Understanding Spark & Kafka

Pauravi Nagarkar (pnagarkar@scu.edu, W1650209)
Tanmay Singla (tsingla@scu.edu, W1644134)

Goal

1. To implement spark code to determine the top 100 most Frequently occurring words and top 100 Most Frequently occurring words having more than 6 characters in a given dataset of 16GB.
2. Gain proficiency in log analytics and implement the log analytics techniques described in the article.
3. Implement a data processing and analysis pipeline using Big Data technologies, including Kafka (producer and consumer), Spark Streaming, Parquet files, and HDFS File System. The objective is to process and analyze log files by establishing a flow that begins with a Kafka producer, followed by a Kafka consumer that performs transformations using Spark.

Problem Statement

Problem 1: Develop Spark code to determine the top 100 most frequently occurring words and the top 100 most frequently occurring words with more than 6 characters in a given dataset of 16GB.

Problem 2: Gain proficiency in log analytics.

1. Create a summary report that identifies the endpoint with the highest number of invocations on a specific day of the week, along with the corresponding count of invocations, for the entire dataset.
2. Find the top 10 years with the least occurrences of 404 status codes in the dataset and provide the corresponding count of 404 status codes for each year.

Problem 3: Implement a data processing and analysis pipeline using Big Data technologies, including Kafka (producer and consumer), Spark Streaming, Parquet files, and HDFS File System. The pipeline should process and analyze log files, starting with a Kafka producer, followed by a Kafka consumer performing transformations using Spark. The transformed data should be converted into Parquet file format and stored in the HDFS.

Bonus Problem: The bonus task requires the development of a clustering algorithm capable of grouping requests based on several factors, including the host that invoked it, the time at which the endpoint was accessed, the status code received, and the data size of the returned information.

System Configurations

- **Apple MacBook Pro**
    - Processor: Apple M1 chip.
    - Cores: 8 number of cores.
    - Memory: 8 GB RAM available on the system.
    - Storage Type: SSd storage with 256 GB storage space
    - Storage size available: Available disk space.
    - Operating System: macOS Ventura
    - Programming Language: Python3
- **Samsung Galaxy Notebook 7**
    - Processor: Intel i7 11th Gen.
    - Cores: 10 number of cores.
    - Memory: 16 GB RAM available on the system.
    - Storage Type: SSd storage with 512 GB storage space
    - Storage size available: Available disk space.
    - Operating System: Windows 11
    - Programming Language: Python3

Programming Model

Apache Spark : Apache Spark is an open-source distributed data processing framework that provides fast and scalable big data analytics. One of the core abstractions in Spark is Resilient Distributed Datasets (RDD), which is an immutable and fault-tolerant collection of objects spread across a cluster. RDDs offer a high-level API for parallel and distributed data processing, enabling users to perform operations like transformations and actions on distributed data. In addition to RDDs, Spark introduced DataFrames, which provide a higher-level abstraction built on top of RDDs. DataFrames offer a structured and schema-based approach to working with data, allowing for efficient and optimized execution of operations. With its powerful capabilities for distributed processing and the flexibility of RDDs and DataFrames, Spark has become a popular choice for processing large-scale datasets and performing complex analytics tasks.

- **RDD:** RDD, short for Resilient Distributed Dataset, is the fundamental data structure in Apache Spark. It represents an immutable distributed collection of objects that can be processed in parallel across a cluster of computers. RDDs provide fault tolerance, high-level data abstractions, and efficient distributed processing capabilities.
- **DataFrame :** In Apache Spark, DataFrame is a distributed collection of data organized into named columns. It provides a high-level API that allows users to perform various data manipulation and analysis operations in a distributed and parallel manner. Under the hood, DataFrames are built on top of the Spark SQL engine, which leverages the power of Spark's distributed computing capabilities.

Problem 1: K Most Popular Words Using Spark

We have implemented the code using both RDD's and DataFrames in Spark in Python. The program counts the frequency of each word in a text file while excluding stopwords. Below are the flow chart diagrams that explains the code.

