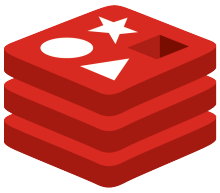


Introduction to Redis

Note to the Reader

This document was generated from the Notebook files used in the training course. It is intended as an alternative way of accessing the iPython Notebooks. We recommend using the executable iPython notebooks if possible.

01 - Connecting to Redis

1 Connecting to Redis

The first step in working with Redis, like most any other database, is setting up a connection to the database server. The server could be running on the same machine as the application program or on a remote machine dedicated to running the database.

This chapter will walk through establishing a connection to Redis on either a local or remote host, and using that connection to implement your first Redis program. To accomplish these tasks, you will need to learn about:

- Redis client libraries
- Database connection parameters
- Establishing connections to a database

After learning the basics of establishing a connection, we will walk through a Redis version of the classic "Hello World" program.

1. Client Libraries

Redis is accessed through a client object that provides methods for executing the Redis commands. Most Redis libraries as well as a host of other NoSQL and relational database libraries use this paradigm of exposing a client object, which is popular across all programming languages.

The most popular client library for Python is the open-source redis-py package. The redis-py package can be installed on a system using any of the popular Python package management systems. We will be using this library for all of our examples.

Creating a Redis client with redis-py is extremely simple—just instantiate an instance of the StrictRedis object. Once constructed, the resulting instance allows you to manipulate data on your Redis server through method calls.

Note

For historical reasons, the redis-py API provides two client objects Redis and StrictRedis. StrictRedis should be used for all new projects. The Redis class is provided for backwards compatibility with older versions of the library.

Note

If you require the connection information for the Redis instance you are running against today, please select the code cell below and execute it using the SHIFT + ENTER keys.

```
In [ ]: pprint.pprint(config)
```

Try running the example below. Select the code cell then press SHIFT + ENTER to execute.

```
In [1]: # example connection parameters
        sample = {
            "host": "redis-server.cloud.redislabs.com", "port": 6379,
            "password": "sekret", "db": 0
        }

        # actual connection parameters are provided by the Notebook environment
        r = redis.StrictRedis(**config) workshop.is_connected(r)
```

```
Connected to DB-0@redis-16464.c8.us-east-1-3.ec2.cloud.redislabs.com:16464
```

After executing the code, you should see an output message indicating your Notebook environment is connected to a Redis database. The message will include the actual hostname, port and virtual database being used, not the information from the sample parameters.

The workshop.is_connected function was created specifically for our Notebook training environment. It is not available in any standard libraries. It is one of a set of functions provided to facilitate today's training.

3. Hello World

Hello World is the traditional program used to introduce new technologies, so why not begin our Redis training with Hello World as well? Since the key-value operations are one of the most popular ways to use Redis, it seems only fitting that we start with the Redis GET and SET commands to build our Hello World program.

Our Redis Hello World program demonstrates several key elements used in almost all Redis programs. Our first Redis program will show how to:

- Import our client library interface
- Connect to a Redis database
- Load our Hello World message into Redis

- Fetch our message from the database
- Print the results from the database as output

Run this sample program by selecting the Notebook code cell and pressing SHIFT + ENTER to execute.

In [2]:
```
import redis

# example connection parameters
sample = {
    "host": "redis-server.cloud.redislabs.com", "port":
    6379,
    "password": "sekret", "db": 0
}

# actual config is provided by the Notebook environment
r = redis.StrictRedis(**config)

r.set("workshop_message", "Hello World!")
message = r.get("workshop_message")

print message Hello
```

World!

After executing the code, you should see the familiar Hello World message printed to the Notebook.

Note

In this example, we've shown all the elements of creating a basic Redis program in Python so that you can see all the details. Most of the examples in the Notebook will hide the boilerplate code for importing the Redis libraries and making a connection to the Redis server, so that you can concentrate on the material for that Notebook.

The structure of our Python code is very straightforward and most Redis programs, even those in other programming languages, will follow a similar pattern. The first step in our program is to import the Redis client libraries. Once our client library is accessible, we declare the connection parameters for our database server and create an instance of the StrictRedis object to connect to the database. Finally, once the database connection is established, we perform commands to read and write data, ultimately printing out the fetched data as the result.

Once you have run the example, edit the code cell and change the way the Hello World program works. See if you can edit the program to store a different message in Redis and change the results.

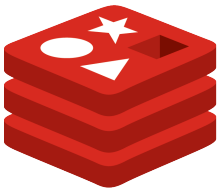
4. Review

In this chapter, we constructed a single connection to a Redis database using the Python redis-py client. We specified the connection information necessary to identify our Redis database, and then created an instance of the StrictRedis client to manipulate data on the database. Once we created our client, we accessed pre-loaded data from our database to implement a Redis version of the classic Hello World program.

This chapter has demonstrated a basic connection setup, but the redis-py library, like most Redis libraries, provides developers with a wide range of advanced connection features including:

- SSL
- Connection Pooling
- Socket parameter tuning
- Operation pipelining

More information about additional connection features can be found in the redis-py documentation.



Tracking Votes With Redis Strings

2. - Tracking Votes with Redis Strings

1 Tracking Votes with Redis Strings

This chapter will look at using the Redis **string** data type to track user votes on a website. User voting can take many forms: up/down votes, thumbs up, likes etc., but essentially all of these experiences are used to record positive and negative impressions from the user in the database.

This chapter will look at using the Redis string data type to maintain a global count of votes on a particular blog post. To developers unfamiliar with Redis, maintaining a vote count using a string might sound incorrect, but the Redis String type is different from the string types in other programming languages you are familiar with.

In this chapter, we will learn about:

- Storing data in Redis strings
- Manipulating data in Redis strings
- How Redis strings differ from string data types
- Working with bitmaps, integers and floating points

At the end of the chapter, you should understand how to use the Redis string data type to implement key-value operations and apply Redis string operations to a variety of development problems.

1. Strings

Redis provides a string data type that is used to associate data with a particular key. Redis strings are the most basic data type available in Redis and one of the first data types that users learn to work with. Our Hello World program in the chapter on connecting to Redis used the string data type.

Most programmers associate string data types with text data like the string data type in most modern programming languages. Redis strings are more like buffers that can store a wide range of data. Redis strings can be used to store:

- Integers
- Floating-point numbers
- Bitmaps
- Strings
- Binary objects

String values in Redis can be up to 512 MB in size, allowing you to store substantial binary objects including images, sound files and object files.

Note

Redis keys are also binary safe, so although most programming examples show Redis keys as strings, you can use any binary sequence as a key in Redis. Keys can also be up to 512 MB in length, but we caution against using very long keys as they can have an adverse effect on performance. Standard practice in Redis is to use structured key names with a schema, such as "comment:1234:reply.to" or "comment:1234:reply-to."

We also recommend that you do not use very short names for Redis keys either; ease-of-use and clarity should be weighed against performance needs. The performance improvement of using a key like "c1234r2" versus "comment:1234:reply.to" are minimal and don't outweigh the loss of clarity.

The current version of Redis (3.2) provides 24 different commands to work with the string data type. Redis provides commands to:

- Read and write string values
- Manipulate strings as integers
- Manipulate strings as floating-point numbers
- Use strings to store bitmaps

Redis allows you several ways to perform calculations on your data *without* having to read the data into your application, manipulate it, and then write it back to the server.

2. Tracking Votes

In our examples, votes will be integer values, both positive and negative, associated with a particular blog post. Our system will track raw votes and each user can either vote up or down for a given post. The resulting vote count is displayed to users as a way of prioritizing posts to read.

3. Data Storage Conventions

Our votes will be stored in Redis using the string data type. One string will be stored for each item under the key item:{item_no}:vote-count and the string will be used to store an integer vote count. An up-vote (positive vote) is tracked as a +1 and a down-vote (negative vote) is tracked as a -1. The votes for a particular item are the sum of all the up and down votes.

