COEN242 PA3 - Kafka & Spark

Hui-yu Liu, Ching-Yueh Huang

I. Introduction

There are two parts in this assignment, part 1 is about using Spark to implement Top K words and part 2 is about using Spark and Kafka for log analytics. The objective is to find Top 100 frequent words and get the words with characters more than 6. By creating a Spark session, configuring log settings, and leveraging Spark's dataframe for data processing, this approach is expected to be more efficient compared to using Python alone or MapReduce function.

In Part 2, we focus on log analytics. Following the article's guidelines, we extract and transform data from a plain log file using PySpark Dataframe API and regular expressions. By utilizing dataframes, we generate the desired output according to specified requirements efficiently. Once we obtain the expected results, we proceed to work with Kafka producers and consumers, employing Spark Streaming for additional transformations to generate parquet files in HDFS.

In this report, we will start with a comprehensive description of the method used and its implementation. Additionally, we will share valuable insights into the optimization techniques and observations that were employed during the testing phases to enhance the performance of the program. We will then focus on summarizing the results obtained from the experiments, offering valuable insights into the effectiveness of our approach. Finally, we will conclude our work by discussing the implications of our findings and exploring potential avenues for future improvements.

II. Methods

A. Part 1

In Part 1, we explored various techniques in Apache Spark to efficiently perform word count. To optimize computation and memory storage, we started by configuring Spark with appropriate settings. For example, in order to allocate a significant portion of memory for caching and storing intermediate data, we increase executor memory allocation by setting 'spark.memory.fraction' and

‘spark.executor.memory’ to 0.8. After experimentation, we discovered that leveraging the DataFrame API provided the most efficient and intuitive approach for processing the data and obtaining the desired results.

The code collects data from two distinct sources: a file named 'stopword.txt' and a large dataset stored in the Hadoop Distributed File System (HDFS) under the path. 'stopword.txt' contains a list of common stopwords, which are typically excluded from word count analysis due to their high frequency and lack of meaningful insights. The main dataset in HDFS serves as the primary data source for word count analysis.

Prior to conducting the word count analysis, the collected data undergoes a series of preprocessing steps. Firstly, the code reads the 'stopword.txt' file and renames the column 'value' to 'stopword'. Subsequently, the code reads the main dataset from HDFS and splits each line into individual words. To ensure consistency and avoid duplication, all words are transformed to lowercase. Additionally, any empty words are filtered out from the dataset. The code then performs a left anti-join with the 'df_stopwords' DataFrame to remove stopwords from the word list. Finally, unnecessary columns are dropped from the 'df_words' DataFrame to streamline the data structure.

B. Part 2

1. Section A

In this section, the initial step involves transforming the log data by extracting specific values using regular expressions. One crucial element is the timestamp column, which contains timezone information. To maintain consistency and disregard the time zone, we opt to exclude it during the transformation process. Additionally, a filter is applied to eliminate any null values in each column, ensuring data integrity throughout the processing phase. After performing these operations, a transformed DataFrame is generated, ready for exploratory data analysis.

In section 3.1, we generate output that highlights the most invoked endpoints on specific days of the week, along with their respective counts. Firstly, we determine the day of the week for each log data entry using the timestamp column. This is achieved by utilizing the date_format("timestamp", "EEEE") code. Next, a group by function is implemented to obtain distinct endpoints for each day of the week. The code groups the df_logs DataFrame by both the "day_of_week"

and "endpoint" columns. It calculates the count of occurrences for each combination and sorts the results in descending order based on the "count" column. The top 10 rows are selected, and specific columns are chosen with alias names assigned to them. The resulting DataFrame is stored in the endpoint_max_count_df variable. Once the desired results are retrieved, the DataFrame is displayed without truncation, showing all columns, using the show function.

On the other hand, to count the least frequent occurrence of the 404 status code, the code begins by filtering the DataFrame to select only the rows where the "status" column has a value of 404. The filtered results are stored in the error_df DataFrame. Subsequently, a new column called "year" is added to the DataFrame. This column extracts the year from the "timestamp" column using the F.year function from the pyspark.sql.functions module. The code then selects the "year" and "status" columns from the DataFrame to obtain the desired output. Following that, the DataFrame is grouped by the "year" column, and the count of occurrences for each year is calculated. The results are sorted in ascending order based on the "count" column, and the top 10 rows are selected, as we are interested in the least frequent occurrences. Finally, the result is displayed using the DataFrame's show function.

2. Section B

To implement the described flow, we can start by creating a Kafka producer in Python. The producer is responsible for reading the log file line by line and publishing each line as a message to a specific Kafka topic. The producer is created using the KafkaProducer class from the Kafka library. The log file path is specified, and each line is encoded as UTF-8 and sent to the Kafka topic using the send method. This ensures that the log data is injected into the Kafka cluster for further processing.

