

Laboratorio 1: Integración de bases de datos NoSQL



Autor: Jose Fernando Salazar Arguedas

Fecha: Agosto 2025

Repositorio: [GitHub Link](#)

Objetivos:

- Este proyecto tiene como objetivo integrar y consultar datos estructurados desde diferentes bases de datos NoSQL (MongoDB, Redis, HBase).
- Investigar cómo montar un servidor de HBase usando un container de Docker y usarlo desde Python con alguna librería disponible.
- Investigar cómo montar un servidor de MongoDB usando un container de Docker y usarlo desde Python con alguna librería disponible..
- Investigar cómo montar un servidor de Redis usando un container de Docker y usarlo desde Python con alguna librería disponible..
- Crear un script de Python que cargue (este dataset) de 2 millones de rows en las 3 bases de datos siguiendo el esquema de cada una.
- Crear un script (o uno por cada base de datos) de Python que haga las siguientes consultas en las bases de datos (debe investigar cómo hacer las consultas, cada consulta debe estar registrada en la documentación):
 - Cuál es la categoría más vendida?
 - Cuál marca (brand) generó más ingresos brutos?
 - Qué mes tuvo más ventas? (En UTC)
- Experimente y analice los tiempos de respuesta de cada consulta en cada base de datos.

Desarrollo:

Montaje de images en docker:

Para la construcción de las imágenes de las bases de datos en docker se desarrolló un archivo **docker-compose.yml** con la siguiente información:

```
services:
  mongodb:
    image: mongo:7
    container_name: mongodb
    ports:
      - "27017:27017"
    volumes:
      - mongodb_data:/data/db

  redis:
    image: redis:latest
    container_name: redis
    ports:
      - "6379:6379"
    volumes:
      - redis_data:/data
    command: ["redis-server", "--appendonly", "yes"]

  hbase:
    image: harisekhon/hbase:latest
    container_name: hbase
    platform: linux/amd64
    environment:
      - HBASE_THRIFT_HTTP=false
    ports:
      - "16010:16010" # Web UI
```

```

- "9090:9090"      # Thrift
- "2181:2181"      # ZooKeeper
volumes:
- hbase_data:/hbase

volumes:
  mongodb_data:
  redis_data:
  hbase_data:

```

Para montar la imagen utilice el comando en terminal:

```
docker-compose up -d
```

Tuve varios problemas con Hbase, entre ellos:

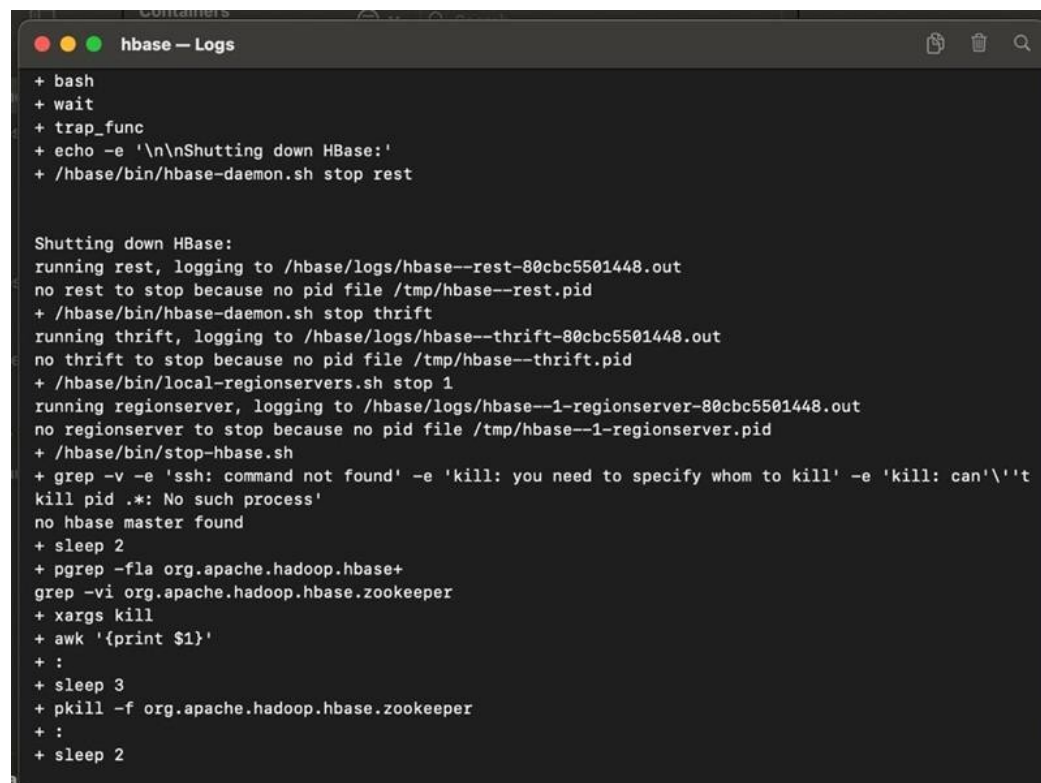
- Si utilizaba el servicio de **Thrift** para realizar las transacciones de datos me daba errores de conexión, esta era intermitente en **Orbstack**; esto debido a que necesita algunas dependencias que deben estar instaladas para poder utilizarlo. Debido a esto se omitió el uso de Thrift:

```

environment:
- HBASE_THRIFT_HTTP=false