To improve the clarity of our code, the Notebook environment includes a few utility functions, allowing you to focus on the Redis implementation details. In this chapter, the main functions that will be used from the workshop package are:

- item_vote_key - generates the proper key from an item ID
- create_item_vote - clears current vote totals from the database
- get_item_vote - returns the vote total for an item
- users_vote_key - returns the key for tracking a user's vote
- clear_users_vote - clears a user's voting record

4. Vote Counting

1. Counting an Upvote

When a user views a blog post on our website, they are given the opportunity to vote on the post. If the user thought the post was a good post and wants to recommend the post to others, they can upvote the post. In our first version of counting upvotes, we aren't going to worry about duplicates, so we need code that will:

- Generate an appropriate Redis key
- Increment an item's vote count

This can be accomplished with a single Redis command: INCR.

The INCR (**INCR**ement) command operates on an integer value stored in Redis as a string and increments the value of the key on the server by one.

Run the example below by selecting the code cell and pressing SHIFT + ENTER

```
In [2]: def upvote_item(r, item_id):
            "Upvotes an item and stores it in the Redis database"

            # get our key
            key = workshop.strings.item_vote_key(item_id) return

            r.incr(key)


        # reset our database state
        workshop.strings.clear_item_vote(r, 20)

        # execute our example
        print 'Vote count for item 20: {}'.format(upvote_item(r, 20)) print

        inspect(workshop.strings.item_vote_key)
```

```
Vote count for item 20: 1
```

After running the sample code, you should see the resulting message, "Vote count for item 20: 1" output below the cell.

Notice in the code, we never create the key item:20:vote-count or check that it exists. Our code always performs an increment operation. We can write our code this way because Redis is designed to provide sensible defaults whenever data is missing. When INCR is called with a key that doesn't exist, Redis assumes a default value of 0, increments it and stores the result.

It is also important to point out that in the proceeding example, we never read the current value of the key into memory. The increment command operates on the value stored in the database. This improves performance for our code and makes it easier to support concurrent updates since we do not require a transaction.

Tip

While Redis is designed to handle missing data with sensible defaults, not all erroneous operations are handled quietly. If the INCR command is applied to a string value that can't be represented as an *integer* (including floating-point numbers), Redis will return an error.

As mentioned previously, our Notebook is intended to be an interactive environment that you can experiment with. Take some time to experiment with the example code to get more familiar with the Notebook environment. Some possible experiments to try:

- Add a Python error to the code
- Add a Redis error to the code
- Use the GET and SET commands from the Hello World program

2. Counting a Downvote

The code and process for handling downvotes is very similar to the code for handling upvotes. At the user-experience level, a visitor to our site can also downvote posts they do not find appealing or believe aren't worth the time for other readers to read. As in our first version of downvoting, we aren't going to worry about a user casting duplicate votes. The structure for the down-voting code is nearly the same as the up-voting code:

- Generate an appropriate Redis key
- Decrement item vote count

This can also be accomplished with a single Redis command - DECR.

The DECR (**DECR*ement) command operates on an integer value stored as a string and decrements the value by one.

Select the code cell below and press SHIFT + ENTER to run the down-vote example.

```
In [ ]: def downvote_item(r, item_id):
            "Downvotes an item and stores it in the Redis database"

            # Get our key
            key = workshop.strings.item_vote_key(item_id) return

            r.decr(key)


        # Reset our database state
        workshop.strings.clear_item_vote(r, 20)

        # Execute our example
        print "Vote count for item 20: {}".format(downvote_item(r, 20))
```

After running the sample code, you should see the message "Vote count for item 20: -1" printed under the cell.

As mentioned previously, this example is nearly identical to the upvote_item example. The only difference between the two examples is the Redis command used: INCR for upvotes and DECR for downvotes. All the points we discussed about INCR—sensible defaults, on-server operation and transactional—also apply to the DECR command.

3. Weighted Votes

Some voting systems prefer to give different weights to upvotes and downvotes. Often a voting system will be biased towards positive votes and weight them higher than negative votes as a simple way to discourage unwanted social behaviors. If we decide to change our voting system to

favor upvotes over downvotes, a simple refactoring to our upvote/downvote routines will implement the change.

In the following example, we modify our initial upvote_item and downvote_item to give upvotes twice the weight of downvotes. We refactor the code to use the INCRBY command instead of the INCR command. The INCRBY (**INCR**ease **BY**) command takes a key and a step or increment parameter and updates the value in the database by the supplied increment. In our case we increment each upvote by two, establishing a two-to-one weighting of positive to negative votes.

Select the code cell below and press SHIFT + ENTER to execute the example.

```
In [ ]: def upvote_item(r, item_id):
            "Upvotes an item and stores it in the Redis database"

            # Get our key
            key = workshop.strings.item_vote_key(item_id) return

            r.incrby(key, 2)

        def downvote_item(r, item_id):
            "Downvotes an item and stores it in the Redis database"

            # Get our key
            key = workshop.strings.item_vote_key(item_id) return

            r.decr(key)

        # Reset our database state
        workshop.strings.clear_item_vote(r, 20)

        # Simulate a vote sequence
        upvote_item(r, 20)
        upvote_item(r, 20)
        upvote_item(r, 20)
        downvote_item(r, 20)

        # See the vote results
        print "Vote count for item 20: {}".format(workshop.strings.get_item_vote(r, 20))
```

The get_item_vote method will return a vote count of 5 after running the sample.

Tip

The *redis-py* library allows you to pass an optional step argument to the incr and decr functions, which get mapped to the INCRBY and DECRBY commands.

---

4. Tracking Votes

Now that we've seen a couple examples of vote-recording functions, we need to flesh out our implementation to demonstrate a more realistic approach to the problem. Most websites don't allow users to vote more than once on a given item. In this iteration of the code, we will update our upvote and downvote methods to check that a user has not already registered a vote on a particular item. A user's vote will only be applied once, and then the function will ignore any additional votes, returning "False."

We will be using the Redis bitmap "type" to maintain a record of votes. As we saw earlier, Redis has commands that treat strings as integers and perform basic arithmetic on the values.

Redis also has commands that treat strings as bitmaps and can perform standard bit operations: test, set, and logical. Redis represents bitmaps as a variable-width vector of bits, with a zero-based indexing scheme. Bitmaps can grow up to Redis' maximum value size (512 MB).

We can augment our original functions with two additional commands, GETBIT and SETBIT, to record votes using Redis' bitmaps. In our code, we will store the users' votes in the key user:{user_id}:votes with a bitmap indexed by the item id. A zero value in the bitmap will indicate no vote and a one value in the bitmap will indicate that the user voted.

To prevent the user from having a duplicate vote we need to:

- Look up the user's vote for a particular item_id
- Check for a recorded vote
- Record that the user voted
- Apply the value of the user's vote

The example code below shows a modified version of our original voting functions with du- plicate checks added. *Please run the example by selecting the code cell and pressing SHIFT + ENTER to execute the code.*

```
In [ ]: def user_voted(r, user_id, item_id):
            "Checks if a user has already voted on an item"
            key = workshop.strings.users_vote_key(user_id)
            voted = r.getbit(key, item_id)

            # Voted is returned as a bit (0/1), cast to Python boolean
            return bool(voted)


        def upvote_item(r, user_id, item_id):
            "Processes a user's upvote and stores it in Redis (with duplicate checking)"

            if not user_voted(r, user_id, item_id): r.setbit(workshop.strings.users_vote_key(user_id),
                item_id, 1)
                r.incrby(workshop.strings.item_vote_key(item_id), 2) return
                True
            else:
                return False


        def downvote_item(r, user_id, item_id):
            "Processes a user's downvote and stores it in Redis (with duplicate checking)"

            if not user_voted(r, user_id, item_id): r.setbit(workshop.strings.users_vote_key(user_id),
                item_id, 1) r.decr(workshop.strings.item_vote_key(item_id))
                retur
            n True
            else:
                return False

        workshop.strings.clear_item_vote(r, 20)
        workshop.strings.clear_item_vote(r, 21)
        workshop.strings.clear_users_vote(r, 3001)
        workshop.strings.clear_users_vote(r, 3002)


        upvote_item(r, 3001, 20)
```

```
upvote_item(r, 3001, 21)
upvote_item(r, 3001, 20)

downvote_item(r, 3002, 20)
downvote_item(r, 3002, 20)

print "Vote count for item 20: {}".format(workshop.strings.get_item_vote(r, 20)) print "Vote count for item 21: {}".format(workshop.strings.get_item_vote(r, 21)) print "Vote count for item 22: {}".format(workshop.strings.get_item_vote(r, 22))
```

The result from our simulated vote sequences should be:

item	vote
20	1
21	2
22	0

Note

One thing that you might notice in the implementation of our new voting function is that there is no explicit transaction or currency control applied. In this chapter, we want to focus on basic String operations and leave currency control to be discussed later.

