that you would expect from Redis.

RediSearch stores data in RAM and can be scaled onto multiple Redis instances.

Geospatial and time-series data

Redis, with its native geo, sorted set, and hash and streams data types is an excellent choice for geospatial and time series data. These data types might be used for location-based recommendations and promotions.

Another geospatial and time-series use case is collection of data from Internet of Things (IoT) devices. These devices and related sensors are constantly generating data and doing so in a manner where their location matters. For example, a traffic sensor noting that the flow of traffic has slowed might be able to relay the message to open additional lanes or that there is another issue that needs attention.

Messaging/queuing

A related use of Redis is handling fast-moving data. In the preceding example, if you have data being generated by thousands and millions of sensors, that needs to be analyzed and processed. It can be collected, streamed, and ingested by Redis using its native publish/subscribe mechanism.

IN THIS CHAPTER

- Understanding the primary data models
- Utilizing patterns and data structures
- Working with modules

Chapter 3
Using Multi-Model Redis: Data Models, Structures, and Modules

Data models represent how the data is stored within a database. An implication of choosing a data model is that your application will then be tied to that model.

In the past, relational models didn't reflect the application or problem domain very well. Instead, relational models emphasize other aspects of data storage. With the rise of NoSQL technologies like Redis, the data model can be a reflection of the application itself.

A multi-model database like Redis enables the data to be represented for multiple use cases simultaneously. This means that the data can be used in the manner most appropriate for the application.

Understanding the Redis Data Models

Using a data store of any kind requires making decisions about how to represent the data within the data store. This model then controls how data is added to the database and how it's retrieved.

Data is stored in Redis using keys. Keys can be just about anything because they're binary safe. For example, you could use an image as a key. Most keys are simple strings, though.

Redis has a variety of commands for working with data of different types. A couple notable commands are encountered in this section, including `SET` and `GET`. The `SET` command creates or changes a value that corresponds to a given key. The `GET` command retrieves the value associated with the given key.

It's worth mentioning that values are overwritten with the `SET` command. That means if you call `SET` twice for the same key, the last value will be the one that is stored and retrieved.

The values that correspond to a given key can be formatted in many ways to create a data model specific to the needs of the organization. This section examines the primary data models in Redis.

Strings and bitmaps

The simplest value type in Redis is a *string.* A value can be added to the database with the `SET` command. When using the `SET` command, a key and a value are the minimum requirements in order to create the entry. For example, to create a key called `user` with a value of `steve`, you simply need to execute this command from the Redis CLI:

```
> SET user steve
```


TIP

Even though double quotes were used for this string value, they aren't strictly necessary when the value is a single word. With that, a simple string value of `steve` has been stored in the database and can then be retrieved with the `GET` command:

```
> GET user
```

Doing so retrieves the following value:

```
"steve"
```

There are numerous other commands that can be executed, and some make sense in a certain context. For example, a common way to use simple string values is as a counter. In these cases, commands like `INCR` (short for *increment*) can be used. Consider this example:

```
SET logincount 1
```

In the command, a new key called `logincount` is created and set to the value of `1`. Then you call `INCR` on that key:

```
INCR logincount
```

When `INCR` is executed, the new value is returned immediately:

```
(integer) 2
```

Of course, you can always retrieve the value with the `GET` command:

```
GET logincount
```

Doing so returns the following:

```
"2"
```


TIP

There are numerous other commands to manipulate and work with string and stringlike data in Redis, though you can't use commands intended for numeric data on string data.

Closely related to strings are bitmaps, which are a form of string storage. Using a bitmap, you can represent many data elements that are simply on (1) or off (0). This is useful for operations where you only need to know those two possible values, such as whether a user is active or inactive. Because it can be only one of two values, you can represent that data efficiently.


TECHNICAL STUFF

The largest size for a single string value is 512MB. This means that you can store 2^{32} possible values inside of one string value in Redis. This size limit will be increasing and may have already increased by the time you're reading this. Check the latest Redis documentation for the current size limit for string values.

There are commands specific to working with bitmaps available in Redis. These commands include `SETBIT` and `GETBIT`, which are used to create or change a value and retrieve a value, respectively. Other commands include `BITOP` and `BITFIELD`.

Lists

Lists are a way to store related data. In some contexts, lists are called arrays but in Redis, a list is a linked list, which means operations to write to the list are very fast. However, depending on where in the list the item is located, its performance is not as fast for read operations. Although not always appropriate because of repeated values, a set (discussed later) can sometimes be used when read speed is crucial.

Lists use one key holding several ordered values, and values are stored as strings. You can add values to the head or tail (called "left" and "right" in Redis) of a list and you retrieve values by their index. Values within a list can repeat, meaning you may have the same value at a different index within the list.

Pushing a value onto a list is accomplished with the `LPUSH` and `RPUSH` commands. These commands place values onto a list either on the left (or head) or to the right (or tail) of the list. For example, creating a two-item list looks like this:

```
LPUSH users steve bob
```

The list now contains two items, indexed beginning at `0`. An individual item can be retrieved using the `LINDEX` command. For example, retrieving the first item in the list looks like this:

```
LINDEX users 0
```

Retrieving the second item looks like this:

```
LINDEX users 1
```

TIP

If you try to retrieve an index that doesn't exist, you'll receive `(nil)` as output.

All items or just a slice of items can be retrieved with the `LRANGE` command. The `LRANGE` command expects to receive the first and

last indexes to retrieve, by number. If you want to retrieve all items in the users list, it looks like this:

```
LRANGE users 0 -1
```

Note the use of the `-1` as the second value. The `-1` means "to the end of the list."

The output from the `LRANGE` command for the users table is as follows:

```
1) "bob"
2) "steve"
```

Also notably, because `LPUSH` was used, the last item, `bob`, becomes the top of the list, or item 1. If this list had been created with `RPUSH`, then `bob` would be the bottom of the list, or item 2.

