Redis Essentials

Redis is the most popular in-memory key-value data store. It’s very lightweight and its data types give it an edge over other competitors.

Redis Essentials is a fast-paced guide that teaches the fundamentals of data types, explains how to manage data through commands, and shares experiences from big players in the industry. We start off by explaining the basics of Redis followed by the various data types such as Strings, Hashes, Lists, and more. Next, the common pitfalls for various scenarios are described, followed by solutions to ensure you do not fall into them.

Next, you will learn how to extend Redis with Lua, get to know security techniques such as basic authorization, firewall rules, and SSL encryption, and discover how to use twemproxy, Redis Sentinel, and Redis Cluster to scale infrastructures horizontally. At the end of this book, you will be able to utilize all the essential features of Redis to optimize your project’s performance.

Who this book is written for
This book is intended for those who want to learn about Redis with or without previous experience. By providing a foundation to understand the capabilities of Redis, this book will teach you how to extend and scale Redis in real-life situations.

What you will learn from this book
- Build analytics applications using Bitmaps and HyperLogLogs
- Enhance scalability with Twemproxy, Redis Sentinel, and Redis Cluster
- Build a time series implementation in Node.js and Redis
- Create your own Redis commands by extending Redis with Lua
- Get to know security techniques to protect your data (SSL encryption, firewall rules, basic authorization)
- Persist data to disk and learn the trade-offs of AOF and RDB
- Understand how to use Node.js, PHP, Python, and Ruby clients for Redis
- Avoid common pitfalls when designing your next solution


Maxwell Dayvson Da Silva       Hugo Lopes Tavares

Redis Essentials
Harness the power of Redis to integrate and manage your projects efficiently

$ 39.99 US
£ 25.99 UK

Prices do not include local sales tax or VAT
In this package, you will find:

- The author's biography
- A preview chapter from the book, Chapter 1 'Getting Started (The Baby Steps)'
- A synopsis of the book’s content
- More information on Redis Essentials
Maxwell Dayvson Da Silva, a self-taught programmer, is the director of technology at The New York Times.

Born in Recife, Brazil, he is a video specialist and is most interested in bringing technology to a global audience. His work has ranged from developing and delivering highly scalable products to innovating and implementing large-scale video solutions. Prior to joining the Times, he worked for Globo, Brazil's leading media network, and Terra, a global digital media company.

Additionally, he has spoken at conferences such as Campus Party, FISL, SET Broadcast and Cable, Streaming Media East, and Streaming Media West. Maxwell has also devoted time to speaking at several Brazilian universities, including UFGRS, IFRS, UDESC, and FEVEALE-RS.

He is a contributor to and creator of some open source projects. You can find them at https://github.com/dayvson. Outside of his professional work, Maxwell regularly combines his passion for art and science to create games and interactive art installations. His son, Arthur, inspires him to seek opportunities to bring science into the lives of young people, both in New York and abroad.

Although Redis Essentials is Maxwell's first book, he has done technical reviewing for two others, Extending Bootstrap and Learning JavaScript Data Structures and Algorithms. You can contact him on LinkedIn at http://www.linkedin.com/in/dayvson.
Hugo Lopes Tavares is a software developer from Brazil who currently works as a platform engineer at Yipit, a technology company focused on data aggregation and analysis. Prior to his work in the United States, Hugo worked on live streaming video development for Globo.com, the Internet branch of Grupo Globo, which is the largest media conglomerate in Latin America.

Having been involved in open source software, he has made a significant impact in this field. He was a main contributor to pip (the Python package installer), wrote improvements to CPython and the Python standard library, coauthored Splinter (a web-testing tool), and contributed to many well-known projects. Some of his contributions can be found at https://github.com/hltbra.

Additionally, Hugo worked at NSI (Information Systems Research Group), carrying out research and development on agile methods and software quality for the Brazilian government. Within his research, he created some testing tools, the most famous of which are Should-DSL and PyCukes, which are mentioned in Python Testing Cookbook, Packt Publishing (Should-DSL has its own section in it).

When Hugo is not doing anything related to technology, he is involved in strength training as an amateur powerlifter.

You can contact him on LinkedIn at https://www.linkedin.com/in/hltbra.
Redis is the most popular in-memory key-value data store. It is very lightweight and its data types give it an edge over other competitors. If you need an in-memory database or a high-performance cache system that is simple to use and highly scalable, Redis is what you should use.

This book is a fast-paced guide that teaches you the fundamentals of data types, explains how to manage data through commands, and shares experiences from big players in the industry.

**What this book covers**

*Chapter 1, Getting Started (The Baby Steps)*, shows you how to install Redis and how to use `redis-cli`, the default Redis command-line interface. It also shows you how to install Node.js and goes through a quick JavaScript syntax reference. The String, List, and Hash data types are covered in detail, along with examples of `redis-cli` and Node.js.

*Chapter 2, Advanced Data Types (Earning a Black Belt)*, is a continuation of the previous chapter. It presents the Set, Sorted Set, Bitmap, and HyperLogLog data types. All the examples here are implemented with `redis-cli` and Node.js.

*Chapter 3, Time Series (A Collection of Observations)*, uses all of the knowledge of data types from the previous chapters to build a time series library in Node.js. The examples are incremental; the library is initially implemented using the String data type, and then the solution is improved and optimized by using the Hash data type. Uniqueness support is added to the String and Hash implementations by using the Sorted Set and HyperLogLog data types, respectively.
Chapter 4, Commands (Where the Wild Things Are), introduces Pub/Sub, transactions, and pipelines. It also introduces the scripting mechanism, which uses the Lua programming language to extend Redis. A quick Lua syntax reference is also presented. A great variety of Redis commands are presented in this chapter, including the administration commands and data type commands that were not covered in the previous chapters. This chapter also shows you how to change Redis’s configuration to optimize different data types for memory or performance.