Next, we can implement the Kafka consumer, which will be subscribed to the same topic as the producer. In our code, a Spark Streaming application is used to consume data from Kafka. The create_read_stream function sets up a streaming query that reads data from the Kafka topic using the readStream API. The data is processed in micro-batches, and the process_micro_batch function is called for each batch. Within this function, the received data is transformed and analyzed using Spark SQL operations. In the provided code, the log lines are parsed using regular expressions to extract relevant fields such as host, timestamp, method,

endpoint, status, and content size. These fields can then be used for further analysis and exploration.

After performing the necessary transformations and analysis, the transformed data is written as Parquet files using the writeStream.format("parquet") method in Spark. The output mode is set to "append" to continuously append new data to the Parquet files. The Parquet files are stored in a specified location within the Hadoop Distributed File System (HDFS). By executing the Kafka consumer, the data flow is initiated, and the results of the analysis are continuously written to Parquet files in HDFS. This approach provides a scalable and efficient solution for log data processing and analysis, as the Parquet file format offers columnar storage, compression, and schema evolution capabilities. Additionally, storing the data in HDFS enables fault-tolerant and distributed processing of large-scale log datasets.

III. Analysis

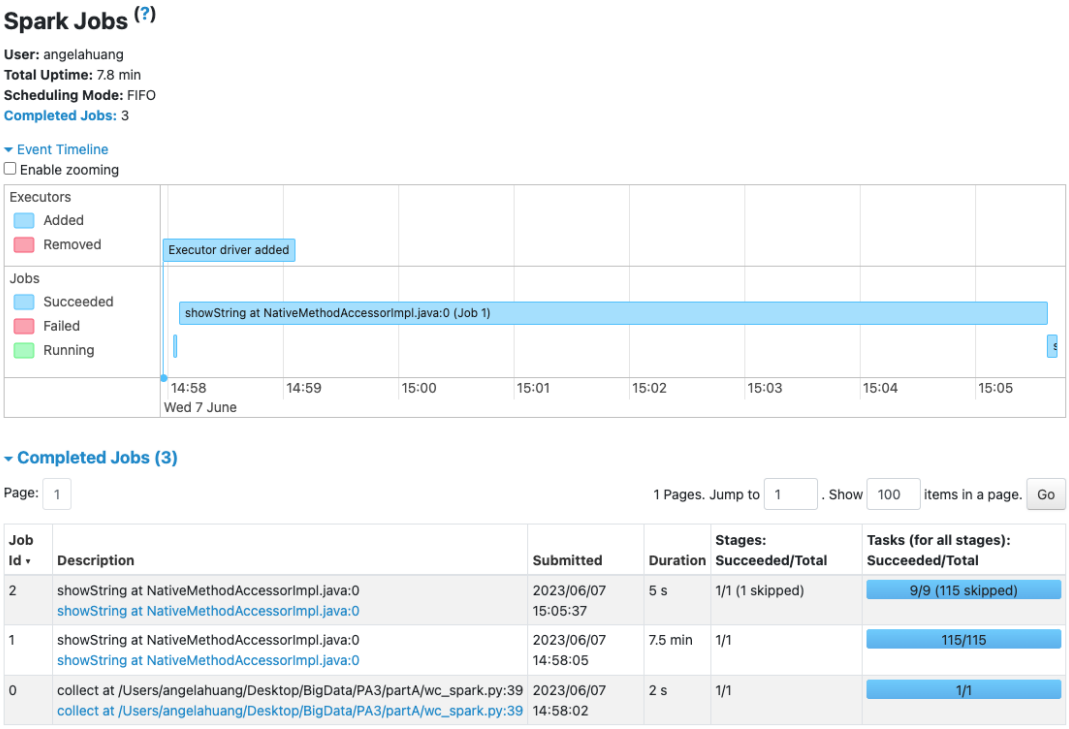
A. Part 1

The analysis conducted in this study harnesses the power of PySpark, a distributed computing framework, to perform word count analysis on a large dataset. PySpark offers several advantages over the traditional MapReduce approach, which was commonly used for similar analysis tasks. A comparison between PySpark and MapReduce can shed light on the benefits of PySpark in this context.

Additionally, PySpark's DataFrame API simplifies data preprocessing and analysis through high-level operations like filtering, joining, and aggregation. This higher-level abstraction improves code readability and reduces the complexity associated with low-level MapReduce programming. In contrast, MapReduce requires explicit definition of map and reduce functions, handling intermediate key-value pairs, and manual serialization and deserialization, resulting in more verbose and intricate code.

The code conducts word count analysis on the preprocessed data. It groups the words in the 'df_words' DataFrame based on the 'word' column and employs the count function to aggregate the frequency of each word. The resulting DataFrame, named 'df_word_counts', contains two columns: 'word', representing

the unique words, and 'count', representing the frequency of each word in the dataset.



