Big Data: Assigment 3 report

Juan Zuluaga, Adam Graham

Departments of Computer Science
Santa Clara University, Santa Clara, California, USA
{jzuluaga,ajgraham}@scu.edu

1 Introduction

This report presents the analysis and implementation of log analytics and data processing using Apache Spark in a Big Data environment.

Part 1 focuses on analyzing text data and determining the most frequent/repeated words in a 16GB dataset. The dataset is processed using Spark, considering both all words and only words with more than six characters. This section aims to familiarize with Spark and explore text data analysis techniques.

Part 2 consists of two sections. In Section A, log analytics techniques are implemented following the guidelines provided in the referenced article. Exploratory data analysis (EDA) is performed, and specific analyses from the article, such as finding the endpoint with the highest number of invocations on a specific day of the week and identifying the top 10 years with the least 404 status codes, are implemented. This section aims to gain insights from log data and understand log analytics techniques using Spark.

Section B of Part 2 introduces a combination of Big Data technologies, including Kafka, Spark Streaming, Parquet files, and HDFS. The objective is to process and analyze log files by establishing a data flow that starts with a Kafka producer, followed by a Kafka consumer performing transformations using Spark. The transformed data is converted into Parquet file format and stored in HDFS. This section highlights the utilization of real-time data processing and storage techniques in a distributed environment.

Overall, this assignment provides an opportunity to apply Spark and Big Data technologies to analyze and process log data, extract insights, and explore advanced techniques such as clustering.

2 PART 1

In Problem 1, the task was to analyze a large text file to extract the top 100 most frequently occurring words and the top 100 words with more than six characters. For this task, Apache Spark, a distributed data processing system, was utilized due to its ability to handle large volumes of data and perform computations in parallel across multiple nodes. The key section of the analysis process is coded in Python using PySpark, a Python library for Spark programming. Here is an in-depth analysis of the code:

The function QuestionOne takes three arguments - exemem, drmem, and resultsize. These arguments are used to configure the Spark application, specifying the executor memory, driver memory, and maximum result size respectively.

The SparkSession is initialized with the various configurations:

- 'spark.executor.memory' specifies the amount of memory allocated to each executor process. In this case, it's set to exemem, which is '10g' by default.
- 'spark.driver.memory' specifies the amount of memory allocated to the driver process. In this case, it's set to drmem, which is '2g' by default.
- 'spark.driver.maxResultSize' specifies the maximum size of a serialized result of all partition data that should be collected to the driver program. Here, it's set to resultsize, which is '3g' by default.
- The master node is set to local mode using .master("local[*]"). Local mode means the Spark application runs in a single JVM and can use all available cores with "[*]".
- 'spark.serializer' is set to org.apache.spark.serializer.KryoSerializer for efficient serialization of objects to send between spark nodes.
- 'spark.sql.shuffle.partitions' is set to 4, this determines the number of partitions to use when shuffling data for joins or aggregations.
- The analysis begins by creating a list of common English words known as "stopwords". These words are typically filtered out during text analysis because they do not contain significant meaning.

A pivotal decision in the analysis performed in Problem 1 was the choice to use the .master("local[*]") configuration during the initialization of the SparkSession. This particular setting denotes running Spark in local mode, wherein all computations are run on a single machine. The asterisk (*) is a wildcard, and in this context, it implies that Spark should utilize all the available cores on the machine to distribute the processing load. By using all available cores, Spark effectively leverages the full processing power of the machine to perform computations in parallel, providing a significant boost to the speed and efficiency of the data processing tasks. This ability to multi-thread tasks is particularly important when dealing with large datasets, as in this problem where we analyzed a large text file.

Choosing this configuration eliminated the need for manually specifying the number of cores or the amount of processing power to be allocated to the Spark application. This aspect significantly simplifies the setup process and ensures that we take full advantage of the machine's computational resources. It's important to note, however, that while this approach works well for local development or smaller datasets, for larger, real-world datasets processed across a cluster of machines, a different master would be used, such as a Spark standalone master, Mesos or YARN.