```

Error Log de Hbase en Orbstack:



```

hbase — Logs
+ bash
+ wait
+ trap_func
+ echo -e '\n\nShutting down HBase:'
+ /hbase/bin/hbase-daemon.sh stop rest

Shutting down HBase:
running rest, logging to /hbase/logs/hbase--rest-80cbc5501448.out
no rest to stop because no pid file /tmp/hbase--rest.pid
+ /hbase/bin/hbase-daemon.sh stop thrift
running thrift, logging to /hbase/logs/hbase--thrift-80cbc5501448.out
no thrift to stop because no pid file /tmp/hbase--thrift.pid
+ /hbase/bin/local-regionervers.sh stop 1
running regionserver, logging to /hbase/logs/hbase--1-regionserver-80cbc5501448.out
no regionserver to stop because no pid file /tmp/hbase--1-regionserver.pid
+ /hbase/bin/stop-hbase.sh
+ grep -v -e 'ssh: command not found' -e 'kill: you need to specify whom to kill' -e 'kill: can\''t
kill pid .*: No such process'
no hbase master found
+ sleep 2
+ pgrep -fla org.apache.hadoop.hbase+
grep -vi org.apache.hadoop.hbase.zookeeper
+ xargs kill
+ awk '{print $1}'
+ :
+ sleep 3
+ pkill -f org.apache.hadoop.hbase.zookeeper
+ :
+ sleep 2

```

Creacion del virtual enviroment:

- Primero cree el archivo **requirements.txt** donde agregue las librerias necesarias para el proyecto:

```
pandas
pymongo
redis
happybase
```

- Para crear el ambiente virtual utilice el siguiente comando en la carpeta raiz del proyecto:

```
python3.13 -m venv .venv
```

Para instalar las librerias necesarias en el ambiente virtual use el comando en la carpeta donde esta el archivo de **requirements.txt**:

-

```
pip install -r requirements.txt
```

Desarrollo de script de python:

- Primero guarde los datos en formato .zip a la carpeta raiz del proyecto para cargarlos y procesarlos.
- Una vez guardados los datos procedi a cargarlos como un DataFrame con ayuda de **pandas**. Ya que todas las bases de datos van a utilizar este **df** revise algunos de los datos primordiales que iba a utilizar para la investigacion:
 - Cantidad de datos y columnas. Datos
 - nulos y vacios.
 - Tipo de datos por columna. Al revisar los datos existen varios factores que podrian afectar mi procesamiento en cada una de las bases de datos, por esto procese los datos modificando ciertos tipos de datos y tambien cambiando datos vacios de la columna **price** por 0.

```
df = pd.read_csv('Laboratorio 1/kz.csv.zip')
#EDA:
print('Encabezados de columnas: ',df.columns)
print('Cantidad de rows: ',len(df))
print(df.info())
print('Datos nulos: \n',df.isna().sum())
print('Datos vacios: \n', (df == '').sum())
df['event_time'] = pd.to_datetime(df['event_time'], errors='coerce')
df['event_time'] = df['event_time'].dt.strftime("%Y-%m-%d %H:%M")
# Reemplazar valores NaN de price con 0
df['price'] = df['price'].fillna(0)
```