However, this is a good point to discuss how our Python method invocations are translated into Redis operations on the database server. Each method call is executed as a single call to and response from the Redis database. In the above example, the call to getbit in the upvote_item is executed in a separate call/response from the setbit method in the upvote_item and downvote_item functions. Since these commands are executed in a separate call and response, our code suffers from the classic concurrent update problem. As written, there is nothing in the code to prevent another execution thread from modifying the data between method invocations in our function.

The chapter on transactions and concurrency control will show you the features of Redis that you can use to maintain proper update consistency. For now, concentrate on learning the basic Redis operations and don't be concerned with concurrency.

Experiment with the implementation of upvoting and downvoting. Go back and update the implementation of upvoting, downvoting and allowing the user to change their vote after the vote is cast.

5. Counting with Floating-Point Numbers

In addition to the ability to treat strings as integers, bitmaps and text, Redis can treat strings as floating-point numbers. Redis provides the INCRBYFLOAT (**INCR**ement **BY FLOAT**) command, which takes a key and an increment (which can be negative), and updates the value of the key in the database by the increment specified.

Tip

Redis does not provide a subtype for floats or any of the other types of data that can be represented using the String data type. The TYPE operation returns the type string when called on any of the subtypes. At the same time, the regular GET and SET STRING operations can also be applied to any of the subtypes.

When you call the GET operation on a bitmap, Redis returns an escaped string representing the bitmap. On an integer or a floating point, you get the string representation of the number. You can even use the textual representation of a number or bitmap (use 0x00 escapes) to set a value in Redis.

All of the string operations can be applied to any subtype, but not all combinations will work.

Optional Exercise: Try using the various string commands on different subtypes (bitmaps, integers, floats, text) and see how the various combinations interact.

In [ ]: `# Use this cell to experiment with string commands`

6. Expanding on GET and SET

We introduced the basic Redis String manipulation commands GET and SET in our first Hello World program. GET and SET have some important variants to learn about which allow the user to operate multiple keys, control the time-to-live of keys and perform basic test-and-set operations.

6.1. Working with Multiple Keys

The Redis GET and SET commands have variants that can operate on multiple keys at a time. The GET command is paired with the MGET (**M**ultiple **GET**) command, which takes a sequence of keys and returns the value of those keys. The MSET (**M**ultiple **SET**) command works the same way —it takes a sequence of key value pairs and sets the values of all those keys.

Tip

The multiple key operations are important because they allow developers to operate on a consistent snapshot of a set of keys. As we mentioned earlier, each method call in our program is executed as a single call/response to the server. If we make multiple GET or SET calls, concurrent modifications could change the data in-between calls.

6.2. Expiring Keys

Keys written to Redis are stored until deleted, provided that persistence is enabled. Without persistence, keys are only stored until the Redis server is shutdown or rebooted. In some applications, cached values are only valid for a limited time period. Redis allows you to optionally provide an expiration time when you set data.

With the SET command, Redis allows you to optionally specify the expiration time of a key using the EX parameter for seconds or the PX parameter for milliseconds. The *redis-py* library allows you to specify both of these optional parameters as keyword arguments to the SET function. Redis provides two other (redundant) functions for setting keys with an expiration time: the PSETEX can be used to set a key with a millisecond expiration time and the SETEX command allows you to set the expiration in seconds. Since these functions are redundant to the SET command options, it is possible that they may be deprecated and removed in future versions of Redis.

6.3. Test and Set Operations

Redis provides a third variant of the SET operation to perform "test-and-set" operations. The Redis SETNX (**SET N**ot e**X**ists) command allows you to provisionally set the value of a key, provided that the key is not already defined. Redis returns a result of 1 if the key was set, and 0 if the key was not set.

The SET command provides optional NX and XX flags that update the key only when it doesn't already exist (NX) or only when it already exists (XX). These flags can be used in combination with the expiration parameters (PX and EX).

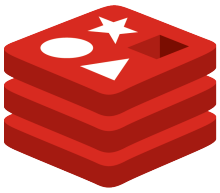
1.7 Review

In our first Hello World program, we introduced the GET and SET operations using the familiar key-value paradigm, with additional functionality and flexibility beyond the basic GET and SET operations provided by the string data type.

In this chapter, we used the string data type to store vote counts for blog posts on a website. We saw how strings could be treated as integers, allowing us to update vote counts directly in Redis using increment and decrement commands. We also saw how strings could be treated as bitmaps to provide compact representations of boolean information. Finally, we learned about several of the variants of GET and SET that support multiple keys, manage time-to-live values and perform test-and-set operations.

Redis strings are not like the strings we think of from programming languages. Redis strings can be used to store many different kinds of data, including data we don't normally think of as string data. We've seen how the GET and SET operations are used to implement a basic key-value store and we've also seen how other operations can be used to manipulate strings as integers, floating points and bitmaps.

This chapter has covered a wide range of different string functionality, but not all of the commands are covered in this chapter. For additional information on Redis strings as well as a complete command reference guide, see the String Commands page at Redis.io.



Tracking Viewed Items Using Lists

Using the LRANGE command, you can read a range of items from a list without modifying the stored list. The developer specifies a start and end position to the LRANGE command, which returns the *inclusive* range specified by those two positions relative to the **left** end of the list. As an example, given the list [0, 1, 2, 3, 4, 5], the range 1, 3 would return the list [1, 2, 3]. Negative indexes can be used to reference from the right of the list.

The LRANGE command takes a start and end position and returns the inclusive range specified by those two positions from the **left**. For example, if I had the list [0, 1, 2, 3, 4, 5], the range 1 to 3 would return the list [1, 2, 3]. Negative indexes can be used to reference from the right (end) of a list. Specifying the range -3,-1 to LRANGE would return [3, 4, 5].

Tip

Although negative indexes in Redis reference from the right of the list, ranges still have to define a contiguous block of elements that run from left to right. So the range -3, -1 will return results while the range -1, -3 returns an empty list.

Like most Redis commands for reading data, LRANGE tries to return a sensible result for missing data. The example code below shows how you can use LRANGE to fetch a user's entire history list.

```
In [ ]: def get_recently_viewed_items(r, user_id):
            "Returns a list of the most recently viewed items of the given user" key =

            workshop.lists.user_history_key(user_id)

            db_results = r.lrange(key, 0, 9) results = []
            for db_result in db_results: results.append(workshop.lists.ViewedItem.from_serialized_json(db_result))

            return results

        # Utility functions to set up our database
        workshop.lists.clear_user_history(r, 3001)
        workshop.lists.add_history(r, 3001)

        pprint.pprint(get_recently_viewed_items(r, 3001))
```

If your example runs as expected, you should see a printout of the list of ViewedItems representing the user's data.

5. Scaling

As your data set changes, you may need to reconsider the implementation used in this particular example. For simplicity, the example code stores a copy of the complete historically viewed item in each element, redundantly storing both the title and url properties across multiple users.

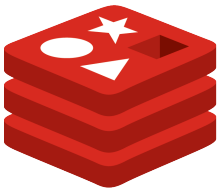
While the amount of memory that a single user might require may be reasonable, the cost of storing that redundant information may become prohibitive as your user population grows.

To support a large data set, you might consider using a more compact encoding than JSON and storing the date along with a key—such as an integer item ID—that can be used to look up the other details about the item.

6. Conclusion

The Redis List data type is a flexible way of storing ordered sequences of data. In our example, we use it to maintain a list of historically viewed items, but the underlying data structure can be used for a variety of features. You could use lists to maintain a user's news feed, implement a widget that displays a series of tweets, or even order user votes over time.

Redis lists can be used for a variety of other tasks. Many developers (and some libraries) use lists to maintain work and task queues while other developers have used lists to exchange messages between processes. More details regarding lists can be viewed on the List Commands page on Redis.io.



Storing User Data in Hashes

04 - Storing User Data in Hashes

1. Storing User Data in Hashes

1.1. Hashes

Redis provides users with a hash table or dictionary data structure that maps keys to particular values with constant access time. Redis uses the term **hash** for this particular data structure. Since Redis uses the term "key" to refer to the name referencing a data structure, the keys used to look up items in a Redis hash data structure are called "fields" and the items retrieved are referred to as "values."