Chapter 5, Clients for Your Favorite Language (Become a Redis Polyglot), shows how to use Redis with PHP, Python, and Ruby. This chapter highlights the features that vary more frequently with clients in different languages: blocking commands, transactions, pipelines, and scripting.

Chapter 6, Common Pitfalls (Avoiding Traps), illustrates some common mistakes when using Redis in a production environment and related stories from real-world companies. The pitfalls in this chapter include using the wrong data type for a given problem, using too much swap space, and using inefficient backup strategies.

Chapter 7, Security Techniques (Guard Your Data), shows how to set up basic security with Redis, disable and obfuscate commands, protect Redis with firewall rules, and use client-to-server SSL encryption with stunnel.

Chapter 8, Scaling Redis (Beyond a Single Instance), introduces RDB and AOF persistence, replication via Redis slaves, and different methods of partitioning data across different hosts. This chapter also shows how to use twemproxy to distribute Redis data across different instances transparently.

Chapter 9, Redis Cluster and Redis Sentinel (Collective Intelligence), demonstrates the differences between Redis Cluster and Redis Sentinel, their goals, and how they fit into the CAP theorem. It also shows how to set up both Sentinel and Cluster, their configurations, and what happens in different failure scenarios. Redis Cluster is covered in more detail, since it is more complex and has different tools for managing a cluster of instances. Cluster administration is explained via native Redis commands and the redis-trib tool.
Redis is a NoSQL (Not only SQL) advanced key-value data store. It is also referred to as a data structure server because of its powerful data types, such as Strings, Hashes, Lists, Sets, Sorted Sets, Bitmaps, and HyperLogLogs. By default, Redis saves all data in the memory, therefore read and write operations are very fast. It can also cause data to persist in the disk. Data persistence in Redis can be achieved by creating a binary snapshot of the stored data or a human-readable file with a sequence of all executed commands over time. These are respectively known as snapshotting and journaling.

Additionally, Redis includes configurable key expiration, transactions, and publish/subscribe features. It also provides Lua scripting to extend Redis to create new commands. Combined, these features transform Redis into the Swiss Army knife of data type storage.

Redis stands for **REmote DictionarY Server**. It was written in C by *Salvatore Sanfilippo* in 2006 and currently has many contributors. There are Redis clients available for over 30 programming languages. The open source project can be found at [https://github.com/antirez/redis](https://github.com/antirez/redis). The official Redis documentation is also a really good resource of knowledge and can be found at [http://redis.io](http://redis.io).

Redis is a well-established open source project and has been used in production for years by big companies, including Twitter, GitHub, Tumblr, Pinterest, Instagram, Hulu, Flickr, and The New York Times.

This chapter is going to show you how to install Redis, introduce the command-line interface, introduce a Node.js client for Redis, and then present three data types in detail: Strings, Lists, and Hashes.
Redis data types are a very extensive subject. There is enough information to write a book that just describes how they work. We will present the most relevant and useful commands for each data type along with real-life use cases in the first two chapters. Chapter 2, Advanced Data Types (Earning a Black Belt), is going to cover other data types: Sets, Sorted Sets, Bitmaps, and HyperLogLogs. After this chapter and the next have explained all data types, Chapter 3, Time Series (A Collection of Observations), will present a time series implementation that uses multiple data types.

Please note that all data types will be shown with the first letter capitalized (for example, Strings, Lists, Bitmaps, Sets, Sorted Sets, and HyperLogLogs) so that we can distinguish between a Redis data type and other existing terms.

Installation

At the time of writing this book, the stable version of Redis was 3.0. All examples presented in this book will work with this version, but it is very likely that newer versions of Redis are going to work as well. Redis is very strict in terms of backward compatibility, so it provides API stability between minor versions. We recommend that you install the latest version of Redis to get the recent bug fixes and performance improvements. Most of the content in this book will remain useful even if you work with a more recent version.

Officially, Redis can be compiled and used on Linux, OS X, OpenBSD, NetBSD, and FreeBSD.

Redis is not officially supported on Windows. However, the Microsoft Open Tech group develops and maintains a Windows port targeting Win64 architecture, which can be found at https://github.com/MSOpenTech/redis. We are not going to cover Windows installation or guarantee that the examples presented in this book will work on Windows.

Installing from source

The first thing we need to do is open a terminal and run the following commands to download and install Redis. The following commands can be executed in any *nix operating system (Ubuntu, CentOS, Debian, OS X, and so on). Some build tools are required to build Redis from source (for example, gcc, make, and so on). On Ubuntu and Debian, these tools can be installed by the package build-essentials.
On OS X, you will need Xcode and Command Line Tools Package installed. After the required build tools are installed, open a terminal window and execute the following commands:

$ curl -O http://download.redis.io/releases/redis-3.0.2.tar.gz
$ tar xzvf redis-3.0.2.tar.gz
$ cd redis-3.0.2
$ sudo make install

Every time you see a dollar sign ($) at the beginning of a code block, it means we are executing the command in a terminal window.

Another way to install Redis is by using package managers, such as `yum`, `apt`, or `brew`. Make sure your package manager has Redis 3.0 or later available.

Hello Redis (command-line interface examples)
Redis comes with several executables. In this section, we are going to focus on `redis-server` and `redis-cli`.