- Resultado del analisis:

```
Encabezados de columnas: Index(['event_time', 'order_id', 'product_id',
                                'category_id', 'category_code',
                                'brand', 'price', 'user_id'],
                                dtype='object')
Cantidad de rows: 2633521
```

```

#      Column      Dtype
<class 'pandas.core.frame.DataFrame'>
0      event_time      object
RangeIndex: 2633521 entries, 0 to 2633520
1      order_id      int64
Data columns (total 8 columns):
2      product_id      int64
3      category_id      float64
4      category_code      object
5      brand      object
6      price      float64
7      user_id      float64

dtypes: float64(3), int64(2), object(3)
memory usage: 160.7+ MB
None
Datos nulos:
event_time      0
order_id      0
product_id      0
category_id      431954
category_code      612202
brand      506005
price  user_id      431954
dtype: int64      2069352
Datos vacios:
event_time
order_id      0
product_id      0
category_id      0
category_code      0
brand      0
price      0
user_id      0
dtype: int64      0

```

Clase para manejar MongoDB:

Una vez procesados y cargados los datos empecé con la creación del script de **MongoDB**. Todo esto utilizando la librería **pymongo** de python.

- Cree una clase llamada **Mongo** para manejar la carga de datos en la db y las consultas.
 - Primero cree la función de inicio con el nombre del host, el puerto de comunicación, nombre de la db y el nombre de la colección de documentos

```

class Mongo:
    def __init__(self, host='localhost', port=27017, db_name='mydb',
                  collection='data'):
        from pymongo import MongoClient
        self.client = MongoClient(host, port)
        self.collection = self.client[db_name][collection]

```

- El parametro **collection** es quien contiene el acceso a la db para la carga, consulta y borrado de datos.
- Al investigar los metodos de carga de datos estos deben tener un formato tipo JSON o como diccionario, ya que MongoDB almacena en este esquema la informacion.
 - Existen 2 metodos para la carga de datos, **insert_one** o **insert_many**, como su nombre lo dice, uno de ellos inserta un documento a la vez y el otro metodo puede insertar multiples a la vez. En nuestro caso debido a la gran cantidad de datos por insertar utilice **insert_many**.
 - La funcion tiene solo un argumento, esta espera que ingrese datos para convertirlos en una lista de diccionarios para ingresarlos a la db como documentos. Se utiliza **insert_many** para enviarlos.

```
def upload(self, data):
    if data is None:
        raise ValueError("Data is empty.")
    else:
        data = data.to_dict('records')
        self.collection.drop() # Limpiar la colección antes de insertar
                                # nuevos datos
        try:
            self.collection.insert_many(data)
        except Exception as e:
            print(f"Error inserting data: {e}")
```

- Para realizar consultas cree una funcion que devuelve datos especificos utilizando llave-valor {'brand': 'Nike'} o devolviendo todos los datos en la collection.
 - La funcion que cree tiene un solo argumento opcional que es **query**, si este se deja en blanco la consulta devolvera todos los datos en la db.
 - MongoDB tiene mucha flexibilidad en cuanto a consultas, existen varios metodos que se pueden utilizar pero se decidio usar **find()** y **aggregate()** por la flexibilidad de manipular los datos que necesite en el momento y analisis posible entre ellos, otros ejemplos de consultas son:
 - **find_one()**: Devuelve el primer documento que coincida
 - **distinct()**: Devuelve los valores unicos de un campo con un key.

```
def request_db(self, query=None):
    if query is None:
        return self.collection.find()
    else:
        return self.collection.find(query)
def request_data(self, query=None):
    return list(self.collection.aggregate(query)) if query else
self.collection.find()
```