A Redis Hash has to have at least one field, and values can be any string. For this example, we will use the Redis Hash data structure to store user session records.

1.1.1. User Records

Languages vary in the patterns they use to represent database records. Some languages map database records into classes, some map them into structs while others map them into dictionaries. For our application, we are going to use the class User to represent the user objects we are working with in our database.

Our user object has five properties: id, username, fname (given name), lname (last name or surname) and email address. It also has a function get_key() to generate the appropriate key for an object. For our database, we will use the convention that user records are stored under the key "user:{id}".

```
In [ ]: # With no arguments, we create an empty user
        empty = workshop.hashes.User()
        print "empty user: {}".format(empty)
        print "empty user key: {}".format(empty.get_key())

        # We can create a dictionary of properties which
        user_info = {
            'id': 147,
            'username': 'ruser',
            'fname': 'Redis',
            'lname': 'User',
            'email': 'ruser@somedomain.net' } ruser =
        workshop.hashes.User(**user_info) print "ruser:
        {}".format(ruser)
        print "ruser key: {}".format(ruser.get_key())
```

1.1.2. Adding Data to the Database

We can store our user objects in the database using the Redis commands for working with hashes. There are two primary commands used to add hash data to Redis: HSET and HMSET. The HSET command allows us to store a single field of a hash into the database.

In the code below, we create a version of our Redis User object to work with and store only the user's email in the database:

```
In [ ]: r_info = {
```

```
        'id': 147,
        'username': 'ruser',
        'fname': 'Redis',
        'lname': 'User',
        'email': 'ruser@somedomain.net' } r_user =
    workshop.hashes.User(**r_info)

    # Store only our user's email in the database
    res = r.hset(r_user.get_key(), 'email', r_user.email) print "Result:

    {}".format(res)
```

The result from the call to hset() will be 1 if a new hash key was created, or 0 if it already exists.

We can also add multiple fields to a Redis Hash in a single call to the database using the HMSET command. In the code below, we create a new version of our Redis User object and then create a new Hash in the Redis database by passing the key and a dictionary of all the fields in a database.

1.1.3. Getting Data from the Database

We can fetch our user records from Redis using three basic patterns: fetch individual fields, fetch multiple fields, and fetch all the fields of a hash. These patterns are supported by the HGET, HMGET, and HGETALL commands respectively. To set up the database for this section, please execute the code below:

```
In [ ]: # Set up example data
    r_info = {
        'id': 300,
        'username': 'ruser',
        'fname': 'Redis',
        'lname': 'User',
        'email': 'ruser@somedomain.net' } r_user =
    workshop.hashes.User(**r_info)
    r.hmset(r_user.get_key(), r_user.__dict__)

    # Initialize key with our example users key
    key = r_user.get_key()
```

In the first example, we can get just the email address of a user from the user key using the HGET call.

```
In [ ]: print "Email: {}".format(r.hget(key, 'email'))
```

The result from this call should be the email ("ruser@somedomain.net") of the sample user we created in the first part of this section.

In the next example, we get both the first name and the last name for our sample user based on its hash key. In this example, we pass the key and list of fields we are interested in to the HMGET call.

```
In [ ]: print r.hmget(key, ['fname', 'lname'])
```

The result from this call is an ordered list of values which matches the order of fields we passed as a list to the hmget() call. Finally, we can get all of the fields of our database record using the HGETALL command:

```
In [ ]: print r.hgetall(key)
```

The result of the hgetall() call is a dictionary mapping the hash fields to the field values.

1.1.4. Flexible Record Structures

Unlike some databases, Redis does not require schemas or predefined structures, so as our application evolves, our user records do not always have to contain the same fields.

In the example below, we create two users, one with a verified field and one without. We can easily store both of these users in the database:

```
In [2]: r1_info = {
            'id': 281,
            'username': 'ruser',
            'fname': 'Redis',
            'lname': 'User',
            'email': 'ruser@somedomain.net'
        } r1_user =
        workshop.hashes.User(**r1_info)

        r2_info = {
            'id': 282,
            'verified': True,
            'username': 'ruser',
            'fname': 'Redis',
            'lname': 'User',
            'email': 'ruser@somedomain.net'
        } r2_user =
        workshop.hashes.User(**r2_info)

        r.hmset(r1_user.get_key(), r1_user.__dict__)
        r.hmset(r2_user.get_key(), r2_user._dict )
```

Out[2]: True

Using HGETALL, we can see that Redis stores the structure of our records independently:

```
In [3]: print "r1: {}".format(r.hgetall(r1_user.get_key())) print
        "r2: {}".format(r.hgetall(r2_user.get_key()))

        print "Verified (r1): {}".format(r.hget(r1_user.get_key(), 'verified')) print
        "Verified (r2): {}".format(r.hget(r2_user.get_key(), 'verified'))
```

```
r1: {'username': 'ruser', 'lname': 'User', 'id': '281', 'fname': 'Redis', 'email': 'ruser@somedo
r2: {'username': 'ruser', 'verified': 'True', 'id': '282', 'lname': 'User', 'fname': 'Redis', '
Verified (r1):
None Verified
(r2): True
```

1.1.5. Working with Hash fields

Redis provides many commands for manipulating the fields of a hash in the database. To set up the database for these examples, please run the following code:

```
In [ ]: r_info = {
            'id': 400,
            'verified': True,
            'username': 'ruser',
```

```
        'fname': 'Redis',
        'lname':'User',
        'email': 'ruser@somedomain.net'
    } r_user =
    workshop.hashes.User(**r_info)
    key = r_user.get_key() r.hmset(key,
    r_user.__dict__)
```

In a development version of our application, we added the verified field to our user. After discussing the change, we decided not to apply it to our system. Our test data has records with the verified field that we want to clean up. We can use the HKEYS command to fetch the fields in our hashes, the HDEL command to remove the field if it is there and the HEXISTS command to verify its removal:

```
In [ ]: print "Initial keys: {}".format(r.hkeys(key))
        print "Delete 'verified': {}".format(r.hdel(key, 'verified')) print "'verified'
        exists: {}".format(r.hexists(key, 'verified'))
```

When we look at the initial state of our hash, we have a verified key. We run our code to restore our test data to the expected format and then verify that it does not exist anymore.

1.1.6. Counting with Hashes

Recall the discussion about counting votes. Just like Redis Strings, the values in a Redis Hash can represent integer and floating-point numbers. The HINCRBY command increments an integer by an integer argument, whereas HINCRBYFLOAT manipulates floating-point values.

1.1.7. Hashes as Key-value Stores

The Redis keyspace is itself just a big hash table, so the Hash data structure can be thought of as a miniature Redis. In the vote counting example, we used one Redis String key for each item's votes counter. Conversely, we can change that design to use a single Hash in which each field corresponds to an item and its value serves as the vote counter:

```
In [ ]: def upvote_item(r, item_id):
            "Upvotes an item and stores it in a Hash"

            return r.hincrby('item_vote_counters', item_id, 1)

        # Vote once for one item an twice for another
        upvote_item(r, 1)
        upvote_item(r, 2)
        upvote_item(r, 2)

        print r.hgetall('item_vote_counters')
```

This design aggregates the different counters inside a single data structure. It provides a logical way for managing related data together, much like using a Hash to store records (that are groups of related fields). However, in this case, the Hash is also somewhat like a table in a relational database —a structure that holds multiple records, each comprised of a primary key (the item ID) and a single value field (the counter).

1.1.8. Reducing RAM Overhead with Hashes

The major benefit derived from using Hashes as key-value stores is savings in RAM overhead.

Each individual key in Redis, regardless of its name and value, requires about 70 bytes (on 64-bit architectures) for administrative purposes. This overhead is negligible with small datasets but can become expensive as the volume increases. That is especially true with keys storing small values such as counters.