`redis-server` is the actual Redis data store. It can be started in standalone mode or in cluster mode. For now, we are only going to use the single-instance mode and later (in Chapter 9, Redis Cluster and Redis Sentinel (Collective Intelligence)) we will cover cluster mode.

`redis-cli` is a command-line interface that can perform any Redis command (it is a Redis client). It makes learning to execute commands in Redis more intuitive.

This chapter is also going to introduce a Node.js client, and later (in Chapter 5, Clients for Your Favorite Language (Become a Redis Polyglot)) we will see how to use Redis with PHP, Python, and Ruby clients.

By default, Redis binds to port 6379, runs in standalone mode, and can be started with this line:

$ redis-server

Since no configuration was specified in this example, Redis will use default configurations.
Getting Started (The Baby Steps)

It will output its PID (process ID) and the port that the clients should connect to, which is 6379 by default.

```
$ redis-server
85041:C 18 Jun 00:38:46.158 * Warning: no config file specified, using the default config. In order to specify a config file use redis-server /path/to/redis.conf
85041:M 18 Jun 00:38:46.159 * Increased maximum number of open files to 10032 (it was originally set to 2560).

Redis 3.0.2 (00000000/0) 64 bit
Running in standalone mode
Port: 6379
PID: 85041

http://redis.io

```

Important note:
The following conventions will be used in this book for `redis-cli`:

- Commands are written in bold, uppercase letters (`SET`).
- Keys are written in italicized, lowercase letters (`GET mykey`).
- Values are written without any text formatting (`SET mykey "my value"`).

The next snippet shows how to connect to the Redis server using `redis-cli`. Once connected, we use the `SET` command to create a key with a string value and then the `GET` command to read the key value:

```
$ redis-cli
127.0.0.1:6379> SET philosopher "socrates"
OK
127.0.0.1:6379> GET philosopher
"socrates"
127.0.0.1:6379>
```
The **HELP** command is useful for learning about command syntax. It displays the command parameters with a summary and examples. See the following example:

```
$ redis-cli
127.0.0.1:6379> HELP SET

**SET** key value [**EX** seconds] [**PX** milliseconds] [**NX**|**XX**]
summary: Set the string value of a key
since: 1.0.0
group: string
```

The **KEYS** command is also useful, as it returns all stored keys that match a pattern (it is a glob-style pattern, like the Unix shell glob pattern). In the following code, all stored key names that start with the letter "p" are returned:

```
$ redis-cli
127.0.0.1:6379> KEYS p*
1) "philosopher"
```

The **redis-cli** is a great tool for debugging and testing commands, but making real examples and applications using **redis-cli** is impractical. This book is going to use the JavaScript language and Node.js to support examples and explanations. We chose JavaScript because of its current popularity. The Node.js website (https://nodejs.org) provides binaries for Mac OS X, Windows, and Linux, which makes installation of Node.js really simple. Keep in mind that this is not a JavaScript book; we are going to use basic features of the language in our examples. If you do not know how to code in JavaScript, do not worry. A quick syntax reference is presented, and it should be enough to understand all the examples in this book.

You can reproduce all the samples presented here in your favorite language. Redis will produce the same results regardless of the programming language.

## Installing Node.js

Download and install Node.js from its website using the available binary packages. At the time of writing this book, the latest version of Node.js was 0.12.4. All examples are guaranteed to work with this version.

Node.js comes with a package manager called **Node Package Manager** (**NPM**), which is responsible for managing and installing all Node.js dependencies and libraries. Think of it as **pip** for Python or **cpan** for Perl.
We recommend that you create a folder called *redis-essentials* to save all the files and libraries necessary for running the examples. We also recommend that you create one folder for each chapter of this book for organization purposes.

All Node.js examples in this book require the library *redis*, which can be installed with NPM:

```
$ cd redis-essentials
$ npm install redis
```

NPM will create a folder called *node_modules*. This is where the *redis* client is installed.

### JavaScript syntax quick reference guide

If you know the basics of JavaScript, you can skip this section. Here is a quick overview of JavaScript:

- Use the keyword `var` to define a variable:
  ```javascript
  var myAge = 31;
  ```

- Use `//` for inline comments and `/* */` for multiline comments:
  ```javascript
  // this is an inline comment
  /* this
     is a
     multi-line
     comment
  */
  ```

- Conditional statements:
  ```javascript
  if (myAge > 29) {
    console.log("I am not in my twenties anymore!");
  } else {
    console.log("I am still in my twenties!");
  }
  ```

- Defining a function:
  ```javascript
  function nameOfMyFunction(argument1, argument2) {
    console.log(argument1, argument2);
  }
  ```

- Executing a function:
  ```javascript
  nameOfMyFunction("First Value", "Second Value");
  ```
• A function can also behave as a class and have methods, properties, and instances. Properties are accessed through the keyword `this`:

```javascript
function Car(maxSpeed) {
    this.maxSpeed = maxSpeed;
    this.currentSpeed = 0;
}
```

• The standard way to create a prototyped method for a function in JavaScript is by using the property `prototype`:

```javascript
Car.prototype.brake = function() {
    if (this.currentSpeed > 0) {
        this.currentSpeed -= 5;
    }
};

Car.prototype.accelerate = function() {
    if (this.currentSpeed < this.maxSpeed) {
        this.currentSpeed += 5;
    }
};
```

• To create an instance of a class in JavaScript, use the keyword `new`:

```javascript
var car = new Car(100);
car.accelerate();
car.accelerate();
car.brake();
```

• Arrays and objects:

```javascript
var myArray = [];
var myObject = {};
```

• Callbacks in JavaScript:

```javascript
var friends = ["Karalyn", "Patrik", "Bernardo"];
friends.forEach(function (name, index) {
    console.log(index + 1, name); // 1 Karalyn, 2 Patrik, 3 Bernardo
});
```

A callback in this example is an anonymous function that is passed to another function as a parameter, so it is called (or executed) inside the other function. As you can see in the preceding example, the `forEach` array method expects a callback function. It executes the provided callback once for each element in the array. It is very common to find asynchronous functions/methods that expect callbacks in JavaScript.
If you want to know more about JavaScript syntax and features, we recommend the Mozilla Developer Network website at https://developer.mozilla.org/en-US/docs/Web/JavaScript.