- Para cerrar la conexión con el servicio se creó una función **close** para evitar el gasto de recursos innecesarios.

```
def close(self):  
    self.client.close()
```

Ejercicio solicitado con MongoDB:

Creé el objeto de mongo y utilicé el método **upload()** en el para subir el df anteriormente cargado y procesado. Utilizando la librería **datetime** consulté el tiempo de procesamiento de inicio a fin:

```
#MongoDB:  
start = datetime.now()  
print("Subiendo datos a MongoDB...")  
mongo = Mongo()  
mongo.upload(df) finish  
= datetime.now()  
print("Tiempo de subida a MongoDB:", finish - start, "segundos", '\n')
```

Una vez cargada la información en MongoDB procedí a realizar las consultas solicitadas:

Cuál es la categoría más vendida?

Utilizando la función **request_data** aplicada al objeto mongo y con argumento el query observado puedo hacer una consulta a **category_id** y **price** para sumar cada precio por categoría y así encontrar la categoría con más ventas, sort -1 lo que hace es ordenarlo de forma descendente y limit 1 solo deja que me de un resultado que contiene ambos datos, category_id y price. Para mostrarlo utilizo indexación y keys para solicitar el dato concreto:

```
start = datetime.now()  
ventas_mongo = mongo.request_data([  
    {"$group": {"_id": "$category_id", "total_sales": {"$sum": "$price"}}},  
    {"$sort": {"total_sales": -1}},  
    {"$limit": 1}])  
finish = datetime.now()  
print("Tiempo de ejecución para la marca más vendida:", finish - start)  
print("Marca con más vendida:", ventas_mongo_brand[0]['_id'], "con ventas totales de:  
", '$', ventas_mongo_brand[0]['total_sales'], '\n')
```

Resultado:

Subiendo datos a MongoDB...

Tiempo de subida a MongoDB: 0:00:12.488443 segundos

Tiempo de ejecución para la categoría más vendida: 0:00:00.941413 segundos

Categoría más vendida: 2.268105428166509e+18 con ventas totales de: \$ 102364774.54

Cuál marca (brand) generó más ingresos brutos?

Utilizando nuevamente la función **request_data** introduzco un query específico con el id **brand** y **price** para encontrar la marca con más ingresos:

```
start = datetime.now()  
ventas_mongo_brand = mongo.request_data([
```

```

    {"$group": {"_id": "$brand", "total_sales": {"$sum": "$price"}}},
    {"$sort": {"total_sales": -1}},
    {"$limit": 1}})
finish = datetime.now()
print("Tiempo de ejecución para la marca más vendida:", finish - start)
print("Marca con más vendida:", ventas_mongo_brand[0]['_id'], "con ventas totales de: ", '$', ventas_mongo_brand[0]['total_sales'], '\n')

```

Resultado:

```

Tiempo de ejecución para la marca más vendida: 0:00:01.105650
Marca con más vendida: samsung con ventas totales de: $ 90052821.66

```

Qué mes tuvo más ventas? (En UTC)

En este caso el query convierte el key **event_time** en un string con el formato año / mes y así calcular cual fue el que tuvo mas ventas en comparacion con otros:

```

start = datetime.now()
ventas_mongo_month = mongo.request_data([
    {"$group": {"_id": {"$dateToString": {"format": "%Y-%m", "date": "$event_time"}},
    "total_sales": {"$sum": "$price"}},
    {"$sort": {"total_sales": -1}},
    {"$limit": 1}})
finish = datetime.now()
print("Tiempo de ejecución para el mes con más ventas:", finish - start, "segundos")
print("Mes con más ventas:", ventas_mongo_month[0]['_id'], "con ventas totales de: ", '$', ventas_mongo_month[0]['total_sales'], '\n')
mongo.close()

```

Por ultimo cierro la conexión con MongoDB. Resultado:

```

Tiempo de ejecución para el mes con más ventas en Redis: 0:00:00.593356 segundos
Mes con más ventas: 2020-08 con ventas totales de: $ 52825929.10000873

```

Clase para manejar Redis:

Cree una clase para manejar **Redis** de la misma manera que **MongoDB**, creando una funcion de inicializacion con todos los parametros necesarios para crear el cliente, seguido una funcion de **upload** donde puedo subir la informacion a Redis y por ultimo una funcion de **request_data** para solicitar la informacion deseada.

