Memoria_Redis_TorroMartinezElena

March 18, 2020

1 TRABAJO PRÁCTICO BASES DE DATOS CLAVE- VALOR: REDIS

Elena Torró Martínez

Asignatura: GESTIÓN Y ALMACENAMIENTO DE INFORMACIÓN NO ESTRUCTURADA
MÁSTER UNIVERSITARIO EN INGENIERÍA Y CIENCIA DE DATOS

Esta práctica tiene como objetivo inicializarse con Redis a través de Python, creando unas funciones que interactúen con la base de datos para almacenar y representar mensajes tipo Twitter.

Se han utilizado las siguientes librerías:

- redis
- pandas
- datetime

1.1 Índice de contenido

1.2 Iniciar Redis

El primer paso es iniciar Redis, conectándonos en local con la base de datos que se ha creado utilizando el `docker-compose` proporcionado. Se ha añadido el comando `flushall` para **reiniciar** la base de datos cada vez que se ejecute el notebook.

[1]:
```
import redis

host='127.0.0.1'
port=6379
```

```
redis_db=redis.Redis(host=host,port=port, decode_responses=True)

redis_db.flushall()
```

[1]: True

1.3 Lectura de datos

Para leer los datos, que se encuentran en formato, vamos a utilizar la librería **Pandas**, y a importar los datos creando una instancia de un DataFrame para poder manipularlos. Este es el contenido de los DataFrames que se van a utilizar: `relations_df` para las relaciones de seguimiento entre los usuarios y `users_df` para los tweets de cada usuario.

[2]:
```
import pandas as pd

relations_df = pd.read_csv('relations.csv')
relations_df.head()
```

[2]:

	User	Follows	Following_Time
0	roxanefeller	cathcooney	13 Jun 2019 05:59:58
1	andyglittle	charleskod	14 Jul 2019 10:07:29
2	andyglittle	seers_helen	18 Jul 2019 09:50:48
3	andyglittle	karin_stowell	31 Aug 2019 15:20:48
4	hanyshita	andyglittle	12 Jul 2019 14:20:46

[3]:
```
users_df = pd.read_csv('twitter_sample.csv')
users_df.head()
```

[3]:

	User	Post_Time	\
0	andyglittle	13 Jul 2019 05:59:58	
1	andyglittle	05 Jul 2019 10:07:29	
2	afparron	11 Jul 2019 09:50:48	
3	drshahrul80	03 Aug 2019 15:20:48	
4	karin_stowell	04 Aug 2019 14:20:46	

	Tweet_Content	Unnamed: 3
0	We've loved being motivated by the stories of …	NaN
1	Thanks for the shout-out on our #MorethanMedic…	NaN
2	#MorethanMedicine-Our Story via @animalhea…	NaN
3	We hope to get some lovely weather on our annu…	NaN
4	This is what #MoreThanMedicine is about. Love …	NaN

1.4 Fechas

Para guardar las fechas vamos a convertirlas a formato `timestamp` utilizando la librería `datetime`. Como Redis, al recuperar una fecha, la devuelve en formato `string` hay que parsearla primero a `float` y después utilizar el formato desearlo para imprimirla.

Se han creado dos métodos: `date_to_datetime` para guardar las fechas y `datetime_to_date` para recuperarlas.

[22]:
```
import datetime

date_format = '%d %b %Y %H:%M:%S'

def date_to_datetime(date_str):
    return datetime.datetime.strptime(date_str, date_format).timestamp()

def datetime_to_date(date_datetime):
    return datetime.datetime.fromtimestamp(float(date_datetime)).
↪strftime(date_format)
```

1.5 Estructura

Un punto que he considerado crucial es definir de una manera legible y accesible la nomenclatura para guardar los distintos valores.

1.5.1 Usuarios

- `users`. Almacena un set con los usuarios existentes. Permite así saber si un nombre de usuario ya ha sido elegido, ya que el set sólo contiene valores únicos.
- `{user_id}`: Valor autoincrementado para generar los ids de usuarios
- `user:{user_id}`: almacena los datos de usuario. En este caso, no guardamos nada porque no tenemos realmente nada más que guardar, pero serviría para guardar algunos campos como por ejemplo, la biografía que muestra el usuario en su perfil o similares.
  - datos:

    ```
    {'username' : 'username'}
    ```
- `{username}`: almacena el id del usuario asociado a un nombre de usuario. Esto se ha decidido así para que el usuario pueda cambiar de nombre y mantener sin ningún otro cambio adicional todos los otros campos asociados a su id de usuario.
  - datos: `user_id`
- `followers:{user_id}`: almacena un set ordenado con los ids de los usuarios a los que sigue el usuario con identificador `{user_id}`, que nos permiten obtener después la lista de followers.
- `followings:{user_id}`: almacena un set ordenado con los ids de los usuarios de los que es seguidor `{user_id}`, , que nos permiten obtener después la lista de followings.

