LEARNING MADE EASY

2nd Limited Edition

Redis™

for dummies®
A Wiley Brand

Discover Redis data structures and modules

Create applications with Redis

Learn about Redis use cases

Courtesy of

redislabs
HOME OF REDIS

Steve Suehring

About Redis Labs

Data is the lifeline of every business, and Redis Labs helps organizations reimagine how quickly they can process, analyze, make predictions with, and take action on the data they generate. As the home of Redis, the most popular open source database, we provide a competitive edge to global businesses with Redis Enterprise, which delivers superior performance, unmatched reliability, and the best total cost of ownership. Redis Enterprise allows teams to build performance, scalability, security, and growth into their applications. Designed for the cloud-native world, Redis Enterprise uniquely unifies data across hybrid, multi-cloud, and global applications, to maximize your business potential.

Learn how Redis Labs can give you this edge at redislabs.com.

With more than 48,000 GitHub stars, 19,000 forks, and 430+ contributors, Redis is an incredibly popular open source project supported by a vibrant community. Redis has been voted the most loved database, rated the most popular database container, the #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


https://developer.redislabs.com

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 and 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™

2nd Limited Edition

by Steve Suehring

for dummies®
A Wiley Brand

Redis™ For Dummies®, 2nd Limited Edition

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

ISBN 978-1-119-82427-5 (pbk); ISBN 978-1-119-82428-2 (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:

Project Editor: Elizabeth Kuball
Acquisitions Editor: Ashley Coffey
Editorial Manager: Rev Mengle
Business Development Representative: Matt Cox
Production Editor: Tamilmani Varadharaj

Table of Contents

Introduction

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

Redis and Redis Enterprise are market-leading, multi-model NoSQL databases that bring NoSQL to organizations both big and small. Redis is open source, and Redis Enterprise software adds several enhancements that are important to the enterprise customer. Redis Enterprise Cloud enables Redis Enterprise deployments on popular cloud providers like Amazon Web Services (AWS), Google Cloud, and Microsoft Azure.

About This Book

This book provides a starting point for those new to NoSQL and those who have heard about NoSQL but would like to see how it might be used in their organization.

The book serves multiple audiences, with chapters geared towards managers and chapters specifically for developers. You don't *need* to read this book from front to back, but you certainly can!

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 an environment available on which you can install things. I show examples in later chapters using Redis that also utilize Docker and GitHub, so having a development environment available will be helpful, but it isn't required — you can follow along even if you don't run the examples yourself.

Icons Used in This Book

Throughout this book, I occasionally use special icons to call attention to important information. Here's what to expect:

The Remember icon points out information you should commit to your nonvolatile memory, your gray matter, or your noggin — along with anniversaries and birthdays!

You won't find a map of the human genome here, but if you seek to attain the seventh level of NERD-vana, perk up! The Technical Stuff icon explains the jargon beneath the jargon!

Tips are appreciated, never expected, and I sure hope you'll appreciate the nuggets of information marked by the Tip icon.

The Warning icon points out the stuff your mother warned you about. Well, probably not, but these paragraphs do offer practical advice to help you avoid potentially costly or frustrating mistakes.

Where to Go from Here

There's a lot more to Redis than is covered in this book. To continue exploring all the capabilities that Redis offers, visit `https://redislabs.com`.

IN THIS CHAPTER

- Getting acquainted with NoSQL
- Understanding what Redis brings to the table
- Creating a multi-model database with Redis

Chapter 1
What Is Redis?

In this chapter, I offer an overview of NoSQL, including the types of NoSQL databases (such as key/value, document, column, and graph). I also compare NoSQL to methods for traditional data persistence. Finally, I introduce Redis, a popular multi-model database server. Redis goes beyond a 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.

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* (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.

Figure 1-1 illustrates the concept of key/value pairs using a phone directory. In this example, a person's name provides the key and the value is then the corresponding phone number.

Phone directory

Key	Value
Paul	(091) 9786453778
Greg	(091) 9686154559
Marco	(091) 9868564334

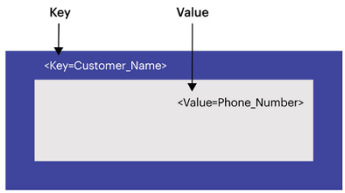

FIGURE 1-1: A phone directory is an example of a key/value pair data store.

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.

Figure 1-2 shows an example of a document database in JavaScript Object Notation (JSON) format. The example shows information about a book, such as its unique identifier, title, author, and year. Each of these items could then be parsed using a JSON parser into key/value, as shown in the figure.

A Document

```
{
"BookID" : "978-1449396091",
"Title"  : "Redis - The Definitive Guide",
"BookID" : "Salvatore Sanfilippo",
"Year"   : "2013",
}
```

Key	Value
BookID	978-1449396091
Title	Redis - The Definitive Guide
Author	Salvatore Sanfilippo
Year	2013

FIGURE 1-2: A document database example containing information about a book.

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. Figure 1-3 depicts relationships between three different entities, each containing labels that help to describe the entity.

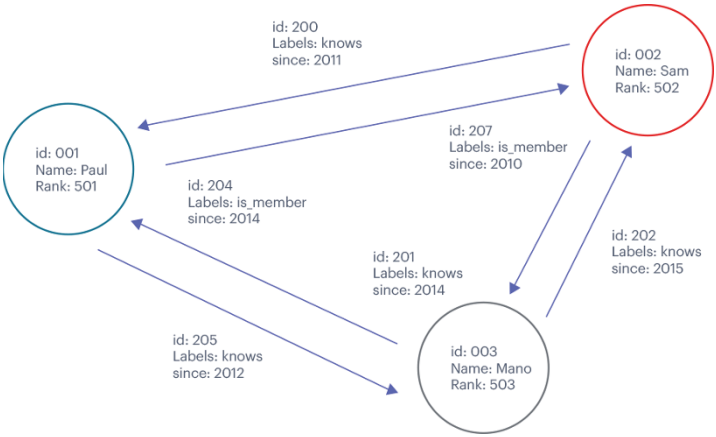

FIGURE 1-3: A graph database depicting relationships between data elements.

Knowing when to use NoSQL versus a relational database

Regardless of the type of NoSQL database, the patterns and tools that you use to work with data are different from the patterns and tools that you typically find with a relational database. The paradigm for storage and the arrangement of the data typically require 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.

A relational database has somewhat less flexibility than software such as Redis that implements NoSQL. 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. Redis can implement multiple representations of a data model. You can find more information on Redis as a multi-model database later in this chapter.

Deciding when to use a NoSQL database

Comparing a NoSQL database to a relational database may have you thinking about the specific uses for NoSQL. NoSQL excels when fast access to large amounts of data is needed. NoSQL is also excellent at enabling developers to work with a flexible data model, as is frequently the case with modern applications. In these scenarios, the data model may not be immediately or fully known. However, developers need to get started programming the application itself and can use NoSQL for flexible data models that support both semi-structured and unstructured data.

When it comes time to move to production, a NoSQL database offers highly available replicated data. Unlike a traditional relational database, NoSQL data is replicated without need for primary read/write and secondary read-only nodes. Replicated data is kept consistent by design, and NoSQL databases utilize a shared-nothing approach to ensure high availability. Both traditional and NoSQL databases handle scaling through sharding but the consistency model required for horizontal scaling requires significant forethought and effort with a relational database.

Sharding is a means to partition or split data into smaller pieces that are distributed to different computing resources. For instance, data may be sharded according to the geographic location where it is most frequently used and then stored in a data center close to the users in order to decrease latency.

Read/write speed is typically much higher with NoSQL when compared with a traditional relational database system. NoSQL also thrives with unstructured data, whereas a relational database needs to have a schema declared prior to any data being entered into the system. Schema changes due to new requirements can require significant rework with a traditional database. Queries with a normalized relational data model can be quite time- and resource-intensive, which is not usually acceptable for real-time application needs. These complex queries don't scale well either.

NoSQL thrives in the modern application development life cycle and operates at Internet scale with ease. Look first to NoSQL databases when you're working with unstructured data that changes frequently or needs to scale up and out, or when large volumes of data are involved.

Seeing Where Redis Fits

Redis is a NoSQL database, but it's also much more. Redis is a multi-model database enabling search, messaging, streaming, graph, and other capabilities beyond that of a simple data store.

Data storage

Redis keeps data in memory for fast access and persists data to storage, in addition to replicating 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
- HyperLogLogs
- Streams
- Geospatial Indexes

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 (Pub/Sub) pattern and additional patterns that make Redis suitable for modern data-intensive applications. Geo data structures include commands that analyze geospatial data in order to calculate distances, find members within a particular distance from each other, and other analysis pertinent to geographically-relevant data. This design enables reduced code complexity, reduced network consumption, and overall faster execution.

Working with Multi-Model Application Requirements

Multi-model databases support multiple data models with a single, integrated back-end server application. Redis provides full multi-model functionality through Redis Modules. Redis Modules are customized, prebuilt functionality that enable the addition of data structures in a modular fashion. As such, Redis Modules open the door to adding other data models, such as graph and full-text search. Redis Modules also lead to other capabilities such as improved JSON support, secondary indexes, linear algebra, SQL support, and image processing. The use of Redis as a multi-model database enables greater flexibility for application developers within an organization.