The DataFrame word_counts_all is created to count the occurrences of each word. Each line is split into words, all punctuations are removed, and all words are converted to lower case for uniformity. Stopwords and empty words are filtered out. The DataFrame is then grouped by each unique word, and the count

of each word is calculated. The DataFrame is sorted in descending order of count to obtain the most frequent words first.

The DataFrame word_counts_6char is created by filtering word_counts_all for words longer than six characters.

Finally, the top 100 rows from word_counts_all and word_counts_6char are displayed, representing the top 100 most frequent words and the top 100 most frequent words with more than six characters respectively.

2.1 Configuration Testing:

We experimented with various configurations for our Spark application, primarily focusing on the parameters spark.executor.memory, spark.driver.memory, and spark.driver.maxResultSize. However, the data, as listed above, reveals that there wasn't a significant change in the performance of our Spark application despite varying these configurations.

This could be attributed to the fact that our processing environment, Google Colab, comes with limited CPU resources. While we can configure the memory allocation for the executor, driver, and result size, these configurations only affect the application's performance when the hardware resources available exceed those allocated to the Spark application.

Since the total CPU resources in a Google Colab environment are somewhat limited, there may not be enough resources to significantly improve the Spark application's performance beyond a certain point, even if the configurations allow for more resources to be utilized. As such, our experiment primarily served to confirm that the local[*] configuration was indeed making the most out of the available resources.

For instance, when looking at the execution times, the fastest time recorded was approximately 97.41 seconds, while the slowest was around 139.53 seconds. While this time change is significant, this dropoff leveled off after changing the ram from 1GB to 2GB, suggesting that increasing the resource allocation did not substantially speed up our processing tasks. Further, the percentage of CPU utilization was consistently high across all configurations, ranging from 89.4

In conclusion, while configuring resource allocation in a Spark application is important and can lead to significant performance improvements, the effectiveness of such configurations is largely dependent on the total resources available. In resource-constrained environments like Google Colab, these improvements might be less noticeable. In such cases, other performance tuning methods such as optimizing code, refining data partitions, or using more efficient data structures might be more beneficial.

3 PART 2

3.1 Section A

In this problem, we had to first explore and analyze how Spark worked, get familiar with the installation, how load the files, and work with streaming tasks.

The nature of the data set itself for this problem was complex since it was somehow unstructured since was mostly logs of the traffic in a network. This script we were provided was a great guide on how to preprocess data of that nature and convert it to a proper data frame that we can actually use to perform further exploratory data analysis using PySpark's DataFrame API.

After exploring this script and learning how to do the most basic operations on Spark. We worked on the additional questions that were requested in the problem statement.

First, we explored the log files by grouping the data by the day of the week and endpoint, and counting the invocations. This analysis helped us identify the endpoint with the highest count for a day of the week, shedding light on the most frequently accessed endpoints on a specific day. The endpoint with the highest count was on Thursday with a count of 2503. Additionally, We also filtered the log files to include only records with 404 status codes, focusing on unsuccessful requests. Extracting relevant fields such as the timestamp, status code, and endpoint, we leveraged PySpark's DataFrame API to perform various analyses. In order to do so, we grouped the data by year and counted the occurrences of 404 status codes, enabling us to rank the years based on the count in ascending order. This allowed us to identify the top 10 years with the least occurrences of 404 status codes, providing insights into the performance and stability of the web service. However, it was a surprise to find out that only one year presented occurrences of 404 status codes, which was 1995 with a count of 10843.

As we were working on these two questions, we were wondering under what conditions this could be useful information for companies. We came to the conclusion that these analyses provide valuable insights for organizations to enhance their error-handling mechanisms, reduce the frequency of 404 status codes, and gain a comprehensive understanding of endpoint invocations based on specific days of the week. By identifying the years with the lowest occurrences of 404 status codes, organizations can prioritize efforts to improve web service performance during those particular years. Additionally, by recognizing the most frequently accessed endpoints on different days of the week, organizations can allocate resources effectively and make informed decisions regarding system enhancements. These analyses empower organizations to optimize their systems and deliver a better user experience.

On the other hand, we encountered that throughout this whole analysis, PySpark's distributed data processing capabilities proved effective in handling large volumes of log files. Its features enabled filtering, data extraction, aggregation, and ranking, facilitating the extraction of valuable insights. Therefore, it can be a very powerful tool for streaming tasks like these with less code, less time and with more optimal memory usage.