1.5.2 Posts

- `{post_id}`: Valor autoincrementado para generar los ids de los posts
- `post:{timestamp}:{post_id}`: Almacena los datos de un post: el id del usuario que lo ha escrito, el contenido del mismo y la fecha en la que lo hizo. Esto facilita la impresión del post. El formato del id se debe a que de este modo podemos ordenarlos por fecha utilizando la función `sort` como pide el enunciado. Además de que, si dos posts han sido publicados justo en la misma fecha, puedan diferenciarse por el id del post.
- `posts:user:{id}`: Almacena la lista de posts por usuario. Como la lista hay que recorrerla entera para mostrar los posts y no buscar un elemento dentro de ella, se ha decidido utilizar una simple lista en lugar de otro tipo de estructura.

1.6 Funciones

1.6.1 Nuevo Usuario

Antes de crear un nuevo usuario, se comprueba si está presente en el set de usuarios de la base de datos. Si está, muestra un error.

[5]:
```
def nuevo_usuario(username):
    if redis_db.sadd('users', username):
        new_id = redis_db.incr('user_id')
        user_id = 'user:{0}'.format(new_id)

        user_data = {
            'username': username
        }

        redis_db.set(username, user_id)
        redis_db.hmset(user_id, user_data)

        return user_data

    else:
        print('ERROR: User {0} already exists'.format(username))
        return False
```

1.6.2 Nuevo Following y Nuevo Follower

Ya que se crean del mismo modo los conjuntos de followings y followers, se ha abstraido la lógica en la función `nuevo_follow`, al que se llama directamente desde `nuevo_following` o `nuevo_follower`, según el caso.

Se han tenido en cuenta una serie de consideraciones:

- 1. Ambos usuarios han de existir en la base de datos

- 2. Un usuario no puede seguirse a sí mismo

[6]:
```
def nuevo_follow(username, follow_username, date_str, follow_type):
    if username == follow_username:
        print('ERROR: Username and follower username are the same')
        return False
    if not redis_db.get(username):
        print('ERROR: User {0} does not exist'.format(username))
        return False
    if not redis_db.get(follow_username):
        print('ERROR: User {0} does not exist'.format(follow_username))
        return False

    user_id = redis_db.mget(username)[0]
    follow_id = redis_db.mget(follow_username)[0]
    user_follow_id = '{0}:{1}'.format(follow_type, user_id)

    timestamp = date_to_datetime(date_str)
    follow_data = {follow_username: timestamp}

    redis_db.zadd(user_follow_id, follow_data)

    return follow_data
```

[7]:
```
def nuevo_follower(username, follower_username, date_str):
    return nuevo_follow(username, follower_username, date_str, 'followers')
```

[8]:
```
def nuevo_following(username, follower_username, date_str):
    return nuevo_follow(username, follower_username, date_str, 'following')
```

1.6.3 Seguir

Esta función simplemente llama, como se expresa en el enunciado, a `nuevo_follower` y `nuevo_following`.

[9]:
```
def seguir(username, follow_username, date_str):
    nuevo_follower(username, follow_username, date_str)
    nuevo_following(follow_username, username, date_str)
```

1.6.4 Nuevo Post

Esta función almacena los datos del post como se ha explicado anteriormente. Para ello, se hace en 3 pasos importantes:

1. Guardar el contenido del post en la estructura `post:{timestamp}:{post_id}` para su futura ordenación

2. Añadir el post a la lista de posts del usuario que escribe el post
3. Recorrer la lista de usuarios que siguen al usuario que escribe el post para que aparezcan en su timeline, y guardar el post solo si empezó a seguir al usuario **antes** de que hubiera publicado el post.

[10]:
```python
def nuevo_post(username, message, date_str):
    if not redis_db.get(username):
        print('ERROR: User {0} does not exist'.format(username))
        return False

    new_id = redis_db.incr('post_id')
    timestamp = date_to_datetime(date_str)
    user_id = redis_db.mget(username)[0]

    post_data = {
        'user_id': user_id,
        'message': message,
        'timestamp': timestamp
    }

    # 1. Save Post Content
    post_sort_id = 'post:{0}:{1}'.format(timestamp, new_id)
    redis_db.hmset(post_sort_id, post_data)

    # 2. Add to users's post list
    posts_user_id = 'posts:{0}'.format(user_id)
    redis_db.lpush(posts_user_id, post_sort_id)

     # 3. Add to followers user's post list
    followers_user_id = 'followers:{0}'.format(user_id)

    for key in redis_db.zscan_iter(followers_user_id):
        following_timestamp = key[1]
        if (following_timestamp < timestamp):
            following_username = key[0]
            user_id = redis_db.mget(following_username)[0]

            posts_user_id = 'posts:{0}'.format(user_id)
            redis_db.lpush(posts_user_id, post_sort_id)
```

1.6.5 Llenar la base de datos

La función `database_setup` lee los valores de los ficheros y llamando a `nuevo_usuario`, `seguir` y `nuevo_post` se encarga de rellenar la base de datos.