The single-model problem

A single data model, a choice typically made at the beginning of a project, can prove to be a development and performance bottleneck as the project and application evolve. At the very least, decisions made about how to represent and interact with data can incur technical debt that is difficult to pay back.

One approach to solve the technical debt issue is with a multi-model database. However, multi-model databases frequently only gloss over the issue by adding application-level or integration-layer code to present the data using a different model. Redis takes a different approach.

The modules solution

Redis modules make it possible to extend Redis functionality using external modules, rapidly implementing new Redis commands with features similar to what can be done inside the core itself. Redis Modules is designed to plug into the open-source Redis database, taking advantage of functionality such as in-memory processing, scalability, and high availability. Through its modularity of design, Redis provides capabilities necessary for many different types of applications.

IN THIS CHAPTER

- Seeing how Redis can help you
- Looking at a real-world example

Chapter 2
What Is Redis Used For?

In this chapter, I give you an overview of some of the specific ways that Redis can help. Chapter 1 shares a bit more about specific components of Redis, including Redis Modules. This chapter concludes with an example of how Redis is used for real-time practical applications.

Identifying How Redis Can Help You

This section examines several 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 — improves the entire user experience.

Real-time analytics

Redis is exceptionally good for real-time analytic calculations like top scores, top-ranked contributors, top posts, and more. Fast-paced calculations of the type needed for instant scoring in a game or last-minute bidding in an auction can use Redis Sorted

Sets. Sorted Sets keep track of ordering automatically, enabling you to obtain top scores or the high bidder with native commands like ZRANGE and ZREVRANGE. Because Sorted Sets do the work behind the scenes when ZADD is used to add values, you no longer need to expend resources sorting the result set.

Fraud detection

Detecting fraud in transactions is always important. As attacks become more sophisticated, the need for specialized and instant fraud detection becomes increasingly vital. As a customer navigates through a shopping or financial transaction, a record of that transaction can be captured in Redis Streams, fed into Redis Bloom for fraud probability scoring, and coupled with RedisAI to provide full analysis of the transaction. As transactions move through the system, technologies such as Redis TimeSeries can help spot trends that may indicate fraud.

Gaming and leaderboards

Although real-time analytics alluded to this use case, the highlight around gaming and leaderboard tracking with Redis is the ability to run at scale and increase data-set size without affecting performance or needing to re-architect the application code. Additionally, Internet-based gaming naturally enables players from around the world who expect instant feedback. The high-availability nature of Redis enables data to be geographically distributed to shorten the distance and, thus, the latency between the player and their scoring status.

Sorted Sets are used for gaming and leaderboards. Commands such as HSET help to store values in a structured manner for easier retrieval. The resulting hash can then be associated with a Sorted Set to take full advantage of the native capabilities for sorting data automatically.

Personalization with session management

A session is loaded when a user logs in or when they're using the application in order to track their 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 Hash data structure enables data to be stored in multiple fields while facilitating the flexibility to use other data structures. In the event of an outage, immediate failover is still available even though the data is maintained within the low-latency in-memory storage.

Recommendation management

Redis Sets enable easy tracking of items by simple tagging, which facilitates a recommendation engine for products. For example, data from users with similar purchases that is stored as a Redis Set can be analyzed for common items by using the SINTER command to look for the intersection of products. The SADD command is used to tag each product with keywords to help with this use case.

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 provides the performance necessary for these applications.

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

- Intelligent caching
- Publish/subscribe (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. Because it's native JavaScript, JSON-formatted data can be used directly in an app without needing to be transformed into another format.

Search

RediSearch is a powerful indexing, querying, and full-text search engine for Redis. Allowing users to search data is difficult.

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. RediSearch enables full-text search experiences with powerful indexing capabilities built in. RediSearch indexing is available across multiple languages (including Chinese, English, French, German, Russian, Spanish, and several others). With indexing, Redis no longer needs to execute a SCAN operation for each query, which speeds up queries significantly. RediSearch enables fast creation of indexes on datasets (Hashes), and RediSearch uses an incremental indexing approach for rapid index creation and deletion. The indexes let you query data at lightning speed, perform complex aggregations, and filter by properties, numeric ranges, and geographical distance.

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 list certainly isn't exhaustive, 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 problem is often solved with caching.

Redis can be used as a means to cache data between the application and the back-end 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 data sets

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 data-set analysis cost-effective. In this use case, Redis Enterprise Flash is used to extend random access memory (RAM).

Full-text fuzzy search

The RediSearch module is used to extend the capabilities of Redis and enable searching through relevant documents in the shortest possible time. 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 this is done with the performance that you'd expect from Redis.

RediSearch enriches search experiences with context-aware suggestions and fuzzy searching. RediSearch stores data in RAM and can be scaled onto multiple Redis instances. RediSearch also allows for Fuzzy Suggestions, meaning you can get suggestions to prefixes even if the user makes a typo in their prefix.

Geospatial and time-series data

Redis, with its native geospatial index, hash, sorted set, and stream data types (see Chapter 1), is an excellent choice for geospatial and time-series data. These data types may 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 may be able to relay the message to open additional lanes or that there is another issue that needs attention. RedisTimeSeries helps with sequencing of events and facilitates trace-back of events in the exact order in which those events occurred.

Messaging/queuing

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

IN THIS CHAPTER

- Identifying the components of Redis
- Exploring the deployment of Redis
- Installing Redis and making your first connection
- Working with Redis clients

Chapter 3
Getting Started with Redis

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 Unix-based operating systems, like Linux and macOS) and through the Windows Subsystem for Linux (version 2). The Redis server is downloaded and installed in just a few steps, and then it's ready to use.

TIP

The installation process for Redis is fully documented in the Quick Start guide available at `https://developer.redislabs.com/create`.

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. Through these clients and drivers, you interact programmatically with data found on the 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.

TECHNICAL STUFF

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:

- **C:** The official Redis client for the C language is hiredis. Also see hiredis-vip for cluster-related C language support.
- **C#:** Two popular clients include ServiceStack.Redis and StackExchange.Redis.
- **Go:** Clients for Google Go and connectors are available; one is Go-Redis and the other is Redigo.
- **Java:** Three popular clients include Jedis, Lettuce, and Redisson, all of which serve slightly different needs.
- **Node.js:** Two clients, ioredis and node-redis, offer a rich feature set for using Node.js with Redis.
- **PHP:** PHP has several clients with PHP; PhpRedis is a recommended client.

Databases, memory, and persistence

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

Those familiar with formalized database creation and definition may be uncomfortable with the seemingly informal process of Redis 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.

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.

Deploying Redis

There are numerous options for working with Redis, largely depending on the goal for the deployment. Redis enables you to install and start using the software quickly on multiple platforms. This section shows some of the ways that you can deploy Redis.

Using Redis Enterprise Cloud

Redis offers a free 30MB plan with major cloud vendors such as Amazon Web Services (AWS), Google Cloud, and Microsoft Azure. When you sign up for Redis Enterprise Cloud at `https://redislabs.com/try-free` you'll receive an email with instructions on how to activate the free plan.

After activation, follow these steps:

1. **Choose a cloud provider and a region for the deployment of Redis Enterprise Cloud (see Figure 3-1), and scroll down to continue on this page.**

   In Figure 3-1, the us-east-1 region for AWS is selected.

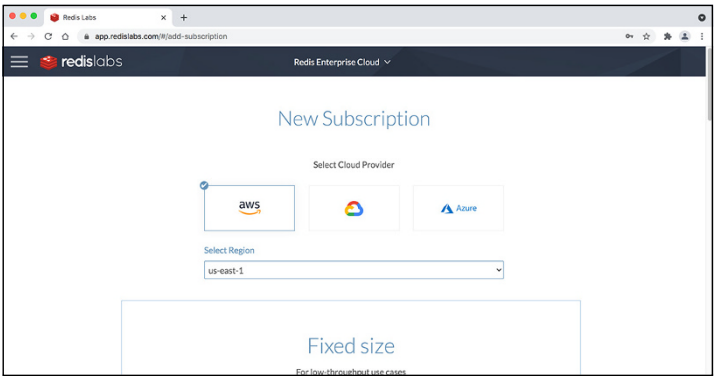

FIGURE 3-1: Choosing a cloud provider and region.

2. **Choose the size to use for the deployment, and click Create.**

   The free 30MB option is available along with other, paid, options. If you're just getting started with Redis, the free 30MB option enables you to quickly get a sense of its

capabilities. Figure 3-2 shows the 30MB option selected with a subscription name of Product-Test. The 30MB option creates a single subscription with up to 30 connections.

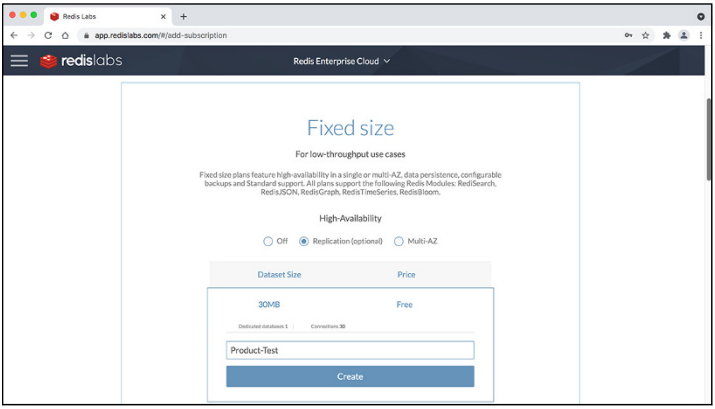

FIGURE 3-2: Creating a Redis Enterprise Cloud subscription.

