



Deep Learning y Big Data con Python

Laboratorio 1

Estudiante:

Jose Javier Arce Solorzano

Profesor:

Adan de Jesus Mora Fallas

# 1. MongoDB

## Creación del .YML

Para poder utilizar docker para hostear el servidor de mongodb, se debe crear el .yml respectivo para que docker pueda realizarlo, la configuracion utilizada para este fin fue la siguiente:

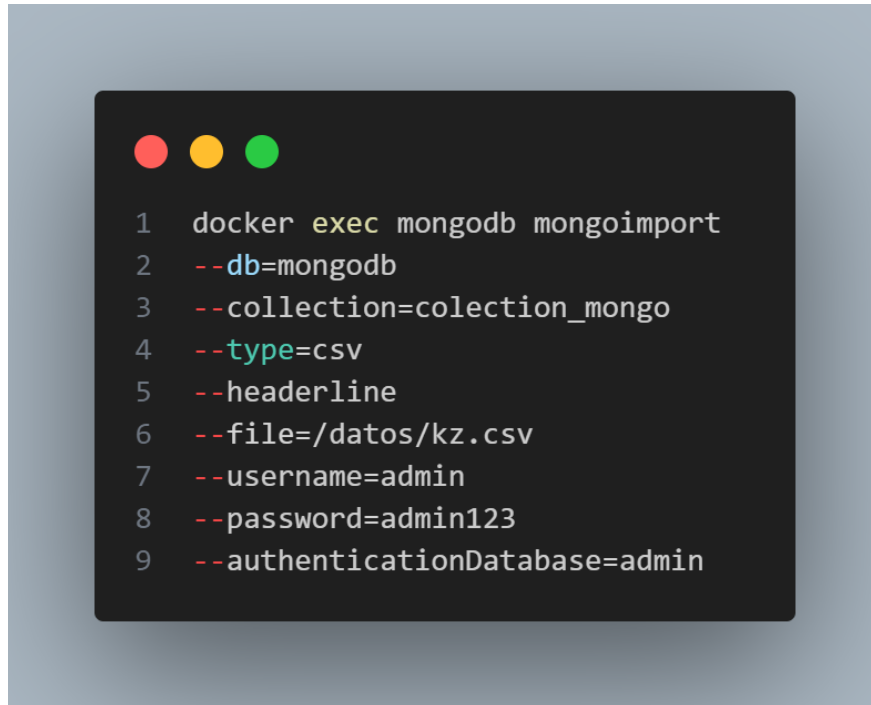


Figura 1: .yml MongoDB

Se setean dos parámetros de username y de contraseña opcionales que mas adelante serán necesarios para la carga de datos. El documento 'kz.csv' es el dataset que contiene los datos que se quieren consultar en la base de datos, es una buena práctica cargarlo como 'volume' ya que así se asegura que este dataset se encuentre dentro del contenedor de docker de donde después se puede consultar, esto es muy útil cuando se tienen demás contenedores y se opera con los valores presentes en otro contenedor, para este laboratorio la carga se realiza de manera local mediante un script de python.

## Carga de los Datos a la base

Para la carga de datos a la base de datos de MongoDB, podemos ejecutar un comando desde la terminal utilizando el contenedor de Docker y atributos propios de la imagen de mongo que nos ofrece esta opción (mongoimport), el tiempo de espera no es mayor a un par de minutos. Este paso es vitalmente importante ya que el mismo define el nombre de nuestra base de datos y de la colección que proximately consultaremos.

