PRACTICA FINAL GVD Y SPARK ML

GRANDES VOLUMENES DE DATOS

Luis Cabello, Javier Rafael Cubas, Christian Jonathan Yang, Mario Gutiérrez, Mario Redondo y Rafael Picón

Índice

1. Introducción

Esta es una memoria final en la que detallamos las resoluciones a los ejercicios planteados como entrega final de la asignatura Grandes Volúmenes de Datos en el curso 24/25.

A lo largo del curso hemos obtenido una serie de conocimientos acerca de los grandes volúmenes de datos que ahora hemos tenido que demostrar de manera práctica. Nuestra actividad, también conocida como LogMaster contaba de dos tareas a realizar:

- **Análisis de Logs**, es una tarea en la que debíamos introducir 4 parámetros de entrada y obtener los hostnames que se habían conectado a un hostname dado.
- **Tiempo Real**, era una actividad en la que debíamos desarrollar una herramienta capaz de analizar datos en tiempo real. Estos datos eran hostnames ya estructurados en el log y hostnames que podían ir entrando a medida que pasaba el tiempo.

Para el desarrollo de la actividad hemos decidido utilizar lo siguiente:

- **Lorca** lo hemos utilizado debido a que tenía una versión de Python que cumplía con los requisitos (3.10.12) y a la que todos teníamos acceso gracias a los usuarios que nos proporcionaron.
- **Apache Spark,** una potente herramienta vista en clase que se utiliza para el procesamiento distribuido. Además, hemos decidido aprender un poco más acerca de esta herramienta y hemos usado DataFrames. Esto son estructuras de datos distribuidas de manera similar a una tabla de una base de datos. Esto facilita la manipulación, consultas y análisis de grandes volúmenes de datos de una manera eficiente y escalable.
- **Github,** era un requisito indispensable para poder tener un registro de las modificaciones de cada miembro del equipo. Por ello, se creó un fork para trabajar sobre la actividad y los miembros realizamos commits a medida que avanzamos. Una vez se dio por terminado el trabajo, se realizó una pull request al repositorio principal.

Para completar las tareas se ha seguido una estructura que nos permitía reutilizar código y la que se sigue para explicar la solución.

2. Archivo utils.py

Este primer archivo utils.py lo podemos encontrar dentro de una carpeta llamada utils. Esta carpeta contiene funciones que se han utilizado en ambos códigos y por lo tanto que hemos reutilizado en lugar de desarrollarlas dos veces. A continuación, definimos las diferentes funciones utilizadas.

2.1. Importaciones de bibliotecas

Comenzamos con la importación de las bibliotecas.

```
import os
import sys
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StructField, LongType, StringType
```

- **os y sys:** Estas bibliotecas permiten manejar archivos y gestionar errores relacionados con la existencia o ausencia de archivos.
- **pyspark.sql:** Contiene las clases y métodos necesarios para configurar sesiones de Spark y definir estructuras de datos.

2.2. Creación del a sesión de spark

```
def crear_sesion_spark(app_name, memoria_ejecutor, memoria_driver, particiones_shuffle):
    return SparkSession.builder.appName(app_name).config(
        "spark.executor.memory", memoria_ejecutor
    ).config(
        "spark.driver.memory", memoria_driver
    ).config(
        "spark.sql.shuffle.partitions", particiones_shuffle
    ).getOrCreate()
```

La función “crear_sesion_spark” se encarga de inicializar una sesión de Spark personalizada con los parámetros proporcionados:

- **app_name:** Nombre de la aplicación Spark.
- **memoria_ejecutor:** Memoria asignada para los ejecutores de Spark.
- **memoria_driver:** Memoria asignada para el driver de Spark.
- **particiones_shuffle:** Número de particiones que se utilizarán en operaciones de shuffle.

Esta función asegura que la sesión Spark esté configurada según las necesidades específicas del proyecto.