3. **Select and configure various parameters related to the database, and click Activate.**

   These parameters include the database name, protocol, whether replication is enabled or disabled, access control and security, the data eviction policy, alerts, and the module(s) to include. Figure 3-3 shows the creation of a database called ProductDB with the other options left at their default.

   After you click Activate, the database configuration parameters are shown.

Now you're ready to connect to the database endpoint and begin working with the database and integrating your application. You can find more information about this next step in the "Taking the First Steps with Redis" section, later in this chapter.

Compiling Redis from source

Redis can be compiled from source code. This option is useful when you have extended needs that aren't available in a pre-packaged option. It's also very helpful when you want to use new features or an early release of Redis. Compiling from source is typically done on a Linux environment; the examples in this section use Debian Linux, though other distributions will have a similar, if not the same, process.

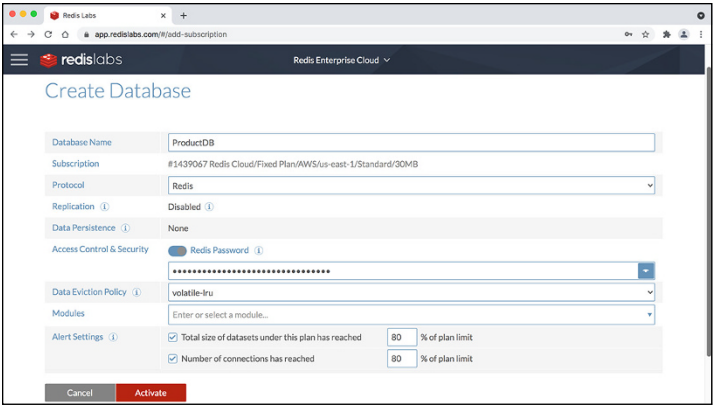

FIGURE 3-3: Creating a database as part of a Redis Enterprise Cloud subscription.

You can download the source code from `https://redis.io/download`. Redis source code is downloaded as a compressed `.tar` archive. After you've downloaded the file, follow these steps:

1. **Uncompress and unarchive the software with the `tar` command.**

   For example, at the time of writing, the latest version of Redis was 6.2.2, making the command as follows:

   ```
   tar -zxvf redis-6.2.2.tar.gz
   ```

   Adjust the command, as necessary, to match the version of Redis that you've downloaded.

   Running the `tar` command with the options shown results in the source code being uncompressed into its own directory. In the example with version 6.2.2, the source code is placed into a directory called `redis-6.2.2`.

2. **Change the directory into the source code directory and begin the compile process.**

   You have some options when it comes to how the compile and the resulting software behave. For example, you can configure Redis to work with systemd. Doing so requires that systemd-related development libraries are available. More information about systemd integration and the other options for compiling Redis are found in the `README.md` file, which can be viewed with any text editor or terminal pager.

Using the default option, changing into the Redis source code directory is accomplished with the `cd` command, and compiling Redis is done with the `make` command.

```
cd redis-6.2.2
make
```

REMEMBER

As before, adjust the command, as necessary, to match the version of Redis that you downloaded.

The compile process begins. If all goes well, you'll receive a message indicating that it's a good idea to run `make test` next. However, if the compile process doesn't work correctly, an error and possibly a backtrace will be shown.

A common reason for errors during the compile process is missing one or more dependencies. The dependency or dependencies will need to be installed. After you've done that, it's a good idea to clean up any cached files from the previously failed compile attempt by executing the `make dist-clean` command.

3. **After the compile has worked, run the tests included with the source code.**

   Running the tests can be done with the following command:

```
make test
```

   The test process will take longer than the compile process, but you'll have more confidence that the installation will work correctly.

With the compile and testing complete, the Redis server is ready. When compiled from source, the server is located in the `src` directory. You can also install the software into the `/usr/local/bin` directory with the following command:

```
sudo make install
```

However, if you're merely testing Redis, running from the `src` directory, like this, works fine and doesn't require sudo privileges:

```
cd src && ./redis-server
```

The server starts and runs on port 6379 by default. You can stop the server by pressing Ctrl+C.

Using Redis in Docker

Assuming that you have Docker installed, you can pull the latest version of the Redis Docker image from DockerHub with the command:

```
docker pull redis
```

After the image has been pulled, run Redis in the container with the `docker run` command. In this example, a container execution called `product-test` will be executed:

```
sudo docker run --name product-test -d redis -p
  6379:6379
```

More information about installing Redis in Docker can be found at `https://hub.docker.com/_/redis`. If you need to install Docker itself, more information can be found at `www.docker.com`.

Homebrewing for macOS

Redis can also be installed natively on a Mac using Homebrew. Installing Redis in a Homebrew environment is as simple as using the following command:

```
brew install redis
```

Additional information about Homebrew can be found at `https://brew.sh`.

Homebrew will download and install the packages necessary for Redis on the Mac. After it's installed, you have a couple of options for how Redis will run on the Mac. You can have Redis start automatically on boot, or you can run it manually.

To start Redis and then have it start automatically on boot, run the following command:

```
brew services start redis
```

To run Redis manually, run the following:

```
redis-server /usr/local/etc/redis.conf
```

The path `/usr/local/etc/redis.conf` is used to specify the configuration file for Redis. If you have a configuration file in a different location, change the path accordingly. The default `redis.conf` file installed by Homebrew includes a set of defaults, such as the IP address and port to bind for the server, among other options. More information on the `redis.conf` configuration file can be found at `https://redis.io/topics/config`.

Taking the First Steps with Redis

The previous section shows four of the ways to install the Redis server, but you had no interaction with the server other than starting it. The Redis CLI is a primary means to ensure that the server is running. This section shows how to install the Redis CLI and how to make your first connection.

Installing the Redis command-line interface

The Redis CLI enables you to interact with the Redis server. Installing the Redis CLI depends on the method that you used to install Redis. If you've installed Redis from source, then the Redis CLI is available within the `src` directory and is started with the following command:

```
./src/redis-cli.
```

Making your first connection

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's installed with the server. The CLI provides a set of commands that enable you to work with the Redis server. Similar to a CLI that you may encounter in Terminal on macOS or Linux or the Command Prompt in Windows, the Redis CLI is the work environment that's 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 database endpoint URL to which you're connecting, and the `<port>` parameter is the port number of the instance.


REMEMBER

For all but development purposes, you'll usually connect to a server running on a different host or IP address. Therefore, the `redis-cli` command will typically be used with the `-h` argument. If the server is listening on a port other than the default 6379, the port option can also be specified.

Credentials can also be specified with the `--user` and `--pass` option or `--askpass` to be prompted for a password. Numerous other options are available that affect the behavior of the `redis-cli` command. You can find these options by using the `--help` option.

After you've connected, the command prompt will change. For example, connecting to the Redis server that is running on the localhost will change the command prompt to:

```
127.0.0.1:6379>
```

If you receive a message such as `Connection refused`, that means the Redis server either isn't running or isn't running on the port or IP address specified.

In addition to connecting with the CLI, the Redis Dashboard for Grafana also enables dashboard creation in Grafana in order to monitor Redis performance and specific data sources.

Working with Redis Clients

This section examines initial programmatic connections for just a few of the languages that are commonly used with Redis. The focus of the section is on making the initial connections with each

of the languages examined. Further details and deeper examples are found in later chapters.

Python

The primary client software for interaction between Python and Redis is the redis-py package, which can be found at `https://redis.io/clients#python`.

Beginning with version 4 of redis-py, Python versions lower than 3 are no longer supported. If you need support for an earlier version of Python, you should use an earlier version of redis-py. Two ways to set up the redis-py package are using setuptools or pip. Using setuptools means downloading the redis-py package from its home page and running the following command:

```
python setup.py install
```

If you prefer, you can use pip to install the redis-py package with the following command:

```
pip install redis
```

Regardless of which method you use, testing the connection using Python can be accomplished with a simple program, such as the following:

```
import redis

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

redis_connection.set('productName','Smart Watch')

print(redis_connection.get('productName'))
```

That simple program connects to a Redis endpoint running on port 6379 of the local computer. With the connection created, a single key-value pair is set, with the key being productName set to the value of Smart Watch. That key is then retrieved and printed.

Java

A number of client connectors are available for using Redis with Java. A connector called Lettuce is popular and can be downloaded and installed through Gradle, Ivy, or Maven, or simply downloaded as a binary. More information about connecting with Java and Lettuce can be found at `https://lettuce.io`, and other Java clients can be found at `https://redis.io/clients`.