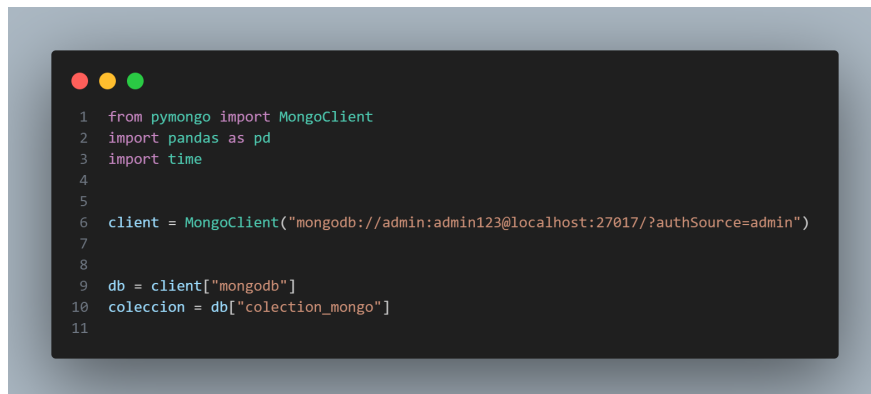


```
1 docker exec mongodb mongoimport
2 --db=mongodb
3 --collection=collection_mongo
4 --type=csv
5 --headerline
6 --file=/datos/kz.csv
7 --username=admin
8 --password=admin123
9 --authenticationDatabase=admin
```

Figura 2: Cargar datos a MongoDB

## Consultas Realizadas

Para realizar las consultas, debemos definir nuestra base de datos y colección dentro del script de Python basándose en cómo las definimos en la sección anterior.



```
1 from pymongo import MongoClient
2 import pandas as pd
3 import time
4
5
6 client = MongoClient("mongodb://admin:admin123@localhost:27017/?authSource=admin")
7
8
9 db = client["mongodb"]
10 coleccion = db["collection_mongo"]
11
```

Figura 3: Consulta MongoDB

Esto lo hacemos gracias a la librería pymongo, definiendo nuestro cliente con MongoClient.

La consulta completa se encuentra en el archivo consulta-mongo.py

## 2. Redis

### Creacion del .YML

Para poder utilizar docker para hostear el servidor de redis, se debe crear el .yml respectivo para que docker pueda realizarlo. La configuración utilizada para este fin fue la siguiente:



Figura 4: .yml Redis

### Carga de los Datos a la base

Para el caso de Redis y Hbase la carga no fue tan simple, ya que no existia una herramienta preestablecida para a la base, lo que implico que se tuvo que hacer mediante una rutina de python, este proceso fue mas tardado ya que debia iterar sobre todo el dataset y conectar con el servidor para enviar los datos. El codigo utilizado fue el siguiente

Para realizar esta carga y las consultas fue necesario instalar la libreria redis de python y pandas, ya que gracias a eso se pudo establecer conexion con el servidor y enviar los datos. Para el caso particular de Redis ya que este almacena los datos como un valor en memoria, la carga podia durar hasta 1 h, se opto hacer la carga mediante paquetes de 500 rows para evitar que siempre que pasara un row debiera comunicarse con el contenedor. Dicho sea de paso se debe tener cuidado, ya que si el valor que se toma como key esta repetido, esto ocasionaría que los datos se reescriban y se pierdan, por lo mismo no solo se toma el order\_id como key, sino tambien su index para evitar se sobrescribieran datos

```

1  import redis
2  import pandas as pd
3
4  r = redis.Redis(host='localhost', port=6379, decode_responses=True)
5
6  csv_path = "kz.csv"
7  df = pd.read_csv(csv_path)
8
9  pipe = r.pipeline()
10 batch_size = 500
11
12 for idx, row in df.iterrows():
13     key = f"orden:{row['order_id']}:{idx}"
14
15     pipe.hset(key, mapping={
16         "event_time": row["event_time"],
17         "product_id": row["product_id"],
18         "category_id": row["category_id"],
19         "category_code": row["category_code"],
20         "brand": row["brand"],
21         "price": row["price"],
22         "user_id": row["user_id"]
23     })
24
25     if (idx + 1) % batch_size == 0:
26         pipe.execute()
27         print(f"{idx + 1} registros insertados...")
28
29 pipe.execute()
30 print("Carga finalizada")