A Hash is a key like any other, so it also requires the same overhead. However, each field (and value) in the Hash can be stored more efficiently compared to the global keyspace. This form of storage requires more processing power to access, but for small Hashes the CPU penalty is minimal. There are two parameters that determine the threshold between small (memory-efficient but CPU-intensive) Hash encoding, a.k.a. ziplist, and the default encoding method:

```
In [ ]: print r.config_get('hash-max-ziplist*')
```

The value of hash-max-ziplist-entries is the maximal number of entries (fields and values) that the Hash can have using the ziplist encoding, and the default value means that Hashes with up to 256 fields are encoded as ziplists. hash-max-ziplist-value is the maximal length of an element (field or value) in a ziplist Hash. Going above the thresholds (but not vice versa) will trigger an automatic conversion from the ziplist encoding to the default one.

1.1.9. Considerations for Working with Large Hashes

Because Hashes can have up to 2^32-1 (4,294,967,295) fields, each potentially a 0.5GB String, they can become quite large. Using commands that operate on the entire Hash—namely HGETALL, HKEYS and HVALS—could become expensive due to the volume of data involved.

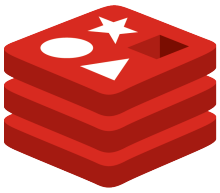
When the requirements make it necessary to fetch the entire contents of a big Hash, consider iterating the data structure instead of reading it entirely with one of the above-mentioned commands. Iterating a Hash, or scanning it, is done with the HSCAN command. Note, however, that HSCAN may return duplicates and is not guaranteed to return data added while iterating.

1.2 Review

In this chapter, we looked at working with the Redis Hash data type to store user records in the database. Redis provides a hash data type that is similar to the hash table and dictionary types provided in most modern programming languages.

We looked at various commands provided by Redis to manipulate hashes and learned how to store a structured object in the database and read it back from the server. We also saw how we could manipulate the members of a hash on the server to update and delete data directly.

This chapter has covered a wide range of different hash functionality, but not all of the commands are covered in this chapter. For additional information on Redis hashes as well as a complete command reference, see the Hash Commands page at Redis.io.



Counting Unique Page Views with Sets

05 - Counting Unique Page Views with Sets

1 Counting Unique Page Views with Sets

Page metrics are an important measurement of the health and vitality of a website. Page view metrics can not only measure customer interest in your site, they can also identify user interface and operational problems in real time. Commonly tracked site metrics include Page Views, Daily Active Users, Unique Views, and Monthly Active Users. Redis can be a vital part of your analytics-processing pipeline, computing aggregate statistics.

In this chapter, we are going to look at how the Redis Set data type is used to calculate and store unique viewers. Among the topics we will cover in this chapter are:

- Sets in Redis and Python
- Specifying data storage conventions
- Using structured keys for data
- Adding views
- Scanning keys
- Secondary indexing

1. Sets

1.1. Redis Sets

Redis provides two types of set data structures, the **Set** and the **Sorted Set**. Either set type would work to calculate unique viewers, but in this chapter we are going to look at examples using the set data type.

Sets in Redis are very similar to the set data structure found in many programming languages or mathematics. Redis sets are an unordered collection of unique elements. Elements in Redis are represented as strings and a byte-wise equality test is used to determine if two elements are equal. Unlike lists, the sets (1, 2, 3), (2, 3, 1), and (3, 1, 2) are identical, as they contain the exact same elements.

There are fifteen different commands to manipulate set data in the current release of Redis (3.2). These commands provide a variety of different operations including:

- Adding elements to and removing elements from a set
- Testing for elements in the set
- Retrieving various elements of a set
- Comparing and combining multiple sets

Redis' set commands generally fall into one of two categories: those that operate on a single set and those that operate on multiple sets. The commands that operate on an individual set normally take a key and a series of one or more elements as parameters. The commands that operate on multiple sets normally take a series of keys as parameters and either return a set as a result or write the result into a destination set.

1.2. Sets in Python

The Python language provides a built-in set data type that is used by Redis client libraries—including **redis-py,** the client we are using for our example code—to represent sets returned from a Redis command.

Sets were a later introduction to the Python language. Sets are constructed in Python using the set([iterable]) constructor. Newer versions of Python, including the one we are using for this Notebook, provide the syntactical shortcut {item1, item2,...}. If you are unfamiliar with Python sets, please see the Python documentation on standard Python types for more information.

2. Page Views

Raw page views are an important metric for websites, but there are other important metrics based on individual users that require additional resources to compute. Unique Daily Page Views, Daily Active Users (DAUs) and Monthly Active Users (MAUs) are all common metrics for measuring the performance of a website.

In this chapter, we are going to look at how Redis can be used as part of an event based analytics pipeline. In our example, Redis will be used to store information about users, allowing us to compute aggregated statistics for unique users on a daily, weekly and monthly basis.

Our analytics system reads processed data from an event log that could be implemented by a wide range of software, including Redis. Our log system provides a stream of interesting events—logins, page views, downloads—that our analytics pipeline can process into interesting metrics and data sets.

Our examples will look at how to process a stream of events into Redis to generate sets of unique users. Then once we have that data loaded into Redis, we show examples of how Redis can be used to compute additional metrics and aggregations from that data.

1.2.1 Data Storage Conventions

Our pipeline will store unique pages viewers in a set associated with both the page and the date. Our system provides us with a unique integer page ID for each page in the system, so we can use that ID plus the date to construct a key of the form page:{page_id}:unique:{year}:{month}:{day} to reference our unique user set. The members of the set will be the integer user ID associated with each database user.

Note

The standard date and time handling libraries in programming languages provide facilities for getting the day, month and year components from a timestamp. In Python the time.gmtime or time.localtime functions provide those conversions. Because we are doing log processing, we will use the gmtime function in our examples. If you are unfamiliar with the time handling functions in Python, please see the time package documentation for more details.

---

3. Recording a Page View

In our first example, we are going to read events from our event stream and record the page view events in Redis. Using the Redis **Set** data structure, our stream processing function will generate sets of unique users.

As our code processes events in the stream, it will:

- Parse the event
- Compute a key based on the event data

- Store the viewer in the viewers set

The Redis SADD (**S**et **ADD**) command adds the specified member to a set, creating the set if it doesn't already exist. Because a member can only exist in the set once, we can add the user to the set every time we record a page view for the user without worrying about duplicates. This gives us our unique user data.

Try running the sample code below by selecting the code cell and pressing SHIFT + ENTER

```
In [ ]: def record_user_page_view(r, pid, year, month, day, uid): "Records a page view in
            Redis to generate unique viewers"

            key = workshop.sets.daily_page_view_key(pid, year, month, day) return
            r.sadd(key, uid)


        def process_page_view_events(r, events):
            """Reads a list of events for the form: (event_id, year, month,
            day, user_id, page_id) and processes them into daily unique
            views
            """

            cnt = 0
            for (eid, year, month, day, uid, pid) in events: workshop.sets.log_page_view_event(eid, pid,
                uid, year, month, day) record_user_page_view(r, pid, year, month, day, uid)
                cnt += 1

            print "Events processed: {}".format(cnt) process_page_view_events(r,

        workshop.sets.sample_page_view_events)
```

When you execute the sample code, you should see a sequence of 20 log-like messages that represent processed page view events from our simulated stream of users, pages, and dates. Each of these log messages correspond to one page view that was recorded in the database.

We can use a utility function created for our Notebook environment, show_database_state, to see the state of our database after we finish processing the stream of data. Execute the code below to see the state of the database:

```
In [ ]: workshop.show_database_state(r)
```

Looking at the output of the database state, you can see how our code has constructed several sets of unique viewers for specific pages and dates. Using the simulated stream provided, you should see three unique users (3001, 3002 and 3003) who viewed page 201 on March 4, 2017.

4. Counting Unique Page Views

Now that our event stream has been processed, we can use the results in Redis to compute a variety of additional metrics. The first example we will look at is how we can compute the daily unique viewers for the site.

With the Redis SCARD (**S**et **CARD**inality) command we can get the cardinality, or size, of a set from the server. The set size is our unique viewer count. In the sample code below, we implement a function to return unique viewers using the SCARD command:

```
In [ ]: def get_unique_views(r, pid, year, month, day):
            """Returns the number of unique views for a page (indexed by id and day of year)"""
```

```
    key = workshop.sets.daily_page_view_key(pid, year, month, day)
    return r.scard(key)


# Fetch page views for March 4, 2017
print "Page 201 Unique views for March 4 2017: ", get_unique_views(r, 201, 2017, 3, 4)
```

At this point we're presented with a problem: finding our data. In the sample code, we already knew that there was data for page 201 on March 4, 2017. For real data, we won't be looking for a known sample in time, so how can we find our data?