2.3. Definición del esquema de datos

La función “definir_esquema” define un esquema para estructurar los datos del archivo CSV. Este esquema nos va a ayudar a manejar el archivo ‘input-file-10000.txt’ como una tabla y por lo tanto como un DataFrame.

```
def definir_esquema():
    return StructType([
        StructField("timestamp", LongType(), True),
        StructField("hostname_origen", StringType(), True),
        StructField("hostname_destino", StringType(), True)
    ])
```

Estructura del esquema:

- **timestamp:** Marca temporal de tipo entero.
- **hostname_origen:** Nombre del host de origen de tipo cadena.
- **hostname_destino:** Nombre del host de destino de tipo cadena.

Este esquema facilita la validación y organización de los datos leídos desde el archivo de entrada.

2.4. Definición del esquema de datos

La función “verificar_archivo” comprueba si el archivo proporcionado existe en el sistema de archivos y si el archivo no se encuentra en la ruta especificada, se imprime un mensaje de error y el programa finaliza, para evitar errores si los hubiese.

```
def verificar_archivo(input_file):
    if not os.path.exists(input_file):
        print(f'El archivo introducido: {input_file} no existe')
        sys.exit(1)
```

2.5. Definición del esquema de datos

La función “cargar_datos_csv” utiliza Spark para leer datos de un archivo txt con un esquema predefinido. De esta manera, como hemos mencionado en el apartado de definir el esquema, podemos convertir el archivo en un Data Frame.

```
def cargar_datos_csv(spark, input_file, schema):
      return  spark.read.schema(schema).csv(input_file,  sep="  ",
header=False)
```

En cuanto a los parámetros de entrada, recibe la sesión de spark, la ruta del archivo que vamos a convertir en DataFrame y el schema que va a tener. Además, en la línea del return podemos identificar dos parámetros más:

- **sep:** Con este parámetro especificamos el tipo de separados que tiene el archivo.
- **header:** Solo admite True y False y se utiliza para indicar si el csv tiene cabecera.

3. Herramienta de análisis de log

En este apartado se detalla la implementación del ejercicio 1, cuyo propósito es analizar archivos de logs. Como nos encontramos en la asignatura de Big Data y debemos mostrar los conocimientos que hemos aprendido en la asignatura además de generar unos códigos que sean óptimos para el manejo de grandes volúmenes de datos, hemos decidido, como hemos mencionado anteriormente, utilizar Apache Spark.

Esta herramienta cuenta con 3 ficheros identificados. El primero el análisis_log.py, un fichero en el que se ha creado la función análisis_log que representa la solución del problema. Además, cuenta con un archivo test.py en el que se han ejecutado 7 pruebas unitarias y de esta manera hemos podido comprobar el correcto funcionamiento de nuestra función y por último un main.py en el que hemos hecho una pequeña prueba con los parámetros enunciados en README.md proporcionado por el profesor.

3.1. Análisis_log.py

3.1.1. Función configuración local

```
def configurar_locale():
    try:
        locale.setlocale(locale.LC_TIME, 'es_ES.UTF-8')
    except locale.Error:
        locale.setlocale(locale.LC_TIME, 'Spanish_Spain.1252')
```

Al realizar el ejercicio, nos dimos cuenta de que nos daba problema de formato las fechas que introducíamos como parámetros. Esto se debía a que las fechas de ejemplo que se nos proporcionaban estaban en español y el sistema estaba en inglés.

Para resolver este problema, se creó esta función que se asegura de cambiar el formato de fecha del sistema para que no tengamos problemas.

3.1.2. Conversión de fechas

```
def convertir_fechas(init_datetime, end_datetime, input_format):
    init_dt = datetime.strptime(init_datetime, input_format)
    end_dt = datetime.strptime(end_datetime, input_format)
    return int(init_dt.timestamp() * 1000), int(end_dt.timestamp() * 1000)
```

Dado que el log nos proporciona las fechas en timestamps, lo que hacemos es formatear las fechas de entrada a un timestamp de manera que podamos realizar luego los filtros de las operaciones utilizando estas fechas.

Por lo tanto, esta función se encarga de primero convertir la fecha a timestamp y luego asegurarnos que está en el mismo formato de 12 dígitos o milisegundos.