Sets

From an application standpoint, sets are somewhat like lists, in that you use a single key to store multiple values. Unlike lists, though, sets are not retrieved by index number and are not sorted. Instead, you query to see if a member exists in the set. Also unlike lists, sets cannot have repeating members within the same key.

Redis manages the internal storage for sets. The result is that you don't work with set values in the same way that you do lists. For example, you can't push and pop to the front and back of a set like you can with a list.

Adding a value to a set is done with the `SADD` command:

```
SADD fruit apple
```

Listing all members of a set is done with the `SMEMBERS` command:

```
SMEMBERS fruit
```

Given that the key called `fruit` exists, the command returns a list of all members in that set. In this case, the only item returned is as follows:

```
1) "apple"
```

You can determine if a given value exists in a set with the `SISMEMBER` command. For example, to see if a value called `"apple"` exists in the `fruit` key, the command is as follows:

```
SISMEMBER fruit apple
```

When the member exists in the set, an integer `1` is returned. If the member does not exist, an integer `0` is returned.

Hashes

Hashes are used to store collections of key/value pairs. Contrast a hash with a simple string data type where there is one value corresponding to one key. A hash has one key, but then within that structure are more fields and values.

You might use a hash to store the current state of an object in an application. For example, when storing information about a house for sale, a logical structure might look like this:

```
houseID: 5150
numBedrooms: 3
squareFeet: 2700
hvac: forced air
```

Representing this with a Redis hash looks like this:

```
HSET house:5150 numBedrooms 3 squareFeet 2700 hvac
  "forced air"
```

Individual fields within the overall `house:5150` hash are retrieved with the `HGET` command. To retrieve the `numBedrooms` field value looks like this:

```
HGET house:5150 numBedrooms
```

The result is as follows:

```
"3"
```

Sorted sets

Sorted sets are used to store data that needs to be ranked, such as a leaderboard. Like a hash, a single key stores several members. The score for each of the members is a number. For example, if you were tracking the number of followers for a group of users, the data might look like this:

User Followers:

steve: 31

owen: 2

jakob: 13

Within Redis, this data can be re-created as a sorted set with the following command:

```
ZADD userFollowers 31 steve 2 owen 13 jakob
```

The `ZRANGE` command is used to retrieve the resulting sorted set. Like the `LRANGE` command, which is used to retrieve values from a list, the `ZRANGE` command accepts the beginning and ending number for retrieval. For example, to retrieve all members of a sorted set looks like this:

```
ZRANGE userFollowers 0 –1
```

When that command is executed, the members are retrieved but not the corresponding scores. To retrieve both the member names and their scores, add the `WITHSCORES` argument to the command:

```
ZRANGE userFollowers 0 –1 WITHSCORES
```

When that command is executed against the previously entered data set, the result is:

```
1) "owen"
2) "2"
3) "jakob"
4) "13"
5) "steve"
6) "31"
```

As you can see from the output of ZRANGE, the members and their scores are ranked by score value, lowest to highest. You can also retrieve the members and their scores in reverse order (that is, highest to lowest) with the ZREVRANGE command:

```
ZREVRANGE userFollowers 0 -1 WITHSCORES
```

The score for an individual member can be incremented by any valid number with the ZINCRBY command. For example, to increment the username jakob by 20, the command is as follows:

```
ZINCRBY userFollowers 20 jakob
```

The resulting score is returned, so in this case the returned value represents the original 13 followers plus 20 more:

```
"33"
```

The result of the ZRANGE or ZREVRANGE will reflect the change to the number of followers, too.

Another way of working with data in a sorted set is to use the ZRANK command to determine where within the sorted set a given member resides.

HyperLogLog

HyperLogLog is a specialized but highly useful data type in Redis. A HyperLogLog is used to keep an estimated count of unique items. You might use the HyperLogLog data type for tracking an overall count of unique visitors to a website.

The HyperLogLog data type maintains an internal hash to determine whether it has seen the value already. If it has, then the value is not entered into the database.

The PFADD command is used to both create a key and add items to a HyperLogLog key:

```
PFADD visitors 127.0.0.1
```

If this is the first time that the value 127.0.0.1 has been seen in the visitors key, then an integer value of 1 is returned to indicate a

successful addition to that database. A 0 is returned if the value already exists.

The PFCOUNT command is used to provide an estimate of the number of unique items within a HyperLogLog.

Patterns and Data Structures

I've introduced the basic data types in Redis. But there are also common ways to use Redis, incorporating the data types that you've already seen. I examine some of these patterns in this section.

Pub/sub

Redis can also act as a fast and efficient means to exchange messages in a publisher/subscriber (pub/sub) pattern. When used in such a way, a publisher creates a key/value pair and zero or more clients subscribe to receive messages.

Creation of the channel to which clients will subscribe is as simple as using the PUBLISH command to create a value. For example, the following command creates or publishes to a channel called weather with a message of temp:85f:

```
PUBLISH weather temp:85f
```

The message is published to the channel called weather regardless of whether there are any clients subscribed. If there is a client subscribed, the client will receive a message like the following:

```
1) "message"
2) "weather"
3) "temp:85f"
```


TIP

Clients subscribe to a channel with the SUBSCRIBE command. It's assumed that the client would know the format of messages and be able to parse the messages received correctly. Messages are opaque to Redis.

Like other data types in Redis, pub/sub publisher channels can be split to create a hierarchical structure by convention. For example, creating a weather channel by zip code might look like this:

```
PUBLISH weather:54481 temp:85f
```

Clients can then subscribe to the specific zip code for weather updates. Clients can also subscribe in a wildcard pattern to all weather sub-keys, with the `PSUBSCRIBE` command:

```
PSUBSCRIBE weather:*
```

Geospatial

Geospatial indexing is a common pattern used for encoding data that relies on latitude and longitude. This pattern and resultant data makes working with spatial data very easy and very fast. After it's added to the data set, you can calculate things like the distance between two data points using built-in functions.