```

Figura 5: Cargar Datos Redis

## Consultas Realizadas

Para la consulta de redis de igual forma se utilizao la libreria redis para establecer cone-xion con el contenedor donde se encontraba el mismo:

```

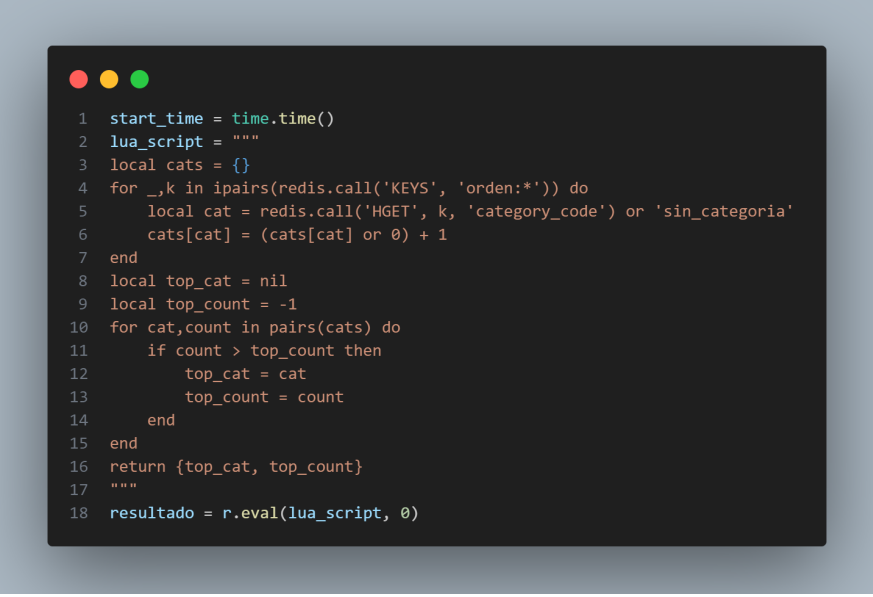
1  r = redis.Redis(host='localhost', port=6379, decode_responses=True)

```

Figura 6: Conexion Redis

Sin embargo aunque la consulta se realizaba casi de forma instantánea, tardaba demasiado tiempo mostrando el resultado en la consola, con el afán de minimizar este tiempo se crearon los denominados lua\_scripts, esto se realizo para correr un script de redis-cli (un programa que permite enviar comandos a un servidor Redis y leer sus respuestas) desde python mediante el de la función eval de la libreria de redis. Se llama Lua script porque el código

que se usa dentro de `redis.eval(...)` está escrito en Lua, que es un lenguaje de programación. Se puede ver un ejemplo en la siguiente imagen:



```
1 start_time = time.time()
2 lua_script = ""
3 local cats = {}
4 for _,k in ipairs(redis.call('KEYS', 'orden:')) do
5     local cat = redis.call('HGET', k, 'category_code') or 'sin_categoria'
6     cats[cat] = (cats[cat] or 0) + 1
7 end
8 local top_cat = nil
9 local top_count = -1
10 for cat,count in pairs(cats) do
11     if count > top_count then
12         top_cat = cat
13         top_count = count
14     end
15 end
16 return {top_cat, top_count}
17 ""
18 resultado = r.eval(lua_script, 0)
```

Figura 7: Uso de lua script para consulta de Redis

### 3. Hbase

#### Creacion del .YML

Para el caso de hbase, en la creación del .yaml era importante también crear el contenedor de zookeeper que hbase necesita.