3.1.3. Filtrados de datos

En este apartado adjuntamos las funciones que hemos utilizado para filtrar los datos que obteníamos y empezar a acercarnos al resultado.

- **Tiempo:**

```
def filtrar_datos_por_tiempo(df, init_timestamp,
end_timestamp):
    return df.filter(
        (F.col("timestamp") >= init_timestamp) &
        (F.col("timestamp") <= end_timestamp)
    )
```

Esta primera función se utiliza para filtrar el DataFrame con los parámetros de entrada proporcionados. El objetivo es quedarlos con los datos que se encuentran dentro del rango de tiempo que el usuario quiere filtrar.

- **Hostname:**

```
def filtrar_por_hostname(df, hostname):
    target_host = hostname.lower()
    return df.filter(
        (F.lower(F.col("hostname_origen")) == target_host) |
        (F.lower(F.col("hostname_destino")) == target_host)
    )
```

Este es el segundo filtro que aplicamos en el que el objetivo es obtener todos aquellos datos en lo que el hostname_origen o el hostname_destino sean iguales al que proporciona el usuario en los parámetros de entrada.

3.1.4. Obtención de host conectados

```
def obtener_hosts_conectados(df, hostname):
    target_host = hostname.lower()
    df_distinct_origin_host = df.filter(
        F.lower(F.col('hostname_origen')) != target_host
    ).select(F.col('hostname_origen').alias('host'))

    df_distinct_destino_host = df.filter(
        F.lower(F.col('hostname_destino')) != target_host
    ).select(F.col('hostname_destino').alias('host'))

    df_result = df_distinct_origin_host.unionByName(
        df_distinct_destino_host).distinct()
    return [row['host'] for row in df_result.toLocalIterator()]
```

Esta sería la última función que utilizaremos en nuestra función principal. El objetivo es obtener un array de todos aquellos hosts que se han conectado al que el usuario ha proporcionado. Para ello, creamos dos DataFrame:

- **df_distinct_origin_host:** donde nos quedamos con todos aquellos nombres de host que sean diferentes al que nos han introducido. Una vez se han filtado, se hace un select de la única columna que nos interesa y se le cambia el nombre a host.
- **df_distinct_destino_host:** Se realiza lo mismo que en el punto anterior, pero con los hosts destino.

Una vez tenemos ambos DataFrames creados, aplicamos un unionByName que se encarga de unir ambas tablas, una seguida de la otra por el nombre de la columna y selecciona los distintos. Por ejemplo:

df_distinct_origin_host
Host1
Host2
Host3
Host4

df_distinct_destino_host
Host1
Host2
Host5
Host6

RESULT
Host1
Host2
Host3
Host4
Host5
Host6

Por último, utilizamos .toLocalIterator() para obtener todos los resultados en un array. Este es un método que, en cuanto a recursos, es mejor que el .collect() que puede ser más conocido. La principal diferencia es que el LocalIterator accede a los datos manteniendo una pequeña parte en la memoria local mientras que el collect baja todos a memoria local para leerlos.

3.1.5. Análisis principal

```
def analisis_log(input_file, init_datetime, end_datetime,
hostname):
    verificar_archivo(input_file)

    spark = crear_sesion_spark("analisisLog", "4g", "4g", "200")
    schema = definir_esquema()
    df_log_csv = cargar_datos_csv(spark, input_file, schema)

    configurar_locale()

    input_format = "%A, %d de %B de %Y %H:%M:%S"
    init_timestamp, end_timestamp = convertir_fechas(
        init_datetime, end_datetime, input_format)

    df_filtered = filtrar_datos_por_tiempo(
```

```
        df_log_csv, init_timestamp, end_timestamp)
    df_filtered = filtrar_por_hostname(df_filtered, hostname)

    hosts = obtener_hosts_conectados(df_filtered, hostname)

    spark.stop()

    return hosts
```

Esta función ejecuta el flujo completo del análisis: carga el archivo, configura Spark, filtra los datos según tiempo y host, y obtiene los hosts conectados. Finaliza cerrando la sesión de Spark y devolviendo los resultados

3.2. Main.py

Este archive main es que hemos creado para ejecutarlo y comprobar que todo funciona antes de crear las pruebas unitarias. Cuenta con los imports, una función main y el main definido que correría cuando ejecutas el programa.

```
import sys
import os
sys.path.append(os.path.dirname(os.path.abspath(__file__)) + '/../')
from analisis_log import analisis_log
```

- **sys y os:** Se utilizan para manipular rutas del sistema y permitir la importación del archivo analisis_log.py desde un directorio superior.
- **analisis_log:** Importa la función principal del análisis desarrollada en el archivo correspondiente.