Creating a data set of locations of radio towers might look like this:

```
GEOADD towers -89.500 44.500 tower1
GEOADD towers -88.000 44.500 tower2
```

You can then calculate the distance between those two towers using the `GEODIST` command:

```
GEODIST towers tower1 tower2
```

The `GEODIST` command returns values in meters by default, but this can be changed to other measures, such as miles:

```
GEODIST towers tower1 tower2 mi
```

Streams

Streams are best thought of as similar to the pub/sub pattern, but with even more power. With pub/sub, data that is published is never stored by the publisher.

Stream consumers create a unique name or identifier for themselves. Because stream publishers store past messages, new

consumers can request to receive all available messages. Additionally, messages can be marked as acknowledged by a given client subscriber.

Streams are created through the `XADD` command, with other commands similar to that of sorted sets such as an `XRANGE` command. You can also view pending messages and perform other powerful operations on streams.

Modules

Redis also has several modules available that further enhance the capability of Redis. This section examines three such modules: Redis Graph, RediSearch, and Redis Time Series.


REMEMBER

Redis Labs offers numerous official modules other than the modules discussed in this section. Some of the official modules include ReJSON, ReBloom, and Redis-ML. Redis has a healthy ecosystem of third-party modules available as well.

Redis Graph

Redis Graph is a module that implements a graph database within Redis. Graph databases provide a method for implementation of graph theory through data. A common example when discussing graph database use cases revolves around identifying relationships between social media users.

With a graph database, each endpoint or node can have zero or more properties. Nodes are then connected to each other through an edge. Like nodes, edges can also have properties of their own.

RediSearch

Another highly useful module is RediSearch. RediSearch is a full-text search engine that features document storage within Redis while enabling high-performance search capabilities.

The RediSearch module enables weighted search results, the use of Boolean logic, autocomplete functionality, and several other common features.

RediSearch can also perform concurrent queries and concurrent indexing. This further enhances performance.

Redis TimeSeries

Storing time-series data is another common task for a database and is also common for NoSQL databases. The Redis TimeSeries module is a high-performance way to store and work with data that is ordered by time.

Data stored with the TimeSeries module can be best thought of like a list but with the added bonus of having a timestamp associated with the data. Time-series-based data facilitates easy metadata retrieval and summarized data queries (such as finding the minimum or maximum timestamp, counting, and so on).

IN THIS CHAPTER

- Understanding clustering and high availability
- Examining transactions and durability

Chapter 4
Redis Architecture and Topology

This chapter focuses on Redis in a production environment, including those elements that organizations need in order to run a highly available enterprise-grade database.

The chapter begins with a look at clustering capabilities of Redis and Redis Enterprise before looking at high availability. Finally, the chapter wraps up with a discussion of transactions and durability in Redis.

Much of the chapter highlights the features that Redis Enterprise brings to a production deployment.

Understanding Clustering and High Availability

A production environment typically requires a certain level of performance and redundancy. Database performance is fulfilled through a number of means, including clustering and sharding.

A database shard is a portion of a larger database. Pieces of a dataset are split among multiple servers, with each server responsible for a subset of the data. Doing so splits the load among the servers.

Redis Enterprise cluster architecture

Redis Enterprise has clustering capabilities built in. With a Redis cluster, portions of a database are shared throughout a set of servers. Each server within a cluster is responsible only for its own set of data.

Cluster management is performed at a different layer of the Redis cluster architecture. This means that requests can be served as quickly as they would be if the server was not running in a cluster.

Within a Redis cluster, a given server is referred to as a node. Each node can be a primary (master) or a secondary (slave) node.

The Redis Enterprise cluster consists of several components:

- **Open Source/Data Layer:** Data is stored and managed at this layer and is the same core as a single instance of open-source Redis.
- **Cluster Manager:** The Cluster Manager is responsible for management of overall cluster health and monitoring, including rebalancing, resharding, provisioning, and de-provisioning nodes, and so on.
- **REST API:** The secure REST API is used for management of the cluster.
- **Zero-Latency Proxy:** Each node of the cluster uses a proxy to provide stateless and multi-threaded communication between client and node.

High availability

Providing high availability in the case of network splits involves running three replicas of the same data simultaneously. In the event of a network failure, the two remaining nodes that can communicate become authoritative.

Organizations using open-source Redis to achieve high availability find the expense of RAM makes doing so costlier and overall

more complex. Redis Enterprise provides high availability without needing a third live replica — instead, it uses a third, much smaller server for quorum resolution in the case of network splits. Providing high availability in this way avoids the need for expensive RAM, which, in any scenario, means direct cost savings.

Redis Enterprise uses in-memory replication between the master and slave. Replication with Redis Enterprise is optimized even more than the open-source Redis. Benchmarks show that Redis Enterprise replication is 37 percent faster than the standard open-source Redis.

Behind the scenes, Redis Enterprise monitors both at the node level and at the cluster level. Node monitoring ensures that processes related to node performance are working correctly. If a node becomes unavailable or unresponsive, the node watchdog begins the shard failover process.

Cluster monitoring with Redis Enterprise watches the health of nodes from an overall view and monitors for network health as well.

TIP

Redis Enterprise also supports multi-AZ (availability zone) deployments.

Running Redis at scale

From an architectural perspective, there are several characteristics found in a fully scaled and production-level Redis deployment. The key to running Redis at scale is using Redis Enterprise. Redis Enterprise makes enterprise-level deployments easy by providing many of the components needed for such an architecture.

Redis Enterprise supports both scaling vertically and scaling horizontally, and the choice is not mutually exclusive. Production environments use scaling to share the load or increase compute capability based on demand.

TECHNICAL STUFF

Scaling up is used when there is available capacity within a server or cluster while scaling out deploys more servers or compute resources and shards the data onto those newly deployed servers.

Redis Enterprise can also scale proxies when necessary. This typically isn't required because proxies are deployed in a redundant configuration and are highly performant on their own. However, when extra capacity at the proxy level is required, Redis Enterprise can do so.