**Hello World with Node.js and Redis**

This section shows the basics of creating a JavaScript program using Redis. It is important to understand this foundation since the upcoming examples use the same principles.

In this book, all filenames, function names, and variable names are italicized. Some sentences follow this convention:

- Create a file called *my-filename.js*.
- Execute the function *myFunctionName*.
- Create a variable called *myVariableName*.

Create a file called *hello.js* with the following code:

```javascript
var redis = require("redis"); // 1
var client = redis.createClient(); // 2
client.set("my_key", "Hello World using Node.js and Redis"); // 3
client.get("my_key", redis.print); // 4
client.quit(); // 5
```

Please note that all the code snippets in this book will have inline comments with numbers. After the code is presented, it will be explained by referencing those numbers.

1. Require the *redis* library in Node.js. This is equivalent to *import* in Go, Python, or Java.
2. Create the Redis client object.
3. Execute the Redis command *SET* to save a String in a key called *my_key*.
4. Execute the Redis command *GET* to get the value stored in *my_key*, and then output it.
5. Close the connection with the Redis server.

Lines 1, 2, and 5 of this example will be used in the majority of the examples that use Node.js.
Run *hello.js* with the *node* command (*node* is the Node.js interpreter):

```
$ node hello.js
```

Reply: Hello World using Node.js and Redis

## Redis data types

After you have understood how Redis data types work, you will be able to design better applications and make better use of the available resources. It will also help you decide whether Redis is the right solution for your problem. The main reason for Redis to have many data types is very simple: one size does not fit all, and different problems require different solutions.

Although you do not need to use all the data types, it is important to understand how they work so that you can choose the right ones. By the end of this book, you will have a full understanding of these data types and know how to improve the performance of your applications using Redis.

### Strings

Strings are the most versatile data types in Redis because they have many commands and multiple purposes. A String can behave as an integer, float, text string, or bitmap based on its value and the commands used. It can store any kind of data: text (XML, JSON, HTML, or raw text), integers, floats, or binary data (videos, images, or audio files). A String value cannot exceed 512 MB of text or binary data.

The following are some use cases for Strings:

- **Cache mechanisms**: It is possible to cache text or binary data in Redis, which could be anything from HTML pages and API responses to images and videos. A simple cache system can be implemented with the commands `SET`, `GET`, `MSET`, and `MGET`.

- **Cache with automatic expiration**: Strings combined with automatic key expiration can make a robust cache system using the commands `SETEX`, `EXPIRE`, and `EXPIREAT`. This is very useful when database queries take a long time to run and can be cached for a given period of time. Consequently, this avoids running those queries too frequently and can give a performance boost to applications.
• **Counting:** A counter can easily be implemented with Strings and the commands **INCR** and **INCRBY**. Good examples of counters are page views, video views, and likes. Strings also provide other counting commands, such as **DECR**, **DECRBY**, and **INCRFLOATBY**.

### String examples with redis-cli

The **MSET** command sets the values of multiple keys at once. The arguments are key-value pairs separated by spaces.

The **MGET** command retrieves the values of multiple key names at once, and the key names are separated by spaces.

The following is a combined example for the preceding commands:

```bash
$ redis-cli
127.0.0.1:6379> MSET first "First Key value" second "Second Key value"
OK
127.0.0.1:6379> MGET first second
1) "First Key value"
2) "Second Key value"
```

The **EXPIRE** command adds an expiration time (in seconds) to a given key. After that time, the key is automatically deleted. It returns 1 if the expiration is set successfully and 0 if the key does not exist or cannot be set.

The **TTL** (**Time To Live**) command returns one of the following:

- **A positive integer:** This is the amount of seconds a given key has left to live
- **-2:** If the key is expired or does not exist
- **-1:** If the key exists but has no expiration time set

```bash
$ redis-cli
127.0.0.1:6379> SET current_chapter "Chapter 1"
OK
127.0.0.1:6379> EXPIRE current_chapter 10
(integer) 1
127.0.0.1:6379> GET current_chapter
"Chapter 1"
127.0.0.1:6379> TTL current_chapter
(integer) 3
127.0.0.1:6379> TTL current_chapter
```

---

[10]
The commands **INCR** and **INCRBY** have very similar functionality. **INCR** increments a key by 1 and returns the incremented value, whereas **INCRBY** increments a key by the given integer and returns the incremented value. **DECR** and **DECRBY** are the opposites of **INCR** and **INCRBY**. The only difference is that **DECR** and **DECRBY** decrements a key.

The command **INCRBYFLOAT** increments a key by a given float number and returns the new value. **INCRBY**, **DECRBY**, and **INCRBYFLOAT** accept either a positive or a negative number:

```
$ redis-cli
127.0.0.1:6379> SET counter 100
OK
127.0.0.1:6379> INCR counter
(integer) 101
127.0.0.1:6379> INCRBY counter 5
(integer) 106
127.0.0.1:6379> DECR counter
(integer) 105
127.0.0.1:6379> DECRBY counter 100
(integer) 5
127.0.0.1:6379> GET counter
"5"
127.0.0.1:6379> INCRBYFLOAT counter 2.4
"7.4"
```

The preceding commands shown are atomic, which means that they increment/decrement and return the new value as a single operation. It is not possible for two different clients to execute the same command at the same time and get the same result—no race conditions happen with those commands.