Definimos la función principal “main()”:

```
def main():
    inputFile = '../input-file-10000.txt'
    init_datetime = 'Martes, 13 de agosto de 2019 01:00:00'
    end_datetime = 'Martes, 13 de agosto de 2019 21:00:00'
    target_host = 'Savhannah'

     result = analisis_log(inputFile, init_datetime, end_datetime,
target_host)

    print(result)
```

Esta función main es la que vamos a llamar al final y es la función en la que declaramos las variables de entrada que nos solicita el enunciado. Ejecutamos la función con las misma e imprimimos el resultado.

Y por último ejecutamos el script principal antes definido “main()”:

```
if __name__ == "__main__":
    main()
```

Este flujo asegura que el programa lea el archivo, analice los datos y muestre los hosts conectados en el rango de tiempo especificado.

3.3. Test.py

En este archivo hemos introducido pruebas unitarias. Era uno de los requisitos que debía de tener la entrega y la hemos llevado a cabo. Para poder hacer estas pruebas unitarias, lo que hemos hecho ha sido utilizar la librería unittest. Es un marco de pruebas unitarias en Python que permite crear, organizar y ejecutar pruebas automatizadas para asegurar la funcionalidad y la calidad del código.

Nosotros hemos realizado 7 pero solo vamos a mostrar una, para que el documento no quede muy extenso. La estructura es siempre la misma solo modificamos los parámetros de entrada de la función y el output esperado.

Para poder realizar las pruebas, hemos tenido que crear una clase:

```
class TestAnalisisLog(unittest.TestCase):
```

Dentro de esta clase se realizan las pruebas con la siguiente estructura:

```
def test_analisis_log(self):
        """
        Primera prueba unitaria para la función analisis_log.
        """

        input_file = '../input-file-10000.txt'
        init_datetime = 'Martes, 13 de agosto de 2019 01:00:00'
        end_datetime = 'Martes, 13 de agosto de 2019 21:00:00'
        target_host = 'Savhannah'

        expected_result = [
            'Deyshawn', 'Rumaldo', 'Jaylien', 'Zarriyah', 'Shaynee',
            'Demarius', 'Borna', 'Elmir', 'Michelene', 'Ajee',
'Tanisha'
        ]

        result = analisis_log(input_file, init_datetime,
end_datetime, target_host)

        self.assertEqual(result, expected_result)
```

Como podemos ver, cada prueba unitaria debe de tener un nombre de función, los 4 parámetros de entrada que requiere nuestra función y el output esperado. Después de esto

se ejecuta nuestra función creada en el otro archivo y con ayuda de la librería comprobamos que sea correcto el resultado o no. En caso de no serlo nos devuelve el resultado que es y el que nosotros esperábamos y en caso de estar correcto devuelve un ok.

4. Herramienta de Tiempo Real

En este apartado se detalla la implementación del ejercicio 2, cuyo propósito es procesar logs en tiempo real utilizando Apache Spark y la biblioteca **Watchdog** para detectar cambios en un archivo. Para ello, se implementan las funciones y clases necesarias.

4.1. Análisis real_time

4.1.1 Clase LogFileHandler

4.1.1.1 Función __init__

```
def __init__(self, input_file, spark, hostname):
        self.input_file = input_file
        self.spark = spark
        self.hostname = hostname.lower()
        self.last_timestamp = int(
            (datetime.now() -
timedelta(hours=1)).timestamp() * 1000)
```

- Inicializa un manejador de eventos para procesar un archivo de logs.
- Almacena el DataFrame de PySpark (`df_log`) y el hostname configurado.
- Define un timestamp inicial que filtra los datos de la última hora.