Redis Enterprise also allows for read replicas using a feature called replica-of. The replica-of feature creates another database that can then also be sharded and configured differently than the original.

Redis on Flash

Redis on Flash, available with Redis Enterprise, enables the database to be stored not only in RAM but also on dedicated flash memory such as a solid-state drive (SSD). With Redis on Flash, keys are maintained in RAM while certain values are placed in flash. Specifically, hot values are maintained in RAM and warm values in flash. Redis intelligently chooses which values to place in flash with the implementation of a least-recently-used (LRU) algorithm.

Examining Transactions and Durability

Having the ability to undo a data write in the event of a problem is key to providing reliable data. This section examines durability and transaction support in Redis.

ACID

Atomicity, consistency, isolation, and durability (ACID) describes overall architectural properties of transactional systems, as typically seen with databases, including NoSQL.

Redis supports all capabilities required to be ACID-compliant. This support is accomplished through various methods:

- **Atomicity:** Redis provides transaction-related commands, including `WATCH`, `MULTI`, and `EXEC`. These commands ensure that operations on the database are indivisible and irreducible.
- **Consistency:** Only permitted writes are allowed to be performed through the validation provided by Redis.
- **Isolation:** Being single-threaded, each single command or transaction using `MULTI`/`EXEC` is thereby isolated.
- **Durability:** Redis can be configured to respond to a client write to confirm that a write operation has been written to disk.

A LOOK AT ACID

ACID is a concept that stretches back many years across multiple iterations of database architectures. ACID describes fundamental characteristics that are needed for enterprise database systems:

- **Atomicity:** The ability to ensure that a write or change to data is either fully written to the database or is not committed at all. In other words, no partial writes that could lead to inconsistencies in the data.
- **Consistency:** The data is correct both before and after a transaction occurs.
- **Isolation:** Helps to ensure consistency by requiring concurrent transactions to be separate from each other.
- **Durability:** Data persistence that ensures that when a transaction is complete, it can be retrieved in the event of a system failure.

TIP

Using Redis with the confirmation for writes can affect performance. The next section discusses durability in more detail.

Durability

There are two methods for providing data persistence in Redis:

- **Append-Only File (AOF):** AOF was described in the preceding section. With AOF and the "every-write" setting, Redis replies to the client after the "write" operation has been successfully written to disk, guaranteeing durability.

  AOF applies to every shard of the database and can be configured to write to the database file every second or on every write. The obvious consequence of writing to the disk on every database write operation is slower performance. The benefit is ensuring durability.

  Redis Enterprise handles AOF different than the open-source version of Redis. With Redis Enterprise, AOF is optimized to increase performance. One of the ways this is done is by configuring AOF only from the slave replica, which means the master sees unhindered high performance.

- **Snapshot:** Snapshots are a point-in-time copy of the database. Snapshots apply to all shards within a database and are used primarily for the aforementioned durability rather than as a backup.

Both AOF and snapshot, along with the enhancements available to each through Redis Enterprise, help to ensure that transactions and, indeed, the data remain durable and available at all times.


TIP

Another method for providing a level of durability is through in-memory persistence, which can be both safer and faster.

IN THIS CHAPTER

- Understanding Redis Enterprise
- Getting started with Redis Enterprise

Chapter 5
Using Redis Enterprise Cloud and Software

Redis Enterprise provides an enhanced, enterprise-ready implementation of Redis. This chapter explains Redis Enterprise and the enhancements that make it appealing for so many production workloads today.

Understanding Redis Enterprise

In a modern enterprise environment, performance and reliability are requirements of all applications. Redis Enterprise is the overall name for the enhanced versions of Redis that are focused on the needs of enterprise users.

There are two primary means to deploy Redis Enterprise:

- Redis Enterprise software can be deployed locally within your data center or cloud provider.
- Redis Enterprise can be used as a fully managed and hosted service, Redis Cloud, available in all major cloud providers (even inside virtual private clouds, or VPCs).
- Redis Enterprise can be deployed in a multi-cloud or hybrid on-premises/cloud architecture.

Both methods for deployment result in a high-performance implementation of Redis. The difference between the two is whether you manage the underlying platform or the platform is managed by Redis Labs.

Regardless of how Redis Enterprise is deployed, you receive the same benefits:

- Seamless scaling
- Always-on availability with instant automatic failover
- Multi-model functionality through modules such as RediSearch, ReJSON, ReBloom, and others
- Full durability and snapshots
- Stellar performance

Redis Enterprise also implements geographical distribution in an active-active manner.

CAP THEOREM, ACTIVE-ACTIVE, AND CRDT

CAP Theorem states that it is impossible for a network-based service (such as a server or data shared across a network) to simultaneously provide more than two out of the following three guarantees: consistency, availability, and partition tolerance. Ideally, you would be able to provide consistency and availability of data in a way that was tolerant of network partitions, but in reality, doing so means making trade-offs between the three properties.

Redis Enterprise works with CAP Theorem properties. To ensure availability, Redis Enterprise replicates or copies data across multiple data centers so that an incoming request can be handled by any of the data centers.

Redis Enterprise must also be able to maintain consistency while keeping data available. Redis Enterprise uses conflict-free replicated data types (CRDTs) to maintain consistency and availability of data. Because CRDTs are available across data centers, data within Redis Enterprise is able to handle *network partitions,* or divisions within the network that might otherwise make some or all of the data inaccessible.

Active-active refers to a replication or application model that distributes requests across multiple data centers. Requests can be serviced by any data center, and conflict resolution is used for both simple and complex data types.

Redis Enterprise Software on Docker

Docker makes it easy to develop, test, and deploy an application by placing applications into distinct containers from which they can be deployed and tested.

Redis Enterprise can be run inside a Docker container. To do so, you must first install Docker. After Docker has been installed, executing a simple command will run Redis Enterprise within a container. For example, on a Linux system, here's the command to run Redis Enterprise in a container:

```
$ docker run -d --cap-add sys_resource --name rp
  -p 8443:8443 -p 12000:12000 redislabs/redis
```

Docker will then download the necessary components and begin running Redis inside of the container.