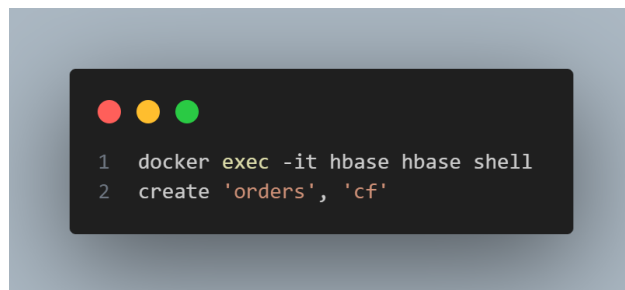


```
1  version: '3.8'
2
3  services:
4    zookeeper:
5      image: confluentinc/cp-zookeeper:7.5.0
6      container_name: zookeeper
7      environment:
8        ZOOKEEPER_CLIENT_PORT: 2181
9        ZOOKEEPER_TICK_TIME: 2000
10     ports:
11       - "2181:2181"
12
13     hbase:
14       image: dajobe/hbase
15       container_name: hbase
16       depends_on:
17         - zookeeper
18       ports:
19         - "9090:9090"
20       volumes:
21         - ./kz.csv:/datos/kz.csv
```

Figura 8: .yaml Zookeeper y Hbase

#### Carga de los Datos a la base

Primero se debe crear dentro del contenedor la base de datos y la 'familia' de columnas donde se guardaran los datos, esto se realiza corriendo estos comando en la terminal:

A terminal window with a dark background and three colored window control buttons (red, yellow, green) at the top left. It contains two lines of text: a command to run a Docker container with Hbase shell, and a command to create a table named 'orders' with a column family 'cf'.

```
1 docker exec -it hbase hbase shell
2 create 'orders', 'cf'
```

Figura 9: Crear tabla en Hbase

Al igual que en Redis, no había una herramienta directa para hacer la carga de datos, se podía utilizar hbase shell que viene integrada en el contenedor, pero no era viable para un dataset tan grande, por lo que se decidió hacerlo mediante una rutina de pythin utilizando la librería happybase para realizar la conexión con el contenedor.

A terminal window with a dark background and three colored window control buttons (red, yellow, green) at the top left. It contains a Python script that uses the happybase library to connect to an Hbase instance, read data from a CSV file, and insert it into a table named 'orders' in batches. The script includes error handling and progress reporting.

```
1 import happybase
2 import csv
3
4 def safe_encode(value):
5     return str(value).encode() if value else b''
6
7
8 csv_file = 'kz.csv'
9 table_name = 'orders'
10 familia = 'cf'
11 batch_size = 500
12
13
14 connection = happybase.Connection('localhost', port=9090)
15 connection.open()
16 table = connection.table(table_name)
17
18 count = 0
19 errores = 0
20
21 with open(csv_file, newline='') as f:
22     reader = csv.DictReader(f)
23     batch = table.batch()
24
25     for idx, row in enumerate(reader):
26         base_key = str(row.get('order_id', '')).strip()
27
28         # Asegura unicidad del row_key, aunque haya duplicados
29         row_key = f'{base_key}_{idx}' if base_key else f'fila_{idx}'
30
31         batch.put(row_key, {
32             f'{familia}:event_time': safe_encode(row.get('event_time')),
33             f'{familia}:product_id': safe_encode(row.get('product_id')),
34             f'{familia}:category_id': safe_encode(row.get('category_id')),
35             f'{familia}:category_code': safe_encode(row.get('category_code')),
36             f'{familia}:brand': safe_encode(row.get('brand')),
37             f'{familia}:price': safe_encode(row.get('price')),
38             f'{familia}:user_id': safe_encode(row.get('user_id')),
39         })
40         count += 1
41
42     if count % batch_size == 0:
43         try:
44             batch.send()
45             print(f'{count} registros insertados...')
46             batch = table.batch()
47         except Exception as e:
48             print(f'Error al enviar batch en fila {idx}: {e}')
49             errores += 1
50
51     batch.send()
52
53 print(f'Carga finalizada con {count} registros. Errores: {errores}')
54
55 connection.close()
```

Figura 10: Cargar datos Hbase

## Consultas Realizadas

Para realizar las consultas se estableció la conexión con hbase mediante happybase



```
1 connection = happybase.Connection('localhost', port=9090)
2 table = connection.table('orders')
```

