Práctica 1: Sistema de reducción de URL's.

Bases de datos no estructuradas

Guillermo Gerardo Andrés Urbano
Fernando Avitúa Varela
Fernando Santa Rita Vizuet

9 de marzo de 2022

Resumen

Se desea implementar un sistema de reducción de URL's que pueda soportar el manejo de usuarios, que cada uno de los usuarios tenga ligas públicas y privadas, tener una *wish list*, y la posibilidad de categorizar dichas URL's. La implementación de dicho sistema requiere el uso de la API de Redis en Python como la estructura de datos en memoria, HTML para el despliegue en el navegador de la aplicación, y Flask como framework.

1. Introducción

De acuerdo a la página oficial [1], Redis es una estructura para almacenar datos en memoria empleada para bases de datos, caché, y provee diferentes estructuras de datos tales como cadenas, hashes, listas, conjuntos, etc.

2. Desarrollo

2.1. Diccionarios anidados en Redis

Aunque estamos haciendo una base de datos que usa llave valor, no existe en redis una manera de anidar tablas de hash. Entonces, para diseñar la estructura que planeamos, tendremos que tener diccionarios cuyos valores tienen la información de las llaves de otros diccionarios. Para mostrar este cambio, consideramos el siguiente ejemplo de un diccionario que podríamos encontrar en Python

$$a = \begin{cases} b : \begin{cases} d : e \\ h : i \end{cases} \\ c : \begin{cases} j : k \\ l : m \end{cases} \end{cases} \tag{1}$$

En redis, esto no es posible; los valores de una tabla hash deben de ser cadenas.

Cada que creamos una tabla hash en redis ejecutamos la función hmset; le pasamos dos argumentos, el primero es el nombre de la tabla hash y el segundo es una serie de cadenas pares

llave-valor. En Python se sigue la misma estructura, sólo que el segundo parámetro es un diccionario que sólo puede tener cadenas. El siguiente código muestra la ejecución

```
hmset('a',{'b':'1','c':'2'})
```

y tiene como resultado la siguiente estructura

$$a = \begin{cases} b : 1 \\ c : 2 \end{cases} \tag{2}$$

Para llegar a la estructura de 1 asociamos cada valor de la tabla hash 2 a otro diccionario que contiene el segundo nivel de llave-valor que se encuentra en 1. Así, tendríamos que llamar nuevamente la función hmset de la siguiente forma

```
hmset('1',{'d':'e','h':'i'})
hmset('2',{'j':'k','l':'m'})
```

y la estructura de los datos en redis sería

$$a = \begin{cases} b : 1 \\ c : 2 \end{cases} \tag{3}$$

$$1 = \begin{cases} d : e \\ h : i \end{cases} \tag{4}$$

$$2 = \begin{cases} j : k \\ l : m \end{cases} \tag{5}$$

Para llegar al valor m en 1 bastaría con hacer la evaluación $x = a[c][l]$; en la estructura que obtenemos en redis hay que buscar la información por pasos como se muestra a continuación

```
nombre_diccionario=r.hget('a','c') # esta función regresa el valor 2
x=r.hget(nombre_diccionario,'l') # esta función regresa el valor m
```

Operaciones CRUD

Para poder realizar manipulaciones de una base de datos es necesario poder hacer las operaciones de creación (C), lectura (R de read), actualización (U de update) y eliminación (D de delete). A continuación se muestra el código usado para implementar cada una de ellas

2.2. Creación

2.2.1. Usuarios

Para empezar, dado el diseño de nuestro problema, es necesario poder agregar usuarios. Cada usuario tiene 3 características asociadas: su username, su nombre y su contraseña. Escogimos nombrar a la tabla hash que tiene la información del user como su username, esta tabla contiene su nombre y contraseña. No es necesario anidar tablas.

A continuación se muestra la función que usamos para agregar usuarios.

```
def agregar_user(username, nombre, password)->bool:
"""Agrega al usuario a la base de datos"""
exito = r.hmset(username, {"nombre": nombre, "password": password})
return exito
```

2.2.2. Ligas públicas

Para cada usuario, las ligas públicas van a un conjunto (set de redis), de esta forma evitamos agregar una liga que ya existe. Después, para que le aparezca la liga corta al usuario que la agregó, se almacena una tabla hash cuyo nombre una concatenación del username y la liga larga y tiene como campos la liga corta y la categoría donde la quiere organizar.