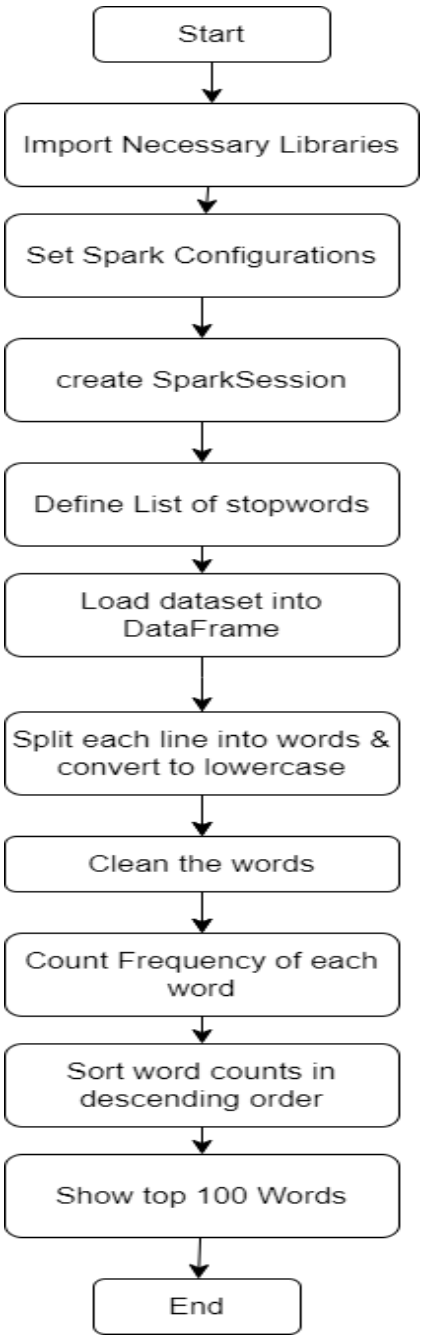

Fig: Flow Chart of Code using DataFrame Api

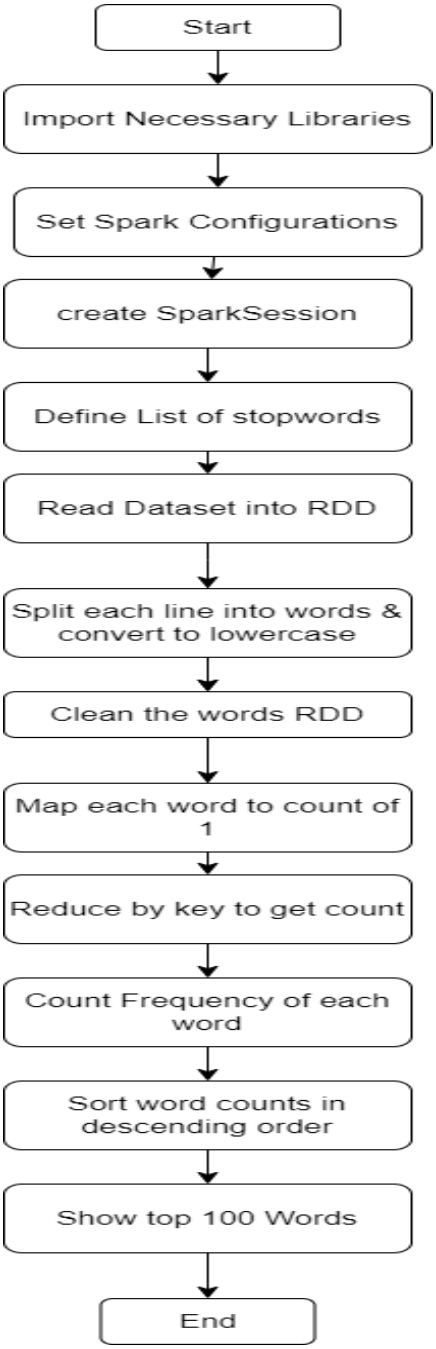

Fig: Flow Chart of Code using RDD

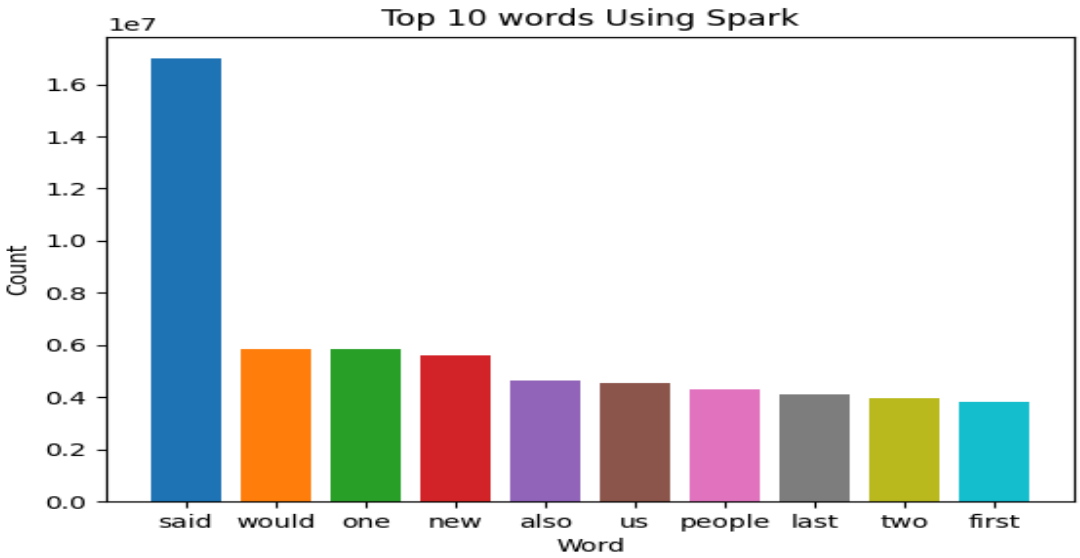

Fig. Frequency of Top 10 words using Spark

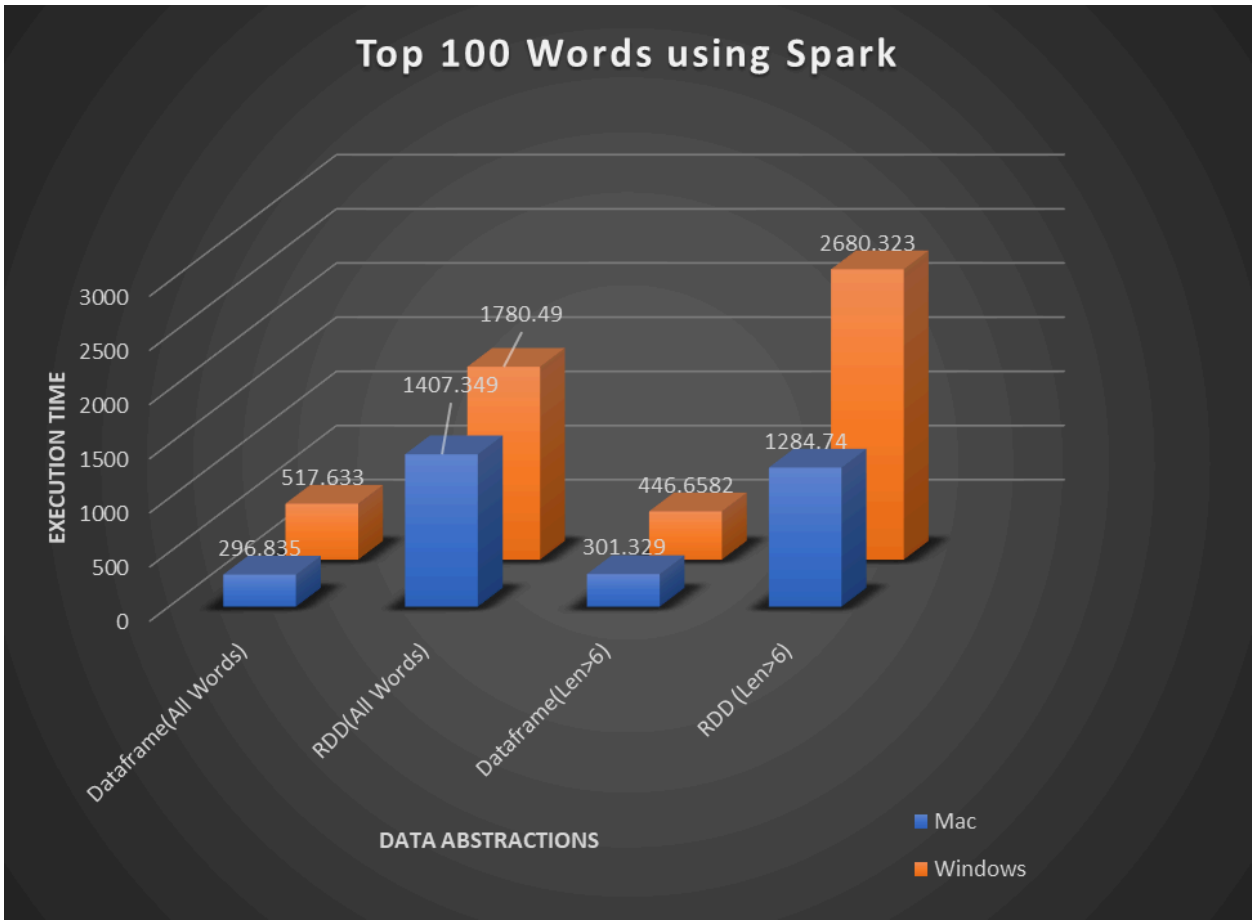

Fig. Execution Time, DataFrame vs RDD, Mac vs Windows

From the figure above, we can see that the program leveraging DataFrames runs faster than RDD in both Mac and Windows. DataFrames run faster in Apache Spark compared to RDDs for several reasons. Firstly, DataFrames benefit from optimized execution through the Catalyst optimizer. This optimizer applies various optimizations like predicate pushdown, column pruning, and code generation to generate efficient query execution plans, resulting in faster processing. Additionally, the higher-level abstraction provided by DataFrames enables more granular query optimization, allowing Spark to optimize and execute operations more efficiently, leveraging the underlying data structures and computing resources. DataFrames are also stored in a columnar format, which enhances performance for analytical workloads by enabling efficient compression, improved data locality, and vectorized processing. Moreover, Spark's Catalyst optimizer and expression evaluation engine further optimize the DataFrame operations by generating optimized bytecode and dynamically compiling code at runtime, reducing interpretation and compilation overhead. These factors collectively contribute to the superior performance of DataFrames over RDDs in Spark.

Detailed Analysis for Top 100 Words:

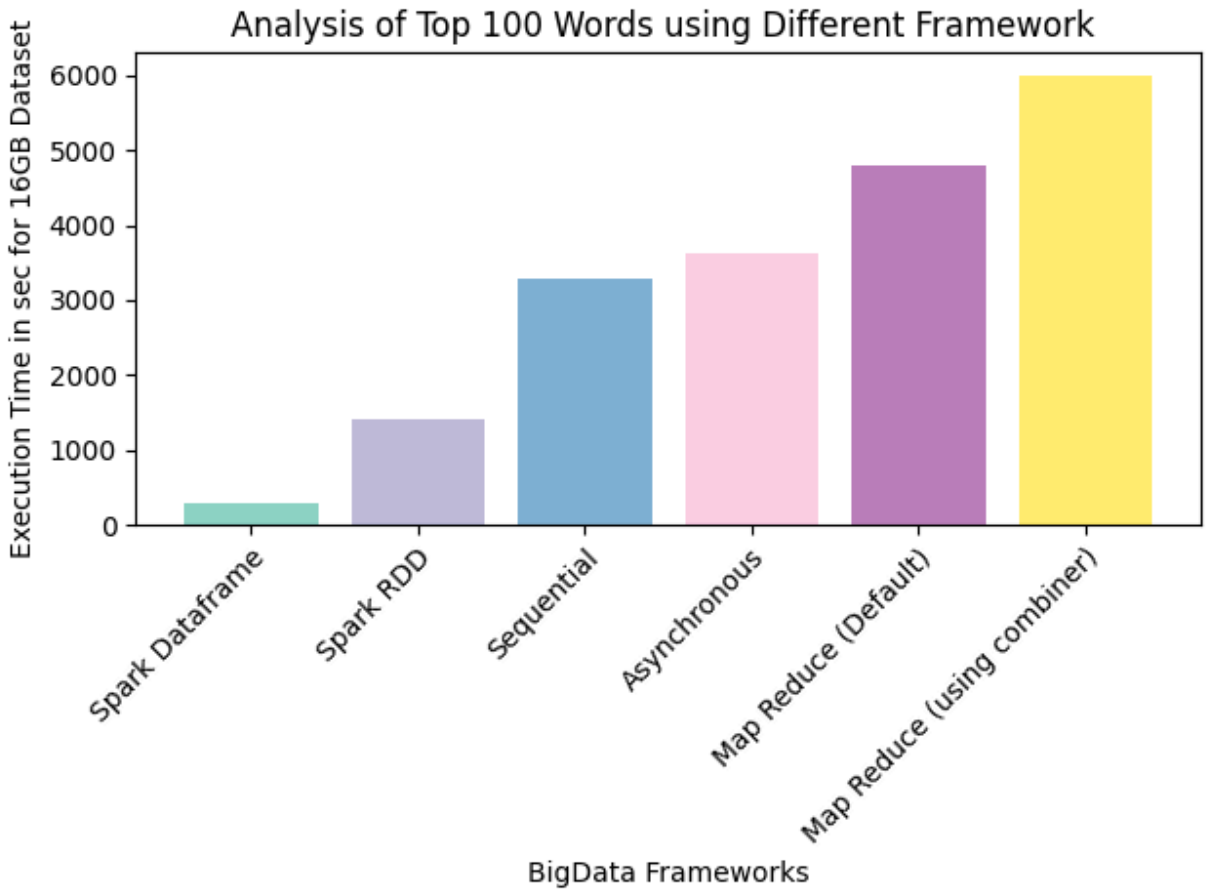

Fig Execution Time taken by each Big Data Framework

In Summary , we can see that Spark Dataframe works better than rest of the big data frameworks. In general we can make the conclusion that sequential sorting approaches have longer processing times compared to parallel processing frameworks like MapReduce and Spark. Spark is faster than traditional sequential processing approaches due to its ability to leverage distributed computing, process data in parallel across multiple machines, and divide the workload into smaller tasks executed simultaneously on different nodes. This parallelization leads to faster processing times compared to sequential execution. Additionally, Spark's in-memory data processing minimizes disk I/O operations, reducing the overhead of reading and writing data to disk and improving performance. Spark further enhances speed through optimization techniques that analyze data processing pipelines, generate optimized execution plans, and minimize unnecessary computations and data movement. Overall, Spark's distributed computing, in-memory processing, optimization techniques, and higher-level abstractions contribute to its faster performance, making it well-suited for efficient and scalable big data analytics and processing tasks.