- Clase **Redis** con funcion de inicializacion:
 - En este caso se configura el host y el puerto de conexion creando client para insertar y consultar datos en Redis:

```

class Redis:
    def __init__(self, host='localhost', port=6379):
        import redis
        self.client = redis.Redis(host=host, port=port)

```


- Funcion upload:

- Redis utiliza un identificador unico para cada record guardado, despues de dar formato a los datos en tipo lista diccionario se procesa uno por uno dando una llave y guardandolo en formato json (por row). Ademas utilice la funcion pipeline para ayudar con la gran cantidad de datos que se debian cargar en Redis, primero proceso todos los datos y despues ejecuto el pipeline para que guarde los datos, asi el servicio no debe ir y esperar una respuesta por cada record guardado. Los metodos de guardado de datos eran muy similares, lo unico que cambia es el esquema que quisiera dar a la db, en mi caso escogi **set** para almacenar cada fila como un JSON, otras alternativas:

- **hset(key, mapping)**: Acceso por campo, estructura de fila.

- **rpush("lista", json)**: Almacena muchos registros secuenciales.

-
-

```
def upload(self, data):
    self.client.flushall() # Limpiar la base de datos antes de
    insertar nuevos datos
    pipe = self.client.pipeline()
    data = data.to_dict('records')
    for i, item in enumerate(data):
        pipe.set(f"item:{i}", json.dumps(item, default=str))
    pipe.execute()
```

- Para evitar duplicados en el ejercicio como debia probar cada vez que realizaba modificaciones al codigo agregue una linea donde limpia la db para evitar colisiones.
- Pipeline mejoro mucho el rendimiento del servicio, antes de utilizarlo tenia mucho tiempo de procesamiento por la enorme cantidad de datos a cargar.

- Funcion request_data:

- Cree una funcion de un argumento **key_pattern** el cual va tener por defecto todos los keys. El bucle va buscar todos los keys que concidan con el key_pattern dado, o en su defecto todos los keys. Una vez finalizada la lista de keys, la funcion **mget()** utiliza como argumento esa lista y devuelve todos los valores de dichas llaves. Para finalizar la funcion retorna una lista con todos los valores a diccionarios decodificados a strings.

```
def request_data(self, key_pattern='item:*'):
    cursor = 0
    all_keys = []

    # Escanea claves que coincidan con el patrón
    while True:
        cursor, keys = self.client.scan(cursor=cursor, match=key_pattern,
        count=1000)
        all_keys.extend(keys)
        if cursor == 0:
            break

    if not all_keys:
        return []
```

```

# Obtener todos los valores en un solo paso
values = self.client.mget(all_keys)

# Decodificar y deserializar JSON
return [
    json.loads(val.decode("utf-8"))
    for val in values if val is not None
]

```

- Para cerrar la conexión con el servicio se creó una función **close** para evitar el gasto de recursos innecesarios.

```

def close(self):
    self.client.close()

```

Ejercicio solicitado con Redis:

Cree el objeto de mongo y utilice el método **upload()** en el para subir el df anteriormente cargado y procesado. Utilizando la librería **datetime** consulte el tiempo de procesamiento de inicio a fin:

```

start = datetime.now()
print("Subiendo datos a Redis...")
redis_client = Redis()
redis_client.upload(df)
finish = datetime.now()
print("Tiempo de subida a Redis:", finish - start, "segundos", '\n')

```

Una vez cargada la información en Redis procedí a realizar las consultas solicitadas:

Cuál es la categoría más vendida?