La función que agrega una liga con usuario se muestra a continuación:

```
def add_liga_publica_user(username, url_larga, categoria) -> bool:

# Agrega a la lista publica
add_liga_publica(url_larga, categoria)

key_pub_usuario = f"{username}_pub"

exito = r.sadd(key_pub_usuario, url_larga)

return exito
```

Esta función agrega la página a un conjunto cuyo nombre comienza con el username y llama a la función *add_liga_publica* definida a continuación

```
def add_liga_publica(url_larga, categoria) -> bool:

llaveDic = f"{url_larga}"

r.sadd("urls", url_larga)
# Hacer la conversion liga publica
url_corta = conversion(url_larga)
```

```
8
9       exito = r.hmset(llaveDic, {"url_corta": url_corta
10                                 , "categoria": categoria})
11
12      return exito
```

2.2.3. Ligas privadas

Para las ligas privadas no se debe crear una tabla hash pública, en vez de eso, almacenamos la la lista en un hash que comienza con el username de aquel que la creo. También se guarda un conjunto de las url para evitar que se repitan.

```
1   def add_liga_privada_user(username, url_larga, categoria) -> True:
2
3       # Creamos el set de ligas privadas
4       r.sadd(f"lpriv_{username}", url_larga)
5
6       # Creamos un diccionario por cada url_larga
7       url_corta = conversion(url_larga)
8
9       exito = r.hmset(
10          f"{username}_{url_larga}", {"url_corta": url_corta
11                                      , "categoria": categoria}
12      )
13      return exito
```

2.3. Lectura

La lectura se hace por usuario. Siguiendo el diseño de la aplicación web, necesitamos separar las ligas en públicas y privadas y cada una de ellas en las categorías que el usuario haya especificado.

La siguiente función lee la base de datos de acuerdo a la estructura que hemos creado hasta ahora y devuelve dos diccionarios de Python; el primero tiene como llaves el nombre de las categorías y como valores la lista de las ligas cortas que pertenecen a dicha categoría, el segundo tiene lo mismo, pero para las ligas públicas del usuario.

```
1   def recuperar_listas(username):
2       cjto_url_pri = r.smembers(f"lpriv_{username}")
3       lista_url_pri = [si.decode("utf-8") for si in cjto_url_pri]
4
5       cjto_url_pub = r.smembers(f"{username}_pub")
6       lista_url_pub = [si.decode("utf-8") for si in cjto_url_pub]
7
8       categorias_pri = {}
```

```
        categorias_pub = {}
        for url in lista_url_pri:
            llaveDic = f"{username}_{url}"
            dict_url = dictBytes_a_dictString(r.hgetall(llaveDic))

            cat = dict_url["categoria"]
            url_corta = dict_url["url_corta"]

            if not categorias_pri.get(cat):
                categorias_pri[cat] = [url_corta]
            else:
                categorias_pri[cat].append(url_corta)

        for url in lista_url_pub:
            llaveDic = f"{url}"
            dict_url = dictBytes_a_dictString(r.hgetall(llaveDic))

            cat = dict_url["categoria"]
            url_corta = dict_url["url_corta"]

            if not categorias_pub.get(cat):
                categorias_pub[cat] = [url_corta]
            else:
                categorias_pub[cat].append(url_corta)

        return categorias_pri, categorias_pub
```

2.4. Actualización

Dado que no tiene mucho sentido editar una liga corta o una liga larga (en este caso mejor se crea una nueva). Se tomó un cuenta sólo la actualización del campo de la categoría.

```
def actualizar_liga(username, liga, categoria_nueva):

key_liga = f"{username}_{liga}"
exito = r.hset(name=key_liga, key="categoria", value=categoria_nueva)

return exito
```

2.5. Borrado

Para borrar ligas resultó más conveniente pasar una liga corta. Esto señaló una falla de nuestro diseño original porque para hacer una operación común tenemos que recorrer el conjunto de las ligas (ya sea públicas o privadas) hasta encontrar la liga corta a borrar. Si se hubiera hecho

el diseño con esto en mente se hubiera puesto la liga corta como nombre del diccionario y así tendríamos una complejidad de tiempo de $O(1)$ en vez de tener $O(n)$, donde n es el número de ligas que tiene cada usuario.