For example, if the `counter` key is 1 and two different clients (A and B) increment their counters at the same time with **INCR**, client A will receive the value 2 and client B will receive 3.

```
Redis is single threaded, which means that it always executes one command at a time. Sometimes, commands are mentioned as atomic, which means that a race condition will never happen when multiple clients try to perform operations on the same key at the same time.
```

```markdown
[11]
```
Building a voting system with Strings using Node.js

This section builds a set of Node.js functions used to upvote and downvote articles. The idea is that there is a set of articles, and users can define their popularity by voting up or down.

Now let's save a small collection of articles in Redis using `redis-cli`. We will only add three article headlines to make the example easier to understand. In a real-world situation, you would use a Redis client for your programming language (rather than `redis-cli`), and the articles would be retrieved from a database:

```
$ redis-cli
127.0.0.1:6379> SET article:12345:headline "Google Wants to Turn Your Clothes Into a Computer"
OK
127.0.0.1:6379> SET article:10001:headline "For Millennials, the End of the TV Viewing Party"
OK
127.0.0.1:6379> SET article:60056:headline "Alicia Vikander, Who Portrayed Denmark's Queen, Is Screen Royalty"
OK
```

To complete this example, we will need two keys in Redis for each article. We have already defined our first key to store the headline of each article. Observe this key name structure: `article:<id>:headline`. The second key name will have a similar structure: `article:<id>:votes`. This nomenclature is important in order to create abstractions. The IDs may be passed around, and even if the key format changes, the application logic will remain the same. Also, it is easy to extend the application if other metadata (URL, summary, and so on) needs to be stored.

Our code will have three functions: the first increments the number of votes in an article by 1, the second decrements the number of votes in an article by 1, and the third displays the article headline and the number of votes. All three functions (`upVote`, `downVote`, and `showResults`) require the article ID as the argument. Perform the following set of steps:

Create a file called `articles-popularity.js` in the `chapter 1` folder where all of the code from this section should be saved:

```javascript
var redis = require("redis"); // 1
var client = redis.createClient(); // 2

function upVote(id) { // 3
    var key = "article:" + id + ":votes"; // 4
    client.incr(key); // 5
}
```

---

[12]
1. Require the redis library in Node.js. This is equivalent to import in other languages.

2. Create a Redis client instance.

3. Create an upVote function that has the article ID as the argument.

4. Define your key name using the article:<id>:votes structure.

5. Use the INCR command to increment the number of votes by 1.

The function downVote is basically the same as upVote. The only difference is that it uses the command DECR instead of INCR:

```javascript
function downVote(id) {
  var key = "article:" + id + ":votes";
  client.decr(key);
}
```

1. Create a function downVote that has the article ID as the argument.

2. Define your key name using the structure article:<id>:votes (just as we did in the upVote function).

3. Use the DECR command to decrement the number of votes by 1.

The function showResults shows the article headline and the number of votes that an article has:

```javascript
function showResults(id) {
  var headlineKey = "article:" + id + ":headline";
  var voteKey = "article:" + id + ":votes";
  client.mget([headlineKey, voteKey], function(err, replies) {
    console.log('The article "' + replies[0] + '" has ' + replies[1] + ' votes');
  });
}
```

1. Use the MGET command to pass an array of keys and a callback function. For every key that does not hold a String value or does not exist, the value null is returned.

   In the anonymous function, the argument replies has two values: index 0, which has the headline, and index 1, which has the number of votes.

2. Display a message with the article headline and number of votes.
Note:
The Node.js client that we are using is strictly asynchronous. All Redis commands have an optional callback function for handling errors and replies from the Redis server.

In the previous MGET example, the only way to handle the key values is by passing a callback to `client.mget()`.

Please make sure you fully understand the idea of callbacks mentioned before. This is necessary in order to understand other examples using Node.js.

It is time to call our functions `upVote`, `downVote`, and `showResults`. Add the following to `articles-popularity.js` too:

```javascript
upVote(12345); // article:12345 has 1 vote
upVote(12345); // article:12345 has 2 votes
upVote(12345); // article:12345 has 3 votes
upVote(10001); // article:10001 has 1 vote
upVote(10001); // article:10001 has 2 votes
downVote(10001); // article:10001 has 1 vote
upVote(60056); // article:60056 has 1 vote

showResults(12345);
showResults(10001);
showResults(60056);

client.quit();
```

Then execute it using the following command line:

```
$ node articles-popularity.js
```

The article "Google Wants to Turn Your Clothes Into a Computer" has 3 votes
The article "For Millennials, the End of the TV Viewing Party" has 1 votes
The article "Alicia Vikander, Who Portrayed Denmark's Queen, Is Screen Royalty" has 1 votes

**Downloading the example code**

You can download the example code files from your account at http://www.packtpub.com for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.
Lists

Lists are a very flexible data type in Redis because they can act like a simple collection, stack, or queue. Many event systems use Redis's Lists as their queue because Lists' operations ensure that concurrent systems will not overlap popping items from a queue—List commands are atomic. There are blocking commands in Redis's Lists, which means that when a client executes a blocking command in an empty List, the client will wait for a new item to be added to the List. Redis's Lists are linked lists, therefore insertions and deletions from the beginning or the end of a List run in $O(1)$, constant time.

The task of accessing an element in a List runs in $O(N)$, linear time, but accessing the first or last element always runs in constant time.