4 Section B

4.1 Chosen Approach

The script we wrote successfully consumes log data in a streaming fashion and perform analysis on it using PySpark, Kafka, and Hadoop. This approach is important when dealing with streaming jobs as they often run indefinitely, thus requiring the capability to process and analyze data in batches or in real-time as it arrives.

However, setting up the environment for such a process can be a non-trivial task, particularly due to the number of dependencies and services involved. The environment setup includes the installation of specific versions of OpenJDK, Kafka, PySpark, and Hadoop. The dependencies are installed using pip and apt-get package managers and may require substantial time and bandwidth, depending on the network and system specifications. Additionally, system variables need to be defined and running services verified to ensure correct operation.

Code Explanation : Dependency and Environment Setup: OpenJDK, Kafka, PySpark, and Hadoop are installed and configured. Zookeeper and Kafka servers are started and their processes verified. Finally, SparkSession is created to initiate PySpark functionality.

Data Ingestion: The script reads a log file and writes the content to a Kafka topic. Kafka is used as a message broker to handle the streaming data.

Data Extraction and Transformation: The data is then consumed from the Kafka topic and converted into a Spark DataFrame. The DataFrame is further transformed by splitting each log line into several columns based on a regular expression pattern, which matches the Common Log Format. This step transforms the raw log data into a structured format suitable for analysis.

Exploratory Data Analysis (EDA): Various analyses are performed on the structured log data such as determining the distribution of HTTP status codes and the most common endpoints accessed. Additionally, the 'time' field is converted to a standard datetime format which allows for time-based analysis.

Data Export: The transformed and structured data is saved as a Parquet file, a columnar storage file format that is highly efficient for analytic queries.

Data Migration to Hadoop File System (HDFS): The Parquet file is migrated from the local system to Hadoop's distributed storage system, HDFS. This step allows for distributed processing of the data across a Hadoop cluster, which is beneficial when dealing with large volumes of data.

4.2 Traditional Approach:

Challenges Encountered in an Ubuntu Virtual Machine Environment: The script above describes two components: a Kafka consumer and a Kafka producer. Both are programmed in Python and are part of a real-time data pipeline where the producer reads log data and publishes it to a Kafka topic, and the consumer retrieves these messages, processes them using PySpark and writes the result

to Hadoop Distributed File System (HDFS) in Parquet format. However, in an Ubuntu Virtual Machine (VM) environment, we encountered problems getting this pipeline to function correctly.

Ensuring Correct Topic Connection: A critical aspect of this data pipeline is that both the producer and consumer connect to the same Kafka topic, allowing the consumer to receive messages the producer sends. In our scenario, we ensured the topic was consistent on both ends; the topic 'logs' was used for both the producer and consumer. The Kafka producer read data from the '42MBSmallServerLog.log' file and published it to the 'logs' Kafka topic. Simultaneously, the Kafka consumer was set to read from the 'logs' topic.

Message Transmission Troubleshooting: We attempted to send messages from the producer to the consumer by running the producer in one terminal and the consumer in another. This approach is standard in Kafka messaging systems, as the producer and consumer typically operate in separate processes, possibly on separate systems. However, despite our efforts, we could not get the messaging system to work.

Ubuntu VM Environment Setup Difficulties: Setting up the environment in an Ubuntu VM presented significant challenges. Our scripts depend on multiple external services, including Kafka, PySpark, and Hadoop, all of which require proper installation and configuration. In our case, the environment's setup within the Ubuntu VM was not successful, resulting in the Kafka consumer and producer failing to function as expected. Further investigation and troubleshooting would be needed to diagnose and resolve these issues. This could involve checking Kafka server logs for any error messages, testing network connectivity between the producer, Kafka, and consumer, and ensuring that all dependencies are correctly installed and configured.

In conclusion, while our Python scripts for the Kafka consumer and producer are theoretically sound, their execution within an Ubuntu VM environment was unsuccessful due to challenges encountered during the setup process. These difficulties underscore the complexity of setting up distributed, real-time data pipelines and emphasize the importance of proper environment configuration.