Problem 2: Log Analysis on NASA dataset

To implement EDA on the given datasets, we utilized Spark's DataFrame Api to read the datasets as DataFrames, used regex expressions to extract meaningful values out of raw data, implemented transformations and actions to do EDA and get the desired output. Below is the flow chart diagram that explains the flow of code.

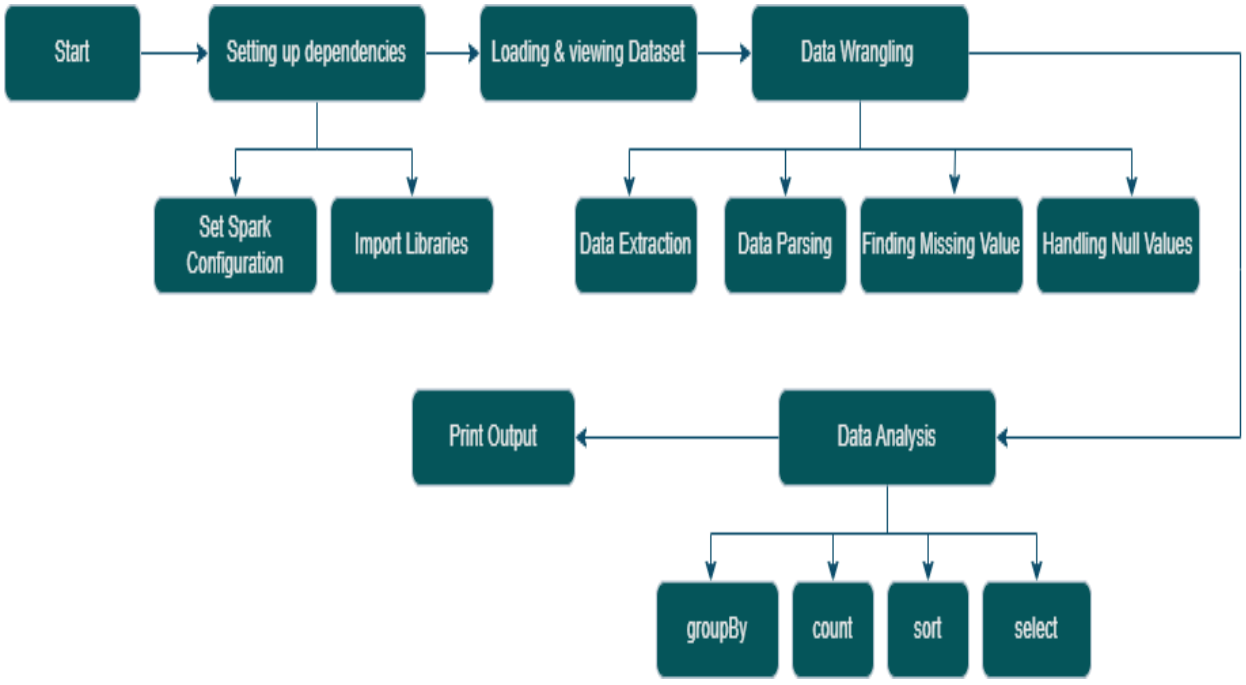

Fig. Flow chart of Log Analysis

Various Transformations and actions used to achieve the results are:-

- Transformations:
  - Filter: Selects rows based on a condition.
  - Select: Retrieves specific columns.
  - GroupBy: Aggregates data based on specified keys.
  - Join: Combines multiple DataFrames based on a common key.
  - Sort: Orders the DataFrame based on one or more columns.
- Actions:
  - Show: Displays the contents of the DataFrame.
  - Count: Returns the number of rows in the DataFrame.
  - Collect: Retrieves all the data from the DataFrame to the driver program.
  - Save: Writes the DataFrame to a specified output format or storage location.

