Digital Forensics: Lab 4a

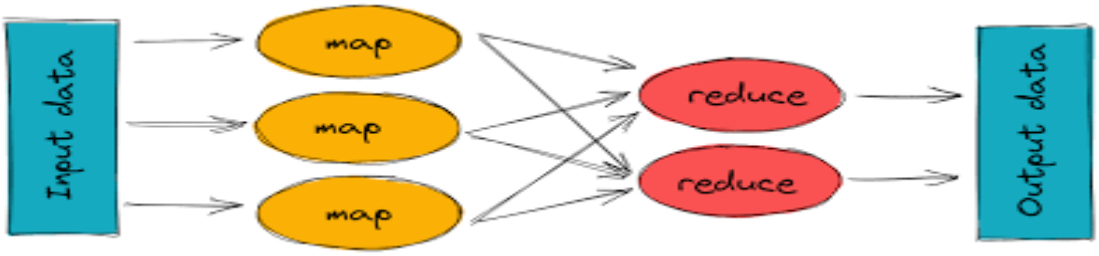
Aim: Design a distributed application using MapReduce which processes log file of a system. List out users who have logged for maximum period on the system.

Tools Used:

- Databricks
- MapReduce
- PySpark

Theory:

MapReduce is a programming model and an associated implementation for processing and generating big data sets with a parallel, distributed algorithm on a cluster. A MapReduce program is composed of a map procedure, which performs filtering and sorting, and a reduce method, which performs a summary operation.



The term "MapReduce" refers to two separate and distinct tasks that Hadoop programs perform. The first is the map job, which takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (key/value pairs).

The reduce job takes the output from a map as input and combines those data tuples into a smaller set of tuples. As the sequence of the name MapReduce implies, the reduce job is always performed after the map job.

MapReduce programming offers several benefits to help you gain valuable insights from your big data:

- Scalability: Businesses can process petabytes of data stored in the Hadoop Distributed File System (HDFS).
- Flexibility: Hadoop enables easier access to multiple sources of data and multiple types of data.
- Speed: With parallel processing and minimal data movement, Hadoop offers fast processing of massive amounts of data.
- Simple: Developers can write code in a choice of languages, including Java, C++ and Python.

Code:

PART-A

The code reads a log file, extracts IP addresses and timestamps, calculates the session duration for each IP, and prints the top 5 sessions by duration. It also prints the total session duration across all users.

```
import re
from datetime import datetime
from pyspark.sql.functions import udf, col
from pyspark.sql.types import StringType, TimestampType
from collections import defaultdict

df1 = spark.read.format("csv").option("header",
"false").load("dbfs:/FileStore/shared_uploads/kevalpanwala08@gmail.com/access_log_
short.csv")

# Define a UDF to parse the log lines
def parse_log_line(line):
    pattern = r'(\d+\.\d+\.\d+\.\d+) - - \[(.*?)\] "(.*?)"'
    match = re.match(pattern, line)
    if match:
        ip_address = match.group(1)
        timestamp_str = match.group(2).split(" ")[0]
        timestamp = datetime.strptime(timestamp_str, "%d/%b/%Y:%H:%M:%S")
        return ip_address, timestamp
    return None, None

# Register the UDF with Spark
parse_log_line_udf = udf(lambda line: parse_log_line(line)[0], StringType())
parse_log_timestamp_udf = udf(lambda line: parse_log_line(line)[1],
TimestampType())

# Apply the UDF to extract the IP address and timestamp
df2 = df1.withColumn("ip_address", parse_log_line_udf(col("_c0"))) \
```

```
        .withColumn("timestamp", parse_log_timestamp_udf(col("_c0"))) \
        .select("ip_address", "timestamp")

# Filter out rows where parsing failed (i.e., ip_address or timestamp is null)
df2 = df2.filter(col("ip_address").isNotNull() & col("timestamp").isNotNull())

# Convert the DataFrame to an RDD for session calculation
logs_rdd = df2.rdd.map(lambda row: (row["ip_address"], row["timestamp"]))

# Group logs by IP address and calculate session durations
user_sessions = logs_rdd.groupByKey().mapValues(list).collectAsMap()
user_durations = {}
for ip, timestamps in user_sessions.items():
    timestamps.sort()
    session_duration = (timestamps[-1] - timestamps[0]).total_seconds()
    user_durations[ip] = session_duration

sorted_user_durations = sorted(user_durations.items(), key=lambda x: x[1],
reverse=True)
for ip, duration in sorted_user_durations[:5]:
    duration_minutes = duration / 60
    print(f"IP: {ip}, Total Session Duration: {duration_minutes:.2f} minutes")

# Calculate the total session duration across all users
total_minutes = sum(user_durations.values()) / 60
print(f"Total session duration across all users: {total_minutes:.2f} minutes")
```