Producer Script (producer.py) The purpose of this script is to read log data from a file and publish it to a Kafka topic. The KafkaProducer is initialized with the local Kafka server's address, localhost:9092. The send_to_kafka function takes a Kafka topic and a file path as parameters. It opens the file, iterates over each line, and sends each line as a separate message to the specified Kafka topic. Note that the line is stripped of any trailing newline characters and encoded to bytes before sending. The file to read (filename) and the Kafka topic to publish to (topic) are hardcoded as "/home/adamg/Desktop/HW3/42MBSmallServerLog.log" and "logs", respectively. The send_to_kafka function is then called with these parameters.

Consumer Script (consumer.py) This script retrieves messages from a Kafka topic, creates a PySpark DataFrame for each message, and writes the DataFrame to HDFS in Parquet format. A run_consumer function is defined which takes a Kafka server address and a topic as parameters. Within this function, a Kafka-

Consumer is initialized with the given topic and server address. Note that a value deserializer is provided to decode the messages from bytes to strings. A SparkSession is also created using the getOrCreate method. The function then enters an infinite loop, where it continuously checks for and processes new messages from the Kafka topic. For each message, it creates a PySpark DataFrame with a single row and column, with the column name being "log_line" and the row value being the message. The DataFrame is then written to HDFS in Parquet format. The path to write to is hardcoded as "/path/to/store/parquet/files/". After processing all messages, the function sleeps for 10 seconds before checking for new messages. The Kafka server address and topic are hardcoded as 'localhost:9092' and 'logs', respectively, and the run_consumer function is then called with these parameters.

5 Extra credit

In this bonus task, we developed a clustering algorithm to group log requests based on various factors, including the host, timestamp, status code, and data size. By leveraging the K-means clustering algorithm, we aimed to identify patterns and similarities among log entries.

During our approach, we found the features of host, time, and data size to be particularly useful in determining cluster membership. These features provided valuable insights into the source, timing, and volume of log requests, enabling effective grouping based on similar characteristics.

We faced challenges in selecting the optimal number of clusters to ensure meaningful results. Through experimentation and evaluation, we determined that a suitable number of clusters for our analysis was 5, balancing granularity and interpretability.

The developed clustering solution uncovered distinct groups within the log data, providing a deeper understanding of log patterns and facilitating targeted analysis. By organizing log entries into clusters, organizations can gain insights into common behaviors, identify anomalies, and make data-driven decisions for system enhancements and troubleshooting.

Below a description of the clusters encountared is presented:

- Cluster 1:Requests with HTTP status 200 and size ranging from 10,000 to 50,000 bytes. Mostly GET requests for HTML and text files. IP addresses in this cluster include 128.171.81.194, 74.59.229.245, and 207.189.81.199.
- Cluster 2:Requests with HTTP status 404 and size below 1,000 bytes. Mostly GET requests for non-existing resources. IP addresses in this cluster include 212.158.71.182, 80.61.114.32, and 2.110.137.72.
- Cluster3:Requests with HTTP status 500 and size above 30,000 bytes. A mix of GET, POST, and DELETE requests. IP addresses in this cluster include 99.149.129.109, 26.81.42.55, and 137.150.150.191.
- Cluster 4:Requests with HTTP status 403 and size ranging from 5,000 to 10,000 bytes. Mostly POST and DELETE requests. IP addresses in this cluster include 116.82.220.168, 198.123.235.207, and 83.211.223.129.

– Cluster 5:Requests with HTTP status 304 and size below 2,000 bytes. Mostly GET requests for resources that have not been modified.IP addresses in this cluster include 10.82.240.107, 128.171.81.194, and 211.251.53.56.

This bonus task was very useful for our knowledge since it expanded our expertise in machine learning and allowed us to explore a problem in which big data and ML intersect. On the other hand, it demonstrated the value of clustering algorithms in extracting meaningful insights from log data. By leveraging the identified features and the established clusters, organizations can enhance their log analysis capabilities and optimize system performance.