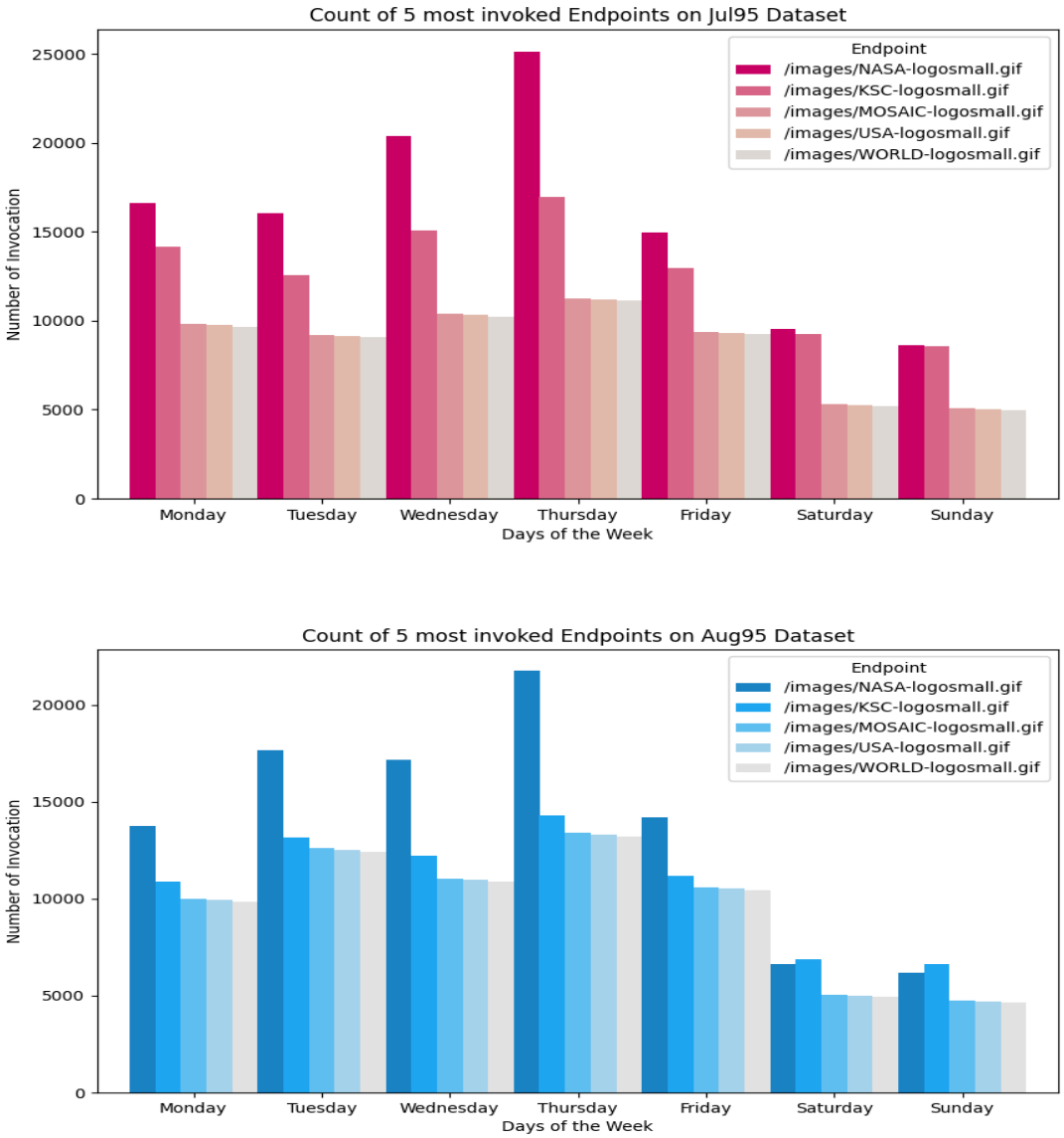

Fig. Highest Count of Endpoints by Day of the Week for NASA datasets

The above graphs presents an analysis of endpoint invocations for the Jul95 and Aug95 datasets. The data shows the frequency of invocations for endpoints on each day of the week. Upon analyzing the Jul95 dataset, it was observed that the endpoint "/images/NASA-logosmall.gif" received the highest number of invocations among all endpoints. Notably, the analysis indicated that the highest invocation count for this endpoint occurred on Thursdays, with a total count of 25,140. Following this, the endpoint "/images/KSC-logosmall.gif" ranked second with a count of 14,291 on the same day.