Figura 11: Conexion Hbase

## 4. Comparativa Consultas

Al realizar las tres consultas a las 3 distintas bases de datos se obtuvieron los siguientes resultados:

```
1 Benchmarking consultas realizadas
2
PROBLEMAS OUTPUT DEBUG CONSOLE TERMINAL PORTS
(venv) jjsajavier@GL: /mnt/c/Users/jjars/OneDrive/Escrito
rio/laboratorio-15 python consulta/consulta-hbase.py
Consulta 1 (Categoría más vendida): 31.7817 segundos
Resultado: {'sin_categoria': 612202}
Consulta 2 (Brand con mayores ingresos): 44.8368 segundos
Resultado: {'samsung', 90852821.6087252}
Consulta 3 (Mes con más ventas): 91.5841 segundos
Resultado: {'05', 483632}
(venv) jjsajavier@GL: /mnt/c/Users/jjars/OneDrive/Escrito
rio/laboratorio-15
(venv) jjsajavier@GL: /mnt/c/Users/jjars/OneDrive/Escrito
rio/laboratorio-15 python consulta/consulta-mongo.py
Consulta 1 (Categoría más vendida): 1.7638 segundos
Resultado: {'_id': '', 'total_ventas': 612202}
Consulta 2 (Brand con mayores ingresos): 1.8888 segundos
Resultado: {'_id': 'samsung', 'total_ingresos': 90852821.6}
Consulta 3 (Mes con más ventas): 7.2369 segundos
Resultado: {'_id': 6, 'total_ventas': 483632}
(venv) jjsajavier@GL: /mnt/c/Users/jjars/OneDrive/Escrito
rio/laboratorio-15
(venv) jjsajavier@GL: /mnt/c/Users/jjars/OneDrive/Escrito
rio/laboratorio-15 python consulta/consulta-redis.py
Consulta 1 (Categoría más vendida): 11.5548 segundos
Resultado: {'sin_categoria': 612202}
Consulta 2 (Brand con mayores ingresos): 11.7867 segundos
Resultado: {'samsung', 90852821.6}
Consulta 3 (Mes con más ventas): 9.2932 segundos
Resultado: {'05', 483632}
(venv) jjsajavier@GL: /mnt/c/Users/jjars/OneDrive/Escrito
rio/laboratorio-15
```

Figura 12: Comparativa Hbase/MongoDB/Redis

## Conclusiones

- En los tres casos los resultados retornados por las tres consultas fue el mismo, lo que permite inferir que la carga de datos y la forma de realizar la consulta fue correcta
- Aquella consulta que menos tiempo tardó realizándose fue la de mongodb, esto puede ser gracias a la manera en la que se puede conectar con la base mediante el uso de la librería pymongo, puede que los procesos que corren por detrás estén mejor optimizados.
- Tanto para redis como para hbase la carga de datos fue el proceso que más tardó, esto debido a que no existe una herramienta propia de alguno de los dos que permita la carga de estos datos de la forma en la que se propuso.
- Al ejecutar las consultas directo en el contenedor de redis usando redis-cli el resultado fue casi instantáneo durando al rededor de 0.0007 s cada una, un resultado esperado para la naturaleza del funcionamiento de redis.

## Referencias

- [1] MongoDB, Inc. *MongoDB Documentation*. Disponible en: <https://www.mongodb.com/docs/manual/>
- [2] Redis. *Redis Documentation*. Disponible en: <https://redis.io/docs/>
- [3] Apache HBase. *Apache HBase Reference Guide*. Disponible en: <https://hbase.apache.org/book.html>
- [4] MongoDB, Inc. *PyMongo – Python driver for MongoDB*. Disponible en: <https://pymongo.readthedocs.io/en/stable/>
- [5] Zhang, D. (2020). *Hands-On Big Data Analytics with PySpark*. Packt Publishing.