6 Challenges

Throughout this assignment, The main challenge we encountered was trying to establish a connection to Kafka. We faced difficulties in configuring the Kafka producer and consumer to connect to the Kafka broker. However, through persistent efforts and exploration of different approaches, we were ultimately able to overcome these challenges and establish a successful connection to Kafka.

Initially, we attempted to connect to a Kafka broker running on our local machine using the default configuration. However, we encountered errors such as "NoBrokersAvailable" and "UnrecognizedBrokerVersion," indicating issues with the connection. We realized that connecting to Kafka required careful configuration of various parameters, including the bootstrap servers, security protocols, and authentication credentials. Therefore we tried to use Confluent-Cloud and create a cluster that could allow us to have a connection to the service, however, many more issues came with this implementation, facing even issues with the queue being "full".

To address these challenges, we explored alternative options and libraries for Kafka integration. We installed the "kafka-python" library, which provided us with more flexibility and control over the Kafka connection. With this library, we were able to configure the producer and consumer with the appropriate settings, including the correct bootstrap servers and authentication credentials.

By leveraging the capabilities of the "kafka-python" library, we successfully established a connection to the Kafka broker. This allowed us to implement the desired functionality for producing and consuming data from Kafka. We were able to send messages to Kafka topics, consume messages from the topics, and process the data using Spark Streaming and other data processing techniques.

The experience of troubleshooting and resolving the Kafka connection issues taught us valuable lessons about the importance of thorough understanding and careful configuration when working with distributed systems like Kafka. We learned the significance of exploring alternative libraries and approaches to overcome challenges and achieve successful integration.

Besides this, we also encountered some other key challenges:

– Handling large dataset: Working with a 16GB dataset and processing log files can be computationally intensive and require efficient memory management.

We encountered challenges related to handling and processing such a large volume of data, optimizing memory usage, and ensuring timely execution.
- Data preprocessing and cleaning: Log files often contain unstructured or messy data. Preprocessing and cleaning the data to extract meaningful information and perform accurate analysis can be challenging. Dealing with different log file formats, handling missing or inconsistent data, and applying appropriate transformations can require careful data preprocessing steps.
- Implementing log analytics techniques: The log analytics section of the assignment may have presented challenges in understanding and implementing the log analytics techniques described in the reference article. Applying exploratory data analysis (EDA) techniques to log data, identifying the endpoint with the highest invocations on a specific day, and determining the top years with the least 404 status codes require careful data manipulation and analysis.
- Integration of Big Data technologies: The second part of the assignment involves working with a combination of Big Data technologies, including Kafka, Spark Streaming, Parquet files, and HDFS. Integrating these technologies and establishing a data flow pipeline can be challenging, especially if you are not familiar with these technologies. Setting up and configuring Kafka, ensuring seamless data transfer between different components, and storing data in the HDFS can require understanding and troubleshooting of the underlying technologies.

7 Conclusion

In conclusion, this assignment provided us with an opportunity to work with a combination of Big Data technologies, including Kafka, Spark Streaming, Parquet files, and HDFS. We successfully processed and analyzed log files by establishing a flow that involved a Kafka producer and consumer, Spark transformations, and the storage of transformed data in Parquet format in HDFS.

Through this assignment, we tackled challenges related to connecting to Kafka and worked diligently to overcome them by utilizing appropriate libraries and configurations. This allowed us to establish a seamless data pipeline from the log files to Kafka, enabling efficient data consumption and transformation using Spark.

We implemented various data analysis techniques, including identifying the most frequent words in the dataset and conducting exploratory data analysis on log data. Additionally, we explored the application of clustering algorithms to group log requests based on different criteria, such as host, time of access, status codes, and data size.

By leveraging machine learning and Big Data technologies, we gained valuable insights into the log data and uncovered patterns and trends that can aid in optimizing error handling mechanisms, minimizing occurrences of specific status codes, and prioritizing resource allocation. The analysis empowered us to make informed decisions regarding system enhancements and performance improvements.

Overall, this assignment provided us with hands-on experience in handling and analyzing large-scale log data, demonstrating the potential of Big Data technologies and machine learning techniques in extracting valuable insights. We have gained a deeper understanding of the challenges and complexities involved in processing and analyzing log files, and we are better equipped to apply these techniques to real-world scenarios in the future.