One solution might be to create a loop in Python and iterate over all of the combinations of dates and page IDs, but that becomes prohibitively expensive and given that most pages will likely have no views at all, this is not an efficient solution to our problem.

Redis, like most databases, provides multiple ways to query your data. The first technique we are going to look at is scanning for the keys.

1. Scanning for Keys

Redis provides the SCAN command to iterate over the keys in the database matching a particular pattern. Redis supports glob-style pattern-matching in the SCAN command. Our example code shows how to iterate over keys that match a particular string pattern:

```
In [ ]: def scan_keys(r, pattern):
            """Returns a list of all the keys matching a given pattern"""

            result = []
            cur, keys = r.scan(cursor=0, match=pattern, count=2)
            result.extend(keys)
            while cur != 0:
                cur, keys = r.scan(cursor=cur, match=pattern, count=2) result.extend(keys)

            return result


    pprint.pprint(scan_keys(r, 'page:*:unique:*'))
```

Tip

Our sample code will output a list of matching keys from the database. For simplicity our function outputs all of the matching keys at once. This works well for our small sample database, but with a large data set, the code would have to be refactored to process the data iteratively.

We can use our scan_keys function in conjunction with our unique_views function to generate a simple report on the number of unique views for each day we have data.

```
In [ ]: def report_unique_page_views(r):
            "Implements a basic report of unique page views for the data in Redis"
```

```
    keys = scan_keys(r, 'page:*:unique:*') keys.sort()

    for key in keys:
        pid, year, month, day = workshop.sets.convert_key_to_components(key)

        views = get_unique_views(r, pid, year, month, day)
        workshop.sets.log_page_view(pid, year, month, day, views)


report_unique_page_views(r)
```

The output from this code should be a very simple report of the days included in our simulated stream and a count of the unique views for that particular day.

Optional

Try adding the show_database_state utility function to the above code and verify that the results of our report function match the state of the database.

---

2. Maintaining a Secondary Index

Using the SCAN command is one way we can find our data; another way is to maintain a ***secondary index*** using Redis data structures to find our data. Maintaining a secondary index requires additional work on our part, but is also an effective way to speed up access to our data.

Implementing secondary indexing is easy and we can use our Redis set data type to implement our index. Secondary indexing can be added, just by modifying our record_user_page_view code to maintain the index at insert time. An updated version of record_user_page_view with secondary index management is shown below. You can reload the sample data by running the code below:

```
In [2]: # Uncomment and run this line if you want to see an example
        print inspect.getsource(workshop.sets.hint_idx_record_user_page_view)


        def record_user_page_view(r, pid, year, month, day, uid): "Records a page view
            in Redis to generate unique viewers"

            # Modify this function to create a secondary index

            key = workshop.sets.daily_page_view_key(pid, year, month, day) return
            r.sadd(key, uid)


        process_page_view_events(r, workshop.sets.sample_page_view_events)
        workshop.show_database_state(r)

def hint_idx_record_user_page_view(r, pid, year, month, day, uid): "Records a page
    view in Redis to generate unique viewers"

    # Modify this function to create a secondary index idx_name =
    workshop.sets.secondary_page_index()
```

```
r.sadd(idx_name, workshop.sets.daily_page_view_key(pid, year, month,  day))

key = workshop.sets.daily_page_view_key(pid, year, month,  day) return
r.sadd(key,   uid)
```

Unless our reporting code takes advantage of the secondary index, building it is a wasted workload on our database. Review the report_unique_page_views function below and try to update it to use the new secondary index stored at index:unique-page instead of doing a key scan.

```
In [ ]: # Uncomment and run this line if you want to see an example
        # print inspect.getsource(workshop.sets.hint_idx_report_unique_page_views)

        def report_unique_page_views(r):
            "Implements a basic report of unique page views for the data in Redis"

            keys = scan_keys(r, 'page:*:unique:*') keys.sort()

            for key in keys:
                pid, year, month, day = workshop.sets.convert_key_to_components(key)

                views = get_unique_views(r, pid, year, month, day)
                workshop.sets.log_page_view(pid, year, month, day, views)


        report_unique_page_views(r)
```

Tip

In the Redis 4.0 release there is a [secondary indexing module] (https://github.com/RedisLabsModules/secondary) built by Redis Labs.

Our example of secondary indexing makes a classic tradeoff: it increases the amount of work when data is inserted to reduce the amount of work required to access our data. This may or may not be the right choice for your work loads, the best way to determine that is through profiling your work load.

3. Monthly Active Users

Our unique viewer storage scheme is flexible enough that we can compute Monthly Active Users (MAUs) quickly from the processed data we already loaded into Redis. The Monthly Active User metric tracks the number of unique visitors to a site aggregated over the entire month.

Using Redis' set manipulation commands, we can take the daily page data and generate statistics for the monthly viewers. Usually MAUs are tracked at the site level, so we will aggregate our statistics for the entire site and not individual pages.

Building up a set of monthly users from our existing data can be accomplished with a simple procedure:

- Use our page index to find Daily Views
- Iteratively build a set of unique monthly viewers
- Count the monthly viewers

This can be accomplished using three Redis commands, the SMEMBERS (**S**et **MEMBERS**) command, the SUNIONSTORE (**S*ET **UNION STORE**) command and the SCARD command that we learned earlier.

The SUNIONSTORE command unions several sets and stores the results. Its parameters are the destination key to store the result and the sequence of one or more sets that are unioned together into the final result. The union operation here refers to the familiar set union operation. If our database has three sets: s1 = (a, b, c), s2 = (a, b, f), and s3 = (b, c, f), the result of calling SUNIONSTORE s4 s1 s2 s3 would be to store the set (a, b, c, f) in s4.

The SMEMBERS command simply returns all of the members of the set provided as a parameter. The result will be returned as a Python set. Remember that because we are working with sets, we cannot depend on the order of the returned results.

In the following sample, we show code to compute our monthly user set and store the results in Redis:

```
In [ ]: def compute_monthly_users(r, year, month):
        "Computes the set of Active Monthly Users and stores in Redis"

        keys = workshop.sets.get_keys_from_secondary_index(r) for key in
        keys:
            k_pid, k_year, k_month, k_day = workshop.sets.convert_key_to_components(key)

            if k_year == year and k_month == month:
                mau_key = workshop.sets.mau_key(k_year, k_month)
                day_key = workshop.sets.daily_page_view_key(k_pid, k_year, k_month, k_day)
                r.sunionstore(mau_key, mau_key, day_key)

    def get_mau_count(r, year, month):
        "Returns the count of Active Monthly Users"
        return r.scard(workshop.sets.mau_key(year, month)) compute_monthly_users(r,

    2017, 3)

    print "Monthly Active Users (MAU) for March 2017: {}".format(get_mau_count(r, 2017, 3))
```

In our sample code, we relied on the fact that Redis uses sensible defaults instead of returning errors whenever possible. We iteratively build our final monthly user set by applying the union operator to our results thus far and the current result we are processing, but we don't have to "special case" the first iteration in the loop, because Redis treats undefined sets like the empty set, so the union proceeds without error.

There are many other commands in Redis that provide the familiar operations on sets. In Redis, there are two variants of each command, one that returns the result and one that stores the result. The set commands provided in Redis are:

Operation	Results Returned	Results Stored
Union	SUNION	SUNIONSTORE
Intersection	SINTER	SINTERSTORE
Difference	SDIFF	SDIFFSTORE

More details about the set operation commands can be found in the documentation page on Redis.io

5. Absent Users

Keeping users engaged with your website is critical, but sometimes despite your best efforts users may stop using your site. Often, you want to send an email or reach out to the users to understand why they stopped visiting and encourage them to come back. We can build this functionality on top of our processed data using additional features from Redis.

For this example, assume that there is already a processed set stored in the key site:users that provides a set of all the user IDs for currently registered accounts. Using Redis, we want to determine all the users that haven't visited our site in the last month, so that we can send them an email inviting them to come back.

Redis also provides the command SISMEMBER (**S**et **IS MEMBER**) to test for membership in a particular set. We can use SISMEMBER in conjunction with our processed data and our user ID set to find missing users.