Though using Docker is beyond the scope of what I cover in this book, it's worth noting that the command shown launches Docker with its `run` subcommand. The `run` subcommand accepts several options, a few of which are used here to make Docker go into the background (`-d`), add Linux capabilities (`--cap-add`), and then execute a container named rp (`--name`), exposing two ports: 8443 and 12000 (`-p`).

Redis Cloud

Redis Cloud is a hosted and managed version of Redis in the cloud. With Redis Cloud, you choose the cloud provider from a list of supported providers like Amazon Web Services (AWS), Google Cloud, and Microsoft Azure. Redis is then deployed and managed for you, so that you can focus on development. It's also available in VPC environments of major cloud providers.

Redis Cloud features high availability, seamless scaling without any downtime, and high performance with linearly scaling performance. It's fully monitored, and there is even a free tier.

Redis Cloud's minimal startup cost and effort make it an excellent solution for development environments. It's also ideal for production environments, thanks primarily to its low monthly cost, high availability, and high performance.

Getting Started with Redis Enterprise

Getting started with Redis Enterprise means selecting a platform, either hosted through Redis Cloud or downloadable software. This section looks at those first steps to begin using Redis Enterprise.

Regardless of which method you choose to get started, you need to sign up at Redis Labs. You can do so at `https://redislabs.com`.

Prerequisites

After you have an account at Redis Labs, you can choose which method you'll use for installing Redis Enterprise: using the cloud or locally. If you're looking to test Redis Enterprise through Redis Cloud, you'll still want to install the command-line interface (CLI) so that you can access the instance after it has been deployed.

For downloadable software, you should have at least 2GB of random access memory (RAM) and 10GB of hard-disk space available for a non-production deployment.

Installing Redis Enterprise locally means selecting one of the supported platforms for Redis Enterprise. The choices include the following:

- Ubuntu
- Red Hat
- Oracle Linux
- AWS AMI
- Docker on Mac or Windows

After you've downloaded it, you'll be able to install Redis Enterprise on the chosen platform.

Connecting

Connecting to an instance typically means using the CLI in order to test the connectivity, create an initial database, and so on. The CLI is accessed through the `redis-cli` command that is installed with the server. The CLI provides a set of commands that enable you to work with a Redis server. Similar to a CLI that you might encounter in Terminal on macOS or Linux or the Command Prompt in Windows, the Redis CLI is the work environment that is used for executing commands when not working programmatically.

If you're accessing a cloud-based instance, you'll need to install a CLI to access Redis Enterprise.

When connecting to a *remote instance* (an instance that isn't located on the same server as the CLI), the command looks like this:

```
redis-cli -h <hostname> -p <port>
```

The `<hostname>` is the name of the host to which you're connecting, and the `<port>` parameter is the port number of the instance.

IN THIS CHAPTER

- Getting started
- Creating a CRUD app

Chapter 6
A Simple Redis Application

In this chapter, you take a look at a basic application created to run on Node.js and demonstrating basic create-read-update-delete (CRUD) operations. The application is not meant to show everything that is possible with Redis; instead, it demonstrates the foundations to help kickstart your development.

Getting Started

This section looks at what you need to begin building a Redis application in Node.js. If you don't want to set up your own development environment, you can build this application using the free plan available with Redis Cloud.

Prerequisites

A simple Redis application has been coded and made available on GitHub. The application stores information about cars as an example and is meant to show how CRUD operations can be achieved with Redis as a storage engine. The application is built in Node.js. Follow the instructions in the GitHub repo at `https://github.com/RedisLabs/redis-for-dummies/` to set up the application.

Front-end application code

The primary location for the front-end code is the file called `index.js`. This file sets up the application for our use. The `index.js` file uses a Node.js HTTP server that has routes backed by Redis calls.

The remaining code in `index.js` is used to route or direct clients to the proper location.

TIP

Even though the file is `index.js`, there is no default web page for this application.

Creating a CRUD App

This section looks at usage for three of the common Redis data types: sets, lists, and hashes. As you work through this section, you might use `MONITOR` from the Redis command-line interface (CLI) in order to see what's happening.

Within the code repository, you'll find a shell script called `sample.sh`. The `sample.sh` script creates sample data that will be used in this section. To execute `sample.sh`, you need to have the Redis server and the Node.js application running.

When you run the script, it will generate sample data records by running `curl` commands against the Node.js server. You'll receive output like the following, though the values for the `id` field in the car descriptions may be different:

```
Adding cars
---------
Added a ford-explorer
Added a toyota im
Added a saab 93 aero
Added a family truckster
Done adding cars.

Adding car descriptions
-----------------------
{"id":"cjhvatfuc00005mfj2zycewid"} <-- Added SUV
```

```
{"id":"cjhvatfv500015mfj8nzqk34c"} <-- Added
  Hatchback
{"id":"cjhvatfvo00025mfjyfxuyp6o"} <-- Added Sedan
{"id":"cjhvatfw700035mfjaupal85f"} <-- Added
  Station Wagon
Done Adding car descriptions.

Adding features
---------------
Added power-steering
Added climate-control
Added car-play
Added disc-brakes
Done adding features.
```

With the sample data created, you can look at how to query each data type with `curl`.

I describe the script throughout the rest of the chapter. In general terms, the application maps verbs from REST calls to Redis commands.

You can add your own sample data or use data from another source for these examples, too.

Cars (sets)

The sample data script added a few cars to the Redis instance. These were added as a set.

REMEMBER

Sets are unsorted collections of unique members. To access them, you query to see if a given value exists.