A List can be encoded and memory optimized if it has less elements than the `list-max-zipmap-entries` configuration and if each element is smaller than the configuration `list-max-zipmap-value` (in bytes). Chapter 4, Commands (Where the Wild Things Are) provides more details on these configurations.

The maximum number of elements a List can hold is $2^{32}-1$, which means there can be more than 4 billion elements per List.

Some real-world use cases of Lists are as follows:

- **Event queue**: Lists are used in many tools, including Resque, Celery, and Logstash
- **Storing most recent user posts**: Twitter does this by storing the latest tweets of a user in a List

In this section, we will show you some List commands using the `redis-cli`, and then present a generic task queue system in Node.js.

**List examples with redis-cli**

Since Lists in Redis are linked lists, there are commands used to insert data into the head and tail of a List. The command `LPUSH` inserts data at the beginning of a List (left push), and the command `RPUSH` inserts data at the end of a List (right push):

```
$ redis-cli
127.0.0.1:6379> LPUSH books "Clean Code"
(integer) 1
127.0.0.1:6379> RPUSH books "Code Complete"
(integer) 2
127.0.0.1:6379> LPUSH books "Peopleware"
(integer) 3
```
The command **LLEN** returns the length of a List. The command **LINDEX** returns the element in a given index (indices are zero-based). Elements in a List are always accessed from left to right, which means that index 0 is the first element, index 1 is the second element, and so on. It is possible to use negative indices to access the tail of the List, in which -1 is the last element, -2 is penultimate element, and so on. **LINDEX** does not modify a List:

```
$ redis-cli
127.0.0.1:6379> LLEN books
(integer) 3
127.0.0.1:6379> LINDEX books 1
"Clean Code"
```

The command **LRANGE** returns an array with all elements from a given index range, including the elements in both the start and end indices. As we mentioned previously, indices are zero-based and can be positive or negative. See the following example:

```
$ redis-cli
127.0.0.1:6379> LRANGE books 0 1
1) "Peopleware"
2) "Clean Code"
127.0.0.1:6379> LRANGE books 0 -1
1) "Peopleware"
2) "Clean Code"
3) "Code Complete"
```

The command **LPOP** removes and returns the first element of a List. The command **RPOP** removes and returns the last element of a List. Unlike **LINDEX**, both **LPOP** and **RPOP** modify the List:

```
$ redis-cli
127.0.0.1:6379> LPOP books
"Peopleware"
127.0.0.1:6379> RPOP books
"Code Complete"
127.0.0.1:6379> LRANGE books 0 -1
1) "Clean Code"
```

**Implementing a generic Queue System**

The following implementation is going to use JavaScript prototypes, and it is going to be similar to a class-based solution seen in many programming languages.
Create a file called `queue.js` in the `chapter 1` folder with the following code:

```javascript
function Queue(queueName, redisClient) {
  this.queueName = queueName;
  this.redisClient = redisClient;
  this.queueKey = 'queues:' + queueName;
  this.timeout = 0;
}
```

1. Create a function called `Queue`, which receives a queue name and the Redis client object as parameters.
2. Save `queueName` as a property.
3. Save `redisClient` as a property.
4. Set the property `queueKey` to the proper Redis key name, based on the function parameter.
5. Set the property `timeout` to zero, which means that when List commands are executed, they will have no timeout.

We need to implement three methods to perform queue operations: `size`, `push`, and `pop`.

The first method we are going to create is `size`:

```javascript
Queue.prototype.size = function(callback) {
  this.redisClient.llen(this.queueKey, callback);
};
```

1. Create the `Queue` method `size`, which expects a callback as an argument.
2. Execute `LLEN` on the queue key name and pass the callback as an argument. This is necessary because the Redis client is asynchronous.

The implementation of the `push` method is as follows:

```javascript
Queue.prototype.push = function(data) {
  this.redisClient.lpush(this.queueKey, data);
};
```

1. Create the `Queue` method `push` that expects one argument. This argument can be anything that can be represented as a string.
2. Execute `LPUSH` by passing the queue key name and the `data` argument.
As this is a generic queue system and Redis lists only store bytes, we assume that all of the data that is sent to the queue can be transformed into a JavaScript string. If you want to make it more generic, you can use JSON serialization and store the serialized string. The previous example used `LPUSH` because we were implementing a queue, and by definition, items are inserted at the front of the queue and removed from the end of the queue. A helpful way to remember this is FIFO (First In, First Out)—we went from left to right.

The implementation of the `pop` method is as follows:

```javascript
Queue.prototype.pop = function(callback) {
    1. // 1
       this.redisClient.brpop(this.queueKey, this.timeout, callback); // 2
};
```

1. Create the `Queue` method `pop`, which expects a callback as an argument.
2. Execute `BRPOP`, passing the queue key name, the queue timeout property, and the callback as arguments.

As we mentioned earlier, elements are inserted at the front of the queue and removed from the end of the queue, which is why `BRPOP` was used (if `RPUSH` was used, then `BLPOP` would be necessary).

The command `BRPOP` removes the last element of a Redis List. If the List is empty, it waits until there is something to remove. `BRPOP` is a blocking version of `RPOP`. However, `RPOP` is not ideal. If the List is empty, we would need to implement some kind of polling by ourselves to make sure that items are handled as soon as they are added to the queue. It is better to take advantage of `BRPOP` and not worry about empty lists.

A concrete producer/consumer implementation is shown next. Different log messages are pushed into the "logs" queue by the producer and then popped by the consumer in another terminal window.