[11]:
```
def database_setup():
    relations_df = pd.read_csv('relations.csv')
    users_df = pd.read_csv('twitter_sample.csv')

    for username in relations_df.User.unique():
        nuevo_usuario(username)

    for index, row in relations_df.iterrows():
        seguir(row['User'], row['Follows'], row['Following_Time'])

    for index, row in users_df.iterrows():
        nuevo_post(row['User'], row['Tweet_Content'], row['Post_Time'])
```

[ ]:
```
database_setup()
```

1.6.6 Obtener followers y followings

Del mismo modo que la función `nuevo_follow`, la función `obtener_follow_list` abstrae la lógica de imprimir la lista de followers o followings de un usuario dado su nombre de usuario.

[13]:
```
def obtener_follow_list(username, follow_type):
    if not redis_db.get(username):
        print('ERROR: User {0} does not exist'.format(username))
        return False

    user_id = redis_db.mget(username)[0]
    followers_user_id = '{0}:{1}'.format(follow_type, user_id)

    for key in redis_db.zscan_iter(followers_user_id):
        print('{0} at {1}'.format(key[0], datetime_to_date(key[1])))
```

[14]:
```
def obtener_followers(username):
    obtener_follow_list(username, 'followers')
```

[15]:
```
obtener_followers('karin_stowell')
```

```
drshahrul80 at 01 Aug 2019 21:58:25
```

[16]:
```
def obtener_followings(username):
    obtener_follow_list(username, 'following')
```

[17]:
```
obtener_followings('karin_stowell')
```

```
drshahrul80 at 24 Jul 2019 10:00:00
andyglittle at 31 Aug 2019 15:20:48
```

1.6.7 Obtener Timeline

Por último, la función `obtener_timeline` devuelve una lista ordenada de manera descendiente de los posts de un usuario, que ha sido creada utilizando la función `nuev_post`.

- **sort_by**: nos permite ordenar por `timestamp`
- **get**: nos devuelve, de la lista de post almacenados externa, los valores del post que queremos, y que son los que utilizamos para imprimirlos formateados.
- **desc**: Ordenamos los posts de manera descendiente ya que queremos leerlos de más reciente a más antiguo.

La función `sort` tiene otro parámetro: `source`, para poder guardar el resultado en la base de datos. Inicialmente, quería guardarlo en un sorted set para después poder filtrarlo utilizando el parámetro `match`, pero `sort` no deja guardar tuplas, que es el formato en el que he querido guardar los posts, por lo que al final no he utilizado redis para filtrar los posts por usuario.

[18]:
```python
def print_post(user, message, timestamp):
    username = redis_db.hmget(user, 'username')[0]
    post_message = message
    post_date = datetime_to_date(float(timestamp))

    print('\n-----')
    print('@{0}: {1} \n{2}'.format(username, post_message, post_date))
    print('-----\n')
```

[19]:
```python
def obtener_timeline(username, tweets_propios=False):
    if not redis_db.get(username):
        print('ERROR: User {0} does not exist'.format(username))
        return False

    user_id = redis_db.mget(username)[0]
    posts_user_id = 'posts:{0}'.format(user_id)
    sort_by = 'post:*'

    sorted_posts = redis_db.sort(
        posts_user_id,
        by=sort_by,
        get=['*->user_id', '*->message', '*->timestamp'],
        groups=['user_id', 'message', 'timestamp'],
        desc=True
    )

    for post in sorted_posts:
        if not tweets_propios:
            if user_id != post[0]:
                print_post(post[0], post[1], post[2])
        else:
```

```
            print_post(post[0], post[1], post[2])
```

[20]:
```
obtener_timeline('alkhalilkouma', False)
```

[21]:
```
obtener_timeline('alkhalilkouma', True)
```

1.7 Conclusiones

Esta es la primera vez que utilizo Redis. Por una parte, me parece muy pontente para hacer pequeñas pruebas y tengo interés para ponerlo en práctica en algún proyecto real. Por otra parte, el sistema de nomenclatura me parece complejo ya que no existe ninguna convención para nombrar las claves. Personalmente, soy fan de las convenciones y más cuando vienen de parte de los propios desarrolladores de una librería. Me resultaría complicado utilizar una base de datos de redis creada por otros que no estuviera debidamente documentada.

También creo que tiene algunas limitaciones, como por ejemplo, que no pueda elegir en qué formato guardo el output de una operación `sort`. En algunos casos, creo que es posible que algunas de estas limitaciones se deban a la librería de python utilizada, que al final actúa como un API para poder interactuar con Redis. Por esto, el aprendizaje es doble: cómo funciona Redis y cómo interactuar con él a través de esta librería. En la librería echo en falta más ejemplos para saber cómo utilizar las distintas funciones.

También echo en falta en los `match` poder ordenar no sólo por valores "iguales" o que hagan match, sino poder filtrar por valores numéricos, con filtros como mayor qué, menor qué, etc.

Mi percepción es que se trata de algo muy simple que requiere un fuerte esfuerzo en la fase de diseño.