Assuming that the dependencies have been resolved, connecting to Redis by using Jedis includes importing the relevant classes, creating a connection pool, and then writing the code for the specific application being created. For example:

```
import redis.clients.jedis.*;

 // Create a Jedis connection pool
 JedisPool jedisPool = new JedisPool(new
  JedisPoolConfig(), "localhost", 6379);

 // Get the pool and use the database
 try (Jedis jedis = jedisPool.getResource()) {

 jedis.set("mykey", "Hello from Jedis");
 String value = jedis.get("mykey");
 System.out.println( value );

 jedis.zadd("vehicles", 0, "car");
 jedis.zadd("vehicles", 0, "bike");
 Set<String> vehicles = jedis.zrange("vehicles",
 0, -1);
 System.out.println( vehicles );

 }

 // close the connection pool
 jedisPool.close();
```

Node.js

Node.js enables rapid development and can be used with Redis. The recommended client for use with Node.js is node-redis, which can be installed with the following command:

```
npm install redis
```

The Redis client software for node-redis will be installed, and a simple program can be created to test the connectivity of the node-redis client. This program is a variation of the example shown within the node-redis documentation found at `https://github.com/NodeRedis/node-redis` and assumes that the Redis server is running on localhost with the default port of 6379. If that isn't the case, the `createClient()` function can be adjusted to connect to a different host or port or both, as follows:

```
const redis = require("redis");
const client = redis.createClient();

client.on("error", function(error) {
  console.error(error);
});

client.set("productName","Watchy Watch",redis.
   print);
client.get("productName",redis.print);
```

Other languages

Redis includes clients for numerous development languages. Many of these languages include their own package management interfaces as well. For example, the Predis implementation for PHP is available through Packagist. Where prepackaged clients are not available, compiling from source is always an option. The end result is that Redis is widely supported with mature and stable client software. More information can be found at `https://redis.io/clients`.

IN THIS CHAPTER

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

Chapter 4
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. Further, most existing database management systems are organized around a single data model that determines how data can be organized, stored, and manipulated.

Relational models frequently fail to 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. Using Redis Modules, data models such as graph and full-text search can be added, and new models can be added through the modularity. This means that the data can be used in the manner most appropriate for the application both today and in the future.

In this chapter, I dive into the data models and types, showing examples of their use.

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 of 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.

TIP

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's 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 command-line interface (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 command, a simple string value of `steve` has been stored in the database and can be retrieved with the `GET` command:

```
> GET user
```

This command retrieves the following value:

```
"steve"
```

Numerous other commands 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 this 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

You can manipulate numerous other commands and work with string and string-like 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.

The largest size for a single string value is 512MB. This means that you can store 2^{32} possible values inside 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.

You can push a value onto a list with the `LPUSH` and `RPUSH` commands, which place values onto a list either on the left (or head) or on 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 the 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 can't 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 the way you can with a list.

You can add a value to a set with the `SADD` command:

```
SADD fruit apple
```

You can list all members of a set 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 whether 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, you would use the following command:

```
SISMEMBER fruit apple
```

If 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 structure 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, use this command:

```
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, you can retrieve all members of a sorted set 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, like this:

```
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 would be 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

Earlier, I introduced the basic data types in Redis. But there are also common ways to use Redis, incorporating these data types. 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 any clients are subscribed. If a client is 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 knows the format of messages and is 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 subkeys, with the PSUBSCRIBE command:

```
PSUBSCRIBE weather:<sup>*</sup>
```

Geospatial indexes

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 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 it can be changed to other measures, such as miles, like this:

```
GEODIST towers tower1 tower2 mi
```

Redis Streams

Streams are modeled after a log data structure, where data is appended like a logfile. This distinction is important for Redis Streams because data is append-only; therefore, data can only be added to or read from a stream.

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

Redis Modules

Redis has several modules that further enhance the capability of Redis. This section examines seven such modules: RediSearch, RedisJSON, RedisTimeSeries, RedisGraph, RedisBloom, RedisAI, and RedisGears.

RediSearch

RediSearch is a full-text search engine that features document storage within Redis while enabling high-performance search capabilities. RediSearch 2.0 enables real-time secondary indexing and powerful querying to create a full-text search engine.

The RediSearch module enables weighted search results, the use of Boolean logic, autocomplete functionality, and several other common features. The use of fuzzy-logic search technology helps to enhance search results for users, especially when combined with autocomplete suggestions.

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

RedisJSON

RedisJSON stores JSON documents in their native format, enabling in-memory manipulation of the corresponding data and data structures. This storage scheme promotes high-velocity use cases without sacrificing performance. For example, user personalization (much of which can be consumed natively in JSON format) is one such use case. Entire documents, such as data like product

catalogs and third-party feeds, can also be stored in order to facilitate a content management scenario. Hierarchical data can be stored as a single compound object, eliminating the need for multiple requests. RedisJSON is unique and distinct from document and JSON data manipulation through Lua, offering significant improvement over other types of storage.

Behind the scenes, the ECMA-404 standard is used as the native format for RedisJSON.

RedisTimeSeries

Storing time-series data is another common task for a database and is also common for NoSQL databases, notably for use cases such as IoT, stock prices, and telemetry. The RedisTimeSeries module is a high-performance way to store and work with data that is ordered by time.

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

With RedisTimeSeries, you can ingest and query millions of samples and events at the speed of Redis. Use a variety of queries for visualization and monitoring with built-in connectors to popular tools like Grafana, Prometheus, and Telegraf.

RedisGraph

RedisGraph 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.

RedisBloom

The RedisBloom module extends Redis core to support additional probabilistic data structures. Specifically, RedisBloom facilitates

the use of four probabilistic data structures in Redis, including a Bloom filter, a cuckoo filter, a count-min sketch, and top-k. Bloom and cuckoo filters provide information on whether an item exists within a set. The count-min sketch and top-k data structure are used to count frequent items, with count-min sketch determining the frequency of items in a stream and top-k providing a list of items that appear most frequently.

RedisAI

RedisAI is a Redis module for executing Deep Learning/Machine Learning models and managing their data. RedisAI uses tensors as a means for modeling machines and deep-learning problems. Some common use cases for the RedisAI module include speech recognition and natural language processing, object detection and visual inspection, and filtering of extremely scaled data, such as social network data.

RedisGears

RedisGears is an engine for data processing in Redis. RedisGears supports batch and event-driven processing for Redis data. To use RedisGears, you write functions that describe how your data should be processed. You then submit this code to your Redis deployment for remote execution.

RedisGears provides a means to execute Python functions and entire Python scripts inside of Redis. Executing functions and scripts not only speeds up processing but also simplifies architecture, enabling serverless architectures.

IN THIS CHAPTER

- Making sense of clustering and high availability
- Looking at transactions and durability

Chapter 5
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 and then turns to 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 data set 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. Redis Enterprise cluster uses a shared-nothing approach — 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 weren't running in a cluster.

Redis Enterprise cluster is linearly scalable with a multi-tenant and symmetric architecture.

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

The Redis Enterprise cluster consists of several components:

- **Redis Shard:** Data is stored and managed at this layer and is the same core as a single instance of open-source Redis.
- **Zero-Latency Proxy:** Each node of the cluster uses a proxy to provide stateless and multi-threaded communication between client and node.
- **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 representational state transfer (REST) application programming interface (API) is used for management of the cluster.

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.

REMEMBER

High availability enables Redis to achieve 99.999 percent uptime.

Organizations using open-source Redis to achieve high availability find the expense of random access memory (RAM) makes doing so costlier and more complex overall. 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 replica. 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 at both the node level and 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-availability-zone (AZ) deployments.

Running Redis at scale

From an architectural perspective, a fully scaled and production-level Redis deployment has several characteristics. The key to running Redis at scale is using Redis Enterprise, which 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. Linear scaling is achieved without the need for nonlinear overhead. Redis Enterprise splits workload across processing cores and nodes and optimizes performance across multiple levels, including connection management,

request scheduling, and execution. This performance optimization is applied regardless of data type or model.

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 it.

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 from the original. Redis Enterprise further enhances performance by automatically resharding and rebalancing the workload.

Redis on Flash

Redis on Flash (RoF), available with Redis Enterprise Software and Redis Enterprise Cloud, enables the database to be stored not only in RAM but also on dedicated flash memory such as a solid-state drive (SSD). With RoF, keys and data are maintained in RAM, while less frequently used 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.

A CLOSER 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 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:** Isolation helps to ensure consistency by requiring concurrent transactions to be separate from each other.
- **Durability:** Durability is data persistence that ensures that when a transaction is complete, it can be retrieved in the event of a system failure.

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.

TIP

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

Durability

Redis Enterprise is a fully durable database. There are two methods for providing data persistence in Redis:

- **Append-Only File (AOF):** 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. Writing every second is fast but not as safe, while writing to the disk on every database write operation means slower performance. The benefit is ensuring durability by writing to the disk for every database write operation.

  Redis Enterprise handles AOF differently from 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 data persistence at the replica of each shard of the database. Performance isn't impacted because the master shard isn't affected. However, replication latency can adversely affect the performance requirements for data persistence. Redis Enterprise provides the ability to enable data persistence on both the master and the replica shards.
- **Snapshot:** Snapshots are point-in-time copies of the database. Snapshots apply to all shards within a database and are used primarily for 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 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

- Getting acquainted with Redis Enterprise Software and Redis Enterprise Cloud
- Using Redis Enterprise Software and RedisInsight

Chapter 6
Using Redis Enterprise Software and Redis Enterprise Cloud

Redis Enterprise Software and Redis Enterprise Cloud provide enhanced, enterprise-ready implementations of Redis. This chapter explains Redis Enterprise Software and Redis Enterprise Cloud and the enhancements that make them appealing for so many production workloads today.