Retrieving the members of a set through the Node application is accomplished by sending a `GET` HTTP request to the cars URL. This results in an `SMEMBERS` call to the car set. Here's an example:

```
curl http://localhost:3000/cars/
```

The `SMEMBERS` command is executed by the server when you make that request. Then you receive the following response:

```
["family-truckster","saab-93-aero","toyota-
  im","ford-explorer"]
```

Other operations are possible, too. For example, an HTTP `PUT` method (executing a `SADD` Redis command) was used to create the original data (in the `sample.sh` script). You can also execute an HTTP `DELETE` method (executing an `SREM` Redis command) to remove a member from the set.

Features (lists)

Another collection of data added by the sample script was features — that is, features you might find in a car. The features were added as a list. Reading data using the application means sending a `GET` request. In this case, retrieving all features looks like this:

```
curl http://localhost:3000/features/
```

Behind the scenes, the `LRANGE` command is executed with `0` and `-1` for the indices, thereby retrieving all values. So, the Redis command is

```
LRANGE features 0 -1
```

Because lists are numerically indexed, you can retrieve based on position within the list. The application supports both start and end index, beginning with `0` for the first item in the list. For example, retrieving all the items beginning with the third item looks like this:

```
curl http://localhost:3000/features/2
```

As before, the `LRANGE` command is executed on the server. Instead of beginning with the `0` index, this time the command begins with index `2` and continues to the end of the list, with the Redis command being:

```
LRANGE features 2 -1
```

You can also set an end index. Retrieving only the third item looks like this:

```
curl http://localhost:3000/features/2/2
```

This time, LRANGE is executed with the same beginning and ending index (2). The full Redis command is

```
LRANGE features 2 2
```

Create, read, update, and delete operations are supported for lists in this application. To create, send a POST request (using LPUSH); to update, send a PUT request (LSET); and to delete, use the DELETE method (LREM).

Car descriptions (hashes)

The sample script added data to the hash data structure in the Redis instance. When the data was added, the first argument in the ZSCORE/HGETALL is the key, containing the unique ID. Working with hash data means creating and requesting that unique ID.

Retrieving the details of a car by its ID looks like this:

```
curl http://localhost:3000/cardescriptions/
  cjhvatfuc00005mfj2zycewid
```


REMEMBER

The unique ID will be different for your database.

When an ID is used in this manner, the ZSCORE command is executed by the server against cardescriptions:collection. This is followed by the HGETALL command. In all, it looks like this:

```
ZSCORE cardescriptions:collection
  cjhvatfuc00005mfj2zycewid
HGETALL cardescriptions:details:cjhvatfuc00005mfj2
  zycewid
```


TIP

The keys used in this chapter are very large. Large keys work well in heavily used applications in order to help avoid overlapping keys. However, you can use more compact unique IDs in your application.

You can see all unique IDs by sending this request:

```
curl http://localhost:3000/cardescriptions/
```

Behind the scenes, the `ZREVRANGEBYSCORE` command is executed when the call to `/cardescriptions/` is made, so this will show all the items in the sorted set. The entire command is

```
ZREVRANGEBYSCORE cardescriptions +inf -inf
```

Like other data types in this application, you can also create (`ZADD`/`HMSET`), patch (`HMSET`), and delete data (`UNLINK`/`ZREM`) stored in hashes.

IN THIS CHAPTER

- Understanding conflict-free replicated data types
- Getting started with conflict-free replicated data types
- Seeing conflict-free replicated data types in action

Chapter 7
Developing an Active-Active/Conflict-Free Replicated Data Type Application

In this chapter, you develop an application using the Active-Active mode of Redis Enterprise implemented via conflict-free-replicated data types (CRDT). I start by defining CRDTs and how they differ from other replication methods.

Getting Acquainted with Conflict-Free Replicated Data Types

This section provides some background on CRDTs. I start by defining them. Then I explain how CRDTs differ from other replication methods. Finally, I offer some thoughts on where and when you'll use CRDTs.

Defining conflict-free replicated data types

CRDTs are a special data structure that enables multiple copies of data to be stored across multiple locations in such a way that each copy can be updated independently. The conflict-free part is due to the fact that this data type can resolve any inconsistencies without intervention.

Looking at how they're different

CRDTs differ from other replication methods in that there doesn't need to be extensive communication between copies — or nodes — involved in a CRDT. When a conflict between two nodes occurs, the condition for choosing which data to use is not based on the wall clock; instead, it's based on a mathematically derived set of rules.

Conflicts are resolved at the database level with CRDTs and are consensus free. This resolution is done without user intervention.

The end result is that CRDTs are faster and provide fault tolerance. Application development is also easier, making the process quicker.

Understanding why and where you need them

CRDTs are valuable for high-volume data that requires a shared state. Additionally, CRDTs can use geographically dispersed servers in order to reduce latency.

The geographic dispersal, also called *geolocal servers*, enables high availability even during network or regional network failures. Disaster recovery also occurs in real time.

TIP

Several data types can be used as CRDTs in Redis, including hashes, strings, strings-as-counters, sets, sorted sets, and lists.

Working with Conflict-Free Replicated Data Types

In this section, you begin to build the application. You install prerequisites in this section and have a good understanding of where you're headed with the application.

Getting an overview of the application

To set up a demonstration in a reasonable amount of time and with a reasonable amount of effort, you'll be creating an environment that simulates a much larger architecture. The overall premise is to use Docker containers for the simulation.

The application being demonstrated runs as a single node and also works great with CRDTs. The end result shows how CRDT-based data converges.

The application simulates a geo-replicated topology to reduce latency. It's worth noting that replication occurs across clusters and not across individual shards or nodes.

Considering the prerequisites

The application uses Docker to make it easy to see the active-active nature of the application and of Redis itself. You need to install Docker before continuing.

TIP

The installation of Docker is beyond the scope of this chapter, but you can find instructions at `https://docs.docker.com/install`.

The application discussed in this chapter also requires the use of multiple Redis Enterprise instances, each of which runs in a Docker container. The application requires more resources than those required for the preceding chapter. For example, the application in this chapter requires 8GB of RAM for each instance of Redis Enterprise.

