José Alberto Benítez Andrades
17th May 2021

Universidad de León
jbena@unileon.es - @jabenitez88

REDIS – Case study

1 INITIAL CONSIDERATIONS:

- It is recommended to use the Linux Operating System, specifically the Ubuntu distribution.

- The Python 3 programming language will be used, specifically the API provided to work with Redis (redis-py).

- Develop the python scripts using Jupyter notebooks, in order to make the delivery easier, as well as the developed code and any comments that the student wants to add.

- Read carefully the indications of the practice statement regarding formatting, data dump order, etc., so that there are no errors in its implementation.

2 PROBLEM TO SOLVE WITH REDIS

In this practical work we are going to design and implement a key-value database using Redis for the management of a microblogging service similar to Twitter. The objective is to become familiar with the use of key-value databases such as Redis and to verify that its structures allow us to build effective data models to implement many types of applications. Our Twitter "clone" will be structurally simple, will work efficiently, and could be distributed among a large number of servers with little effort (although the latter is outside the scope of this practical work). ´

As mentioned above, we will use the Redis management library provided by Python, called redis-py, thanks to the great functionality it offers for our purposes. The first step in the practice will consist of setting up the necessary environment for the use of a Redis database, whose server operates on port 6379. For this purpose, a file "docker-compose.yml " is provided to build the environment. Docker-compose needs to be installed with the command "sudo apt-get install docker-compose". Once this is done, run the command "sudo docker-compose up -d" inside the folder where you have copied the "docker-compose.yml" file. This command will download and install Redis from the dockerhub repository, opening port 6379 to make connections and queries against the database. You should download it from https://github.com/jabenitez88/uw-redis-seminar .

Once we have our Redis database up using docker-compose, it will be necessary to install the redis-py Python package. To do this, we will run "pip install redis" in the console or in the Jupyter notebook used to develop the practice.

Subsequently, the connection to the database is achieved using the following code:

```
import redis

redis_db=redis.Redis(host='127.0.0.1',port=6379, password='')
```

As we can see, we will connect to our own machine (localhost) using the default port 6379. In our case, we will not need to include any password to make this connection.

3 IMPLEMENTATION

The development of this practical work is based on three different parts. In the first part, the design of the database will be implemented by means of a series of Python functions that allow the introduction of new data. In the second part, the previously designed database will be populated and finally, in the third part, functions will be defined to carry out tests to check its correct operation.

3.1 DATABASE DESIGN

The functions to implement to build our key-value database in Redis that mimics Twitter are the following:

- new_user: This function will receive the name of the new user and generate a new entry in the database using an incremental identifier (provided by Redis) for each user. This incremental identifier can be used to differentiate each of the keys containing users, so that these keys are similar to the following: "user:[id]", where "id" is the identifier of the user. For each user, we must store the user's name. In the same way, it is recommended to store all the users and their identifiers within the same unique data structure whose key can be "users".
- new_follower: One of the basic functionalities of Twitter is the possibility of following existing users. To implement this functionality we will create the function "new_follower", which will receive the name of a user and the name of the user that will become a "follower" of the first one, as well as a "timestamp". It is recommended to use key names similar to those used for the users, in the form "followers:[id]", where "id" will be the identifier for the user. This data structure will contain, for each user, a set of tuples. Each tuple will store the identifier of a "follower" and the "timestamp" that represents the moment in which he/she started to follow him/her.
- new_following: By means of this function, contrary to the previous function, we will store the users followed by a specific user. It also receives an original user, the user to follow and a "timestamp". The data structure used will be similar, and its key can have the form "following:[id]", where "id" is the identifier of the original user. For each user, the identifiers of the users they are following and the time at which they started to be followed shall also be stored. - follow: The "follow" function shall receive an original user, a user to follow, and a timestamp and make use of the two auxiliary functions above to update the information relating to the original user's follower and followings. The previous functions (new_follower and new_following) can only be called from this function, never directly.
- new_post: This function will allow us to include new posts in our database. It will receive as parameters the user who creates the message, the body of the message and a "timestamp" representing the time of creation of the message. For each message a key will be created whose name follows the structure already used: "post:[idPost]", where "idPost" will be an incremental counter different from the one used for the users. This structure will contain, for each post, the identifier of the user who created it, the time of its creation and the body of the message. In addition, for a better management of the posts, another structure "posts:[id]" will be created, where "id" will be a user identifier. In this structure we will store the list of "idPost" of all the posts belonging to this user. It is important that this list not only stores the posts created by the user, but also the

ones created by your "followings" (users you follow), for better management of access to posts later on.

Both for these functions and for those to be defined in the "Tests" section, the database created with the function shown in section 2 can be added as an additional parameter of the function, or a single global variable can be used within the functions for the calls to it. It is requested to clearly document which of the two options is used.

3.2 DATA SET

Once the functions that allow us to implement the database itself have been defined, we will populate our database with a database itself, we will populate our database with a previously defined set of data. previously defined dataset. This dataset is provided to the provided to the student by means of two CSV (Comma Separated Values) files: "twitter_sample.csv" and "twitter_sample.csv". Values): "twitter_sample.csv" and "relations.csv":

- The file "twitter_sample.csv" contains a total of 111 tweets for each of which the name of the user who created it (User column), the time of creation of the tweet (Post_Time column) and the body of the message (Tweet_Content column) are presented. For this work we consider that users exist all at once from the first moment, so the first step required to populate the database is the creation of all users, through the function "new_user". That is, it does not store an account creation time, although it does store the time at which one user follows another (important for implementing the query functions).
- The "relations.csv" file contains a total of 22 relations between users, each of which contains the original user (User column), the user being followed (Follows column) and the time at which the user is followed (Following_Time column).

We are asked to use these two files to, after formatting them according to our interests, populate our database by initially entering all the users using the "new_user" function. Next, the relationships between them using the "follow" function, and finally all tweets posted using the "new_post" function. It is important to note that the dates and times in the CSV files are not in "timestamp" (float) format, but are dates in text format with a "dd mmm yyyy hh:mm:ss" structure, i.e. two digits for the day, three letters for the month and 4 digits for the year, separated by spaces, and then (after an additional space), two digits for the hour, two for the minutes and two for the seconds, separated by the ":" character. It will be necessary to convert this format to a format readable by some python library dedicated to time processing (for example "datetime") and then convert the complete date into a "timestamp".

3.3 VALIDATION

Once the database has been created and populated, it is time to perform operations on it. The implementation of the following 3 functions is requested:

- get_followers: This function will receive a username and return or print a list of all the names of the users who follow you, and when they started following you (in date format, not timestamp), sorted in time.
- get_followings: This function will receive a username and return or print a list of all the names of the users you follow, and when you started following them (in date format, not timestamp), sorted in time.

- get_timeline: This function will receive a username and a boolean parameter named "own_tweets". It will return or print to the screen all tweets corresponding to that user (your own or published by the users you follow), sorted by date of publication of the tweet. The "SORT" function provided by Redis within this function must be used to generate the tweet sorting. In addition, only those tweets from users followed by the original user that are after the time when the original user started following them will be displayed. The boolean parameter " own_tweets " will allow us to indicate whether we want to show both our own tweets and those of the followed users (own_tweets = True), or only those of the followed users (own_tweets = False). The exact output will be the name of the user who wrote the tweet, the body of the tweet and the time of publication (in date format, not timestamp).

It is not necessary (although it can be done) to include cells in the notebook with the tests of these functions (just documenting them correctly would be enough), as the teaching team will run a battery of tests on the three functions to check their correct operation.