The SISMEMBER function takes as parameters a set key and a member, and determines if that element is a member of the specified set. It returns a zero or one depending on if the element is or is not a member.

To find absent users we need a function that will:

- Retrieve our registered user set
- Check each member to see if they visited the site in the current month
- Store the absent users in a result set (for other systems to use)

Our sample code below shows the Redis commands to implement this:

```
In [ ]: # Create our sample data
        workshop.sets.create_all_users_set(r)

        def generate_absent_users(r, year, month):
            "Computes the absent users for a given year and month and stores result in Redis"

            mau_key = workshop.sets.mau_key(year, month)
            absent_users_key = workshop.sets.absent_user_key(year, month)

            all_users_key = workshop.sets.all_user_key() users =
            r.smembers(all_users_key)
            for user in users:
                if not r.sismember(mau_key, user):
                    r.sadd(absent_users_key, user)
```

```
def get_absent_users(r, year, month):
    "Returns the absent users for a given year and month"

    absent_users_key = workshop.sets.absent_user_key(year, month) return
    r.smembers(absent_users_key)

generate_absent_users(r, 2017, 3)
pprint.pprint(get_absent_users(r, 2017, 3))
```

Many of you may have balked slightly at our naive implementation of this function, since Redis provides more efficient operations (which we have already talked about) with which to compute this same result.

In the cell below, try and reimplement the generate_absent_users function from the same set of data, but using more efficient Redis commands.

```
In [ ]: # Your new code goes here
def generate_absent_users(r, year, month):

    pass
```

6. Scaling

Many of the samples shown here today would work well for small websites, but as the size of your dataset grows you will have to reconsider some of the techniques presented.

One of the first things to consider is how you fetch items from Redis. Most of the implementations in our sample code read entire sets of keys into memory at once; this is fine when the sets are small, but as the sets get larger you will need to look into progressively fetching the results of a query. You may need to refactor your client application to use an iterator or generator pattern to operate on a subset of results at a time.

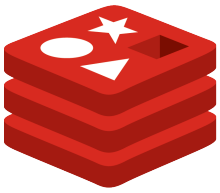
You may also need to reconsider how you store the data in Redis. The members of a set are stored as string and equality is determined using a byte-wise string compare. There are other, more compact ways of representing this data in Redis. One way of reducing the amount of storage required for larger datasets is to use Redis' bitmap operators that treat strings as a vector of bits. For more information on bitmap operators see the String documentation at Redis.io.

7. Review

This chapter looked at ways we can use Redis sets to process analytics data. We first looked at how to process a stream of events to generate a list of Daily Unique Users. Then we learned how to extend our application to count monthly unique users. Finally, we discussed how to identify absent users. In the process we learned to use a variety of Redis commands including:

- SADD
- SCARD
- SCAN
- SUNIONSTORE
- SMEMBERS
- SISMEMBERS

We saw how Redis provides commands that modify the membership of sets as well as commands that operate on multiple sets to return or store a result. The details of all the commands Redis provides to work with sets can be found in the Set Documentation onRedis.io.



Serving User Recommendations with Sorted Sets

06 - Managing Recommendations with Sorted Sets

1 Serving User Recommendations with Sorted Sets

Personalization is an important part of increasing user traffic and growing a successful website. Various algorithms exist to predict what content a user might be interested in given their browsing history. Many third party systems exist to compute recommendations, but recommendations aren't useful if they can't be served to users quickly.

In this chapter, we are going to look at how the Redis **Sorted Set** set data type can be used to serve up blog recommendations. Among the topics we will cover in this chapter are:

- Sorted Sets versus Sets
- Specifying data storage conventions
- Storing recommendations
- Viewing recommendations

1. Sorted Sets

Redis provides two types of set data structures: the Set and the Sorted Set. In the previous chapter we looked at using the Set type to calculate Unique Views, and in this chapter we are going to look at using the other set type, Sorted Sets, to serve up user recommendations.

Sorted Sets in Redis are similar to the basic set data structure, except that every element in the set is associated with a floating-point value, called the "score." Like Redis Sets, Sorted Sets consist of a collection of unique elements but the set is ordered according to two rules:

- Elements with different scores are ordered by score
- Elements with same scores are ordered by string comparison on the element

Using these two rules, the set is maintained in ascending order with the smallest element at position zero. As of the 3.2 release, Redis provides 21 different commands to operate on sorted sets. These commands allow the user to perform a variety of operations:

- Add and remove elements from the set
- Retrieve elements of the set based on score or element name
- Update element scores
- Combine multiple sets
- Determine the size of sets

You may also hear Redis sorted sets referred to as "zsets."

2. Blog Post Recommendations

In this chapter, we are going to look at building a system for quickly serving blog post recommendations to users of our website. In our system, recommendations will be generated from a variety of data sources that are processed offline to generate a list of recommendations that will then be loaded into Redis for serving to visitors.

The recommendation engine will deliver a new set of recommendations for users all at once. The

recommendations data will be structured as a user_id followed by a list of (page_id, score) tuples. A variable number of recommendations is generated for each user.

3. Data Storage Conventions

Our recommended blog posts will be stored in the database using the sorted set data type. The recommended blog posts for a particular user will be stored under the key user:{user_id}:recommendations. The recommendation data will consist of the page_id of the blog post and a score between -1.0 to 1.0, with 1.0 being highly recommended and a -1.0 being not recommended.

To improve the clarity of our examples, the Notebook environment includes several utility functions. In this chapter, the primary utility functions we will use are:

- user_recommendation_key - generates the appropriate key for the user recommendations
- clear_user_recommendations - removes the recommendations for the given user
- show_database_state - textual representation of the database state

4. Storing Recommendations

The first step in building our example recommendation system is building a system to load the data from the recommendation backend into Redis. As our code processes data from the backend recommendation system, it needs to:

- Parse the recommendation data
- Compute a key from the data
- Remove the key if it already exists
- Store the recommendations in Redis

The Redis ZADD command adds the specified member plus score to the sorted set, creating the sorted set if it doesn't exits. In our sample system, an entire batch of recommendations is loaded at once, so we can't just use the ZADD command, because then we would never clear the old recommendations.

Run our example by selecting the cell and pressing SHIFT + ENTER

```
In [1]: def update_user_recommendations(r, user_id, recommendations):
            "Update or create a set of recommendations for the user"

            key = workshop.zsets.user_recommendation_key(user_id)

            members = {}
            for page_id, score in recommendations:
                page_id = str(page_id)
                members[page_id] = score

            r.zadd(key,
            **members) return
            len(members)

        def parse_recommendations(r, recommendation_list):
            "Parses a list of recommendations and updates for a given user"

            for user_id, recommendations in recommendation_list:
                update_user_recommendations(r, user_id, recommendations)

        workshop.zsets.clear_user_recommendations(r, 3001)
        parse_recommendations(r, workshop.zsets.sample_user_recommendations)
```

```
print "Recommendations loaded for user 3001: {}".format(workshop.zsets.get_user_recommen
```

```
Recommendations loaded for user 3001: [('201', -1.0), ('202', -0.8), ('203', 0.4), ('206', 0.7),
```

When you execute the sample code, you should see a message indicating that 4 recommendations were loaded for user 3001. We can use our show_database_state utility function to see the state of our database after loading the recommendations.

```
In [2]:

workshop.show_database_state(r)
```

One thing to note about our code to load recommendations in Redis, is that our code must iterate over the recommendation list and "reverse" the individual score data. Elements in a Redis Sorted Set are specified in score member order. To store a recommendation for page 201 with a score .7 in Redis, you would execute the command:

```
In [3]: r.zadd('sample_recommendation', .7, 201)

Out[3]: 1
```

5. Retrieving Recommendations

The next step in building our sample recommendation system is to serve up recommendations to visitors. We are going to look at two different ways of providing recommendations to the users: top N and above a threshold.

1. Top N Retrieval

The first example method we are going to look at returns an arbitrary number of the best recommendations available for a user. To retrieve recommendations in this fashion, our code needs to:

- Compute a key for the requested user
- Retrieve the top ranked items

This can be accomplished with a single Redis command: ZREVRANGE. The ZREVRANGE command returns a range of a sorted set, treating the set as if it was ordered from highest to lowest. So if we use ZREVRANGE to request the range 0 to N, we get our *top* N elements.