Further examination of the data revealed that for every day of the week in the Jul95 dataset, the endpoint "/images/NASA-logosmall.gif" consistently had the highest invocation count. This suggests a consistent and significant usage of this particular endpoint throughout the week.
Similar analysis was conducted for the Aug95 dataset to identify the endpoint invocations. Once again, it was found that the endpoint "/images/NASA-logosmall.gif" had the highest count of invocations on Thursdays, totaling 21,780. In comparison, the endpoint "/images/KSC-logosmall.gif" experienced the highest invocation counts on Saturdays (6,892) and Sundays (6,654).

In conclusion, the analysis of endpoint invocations for the Jul95 and Aug95 datasets revealed that the endpoint "/images/NASA-logosmall.gif" consistently received the highest number of invocations for each day of the week. This indicates its significant usage and popularity among users. Additionally, the endpoint "/images/KSC-logosmall.gif" demonstrated increased invocation counts on weekends, particularly on Saturdays and Sundays.

Problem 3: Spark Integration with Kafka

To integrate Spark with Kafka, we utilized Structured Streaming, a powerful feature provided by Spark that enables real-time data processing. Our goal was to leverage Kafka's high-throughput streaming capabilities and Spark's advanced analytics capabilities to perform analysis on the data consumed from Kafka topics.

Approach 1: Running Producer and Consumer Sequentially

In our initial approach, we implemented the producer and consumer to run sequentially. While this approach worked well for small datasets, it posed challenges when dealing with large datasets. Sequential processing introduced additional latency between message production and consumption, leading to delays in processing. This delay can significantly impact real-time or near-real-time applications.
Running the producer and consumer sequentially also limited the parallelism and scalability benefits offered by Kafka. The consumer could only process messages that had been produced and committed by the producer, creating a bottleneck in the pipeline. This constraint hindered the

system's overall throughput, preventing us from fully utilizing Kafka's capabilities for handling high-throughput data streaming.

Approach 2: Running Producer and Consumer Parallely in Threads

To overcome the challenges of sequential processing, we explored parallel processing using threads. By introducing multiple threads, we aimed to reduce the latency and improve overall system performance. With parallel processing, messages could be processed concurrently, minimizing the time between production and consumption.

However, during implementation, we encountered resource allocation issues. The resources allocated to the application were being divided between the threads, limiting the full potential of both Spark and Kafka. To optimize our code and address this challenge, we decided to implement the producer and consumer as separate applications.

Approach 3: Running Producer and Consumer Parallely as Separate application

In this optimized approach, we ran the producer and consumer as separate applications, allowing them to execute independently and in parallel. This ensured that each application had dedicated resources and avoided interference between their execution. By running the producer and consumer in parallel as separate applications, we were able to effectively utilize the available resources and optimize the overall execution.

We observed significant improvements in system performance, reduced latency, and increased throughput by decoupling the producer and consumer processes. This approach leveraged the strengths of both Spark and Kafka, enabling efficient data processing and analysis for real-time or near-real-time applications. Below are the flow charts for producer and consumer application:-