The example application also uses Node.js. You may have already installed Node.js as part of Chapter 6, but if not, you can get more information on installing Node.js at `https://node.js.org`.

Finally, the files for the application itself are contained on GitHub and can be found at `https://github.com/RedisLabs/redis-for-dummies`. Within that repository, the directory `/crdt-application/` contains the files for the application in this chapter.

Starting the containers

To start Docker and the Redis Enterprise containers, run `create_redis_enterprise_clusters.sh`.

TIP

You may need to make the script executable, depending on your platform. This typically entails running `chmod 700 <scriptname.sh>` from a command prompt or terminal window.

Running the script creates two network (one for each cluster) and two clusters:

- **172.18.0.2:** Runs Redis on port 12000 and an administrative port of 9443. The administrative port is forwarded to port 8443 on your local development environment.
- **172.19.0.2:** Runs Redis on port 12002 and an administrative port of 9445. The administrative port for this cluster is forwarded to port 8445 on your local development environment.

The `create_redis_enterprise_clusters.sh` script also connects the two networks so that they can communicate.

The creation script takes a few minutes to execute, depending largely on the amount of resources such as CPU and RAM available and whether Docker needs to download the image. You can test whether the instances are up and running by pointing a web browser to `http://localhost:8443` and `http://localhost:8445`. If you see a setup prompt, the instances are working.

WARNING

Do *not* follow the prompt. You'll set up the clusters automatically using a script.

Although there is a user interface for creating the clusters, you'll do so automatically via the command line. To do so, run `setup_redis_enterprise_clusters.sh`.

This script will configure two clusters in each Docker container. The clusters have the sample username of r@r.com and a password of test.

WARNING

Do not use these clusters in a production environment or in an environment that may be otherwise compromised. The clusters should be used for testing only and have very little security hardening.

The final step to get the clusters configured is to run `join_redis_enterprise_clusters_crdb.sh`. You may need to change the permissions on this script, just as with the previous scripts executed in this section. See the tip earlier in this section for more details.

The `join_redis_enterprise_clusters_crdb.sh` script accesses the Redis API to join the clusters. This same task could be accomplished through the user interface, too, but using the API makes it easy.

The final outcome of the script will be to join the two clusters in a conflict-free replicated database spanning both clusters.

Testing the conflict-free replicated data type

The first step in testing the CRDB is to connect to each cluster. Execute the following command to connect to the first cluster:

```
redis-cli -p 12000
```

This command invokes the Redis command-line interface (CLI) and attempts to connect using port 12000.

If you receive a `>` prompt, you're connected and you can execute the following commands within the CLI:

```
SET test hi
EXIT
```

Now connect to the second cluster. To do so, use the `redis-cli` command, but this time use port 12002, like so:

```
redis-cli -p 12002
```

As before, if you're connected, you'll see a `>` prompt.

When you're done within the CLI, type **EXIT** to end your CLI session. This command was included in the previous example but is not shown in subsequent examples.

Retrieve the previously set test key and then change the test key by running the following commands:

```
GET test
"hi"
SET test howdy
```

Finally, connect back to the first cluster and retrieve the test key:

```
redis-cli -p 12000
```

At the prompt, retrieve the test key:

```
GET test
"howdy"
EXIT
```

If these tests are successful, the clusters are communicating properly.

Watching Conflict-Free Replicated Data Types at Work

The code example illustrates a simulated Internet of Things (IoT) configuration that tracks cars as they enter a monitored street. In the configuration, multiple sensors are simulated in order to report cars passing by to track which roads they're on and which position marker on the road they've passed.

In the simulation, each sensor can be connected to the geographically closest cluster to achieve the lowest latency.

As simulated cars pass a marker, they're idempotently added to a set using the `SADD` command and then added to a hash that contains an incrementing counter (`HINCRBY`) to indicate how many markers have been passed.

Setting up the examine code environment

The sample code requires its own environment installed through Node.js. It's worth noting that the example code shows just one way to use CRDTs. There are numerous others, and the example commands can usually be executed whether in cluster mode or when using them on a single cluster or even in a single instance.


TIP

The instructions here give the most common example command. See the `README` file within the example code for specific details, updates, and notes about the example code.

From within the `/cdrt-application/` directory, execute the following:

```
npm install
```

Viewing the example with a healthy network

Behind the scenes, several Redis commands are executed by the code. These commands add a set and perform other related commands in order to achieve the desired result.

The example code runs the following commands:

```
SADD all-roads {passed road from command line}
MULTI
SADD roads:{passed road from command line} {passed
  plate}
HINCRBY road-marker:{passed road from command
  line} {passed plate} 1
EXEC
```


TIP

You can view the commands in real time on the first cluster by executing the following from within another window:

```
redis-cli -p 12000
> MONITOR
```

Connect to the second cluster by changing the port to 12002 instead of 12000 in order to see the commands being executed on the second cluster.

The client can be connected to either cluster. On the first cluster, execute the following:

```
node car.js marker  91street --plate 1234
  --connection ./rp2.json
{
  "entered": "91street",
  "onRoadPreviously": false,
  "marker": 1
}
```

On the second cluster, execute the following:

```
node car.js marker  91street --plate 1234
  --connection ./rp1.json
{
  "entered": "91street",
  "onRoadPreviously": true,
  "marker": 2
}
```

Note how the incremented value is coordinated across the clusters.