4.1.1.2 Funcion process_logs

```
def process_logs(self):
        schema = definir_esquema()
        self.df_log = cargar_datos_csv(self.spark,
self.input_file, schema)

        df_last_hour = self.df_log.filter(
            F.col("timestamp") >= self.last_timestamp)

        connected_to_host = df_last_hour.filter(
            F.lower(F.col("hostname_destino")) ==
self.hostname
        ).select("hostname_origen").distinct()

        connected_from_host = df_last_hour.filter(
            F.lower(F.col("hostname_origen")) ==
self.hostname
        ).select("hostname_destino").distinct()

        most_connections =
df_last_hour.groupBy("hostname_origen").count().orderBy(
```

```
            F.col("count").desc()
        ).limit(1)

        print("\n--- Resultados de la última hora ---")
        print("Hostnames conectados al host configurado:")
        connected_to_host.show()

        print("Hostnames que recibieron conexiones del host
configurado:")
        connected_from_host.show()

        print("Hostname con más conexiones en la última
hora:")
        most_connections.show()

        self.last_timestamp = int(
            (datetime.now() - timedelta(hours=1)).timestamp()
* 1000)
```

1. Filtra logs de la última hora usando `self.last_timestamp`.
2. Calcula:
    - Los hostnames que conectaron al hostname configurado.
    - Los hostnames que recibieron conexiones desde el hostname configurado.
    - El hostname con más conexiones.
3. Imprime los resultados en consola.
4. Actualiza `self.last_timestamp` para procesar futuros cambios en tiempo real.

4.1.1.3 Función on_modfied

```
def on_modified(self, event):
        if event.src_path == self.input_file:
            print(f"Archivo modificado: {event.src_path}")
            self.process_logs()
```

- Este método se activa cuando el archivo observado es modificado.
- Comprueba si el archivo cambiado es el archivo de entrada y, si es así, llama a process_logs.

4.1.2 Función monitor_log_file

```
def monitor_log_file(input_file, spark, hostname):
    event_handler = LogFileHandler(input_file, spark,
hostname)
    observer = Observer()
    observer.schedule(event_handler, path=os.path.dirname(
        input_file), recursive=False)
    observer.start()
    print(f"Monitorizando cambios en {input_file}...")
    try:
```

```
        while True:
            time.sleep(1)
    except KeyboardInterrupt:
        observer.stop()
    observer.join()
```

- Crea un manejador de eventos (`LogFileHandler`) para el DataFrame de logs y el hostname configurado.
- Configura un observador (`Observer`) para detectar cambios en el archivo de logs.
- Mantiene el programa en ejecución mientras monitorea los cambios.
- Detiene el observador si se interrumpe manualmente (`Ctrl + C`).

4.1.3 Función real_time

```
def real_time(input_file, hostname):
    verificar_archivo(input_file)

    spark = crear_sesion_spark("realtime", "4g", "4g", "200")
    monitor_log_file(input_file, spark, hostname)

    spark.stop()
```

- Verifica si el archivo de entrada existe.
- Crea una sesión de Spark con los parámetros configurados.
- Define el esquema del DataFrame y carga los datos del archivo de logs.
- Llama a monitor_log_file para iniciar la monitorización en tiempo real.
- Finaliza cerrando la sesión de Spark.

4.2. Análisis main.py

El archivo `main.py` sirve como punto de entrada para ejecutar el código desarrollado en `real_time.py`.

4.2.1 Código main.py

```
import sys
import os
sys.path.append(os.path.dirname(os.path.abspath(__file__)) +
'/../')
from real_time import real_time
def main():
    input_file = './input-file-10000.txt'
    hostname = 'Prueba'
    real_time(input_file, hostname)

if __name__ == "__main__":
    main()
```

Importaciones:

- Usa `sys` y `os` para manipular rutas del sistema.
- Importa la función principal `real_time` desde el archivo correspondiente.