The complete `Queue` code, saved as `queue.js`, is as follows:

```javascript
function Queue(queueName, redisClient) {
    this.queueName = queueName;
    this.redisClient = redisClient;
    this.queueKey = 'queues:' + queueName;
    // zero means no timeout
    this.timeout = 0;
}

Queue.prototype.size = function(callback) {
    this.redisClient.llen(this.queueKey, callback);
};
```
Queue.prototype.push = function(data) {
    this.redisClient.lpush(this.queueKey, data);
};

Queue.prototype.pop = function(callback) {
    this.redisClient.brpop(this.queueKey, this.timeout, callback);
};

exports.Queue = Queue; // 1

1. This is required to expose Queue to different modules. This explicit export is specific to Node.js, and it is necessary in order to run require("./queue").

Create a file called producer-worker.js in the chapter 1 folder, which is going to add log events to a queue named "logs", and save the following:

```javascript
var redis = require("redis");
var client = redis.createClient();
var queue = require("./queue"); // 1
var logsQueue = new queue.Queue("logs", client); // 2
var MAX = 5;
for (var i = 0 ; i < MAX ; i++) { // 3
    logsQueue.push("Hello world #' + i); // 4
}
console.log("Created " + MAX + " logs"); // 5
client.quit();
```

1. Require the module queue, which we've already created and saved as queue.js.
2. Create an instance of the function Queue defined in the queue.js file.
3. Create a loop that runs five times.
4. Push some logs into the logs queue.
5. Print the number of logs created.

Execute the producer file to push logs into the queue:

```
$ node producer-worker.js
Created 5 logs
```

Save the following code in a file called consumer-worker.js:

```javascript
var redis = require("redis");
var client = redis.createClient();
var queue = require("./queue"); // 1
var logsQueue = new queue.Queue("logs", client); // 2
```
function logMessages() { // 3
  logsQueue.pop(function(err, replies) { // 4
    var queueName = replies[0];
    var message = replies[1];
    console.log("[consumer] Got log: " + message); // 5

    logsQueue.size(function(err, size) { // 6
      console.log(size + " logs left");
    });

    logMessages(); // 7
  });
}

logMessages(); // 8

1. Require the queue module (this is the queue.js file).
2. Create a Queue instance named logs and pass the Redis client to it.
3. Create the function logMessages.
4. Retrieve an element from the queue instance using the pop method. If the List is empty, this function waits until a new element is added. The timeout is zero and it uses a blocking command, BRPOP, internally.
5. Display a message retrieved from the queue.
6. Display the queue size after popping a message from the queue.
7. Call the function (recursively) to repeat the process over and over again. This function runs forever.
8. Call logMessages to initialize the queue consumption.

This queue system is completed. Now run the file consumer-worker.js and watch the elements being popped in the same order in which they were added by producer-worker.js:

$ node consumer-worker.js
[consumer] Got log: Hello world #0
4 logs left
[consumer] Got log: Hello world #1
3 logs left
[consumer] Got log: Hello world #2
2 logs left
[consumer] Got log: Hello world #3
1 logs left
[consumer] Got log: Hello world #4
0 logs left
This file will run indefinitely. More messages can be added to the queue by executing `producer-worker.js` again in a different terminal, and the consumer will continue reading from the queue as soon as new items are added.

The example shown in this section is not reliable enough to deploy to production. If anything goes wrong with the callbacks that pop from the queue, items may be popped but not properly handled. There is no such thing as a retry or any way to track failures.

A good way of solving the reliability problem is to use an additional queue. Each element that is popped from the queue goes to this additional queue. You must remove the item from this extra queue only if everything has worked correctly. You can monitor this extra queue for stuck elements in order to retry them or create failure alerts. The command `RPOPLPUSH` is very suitable for this situation, because it does a `RPOP` in a queue, then does a `LPUSH` in a different queue, and finally returns the element, all in a single step—it is an atomic command.

**Hashes**

Hashes are a great data structure for storing objects because you can map fields to values. They are optimized to use memory efficiently and look for data very fast. In a Hash, both the field name and the value are Strings. Therefore, a Hash is a mapping of a String to a String.

Previously, in the String example, we used two separate keys to represent an article headline and its votes (`article:<id>:headline` and `article:<id>:votes`). It is more semantic to use a Hash in that case because the two fields belong to the same object (that is, the article).

Another big advantage of Hashes is that they are memory-optimized. The optimization is based on the `hash-max-zipmap-entries` and `hash-max-zipmap-value` configurations. Chapter 4, Commands (Where the Wild Things Are), provides more details on these configurations.

Internally, a Hash can be a ziplist or a hash table. A ziplist is a dually linked list designed to be memory efficient. In a ziplist, integers are stored as real integers rather than a sequence of characters. Although a ziplist has memory optimizations, lookups are not performed in constant time. On the other hand, a hash table has constant-time lookup but is not memory-optimized.
Instagram had to back-reference 300 million media IDs to user IDs, and they decided to benchmark a Redis prototype using Strings and Hashes. The String solution used one key per media ID and around 21 GB of memory. The Hash solution used around 5 GB with some configuration tweaks. The details can be found at http://instagram-engineering.tumblr.com/post/12202313862/storing-hundreds-of-millions-of-simple-key-value.

This section is going to show the most used Hash commands using redis-cli, and then present an application that stores movie metadata in Node.js (similar to the http://www.imdb.com website).

**Using Hashes with redis-cli**

The command `HSET` sets a value to a field of a given key. The syntax is `HSET key field value`.

The command `HMSET` sets multiple field values to a key, separated by spaces. Both `HSET` and `HMSET` create a field if it does not exist, or overwrite its value if it already exists.