Now add another car:

```
node car.js marker 118avenue --plate 4567
  --connection ./rp2.json
node car.js marker 118avenue --plate 4567
  --connection ./rp1.json
```

Viewing the roads on either cluster shows synchronization in action. To view the roads, run the following command:

```
node car.js viewroads --connection ./rp1.json
{
  "118avenue": {
    "4567": "2"
  },
  "91street": {
    "1234": "2"
  }
}
```

```
node car.js viewroads --connection ./rp2.json
{
  "118avenue": {
    "4567": "2"
  },
  "91street": {
    "1234": "2"
  }
}
```

As you can see from the results, both clusters are the same and, thus, synchronized. Behind the scenes, the `viewroads` command executes the following:

```
SMEMBERS all-roads
```

Then it executes the following for each member of the road set:

```
HGETALL road-marker:{a member from the previous
  set}
```

Breaking the network connection between clusters

In this section, you use Docker to simulate a break in the network connection. Execute the following script, after making it executable if necessary:

```
split_network.sh
```

After that command has been executed, the client software from the example code is still communicating with each cluster but the clusters themselves are no longer communicating with each other.

Viewing the example in a split network

Now you'll execute commands to demonstrate how the example operates in a split network configuration.

Run the following:

```
node car.js marker 118avenue --plate 4567
   --connection ./rp1.json
node car.js marker  91street --plate 1234
   --connection ../rp2.json
```

Then run `viewroads`:

```
node car.js viewroads --connection ./rp1.json
{
  "118avenue": {
    "4567": "3"
  },
  "91street": {
    "1234": "2"
  }
}
node car.js viewroads --connection ./rp2.json
{
  "118avenue": {
    "4567": "2"
  },
  "91street": {
    "1234": "3"
  }
}
```

Now that the two networks are split, the clusters no longer maintain synchronization with each other. Updates can continue on each cluster while the network is split. However, the clusters can rejoin at any time and no updates will be lost when the clusters rejoin.

Rejoining the network

Reconnect the networks with the `rejoin_network.sh` script, making it executable if necessary:

```
rejoin_network.sh
```

It will take a few seconds for the clusters to discover that they are reconnected, after which time the clusters will reconnect and synchronize without any intervention. All data will be converged using CRDT semantics, and no data will be lost.

Looking at the example in a rejoined network

Now let's look at the example in a network that has been reconnected after a split.

Execute the following:

```
node car.js viewroads --connection ./rp2.json
{
  "118avenue": {
    "4567": "3"
  },
  "91street": {
    "1234": "3"
  }
}
node car.js viewroads --connection ./rp1.json
{
  "118avenue": {
    "4567": "3"
  },
  "91street": {
    "1234": "3"
  }
}
```

Now pass a marker on each road to see how the data is synchronized again:

```
node car.js marker  91street --plate 1234
   --connection ./rp2.json
{
  "entered": "91street",
  "onRoadPreviously": true,
  "marker": 4
}
```

```
node car.js marker  91street --plate 1234
  --connection ./rp1.json
{
  "entered": "91street",
  "onRoadPreviously": true,
  "marker": 5
}
```

As previously stated, the example shown in this chapter is just one of many ways that a CRDT can be used. The underlying and essential elements of Redis are the same when using them standalone or with a single cluster.

Chapter 8
Ten Things You Can Do with Redis

This whole book is about what Redis can do for you. This chapter lists ten things you can do with Redis.

TIP

A single Redis cluster can be used to do any of these ten things, regardless of whether it's a transactional or analytical workload.

- **Use it as your primary database.** Redis is not just a NoSQL database. It goes well beyond NoSQL to implement numerous features for today's enterprise customers. Redis is more than simple key/value storage — it provides multiple data models and multiple methods to access data.

  Redis can be utilized by the entire application stack within an organization.
- **Cache most frequently used pieces of data.** Load data from slower data sources into Redis and provide near-instant response times. Redis keeps data in random access memory (RAM) to make retrieval fast.
- **Use it for session storage.** Session storage requires very fast response times, both for writing data as users progress through an application and for reading that information back.

Redis is an excellent fit for session storage due to its native data-type storage that mirrors the kind of storage needed for storing session data.

- **Decouple services.** Redis streams and the publish/subscribe pattern enable service decoupling. Services can write to and read from Redis streams or can publish and subscribe send messages using Redis as the facilitator of the pub/sub pattern.
- **Provide rate limiting.** Redis can be used to rate-limit users and endpoints. The high-performance, real-time nature of Redis means that tracking can be done in real time along with the users and endpoints.
- **Ingest data quickly.** Redis is known for its capability to work with large amounts of data at speed. Consuming or taking in data in large quantities and then processing it or handing it off for further processing makes Redis a great choice for data ingest.
- **Build real-time leaderboards.** Native data types that promote sorting and counting operations enable Redis to be used as the back end for real-time leaderboards.
- **Build a store finder.** Redis includes GEO-based data types that natively handle geospatial data like latitude and longitude calculations. A store finder is another use case where Redis is the compelling solution.
- **Perform analytics efficiently.** Data that needs to be processed can be stored in Redis in a compact manner. Data that may take terabytes in another storage medium can be processed in such a way that it requires significantly less resources when you use Redis. For example, probabilistic data structures can be used that then help to maintain counts, frequencies, and percentiles very efficiently.
- **Index large amounts of data.** Redis handles large amounts of data well. As an organization and its application portfolio grow, so does the amount of data. Redis has the flexibility and extensibility (through modules) to store data for multiple consumers and the performance and efficiency to store large amounts of data for established and new organizations alike.

Learn how Redis powers modern applications

Enhance your database skills by learning about Redis, a highly popular multi-model in-memory NoSQL database. Understand the primary data models, patterns, and structures used in Redis and see how to develop a high performance application. Go beyond the basics to understand how Redis is powering the instant experience in the real world with use cases such as caching, session stores, geospatial indexing, full-text search, and time-series processing.

Inside...

- Examine Redis data models, structures, and patterns
- See real-world examples of Redis-powered applications
- Explore Redis clustering and high availability
- Get exposure to Redis modules like Search and Graph

Steve Suehring is an Assistant Professor of Computing and New Media Technologies at University of Wisconsin–Stevens Point. Prior to joining the faculty at UWSP, Steve worked for nearly 20 years in industry. Steve is also the author of several technology books.

ISBN: 978-1-119-52080-1
Not For Resale

for dummies®
A Wiley Brand

WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley’s ebook EULA.