Run the sample code below using SHIFT + ENTER to see how ZREVRANGE works.

```
In [4]: def top_recommendations(r, user_id, N):
            "Returns the top N recommendations for the user"

            key = workshop.zsets.user_recommendation_key(user_id) return
            r.zrevrange(key, 0, N - 1)

        print "Top 2 recommendations for user 3001: ", top_recommendations(r, 3001, 2)
```

```
Top 2 recommendations for user 3001:  ['204', '205']
```

After the code runs, you should see a message displaying the top two recommendations for

user 3001. Like many other Redis commands, ZREVRANGE gracefully handles missing data instead of throwing an error. If the requested key does not exist, an empty result is returned and if the range requested is larger than the number of members available, the smaller range is returned.

In this code example, we only return the members of the sorted set in reverse sorted order. If we wanted to get both the members and the scores, we could add the WITHSCORES modifier to the command to return both. In the *redis-py* library, this is accomplished by passing the withscores=True argument to the function call.

Note

Most of the examples in this exercise work with reverse ranges, because we are interested in the *top* recommendations for a user. Redis provides "forward" versions of all the reverse (REV) commands that we are using. The forward versions treat the sorted set as ordered in the ascending direction.

---

2. Threshold Recommendations

The purpose of our recommendation system is to increase traffic to our website by pointing out blog posts that will be particularly engaging for the user. Displaying the top 2, 5 or N recommendations works for many users, but it doesn't guarantee quality recommendations for all our users.

Our recommendation system works by giving posts a score from -1 to 1, indicating how likely a user is to like the particular post, so there may be no strongly recommended items for a given user. In that case, it might be better not to provide any recommendations at all rather than a low scoring post. We can implement this version of returning recommendations using code that:

- Computes a key for the requested user
- Returns all keys above the requested threshold

This can also be accomplished with a single Redis command: ZREVRANGEBYSCORE. The ZREVRANGEBYSCORE command is very similar to the ZREVRANGE command in that it returns a list of members in reverse order. The difference between the two commands is that ZREVRANGE takes a range of elements to determine the results while ZREVRANGEBYSCORE takes a range of scores.

In the next example, we return only return strong recommendations that are above a provided threshold. *Run the example below by selecting the code cell and pressing SHIFT + ENTER*

```
In [5]: def top_recommendations(r, user_id, threshold):
            "Returns the top recommendations for user based on a minimum score"

            key = workshop.zsets.user_recommendation_key(user_id)
            return r.zrevrangebyscore(key, 1.0, threshold, withscores=True)

        print "Top recommendations for user 3001: ", top_recommendations(r, 3001, .7)
Top recommendations for user 3001:  [('204', 0.9), ('205', 0.8), ('206', 0.7)]
```

You should see a list of the top recommendations (along with their scores) for user 3001. Our

threshold-based code returns all of the recommendations within a given range, but for some users that could be a substantial number of recommendations. It would be nice to continue to limit our results to a certain number of strongly recommended items. This can be accomplished by adding the LIMIT argument to our command. The LIMIT argument takes an offset and a count and applies it to our range.

The sample code below combines both the top-N and threshold approaches, to return the top-N results above a certain threshold for a user.

Run the sample code by selecting the code cell and pressing SHIFT+ENTER

```
In [6]: def top_recommendations(r, user_id, threshold, N=2):
            "Returns the top N (default=2) recommendations above the giventhreshold"

            key = workshop.zsets.user_recommendation_key(user_id)
            return r.zrevrangebyscore(key, 1.0, threshold, withscores=True, start=0, num=N) print "Top

        recommendations for user 3001: ", top_recommendations(r, 3001, .7)

Top recommendations for user 3001:          [('204', 0.9), ('205', 0.8)]
```

After executing the sample code, you should see two recommendations for user 3001 that are both above the 0.7 score threshold.

Note

The scores passed to the ZREVRANGEBYSCORE command can be specified as intervals. You can use "(" to indicate an open or non-inclusive interval and "]" to indicate a closed or include interval. Additionally, the arguments+inf and -inf can be used to indicate positive and negative infinity, which allows you to select all of the possible scores without knowing the range beforehand.

---

6. Retiring Recommendations

Once a user has visited a particular recommendation, it is much less valuable as a future recommendation. Our recommendation system is supposed to increase the number of posts a user reads, so recommending them the same posts over and over generally won't increase our traffic. There are different ways we could handle this; we could remove a recommendation once a user visits it or we could reduce its score, making it less likely to be served to the user.

6.1. Removing Recommendations

The first way we can handle "used" recommendations is to remove the recommendation from the sorted set. This can be accomplished with a single Redis command: ZREM. The ZREM command removes a sequence of one or more members from a sorted set. The following example uses ZREM to remove a recommendation once it is used:

```
In [ ]: def use_recommendation(r, user_id, page_id):
            "Removes a recommendation for a user after they visit the recommendation"
```

```
        key = workshop.zsets.user_recommendation_key(user_id)
        r.zrem(key,    page_id)


    parse_recommendations(r, workshop.zsets.sample_user_recommendations)

    print "Top recommendations for user 3001: ", top_recommendations(r, 3001, .5, N=4)
    use_recommendation(r, 3001, 206)
    print "Top recommendations for user 3001: ", top_recommendations(r, 3001, .5, N=4)
```

After running the above example, you should see the recommendations for user 3001 before and after removing page 208.

6.2. Reducing Scores

Reducing the score of a recommendation once it's been visited is another way we can handle a "used" recommendation. This will mean that the strong recommendation is still in our set to consider, but we give more weight to highly recommended items that haven't been viewed. This can also be accomplished with two commands: ZINCRBY and ZREMRANGEBYSCORE. The ZINCRBY command takes a key, an increment and a member as parameters and updates the member's score by the specified increment. The ZREMRANGEBYSCORE command is used to remove the recommendation if its score is less than -1. The ZREMRANGEBYSCORE command takes a range of scores and removes all of the members in that range.

In our example below, we reduce the score of a post by .1 after it has been read, then remove any post that has a score less than -1 to preserve our -1 to 1 range of scores:

```
In [ ]: def use_recommendation(r, user_id, page_id):
            "Reduces the score of a recommendation after the specified user visits"

            key = workshop.zsets.user_recommendation_key(user_id)
            r.zincrby(key,   -0.2,   page_id)

            r.zremrangebyscore(key,  "-inf", -1)

        parse_recommendations(r, workshop.zsets.sample_user_recommendations)

        print "Top recommendations for user 3001: ", workshop.zsets.get_user_recommendations(r,
        use_recommendation(r, 3001, 208)
        print "Top recommendations for user 3001: ", workshop.zsets.get_user_recommendations(r,
```

After running the above example, you should see the recommendations for user 3001 before and after reducing the score for page 208.

7. Working with Sets

In the examples we've looked at so far, nothing we've done has really distinguished sorted sets from an ordered list, but sorted sets are called "sets" for a reason. Redis supports set operations (operations on multiple sets) on Sorted Sets as well. Redis provides two set operations on Sorted Sets:

- ZINTERSTORE - intersection and store
- ZUNIONSTORE - union and store

These operations perform the familiar set intersection or union operations and store the results in a destination key. Arguments to the commands allow you to control how the scores from the two sets will be combined. Redis allows you to control how the scores for members are combined. Scores can be combined using weights, aggregation (sum), or min/max. See the ZUNIONSTORE command documentation for detailed information on combining scores.

2. Review

In this chapter we looked at ways we can use Redis sorted sets to serve up recommendation data to users. We first looked at loading recommendation data into Redis from an external system, and then looked at two different ways of retrieving recommendations from Redis (top-N and thresh- old). Finally, we reviewed two ways (deletion and score reduction) to retire used recommendation. Since most of our examples involved member manipulation, we took a bit of time to look at the set operators that can be applied to Sorted Sets.

In the process, we learned how to use a variety of Redis commands:

- ZADD
- ZRANGE
- ZREVRANGE
- ZRANGEBYSCORE
- ZREVRANGEBYSCORE
- ZINCRBY
- ZREMRANGEBYSCORE
- ZINTERSTORE
- ZUNIONSTORE

Redis provides a wide range of commands that modify Sorted Sets. We've looked at several of the commands in this chapter, but the details of all the commands available can be found at Sorted Set Documentation on Redis.io.