Función `main()`:

- Define las variables:
  - `input_file`: Archivo de logs a analizar.
  - `hostname`: Hostname que se analizará.
- Llama a la función `real_time` para ejecutar el procesamiento.

Ejecución:

- Comprueba si el script se está ejecutando como principal (`if __name__ == '__main__':`).
- Si es así, ejecuta la función `main()`.

4.3. Análisis test.py

4.3.1. Importacion de módulos

```
import time
import random
import sys
import os
```

time: Proporciona funciones relacionadas con el tiempo.
random: Permite seleccionar valores aleatorios.
sys y os: Ayudan a manipular el sistema operativo, como rutas de archivos o directorios.

```
sys.path.append(os.path.dirname(os.path.abspath(__file__))  +
'/../')
from utils.utils import verificar_archivo
```

Se agrega una ruta para importar módulos personalizados desde un directorio padre.
Se importa una función desde utils.utils

4.3.2. Lista de nombres de hosts

```
HOSTNAMES = ["Alpha", "Beta", "Gamma", "Delta", "Savhannah",
      "Zeta", "Epsilon", "Theta"]
```

Lista de nombres de host que representa los puntos de origen de las conexiones.

4.3.3. Función principal: generar_conexiones

```
def generar_conexiones(file_path, num_conexiones=10,
intervalo=1):
    """
    Genera un archivo de logs con conexiones ficticias.

    Args:
        file_path (str): Ruta del archivo donde se guardarán
los registros.
        num_conexiones (int): Número de conexiones a generar.
        intervalo (int): Intervalo en segundos entre cada
conexión generada.
    """
```

```
    print(f"Generando {num_conexiones} conexiones ficticias
en '{file_path}'...")

    verificar_archivo(file_path)

    with open(file_path, 'a') as log_file:
        for _ in range(num_conexiones):
            timestamp = int(time.time() * 1000)

            origen = random.choice(HOSTNAMES)
            destino = "Prueba"

            log_line = f"\n{timestamp} {origen} {destino}"

            log_file.write(log_line)

            print(f"Conexión generada: {log_line.strip()}")

            time.sleep(intervalo)
```

Aqui tenemos la funcion completa que ahora iremos desglosando y explicando poco a poco.

```
def generar_conexiones(file_path, num_conexiones=10,
intervalo=1):
```

Para comenzar, recibe 3 argumentos de entrada; file_path, para tener la ruta al archivo donde se guardan los logs; num_conexiones, para saber la cantidad de conexiones a generar; e intervalo, para saber el tiempo en segundos entre la generación de cada conexión.

```
verificar_archivo(file_path)
with open(file_path, 'a') as log_file:
    for _ in range(num_conexiones):
```

Tras ello, llama a la función verificar_archivo para garantizar si este existe, lo abre en modo de adición y ejecuta un bucle.

```
timestamp = int(time.time() * 1000)

origen = random.choice(HOSTNAMES)
destino = "Prueba"

log_line = f"\n{timestamp} {origen} {destino}"

log_file.write(log_line)

print(f"Conexión generada: {log_line.strip()}")

time.sleep(intervalo)
```

Dentro de este bucle comienza obteniendo una marca de tiempo en milisegundos y después, selecciona un nombre de host de la lista HOSTNAME de forma aleatoria fijando un destino llamado “Prueba”.
A continuación, construye la línea del log con el formato <timestamp> <origen> <destino> y lo escribe en el archivo de logs.
Para finalizar, imprime por consola la línea generada y espera antes de generar la siguiente conexión.

4.3.4. Bloque principal

```
if __name__ == "__main__":
    archivo_logs = "input-file-10000.txt"
    conexiones = 1
    intervalo_segundos = 2

    generar_conexiones(archivo_logs, conexiones,
      intervalo_segundos)
```

Este bloque comienza estableciendo una condición de que se ejecute solo si es el archivo principal, no un módulo importado.
Tras ello, define la ruta del archivo de logs, establece el número de conexiones a generar y configura el intervalo entre conexiones.
Para finalizar, llama a la función generar_conexion la cual hemos explicado antes ya con los parámetros definidos.