Understanding Redis Enterprise Software and Redis Enterprise Cloud

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 three primary means to deploy Redis Enterprise:

- Redis Enterprise software can be deployed locally within your data center or cloud provider and can be deployed in a multi-cloud or hybrid on-premises/cloud architecture.
- Redis Enterprise VPC and Redis Enterprise Cloud are fully managed services operated by Redis Labs. Redis Enterprise VPC enables you to run Redis Enterprise within your own virtual private cloud (VPC) on Amazon Web Services (AWS), but Redis Enterprise Cloud runs on instances that are owned by Redis Labs. Redis Enterprise Cloud is available on all major cloud providers.

Both methods for deployment result in a high-performance implementation of Redis. The difference between the two is with the management of the underlying platform. With Redis Enterprise software, you manage the infrastructure; with Redis Enterprise Cloud, the platform is managed by Redis Labs.

TIP

You can find more information about Redis Enterprise Cloud, including deploying an instance, in Chapter 3.

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, RedisJSON, RedisBloom, and others
- Full durability and snapshots
- Stellar performance

CAP THEOREM AND CRDT

CAP theorem states that it's 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
- 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.

Getting Started with Redis Enterprise Software and RedisInsight

Getting started with Redis Enterprise means selecting a platform, either hosted through Redis Enterprise Cloud or as downloadable software. This section looks at those first steps to begin using Redis Enterprise software. Getting started with Redis Enterprise Cloud is covered in Chapter 3.

REMEMBER

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

Meeting the prerequisites

After you have an account at Redis Labs, you can choose which method you'll use for installing Redis Enterprise: 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 following supported platforms for Redis Enterprise:

- Docker on macOS or Windows
- Kubernetes
- Oracle Linux
- Red Hat
- Ubuntu

Redis Enterprise software can be downloaded from `https://redislabs.com/redis-enterprise-software/download-center/software`. After you've downloaded Redis Enterprise, you'll be able to install it on your chosen platform.