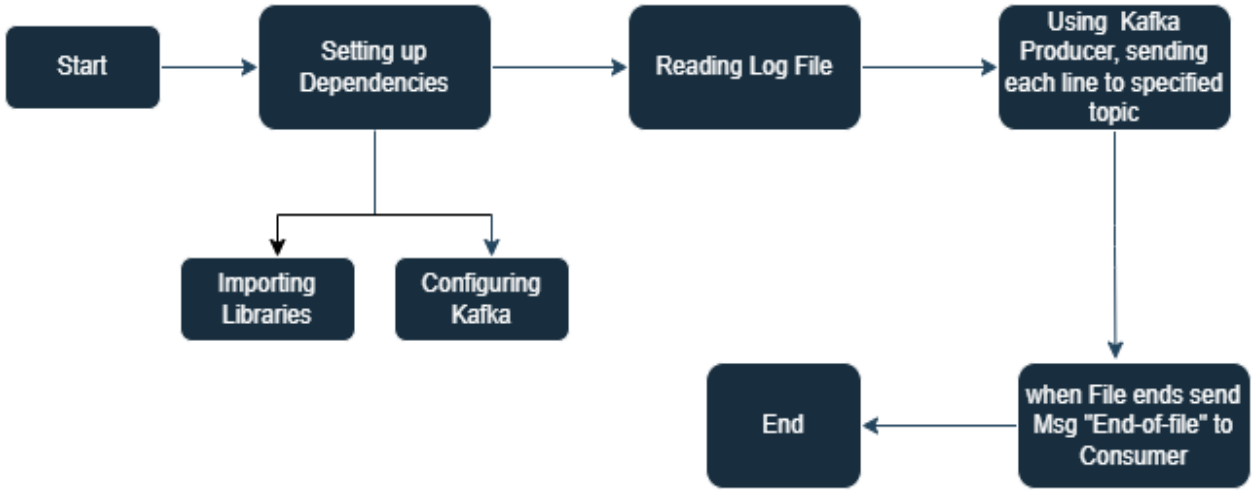

Fig. Flow Chart for Producer

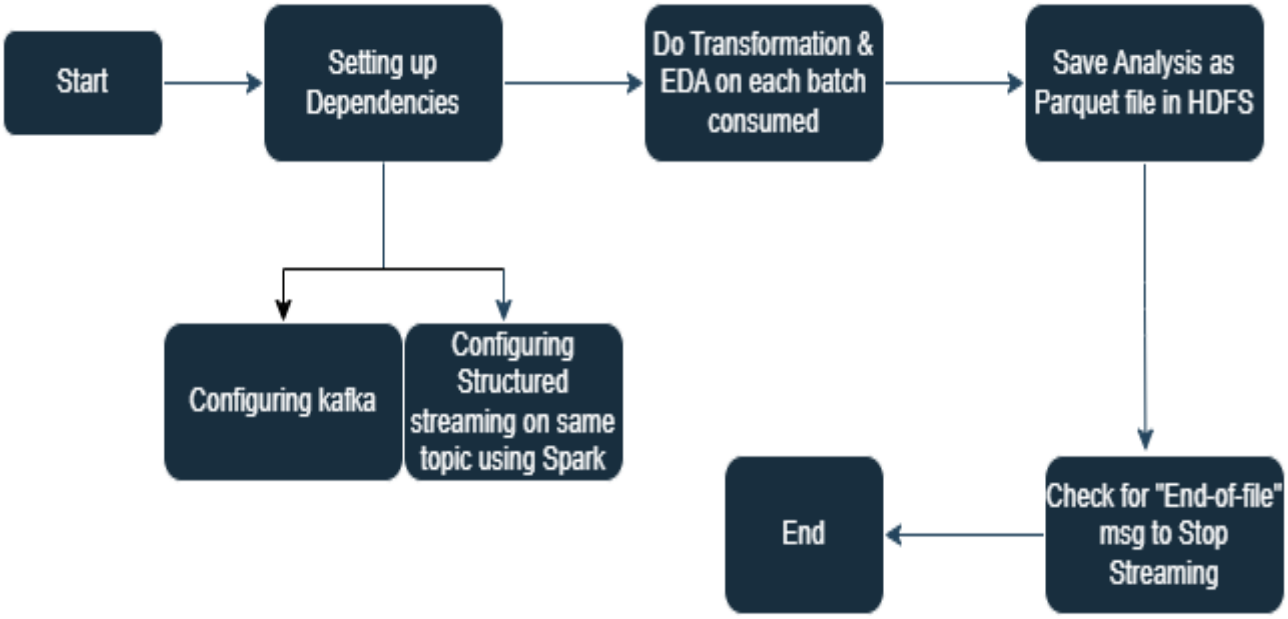

Fig. Flow chart for Consumer

Challenges Faced:

- For the Larger Dataset (17GB log file), it took too long to complete the execution. Due to which, we were unable to play around with different configurations.
- As the Log Analysis requires multiple transformations to be stored and cached as RDD's, for 17GB file, program keeps on crashing as laptop ran out of memory.
- There were multiple issues while installing Kafka on Windows. Finally, for windows Kafka was installed on Docker to run the program.

Bonus Problem:

We have implemented a K-means clustering algorithm to group requests based on several factors. K-means is a popular clustering algorithm that partitions data into a predefined number of clusters. In our case, we have chosen Method and Status Codes as the parameters for clustering. These parameters were selected because their values are distinctive and can effectively differentiate between different types of requests.

To use the Method column, which is a string column, we have created an index for each unique value of the column. This allows us to represent the categorical values numerically. Additionally, we consider the Status Codes as a numerical parameter. Both the Method index and Status Codes are combined to form a feature vector called 'features', which is used to train the K-means model. The training process of the model occurs only for the First Batch of requests. During this phase, the K-means algorithm iteratively partitions the data into clusters, aiming to minimize the distance between data points within each cluster. The resulting clusters represent distinct groups based on the similarities between requests in terms of Method and Status Codes.

For subsequent batches of requests, the trained model is used to map new requests to pre-established clusters. When a new request is received, the Method and Status Codes are extracted from the request and transformed into a feature vector. This feature vector is then assigned to the nearest cluster using the trained K-means model.

References:

- https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#input-sources.
- https://towardsdatascience.com/scalable-log-analytics-with-apache-spark-a-comprehensive-case-study-2be3eb3be977
- https://spark.apache.org/docs/latest/ml-clustering.html
- https://spark.apache.org/docs/latest/api/python/index.html