The command `HINCRBY` increments a field by a given integer. Both `HINCRBY` and `HINCRBYFLOAT` are similar to `INCRBY` and `INCRBYFLOAT` (not presented in the following code):

```
$ redis-cli
127.0.0.1:6379> HSET movie "title" "The Godfather"
(integer) 1
127.0.0.1:6379> HMSET movie "year" 1972 "rating" 9.2 "watchers" 10000000
OK
127.0.0.1:6379> HINCRBY movie "watchers" 3
(integer) 10000003
```

The command `HGET` retrieves a field from a Hash. The command `HMGET` retrieves multiple fields at once:

```
127.0.0.1:6379> HGET movie "title"
"The Godfather"
127.0.0.1:6379> HMGET movie "title" "watchers"
1) "The Godfather"
2) "10000003"
```

The command `HDEL` deletes a field from a Hash:

```
127.0.0.1:6379> HDEL movie "watchers"
(integer) 1
```
The command \texttt{HGETALL} returns an array of all field/value pairs in a Hash:

\begin{verbatim}
127.0.0.1:6379> HGETALL movie
1) "title"
2) "The Godfather"
3) "year"
4) "1972"
5) "rating"
6) "9.2"
\end{verbatim}

It is possible to retrieve only the field names or field values of a Hash with the commands \texttt{HKEYS} and \texttt{HVALS} respectively.

In the next section, we are going to use Hashes to implement a voting system similar to the one presented with Strings.

\section*{A voting system with Hashes and Node.js}

This section creates a set of functions to save a link and then upvote and downvote it. This is a very simplified version of something that a website like \url{http://www.reddit.com} does.

Create a file called \texttt{hash-voting-system.js} in the \texttt{chapter 1} folder, where all of the code from this section should be saved:

\begin{verbatim}
var redis = require("redis"); // 1
var client = redis.createClient(); // 2

function saveLink(id, author, title, link) { // 3
  client.hmset("link:" + id, "author", author, "title", title, "link", link, "score", 0); // 4
}

function upVote(id) { // 1
  client.hincrby("link:" + id, "score", 1); // 2
}
\end{verbatim}

1. Require the module \texttt{redis}.
2. Create a Redis client instance.
3. Create a function \texttt{saveLink} that has \texttt{id}, \texttt{author}, \texttt{title}, and \texttt{link} as arguments.
4. Use \texttt{HMSET} to create a Hash with all fields.

The \texttt{upVote} and \texttt{downVote} functions use the same command (\texttt{HINCRBY}). The only difference is that \texttt{downVote} passes a negative number:

\begin{verbatim}
function upVote(id) { // 1
  client.hincrby("link:" + id, "score", 1); // 2
}
\end{verbatim}
function downVote(id) { // 3
    client.hincrby("link:" + id, "score", -1); // 4
}

1. Create an upVote function, which has the link ID as the argument.
2. Use the command HINCRBY to increment the field score value.
3. Create a downVote function, which has its link ID as the argument.
4. Use the HINCRBY command to decrement the field score value. There is no HDECRBY command in Hash. The only way to decrement a Hash field is by using HINCRBY and a negative number.

The function showDetails shows all the fields in a Hash, based on the link ID:

function showDetails(id) { // 1
    client.hgetall("link:" + id, function(err, replies) { // 2
        console.log("Title:", replies['title']); // 3
        console.log("Author:", replies['author']); // 3
        console.log("Link:", replies['link']); // 3
        console.log("Score:", replies['score']); // 3
        console.log("-----------------------------");
    });
}

1. Create a function showDetails that has link ID as the argument.
2. Use the HGETALL command to retrieve all the fields of a Hash.
3. Display all the fields: title, author, link, and score.

Use the previously defined functions to save two links, upvote and downvote them, and then display their details:

upVote(123);
upVote(123);
saveLink(456, "hltbra", "Hugo Tavares's Github page", "https://github.com/hltbra");
upVote(456);
upVote(456);
downVote(456);
showDetails(123);
showDetails(456);

client.quit();

Then execute hash-voting-system.js:

$ node hash-voting-system.js

Title: Maxwell Dayvson's Github page
Author: dayvson
Link: https://github.com/dayvson
Score: 2

--------------------------
Title: Hugo Tavares's Github page
Author: hltbrea
Link: https://github.com/hltbrea
Score: 1

--------------------------

The command HGETALL may be a problem if a Hash has many fields and uses a lot of memory. It may slow down Redis because it needs to transfer all of that data through the network. A good alternative in such a scenario is the command HSCAN.

HSCAN does not return all the fields at once. It returns a cursor and the Hash fields with their values in chunks. HSCAN needs to be executed until the returned cursor is 0 in order to retrieve all the fields in a Hash:

$ redis-cli
127.0.0.1:6379> HMSET example "field1" "value1" "field2" "value2" "field3" "value3"
OK
127.0.0.1:6379> HSCAN example 0
1) "0"
2) 1) "field2"
   2) "value2"
   3) "field1"
   4) "value1"
   5) "field3"
   6) "value3"
Summary

This chapter began with information about Redis's history and some of its design decisions. We explained how to install Redis and demonstrated that the `redis-cli` tool can be a very powerful tool for debugging and learning Redis.

Some examples in this book that require a programming language are implemented in Node.js. Therefore, a quick reference to JavaScript's syntax and Node.js installation were shown.

Redis data types is an extensive subject, and it has been split into two chapters. This chapter explained how Strings, Lists, and Hashes work. The next chapter will cover Sets, Sorted Sets, Bitmaps, and HyperLogLogs and give practical examples.
Where to buy this book

You can buy Redis Essentials from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.