Utilizando la función **request_data** aplicada al objeto **redis_client** guarde toda la consulta en una variable llamada **ventas_redis**, seguido cree una iteración en esa variable donde almacene las categorías en una variable y el precio en otra. Después, una condicional verifica si existe la categoría que se guardó en la variable **category** y le añade el precio en el diccionario **ventas_redis_grouped**, para así llevar el conteo de ventas por cada categoría. Una vez estén todos los records revisados la variable

ventas_redis_sorted suma cada precio por categoría, lo almacena y lo ordena de forma descendente para que la categoría con más ventas quede de primera en la lista.

```

start = datetime.now()
ventas_redis = redis_client.request_data()
ventas_redis_grouped = {}
for item in ventas_redis:
    category = item.get('category_id')
    price = item.get('price', 0)
    if category in ventas_redis_grouped:
        ventas_redis_grouped[category] += price
    else:
        ventas_redis_grouped[category] = price

```

```

ventas_redis_sorted = sorted(ventas_redis_grouped.items(), key=lambda x: x[1],
reverse=True)
finish = datetime.now()
print("Tiempo de ejecución para la categoría más vendida en Redis:", finish - start,
"segundos")
if ventas_redis_sorted:
    print("Categoría más vendida:", ventas_redis_sorted[0][0], "con ventas totales de:
", '$',ventas_redis_sorted[0][1], '\n')

```

Resultado:

```

Subiendo datos a Redis...
Tiempo de subida a Redis: 0:00:28.832761 segundos
Tiempo de ejecución para la categoría más vendida en Redis: 0:00:17.560933 segundos
Categoría más vendida: 2.268105428166509e+18 con ventas totales de: $ 102364774.54003766

```

Cuál marca (brand) generó más ingresos brutos?

Utilizando nuevamente la funcion **request_data** realizo el mismo procesamiento anterior pero con los items de **brand** y **price** para encontrar la marca con mas ingresos:

```

start = datetime.now()
ventas_redis_brand = {}
for item in ventas_redis:
    brand = item.get('brand')
    price = item.get('price', 0)
    if brand in ventas_redis_brand:
        ventas_redis_brand[brand] += price
    else:
        ventas_redis_brand[brand] = price
ventas_redis_brand_sorted = sorted(ventas_redis_brand.items(), key=lambda x: x[1],
reverse=True)
finish = datetime.now()
print("Tiempo de ejecución para la marca más vendida en Redis:", finish - start)
if ventas_redis_brand_sorted:
    print("Marca con más vendida:", ventas_redis_brand_sorted[0][0], "con ventas
totales de: ", '$',ventas_redis_brand_sorted[0][1], '\n')

```

Resultado:

```

Tiempo de ejecución para la marca más vendida en Redis: 0:00:00.399284
Marca con más vendida: samsung con ventas totales de: $ 90052821.66002268

```

Qué mes tuvo más ventas? (En UTC)

En este caso busco el key **event_time** en un string, si encuentra una fecha entonces busca el precio y lo añade al diccionario para realizar el calculo:

```

start = datetime.now()
ventas_redis_month = {}
for item in ventas_redis:
    event_time = item.get('event_time')
    if event_time:
        month = event_time[:7] # Formato YYYY-MM
        price = item.get('price', 0)
        if month in ventas_redis_month:
            ventas_redis_month[month] += price

```

```

        else:
            ventas_redis_month[month] = price
        ventas_redis_month_sorted = sorted(ventas_redis_month.items(), key=lambda x: x[1],
            reverse=True)
        finish = datetime.now()
        print("Tiempo de ejecución para el mes con más ventas en Redis:", finish - start,
            "segundos")
    if ventas_redis_month_sorted:
        print("Mes con más ventas:", ventas_redis_month_sorted[0][0], "con ventas totales
            de: ", '$', ventas_redis_month_sorted[0][1], '\n')
    redis_client.close()

```

Resultado:

```

Tiempo de ejecución para el mes con más ventas en Redis: 0:00:00.593356 segundos
Mes con más ventas: 2020-08 con ventas totales de: $ 52825929.10000873

```

Clase para manejar Hbase:

Cree una clase para manejar **Hbase** de la misma manera que **MongoDB** y **Redis**, creando una funcion de inicializacion con todos los parametros necesarios para crear el cliente, seguido una funcion de **upload** donde puedo subir la informacion a Redis y por ultimo una funcion de **request_data** para solicitar la informacion deseada.