Installing Redis Enterprise in a Docker container

Redis Enterprise can be installed on a number of platforms, some of which are discussed in Chapter 3. This section shows deployment using 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, first you have to 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 9443:9443 -p 12000:12000
  redislabs/redis
```

Docker will then download the necessary components and pull the image from DockerHub and begin running Redis inside the container.

TECHNICAL STUFF

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 three ports: 8443, 9443, and 12000 (`-p`).

After installation, Redis Enterprise requires some setup. The setup process is accomplished through a web browser by connecting to `https://localhost:8443`.

TIP

Because there is almost certainly no Secure Sockets Layer (SSL) certificate installed for localhost, you may receive a warning from your web browser because it can't verify the validity of the SSL connection.

If you continue through the warning, you'll get to a setup screen like the one shown in Figure 6-1. When you reach this screen, follow these steps:

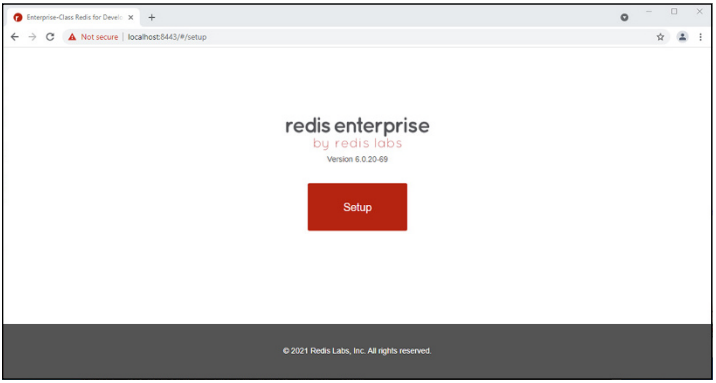

FIGURE 6-1: Beginning the setup process for Redis Enterprise software.

1. **Click Setup.**

   The process of node configuration, shown in Figure 6-2, begins.

   In Figure 6-2, the values are left at their defaults with the exception of the cluster name, which is set to producttest.example.com.

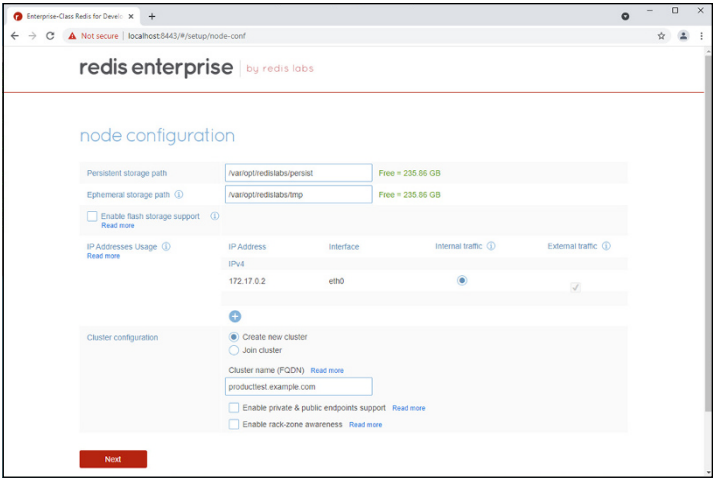

FIGURE 6-2: Setting properties for the cluster.

2. Click Next.

The Cluster Authentication page appears (see Figure 6-3).

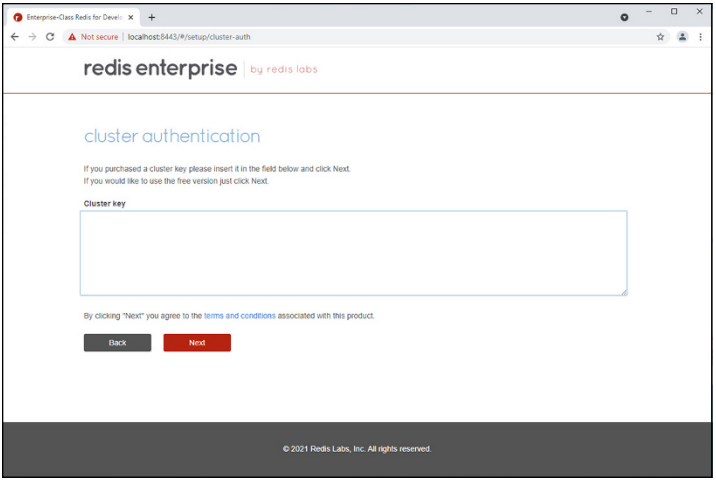

FIGURE 6-3: Optionally including a cluster key.

3. **Enter a cluster key if you want. Either way, click Next.**

   The Set Admin Credentials screen appears. Figure 6-4 shows the email address being set to `redisadmin@example.com` and a password being set as well.

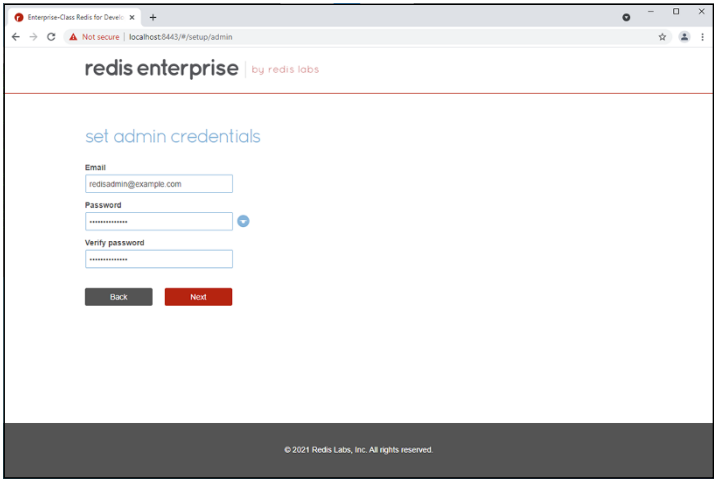

FIGURE 6-4: Configuring credentials for cluster administration.

4. **Set the email and password, and click Next.**

   The Create New Database screen (see Figure 6-5) appears.

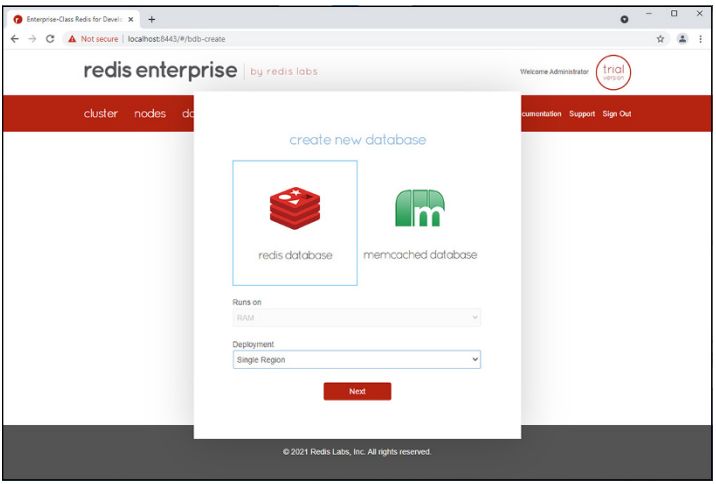

FIGURE 6-5: Creating a database, either Redis or memcached.

5. **Select Redis Database and click Next.**

   The Create Database screen (see Figure 6-6) appears.

   In Figure 6-6, a database name of productdb is created with a 1GB memory limit. The Default Database Access field also has a password filled in. Other values are left at their default but can be customized based on your needs.

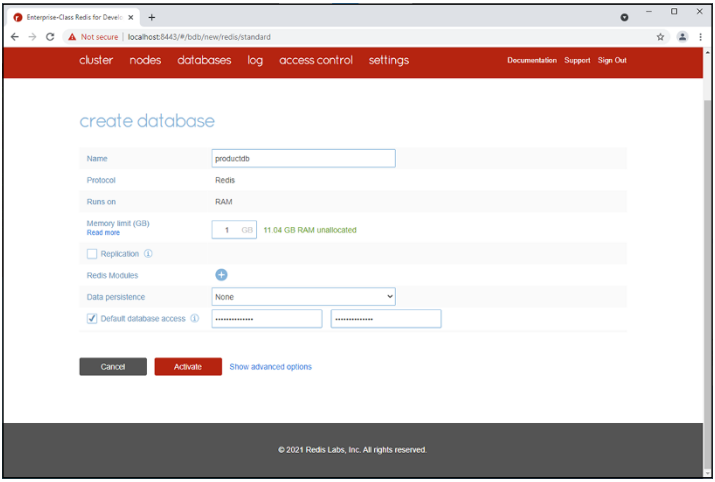

FIGURE 6-6: Setting properties for a database as part of the cluster.

6. **Fill out the fields as needed, and click Activate.**

   After clicking Activate, the database is created.

Understanding concepts and architecture

Redis Enterprise is set up in a clustered environment, with multiple identical nodes ready to serve requests. The nodes themselves include two layers:

- **Management layer:** The management layer enables administration of the cluster with things like the Cluster Manager to manage placement of shards and failure detection and mitigation.

- **Data access layer:** The data access layer manages connections to the data itself, with clients connecting to the primary shard and then secondary shards being maintained by the primary shard.

In-memory replication is used for synchronization, with wide-area network (WAN) replication also available for maintaining synchronization across data centers. Shard management is invisible to the clients and is managed completely by the master shard.

Connecting with RedisInsight

RedisInsight provides an intuitive and efficient GUI for Redis, allowing you to interact with your databases and manage your data — with built-in support for most popular Redis modules. It provides tools to analyze the memory, profile the performance of your database usage, and guide you toward better Redis usage. RedisInsight is available on multiple platforms and can be installed in a variety of ways, including through its own installer. Additional information, including a link to the RedisInsight installer, is available at `https://docs.redislabs.com/latest/ri`.

When RedisInsight is installed, it will listen for connections on port 8001 by default. Follow these steps:

1. **Navigate to** `https://localhost:8001`**.**

   A welcome page appears where you can select a path depending on whether you already have a database or you need to create one (see Figure 6-7). This section assumes that you already have a Redis database.

2. **Select I Already Have a Database.**

   You're presented with several additional options depending on whether you'll be working with an individual database, a cloud database, a cluster, or an Amazon ElastiCache database (see Figure 6-8).


TIP

   RedisInsight can auto-discover databases. If you click Connect to Your Redis Enterprise Cluster, you get the screen shown in Figure 6-9.

   As shown in Figure 6-9, the cluster port for the recently deployed Redis Enterprise cluster is used, along with the credentials created for that deployment. RedisInsight is able to discover the producttest database that was added earlier in this chapter, and that database then becomes available for addition to RedisInsight (see Figure 6-10).

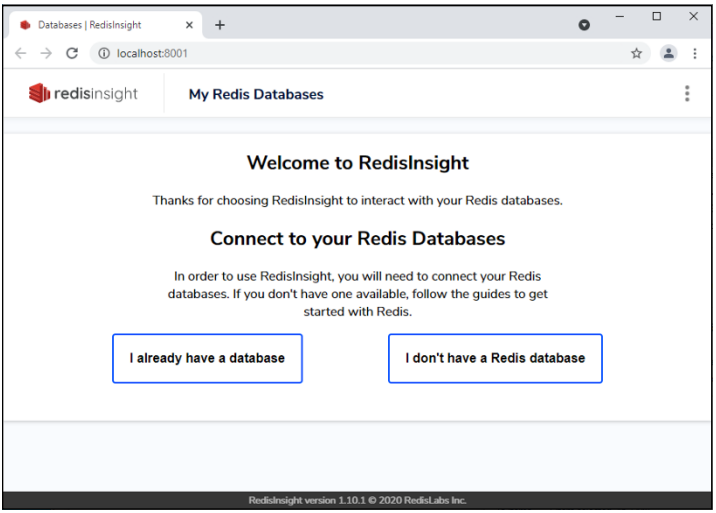

FIGURE 6-7: Choosing whether there is an existing database or a database needs to be created.

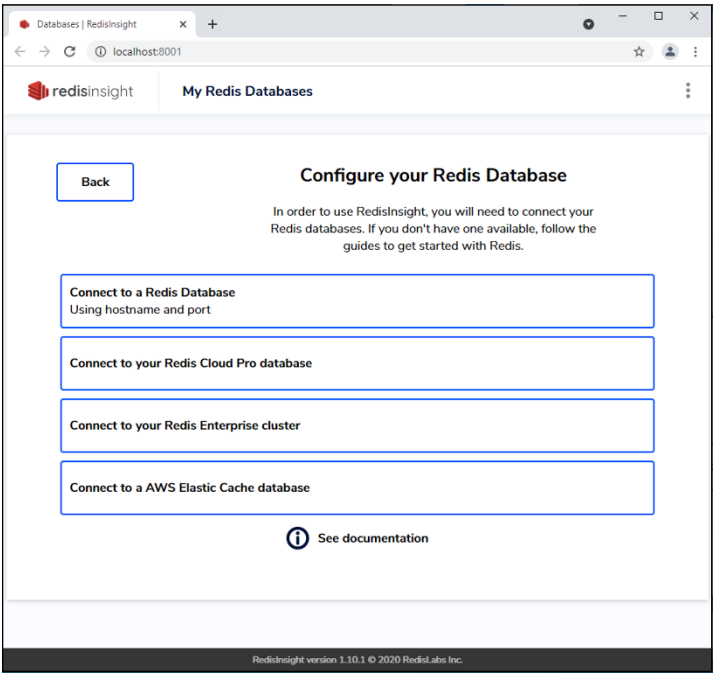

FIGURE 6-8: Various options for working with a database in RedisInsight.

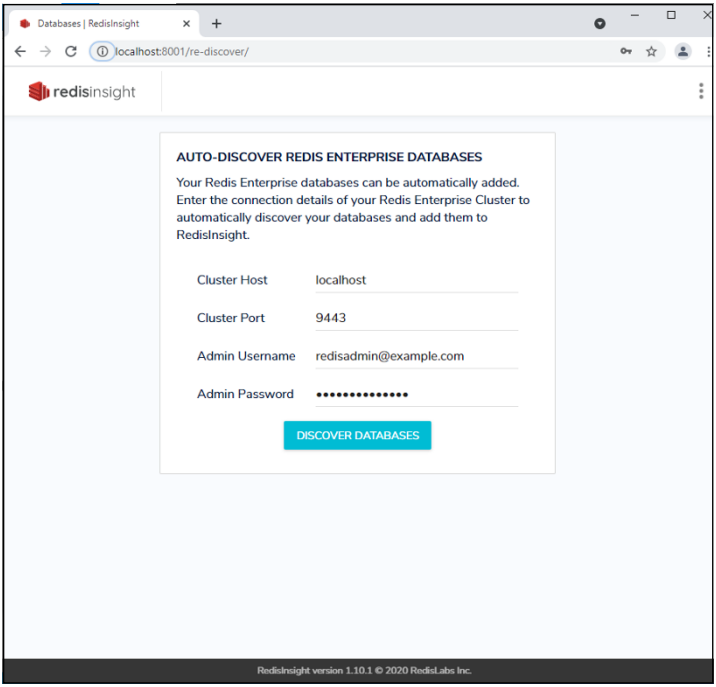

FIGURE 6-9: Configuring auto-discovery settings.

With a database discovered and added to RedisInsight, you can then manage and interact with the data, better understand resource usage, develop queries, and perform other tasks. Figure 6-11 shows the Overview page within RedisInsight. Notice the summary and graphs to give a quick, dashboard-style overview of the database. The various actions that are possible are also shown in Figure 6-11, including accessing a data browser, the CLI, several modules, and much more.

RedisInsight provides a rich, task-centric experience tailored to modern application development and administrative needs. For example, memory analysis can be performed to improve the performance of the application. Slowlog analysis, or troubleshooting slow performance of queries, can also be done through RedisInsight. See `https://developer.redislabs.com/explore/redisinsight` for more details on the tasks that can be accomplished with RedisInsight.

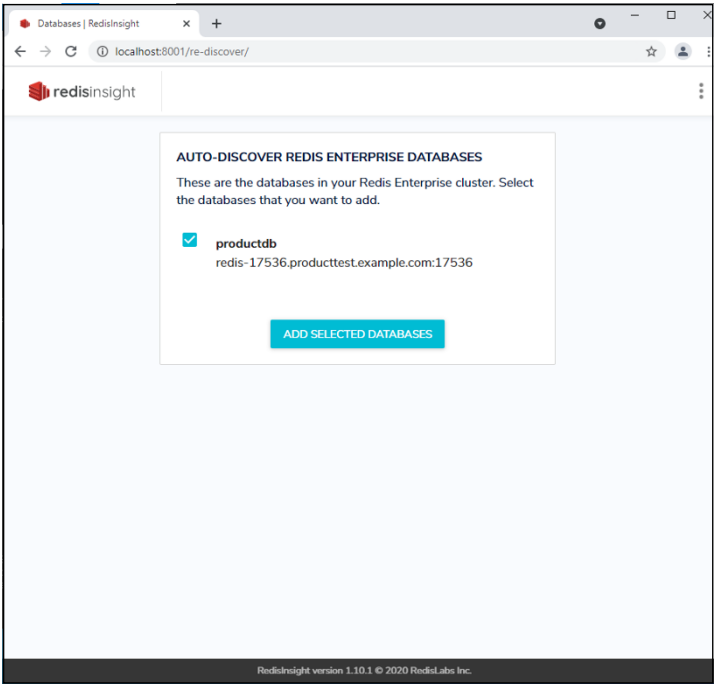

FIGURE 6-10: An auto-discovered database in RedisInsight.

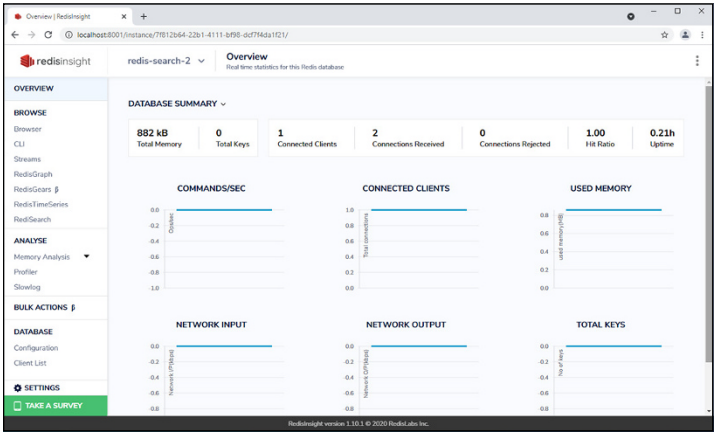

FIGURE 6-11: The Overview screen in RedisInsight.

IN THIS CHAPTER

- Beginning to build a Redis application
- Using three common data types

Chapter 7
A Simple Redis Application

In this chapter, you take a look at a basic application created to run on Node.js demonstrating basic create-read-update-delete (CRUD) operations.

REMEMBER

The application isn't meant to show everything that's possible with Redis. Instead, it demonstrates the foundations to help kick-start 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 Enterprise Cloud.

Prerequisites

Redis has created a simple Redis application that can be found 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 may want to 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. With the Redis server and Node.js running, execute the sample.sh script:

```
./sample.sh
```

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 this chapter. In general terms, the application maps verbs from representational state transfer (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 car's 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 may 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 the following:

```
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 as follows:

```
LRANGE features 2 2
```

The LRANGE command reads data but other CRUD operations are supported for lists in this application. To fulfill the create portion of the CRUS acronym, 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 adds data to the hash data structure in the Redis instance. When the data is 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 cjhvatfuc00005
  mfj2zycewid
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 as follows:

```
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

- Exploring RediSearch with a sample application
- Querying and indexing data

Chapter 8
Building an Application with RediSearch

RediSearch is a powerful indexing, querying, and full-text search engine for Redis, available on-premises and as a managed service in the cloud. In this chapter, I walk you through an application demonstrating RediSearch. The application uses sample data related to movies. The chapter features RedisInsight, a free web-based management interface for Redis. RedisInsight is used to explore and interact with data through an intuitive and efficient GUI, with built-in support for most popular Redis modules.

Using RediSearch for Movie Data

Before you can use RediSearch, you have to install it. After installation, some sample data will be inserted into the database. This section covers both of these tasks.

Installing RediSearch

RediSearch can be installed into an existing Redis instance by compiling from source. The RediSearch module is also available

with Redis Enterprise Cloud or Redis Enterprise Software. Finally, RediSearch can also be installed using Docker. This section focuses on a Windows-based Docker installation and assumes that Docker has already been installed.

The Docker command to install RediSearch is as follows:

```
docker run -it --name redis-search-2 -p 6379:6379
  redislabs/redisearch:2.0.2
```

When you run this command, the container is downloaded and a single Redis instance running RediSearch is installed. Data can now be visualized using RedisInsight.

You can download RedisInsight from `https://docs.redislabs.com/latest/ri`. Chapter 6 includes additional information about RedisInsight, including how to make the initial connection to the database. For this example, RedisInsight will be connected to a single Redis instance through the Connect to a Redis Database option. When you select that option, the Add Redis Database dialog (see Figure 8-1) appears. Using the sample added earlier in this section, the Host should be set to localhost, the Port should be set to 6379, and the Name should be set to redis-search-2. (You can leave the Username and Password fields blank for this example, but for an instance that could be accessed from outside this single machine, you'll need to set them.)

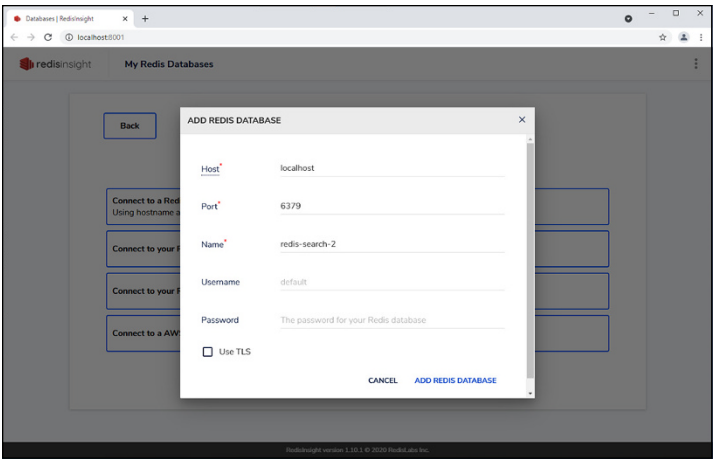

FIGURE 8-1: Adding a Redis database to RedisInsight.

Inserting data

You can insert data for the sample application through the command-line interface (CLI) accessed from within RedisInsight. From the My Redis Databases page of RedisInsight (shown in Figure 8-2), click the redis-search-2 database.

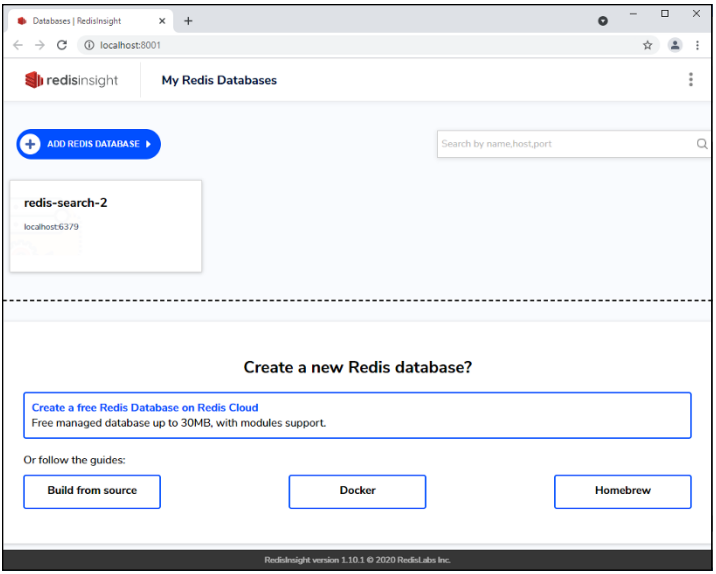

FIGURE 8-2: Viewing databases available in RedisInsight.

When you access the CLI through RedisInsight, you can enter a command within the text box at the bottom of the page. Whatever you enter in the text box will be executed when you press Enter. For the movies example, add the following three movies through the CLI; press Enter after each `HSET` command:

```
HSET movie:11003 title "The Godfather" plot "The
  aging patriarch of an organized crime dynasty
  transfers control of his clandestine empire to
  his reluctant son." release_year 1972 genre
  "Drama" rating 9.2 votes 1563839 imdb_id
  tt0068646
HSET movie:11004 title "Heat" plot "A group of
  professional bank robbers start to feel the heat
  from police when they unknowingly leave a clue
```

```
   at their latest heist." release_year 1995 genre
   "Thriller" rating 8.2 votes 559490 imdb_id
   tt0113277
HSET "movie:11005" title "Star Wars: Episode VI -
   Return of the Jedi" genre "Action" votes 906260
   rating 8.3 release_year 1983  plot "The Rebels
   dispatch to Endor to destroy the second Empire's
   Death Star." ibmdb_id "tt0086190"
```

The end result of adding those three movies to the database through the RedisInsight CLI is shown in Figure 8-3.

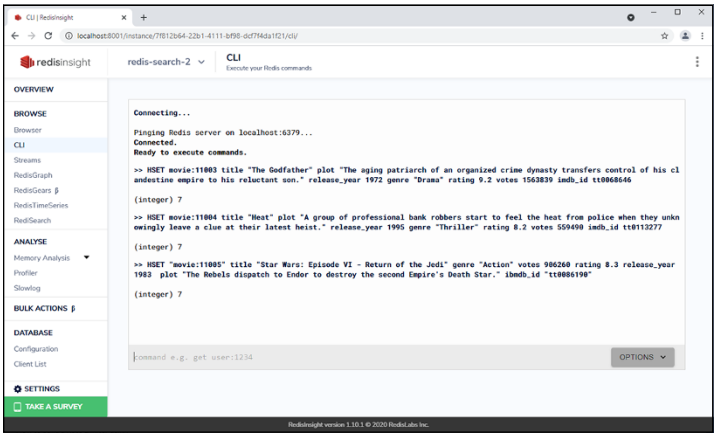

FIGURE 8-3: Inserting data using the CLI in RedisInsight.

Working with Data and Indexes

This section continues the movies database example being developed to demonstrate RediSearch. Within the section, the data you just inserted will be queried, an index will be added, and the data will be searched.

Querying data

You can query data through RedisInsight through the CLI or by using the Browser tool in RedisInsight. For example, to retrieve the title of the movie with key 11003, use the `HMGET` command, like this:

```
HMGET movie:11003 title
```

The result is:

```
1) "The Godfather"
```

Querying through the Browser, which you can navigate to by clicking the Browser tool in RedisInsight, provides a richer experience. You can simply click a given key, such as `movie:11004`, to retrieve all the values for that entry (see Figure 8-4).

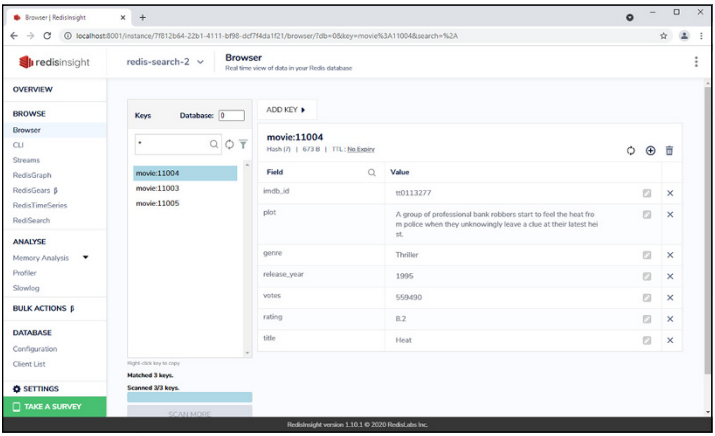

FIGURE 8-4: Querying data through the Browser in RedisInsight.

The Browser enables you to change and delete individual fields within a key or the entire entry itself.

Adding and viewing indexes

With core Redis data structures, you have to manage the index manually, adding significant amounts of code to the application. However, with RediSearch, an index can simply be defined and associated with data. Redis will then manage the index and the query engine can be used to search or query the data using secondary indexes.

Moving forward with the example, indexes are necessary for each field that will be searchable and are created through the CLI with the `FT.CREATE` command. For example, to create an index on

movie title, genre, rating, and year of release, the following command is used:

```
FT.CREATE idx:movie ON hash PREFIX 1 "movie:"
   SCHEMA title TEXT SORTABLE release_year NUMERIC
   SORTABLE rating NUMERIC SORTABLE genre TAG
   SORTABLE
```

When executed through the CLI, the result of this command is a simple:

```
"OK"
```

The command `FT.INFO` is used to display detailed information about the index, including its definition and various statistics about the number of documents indexed, the resources needed to create the index, and more. For example, the following command will display the index information for the newly created index:

```
FT.INFO idx:movie
```


WARNING

Because indexes consume memory and resources, you should carefully consider which fields are necessary to be searched within the application.

Figure 8-5 shows RedisInsight in use for the movie index.

Searching data

With the previous steps complete, it's time to search the sample data. With RedisInsight, clicking RediSearch displays a search text box where you can conduct a search of the index created in the previous step. For example, Figure 8-6 shows a search that has been executed for the movie *Heat.*

You can also search using the `FT.SEARCH` command through the CLI. You can find more information on querying with `FT.SEARCH` at `https://oss.redislabs.com/redisearch/master/Commands/#ftsearch`.

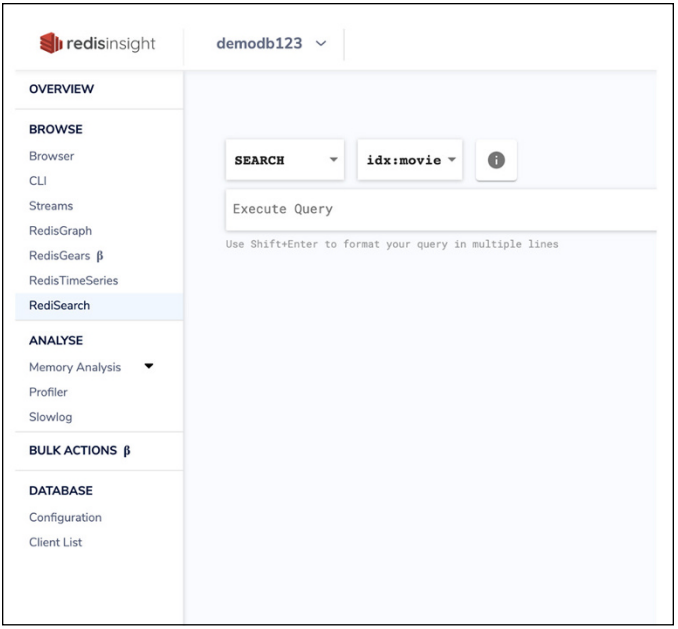

FIGURE 8-5: Using RediSearch for the movie index in RedisInsight.

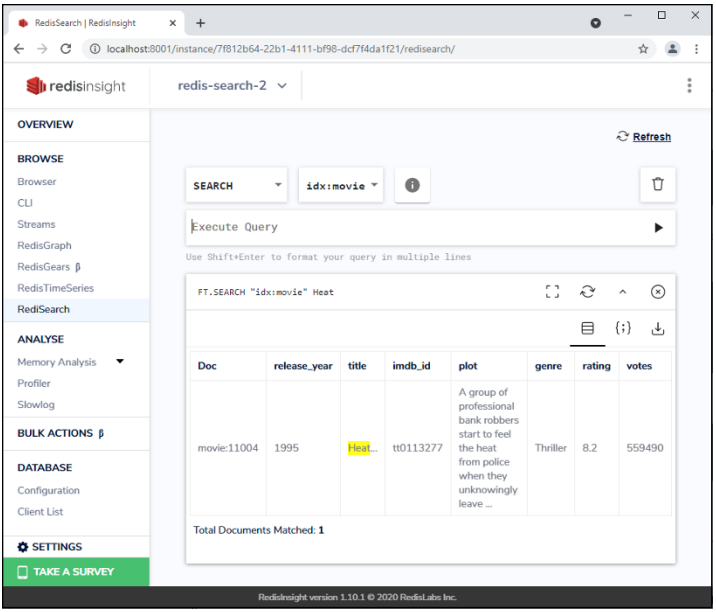

FIGURE 8-6: Using RediSearch within RedisInsight to search for a movie.

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 9
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 (CRDTs). I start by defining CRDTs and explain 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 (also called geolocal) servers in order to reduce latency. With Active-Active databases, applications can read and write to the same data set from different geographical locations seamlessly and with latency less than 1 ms, without changing the way the application connects to the database.

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

Active–Active databases support Lua scripts. However, only the resulting write commands, known as the *effects*, of the scripts are replicated rather than the entire Lua script. This is known as *effects replication mode.*

Several data types can be used as CRDTs in Redis, including:

- Bitfield
- Float counters
- Geospatial
- Hashes (hash fields are treated as strings or counters)
- HyperLogLog
- Integer counters
- Lists
- Sets
- Sorted Sets
- Strings
- Streams

Working with Conflict-Free Replicated Data Types

In this section, you begin to build the application. You install prerequisites and get 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 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 Chapter 8. For example, the application in this chapter requires 8GB of random access memory (RAM) for each instance of Redis Enterprise.

The example application also uses Node.js. You may have already installed Node.js, but if not, you can get more information on installing it 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, follow these steps:

1. **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 networks (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 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 in the next step.

2. **Run** `setup_redis_enterprise_clusters.sh`**.**

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

WARNING

Don't 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.

3. **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 (CRDB) spanning both clusters.

Testing the conflict-free replicated data type

The first step in testing the CRDB is to connect to each cluster. Follow these steps:

1. **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
```

2. **Execute the following command to connect to the second cluster:**

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

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


TIP

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.

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

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

4. **Connect back to the first cluster:**

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

5. **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.


TECHNICAL STUFF

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 example 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 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 commands:

```
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're 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
}
```

REMEMBER

The example 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 on their own or with a single cluster.

Chapter 10
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 the 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. For example, Redis is frequently used for shopping carts.

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 (pub/sub) pattern enable service decoupling. Services can write to and read from Redis Streams or can publish and subscribe to messages using Redis as the facilitator of the pub/sub pattern.
- **Provide rate limiting.** Redis can be used to rate-limit users and end points to protect and improve the availability of API-based services. The high-performance, real-time nature of Redis means that tracking can be done in real time along with the users and end points.
- **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 ingestion. For example, financial applications that work with stock prices to store, aggregate, and query stock prices and financial information at high speeds are a great fit for Redis.
- **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. This use is frequently found in Internet of Things (IoT) sensor applications.
- **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, time-series processing, real-time analytics, fraud detection, gaming, and leaderboards.

Inside...

- See real-world examples of Redis-powered applications
- Learn about coding with Redis clients like Python, Java, and NodeJS
- Explore Redis clustering and high availability
- Get exposure to Redis modules like RediSearch, RedisJSON, RedisGraph, and RedisTimeSeries

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: 978-1-119-82427-5
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.