Como el usuario puede ver si la liga es pública o privada, se separaron las funciones. De esta forma sólo se recorren las ligas ya sea privadas o públicas.

A continuación se presenta la función usada para eliminar ligas públicas

```
def borrar_liga_pub(username, url_corta):

    for url_larga in r.smembers(f"{username}_pub"):

        dicc_nombre = f"{url_larga.decode('utf-8')}"
        url_corta1 = r.hget(dicc_nombre, "url_corta")

        if url_corta == url_corta1.decode("utf-8"):
            # Se quita la liga del conjunto username_pub
            r.srem(f"{username}_pub", url_larga)

            # Se quita del conjunto url
            r.srem("urls", url_larga)

            # Se elimina el diccionario con llave dicc_nombre
            r.delete(dicc_nombre)

            break  # termina el loop si ya se encontró

    return True
```

Y ahora la que elimina las ligas privadas

```
def borrar_liga_priv(username, url_corta):

    for url_larga in r.smembers(f"lpriv_{username}"):

        dicc_nombre = f"{username}_{url_larga.decode('utf-8')}"
        url_corta1 = r.hget(dicc_nombre, "url_corta")

        if url_corta == url_corta1.decode("utf-8"):
            # Se quita la liga del conjunto lpriv_username
            r.srem(f"lpriv_{username}", url_larga)

            # Se elimina el diccionario con llave dicc_nombre
            r.delete(dicc_nombre)

            break  # Termina el loop si ya se encontró
```

```
16
17      return True
```

Vemos que tanto nos costó el error de no tener la liga corta como llave, ya que un proceso que debería ser muy sencillo se vuelve muy elaborado.

3. Resultados

3.1. Creación

3.1.1. User

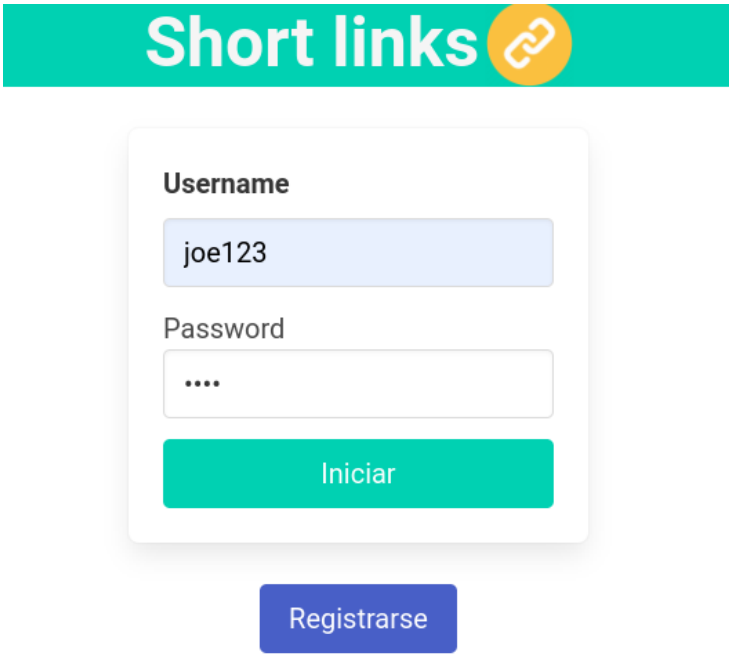
Usando la función definida en la sección 2.2.1, se crea el usuario hugo

```
1  agregar_user('hugol','hugo','123')
```

en la base de datos se va a crear una tabla de hash como la que se muestra a continuación

$$hugol = \begin{cases} nombre : hugo \\ password : 123 \end{cases}$$

En la aplicación web esto se hace en la página de registro como se muestra en la siguiente figura:



3.1.2. Ligas públicas y privadas

Llamamos la función definida en la sección 2.2.2 para crear una lista pública

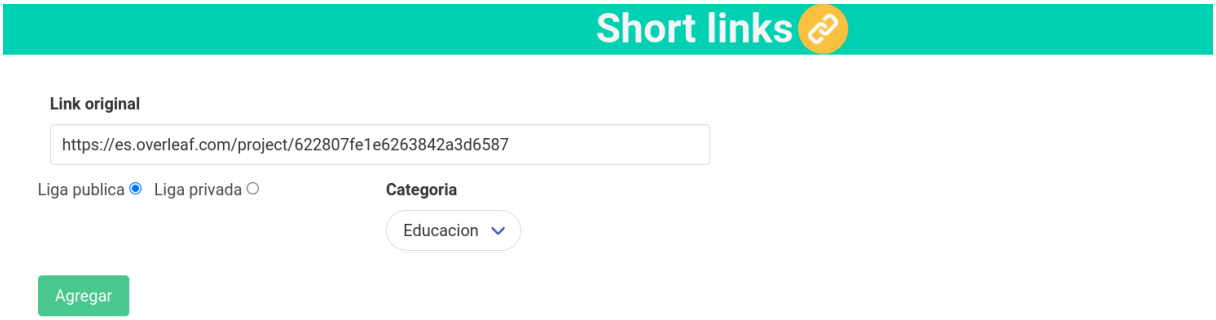
```
add_liga_publica_user(username='hugol',url_larga='youtube.com'
                      ,categoria='Entretenimiento')
```

esto tiene como resultado agregar un elemento al conjunto de ligas públicas, url, agregar un elemento al conjunto de ligas públicas del usuario y crear un diccionario con la información de la liga:

$$\begin{aligned} urls &= \{\ldots, youtube.com\} \\ hugol\_pub &= \{\ldots, youtube.com\} \\ youtube.com &= \begin{cases} url\_corta : bit.ly/sdlf \\ categoria : Entretenimiento \end{cases} \end{aligned} \tag{6}$$

donde los puntos suspensivos indican que puede que existan más ligas públicas o privadas.

En la aplicación web se hace como se muestra en la siguiente figura



Si se llama la función definida en la sección 2.2.3 con los siguientes argumentos

```
add_liga_privada_user(username='hugol'
                      ,url_larga='colab.research.google.com'
                      ,categoria='Educacion')
```

se agrega un elemento al conjunto y se crea un hash en la base de datos:

$$lpriv\_hugol = \{\ldots, colab.research.google.com\} \tag{7}$$

$$hugol\_colab.research.google.com = \begin{cases} url\_corta : bit.ly/dfdk \\ categoria : Educacion \end{cases} \tag{8}$$

en la aplicación web sólo se seleccionaría el botón de liga privada en la forma.

3.2. Lectura

De acuerdo a las ligas que hemos agregado en las secciones anteriores, obtenemos lo siguiente llamando la función de la sección 2.3

```
recuperaListas('hugol')
```

nos da como resultado

$$\Big(\{Educacion : [bit.ly/dfdk]\}, \{Entretenimiento : [bit.ly/sdlf]\}\Big)$$

donde la primera entrada de la tupla aparece por la inserción que se hizo en 7, mientras que la segunda entrada se debe a la que se hizo en 6.

Esta separación nos permite desplegarlas en la página web del perfil del usuario, como se muestra a continuación.

3.3. Actualización

El efecto de ejecutar la función definida en la sección 2.4 es el siguiente

```
actualizar_liga(username='hugol', liga='youtube.com'
                ,categoria_nueva='Videos')
```

es sólo modificar el valor de la llave categoría en el hash de la liga 6, como se muestra a continuación

$$youtube.com = \begin{cases} url\_corta : bit.ly/sdlf \\ categoria : Entretenimiento \end{cases} \Rightarrow \begin{cases} url\_corta : bit.ly/sdlf \\ categoria : Videos \end{cases}$$

Este paso no se implementó en la aplicación web.

3.4. Borrado

No se muestra el efecto de llamar estas funciones ya que sólo quitan el hash de la liga base de datos y el elemento del conjunto de ligas (ya sea público o del usuario).

En la página web el borrado se ve como en la siguiente figura

En esta figura se muestra el antes y después de borrar ligas públicas

4. Conclusiones

La implementación de una base de datos con redis fue complicada de entender al inicio, pero fue siendo más fácil cada vez. El intercambio entre complejidad de entender y la rapidez de la base de datos no es tan grande como pensábamos al inicio. Ya cuando pudimos automatizar todas las operaciones CRUD la base de datos lo hizo muy rápido.

Referencias

[1] *Redis*, https://redis.io/, consultado el 09/03/2022.