- Clase **Hbase** con funcion de inicializacion:
 - En este caso se configura el host creando **table** para insertar y consultar datos en Hbase. Es muy importante verificar que la tabla se este creando ya que puede dar errores a la hora de subir la informacion al servicio.

```

def __init__(self, host='localhost'):
    import happybase
    self.connection = happybase.Connection(host=host)
    self.table = self.connection.table('my_table')

```

- Funcion de upload:
 - Hbase tiene su propio esquema de como manejar la informacion al subirla, comparada a las otras db al trabajar con ella se siente mas rigida. Nuevamente los datos deben de ir en formato JSON o lista de diccionarios, en esta funcion se utilizan 2 argumentos, **data** y **batch_size**, por defecto inicia en 10000. Al ser tantos datos los que se van a subir al servicio decidi utilizar batch, el cual ejecuta en batches la subida de informacion en la base de datos, esto agiliza el procesamiento. Por otra parte el esquema que utiliza hbase es de la siguiente manera: llave(b'cf:columna:valor...') Asi que antes de subir la informacion es necesario dar ese formato a los datos asi como tambien convertir los datos en formato json bytes para que acepte los datos.

```

def upload(self, data, batch_size=10000):
    cleaned_data = self.clean_data_for_hbase(data)
    data = cleaned_data.to_dict('records')
    with self.table.batch(batch_size=batch_size) as batch:

```

```

for i, item in enumerate(data):
    row_key = f'row{i}'

    encoded_data = {b'cf:' + k.encode('utf-8'):
                     v if isinstance(v, bytes) else str(v or '').encode('utf-
8')}

    for k, v in item.items():
        batch.put(row_key.encode('utf-8'), encoded_data)

```

- Funcion request_data:

- La funcion de request es muy similar a Redis, se utiliza **scan** para buscar una clave especifica, y si no se especifica la clave devolvera toda la coleccion de datos:

```

def request_data(self, row_prefix=None):
    if row_prefix:
        rows = self.table.scan(row_prefix=row_prefix.encode('utf-8'))
    else:
        rows = self.table.scan()
    return rows

```

Ejercicio solicitado con Hbase:

Cree el objeto de mongo y utilice el metodo **upload()** en el para subir el df anteriormente cargado y procesado. Utilizando la libreria **datetime** consulte el tiempo de procesamiento de inicio a fin:

```

start = datetime.now()
print("Subiendo datos a HBase...")
hbase_client = HBase()
hbase_client.upload(df)
finish = datetime.now()
print("Tiempo de subida a HBase:", finish - start, "segundos", '\n')
data = hbase_client.request_data()

```

Una vez cargada la informacion en Redis procedi a realizar las consultas solicitadas:

Muy similar a redis, hbase no tiene consultas nativas para el analisis de datos, por lo que debo solicitar toda la informacion de la db y procesarla por aparte en un ciclo extrayendo la informacion que necesito para posteriormente realizar mi analisis.

```

category_sales = defaultdict(float)
brand_sales = defaultdict(float)
sales_by_month = defaultdict(float)

for _, row in data:
    row_data = {k.decode('utf-8'): v.decode('utf-8') for k, v in row.items()}

    category = row_data.get('cf:category_id')
    sales = row_data.get('cf:price')
    brand = row_data.get('cf:brand')

```

```

date = row_data.get('cf:event_time')

if category and sales:
    try:
        category_sales[category] += float(sales)
    except ValueError:
        continue # Ignorar valores no numéricos
if brand and sales:
    try:
        brand_sales[brand] += float(sales)
    except ValueError:
        continue # Ignorar valores no numéricos
if date and sales:
    try:
        date_obj = datetime.strptime(date, '%Y-%m-%d %H:%M') # o el formato que
estés usando
        month_str = date_obj.strftime('%Y-%m')
        sales_by_month[month_str] += float(sales)
    except ValueError:
        continue

```

Este código procesa en paralelo las 3 consultas solicitadas en el ejercicio separando la información requerida en 3 diccionarios diferentes

Cuál es la categoría más vendida?

```

category_most_sold = max(category_sales.items(), key=lambda x: x[1])
print('La categoría con mas ventas fue', category_most_sold[0], 'con: $',
category_most_sold[1])

```

Resultado:

Tiempo de subida a HBase: 0:01:13.958577 segundos

Tiempo de procesamiento para análisis: 0:00:39.453153 segundos.