PART-B

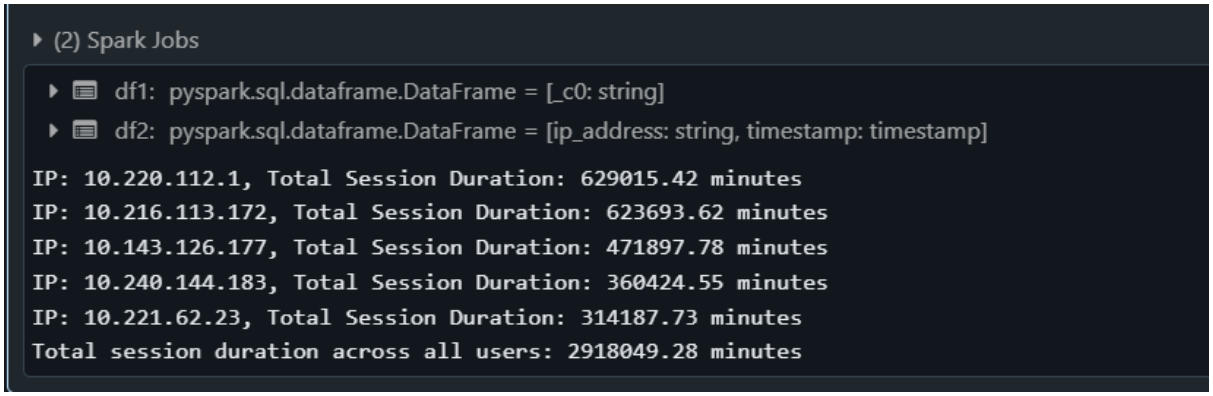
The code reads a log file, extracts IP addresses and timestamps, counts the occurrences of each IP address, sorts them by frequency, and displays the top IPs by the number of occurrences. The result can also be saved to a CSV file if needed.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import udf, col
from pyspark.sql.types import StringType, TimestampType
import re
from datetime import datetime

spark = SparkSession.builder.appName("LogAnalyzer").getOrCreate()
log_df = spark.read.format("csv").option("header",
"false").load("dbfs:/FileStore/shared_uploads/nivdshi2003@gmail.com/access_log_sho
rt.csv")
```

```
# Define the function to parse log lines
def parse_log_line(line):
    pattern = r'(\d+\.\d+\.\d+\.\d+) - - \[(.*?)\] "(.*?)"'
    match = re.match(pattern, line)
    if match:
        ip_address = match.group(1)
        timestamp_str = match.group(2).split(" ")[0]
        timestamp = datetime.strptime(timestamp_str, "%d/%b/%Y:%H:%M:%S")
        return ip_address, timestamp
    return None, None

# Define UDFs for extracting IP and timestamp
def extract_ip(line):
    ip, _ = parse_log_line(line)
    return ip

def extract_timestamp(line):
    _, ts = parse_log_line(line)
    return ts

# Register UDFs with Spark
extract_ip_udf = udf(extract_ip, StringType())
extract_timestamp_udf = udf(extract_timestamp, TimestampType())

# Apply UDFs to the DataFrame to get separate IP and timestamp columns
parsed_df = log_df.withColumn("ip_address", extract_ip_udf(col("_c0"))) \
                  .withColumn("timestamp", extract_timestamp_udf(col("_c0")))

parsed_df = parsed_df.filter(col("ip_address").isNotNull())

ip_frequency_df = parsed_df.groupBy("ip_address").count()

sorted_ip_frequency_df = ip_frequency_df.orderBy("count", ascending=False)

sorted_ip_frequency_df.show(truncate=False)
```

Output:

PART-A



PART-B