La categoría con mas ventas fue 2.268105428166509e+18 con: \$ 102364774.54003564

Cuál marca (brand) generó más ingresos brutos?

```

brand_most_sold = max(brand_sales.items(), key=lambda x: x[1])
print('La categoría con mas ventas fue', brand_most_sold[0], 'con: $',
brand_most_sold[1])

```

Resultado:

La categoría con mas ventas fue samsung con: \$ 90052821.6599907

Qué mes tuvo más ventas? (En UTC)

```

best_month = max(sales_by_month, key=sales_by_month.get)
print('El mes con mas ventas fue:', best_month[0], 'con: $', best_month[1])

```

Resultado:

El mes con más ventas fue: 2020-08 con: \$ 52825929.10001593

Conclusiones:

Los resultados solicitados fueron identicos, sin embargo la manera en que se procesan y suben los datos a cada db es muy diferente. Por ende los tiempos de procesamiento fueron muy diferentes.

MongoDB fue el que obtuvo el mejor rendimiento con respecto a los demas, tanto en subida como en consultas.

Por obvias razones cada una de ellas fue pensada para un fin especifico y por esos motivos no todas se comportan o responden de la misma manera. Dentro de los aspectos claves que pude observar que marca una gran diferencia al comparar cada base de datos es la facilidad de procesar la informacion y analizarla.

Mientras que para MongoDB existe una gama amplia para la consulta y el analisis de datos, Redis y Hbase son muy poco maleables en ese aspecto. Son mis primeras impresiones pero ya por el hecho de que debo crear ciclos para procesar la informacion y obtener analisis de los datos solicitados hace que sea un factor clave en velocidad y respuesta.

Este ejercicio me parecio excelente para comprender como trabaja cada servicio y observar como es trabajar con grandes cantidades de data.

- Me encanto trabajar con MongoDB, fue muy flexible y me recordo mucho a SQL en el sentido de que puedo manipular mis consultas para obtener resultados especificos sin necesidad de yo realizar un procesamiento externo.
- Redis me parecio un ejemplo de como utilizar algo genial de la peor manera, ya que estaba procesando millones de datos a la vez en memoria cuando unicamente iba a utilizar unas cuantas columnas para realizar mi analisis. Esto me ensena que son cosas que deberia tener en consideracion si necesitara utilizar este servicio.
- Hbase fue una pesadilla, desde la configuracion, errores sin descripcion y poco feedback de parte del cliente. Tuve problemas desde que inicie la imagen, debido a que estuve intentando utilizar Thrift como sugiere en algunos casos.

MacOS necesita algunas dependencias para poder trabajar con ese servicio y al no estar hacia que la imagen se cayera. Una vez resolví, con ayuda del profesor, el tema de la imagen empece a tener problemas con la insercion de datos a la tabla, no me mencionaba nada acerca que no existiera la tabla donde estaba intentando insertar mis datos, si no que me daba directamente un error de tipo de datos. Logre percatarme que era la tabla porque empece a realizar un debug del cliente para corroborar que todo estuviera inicializandose correctamente, una vez me di cuenta que no se estaba creando dicha tabla revise el codigo y force la creacion de la misma; una vez hecho esto comenzo a trabajar con normalidad.

Bibliografia:

Pymongo:

- <https://pymongo.readthedocs.io/en/stable/>

Redis:

- <https://redis.readthedocs.io/en/stable/>

Hbase:

- <https://happybase.readthedocs.io/en/latest/>