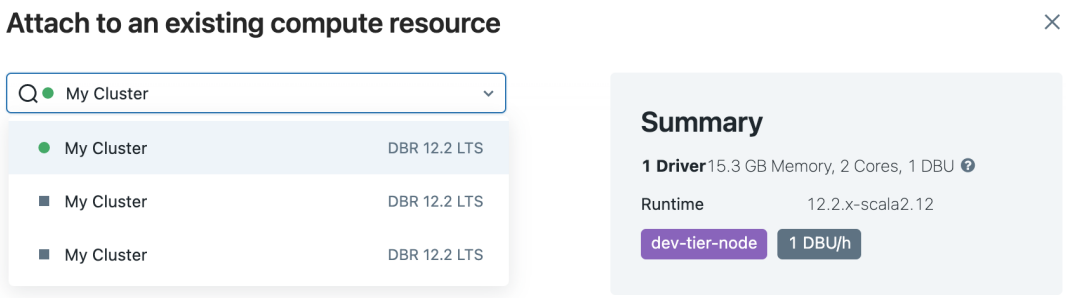
In order to check the progress of Spark tasks, we enable Spark event logging. The results obtained from analyzing the event logs reveal that the word count task using Spark takes less than 10 minutes to complete. Such great performance is 4 times faster compared to the previous implementation using MapReduce which takes half an hour to complete in PA2. Without performing jobs in sequential manner, Spark distributed workloads across multiple nodes and performed the operations in parallel and executed tasks in different nodes simultaneously. Plus, we optimized the task by narrowing down the amount of data to be processed. Prior to performing the word count, we strategically dropped unnecessary columns from the dataframe. This data pruning technique helped reduce the workload and further accelerated the task execution.

B. Part 2

1. Section A

This section is for getting familiar with log analytics, it is one of the most popular and effective enterprise case-studies which leverage analytics today. In

this case study, we will analyze log datasets from NASA Kennedy Space Center web server in Florida. These two datasets contain two months' worth of all HTTP requests to the NASA Kennedy Space Center WWW server in Florida. We will take the data from July 01 to July 31 as an example, which is NASA_access_log_Jul95.gz. There are two ways to do this analysis, one is local setup and one is cloud based setup. In this report we would take the cloud based setup on databricks as an example.



To facilitate the analysis, we began by uploading the log file to the Databricks File System (DBFS). With the file readily available, we utilized the `spark.read.text()` function to read its contents. In order to perform the subsequent analysis, we needed to transform the timestamp column into a standardized format of 'dd/MMM/yyyy:HH:mm:ss Z'. To accomplish this, we employed the `to_timestamp` function from the `pyspark.sql.functions` module.

Furthermore, as the log data contains timestamps in a specific timezone, we adjusted the timezone settings in Spark to align with the data's timezone. Specifically, we configured the Spark session's timezone to 'Asia/Calcutta', ensuring accurate timestamp interpretation throughout the analysis process.

By parsing the input log events, we created distinct columns such as 'host', 'timestamp', 'method', 'endpoint', 'protocol', 'status', and 'content_size'. These columns provide granular information about each log entry, enabling us to perform in-depth analysis and gain insights from the data.

```python
def with_parsed_fields(df_logs):
    host_pattern = r'(^\S+\.[\S+\.]+\S+)\s'
    ts_pattern = r'\[(\d{2}/\w{3}/\d{4}:\d{2}:\d{2}:\d{2} \+\d{4})]'
    method_uri_protocol_pattern = r'\"(\S+)\s(\S+)\s*(\S*)\"'
    content_size_pattern = r'\s(\d+)$'
    status_pattern = r'\s(\d{3})\s'

    return df_logs.select(
        F.regexp_extract('value', host_pattern, 1).alias('host'),
        F.to_timestamp(
            F.regexp_extract('value', ts_pattern, 1),
            'dd/MMM/yyyy:HH:mm:ss Z'
        ).alias('timestamp'),
        F.regexp_extract('value', method_uri_protocol_pattern, 1).alias('method'),
        F.regexp_extract('value', method_uri_protocol_pattern, 2).alias('endpoint'),
        F.regexp_extract('value', method_uri_protocol_pattern, 3).alias('protocol'),
        F.regexp_extract('value', status_pattern, 1).cast('integer').alias('status'),
        F.regexp_extract('value', content_size_pattern, 1).cast('integer').alias('content_size')
    )
```

df_logs: pyspark.sql.dataframe.DataFrame = [host: string, timestamp: timestamp ... 5 more fields]

Table +

	host	timestamp	method	endpoint	protocol	status	content_size
1	199.72.81.55	1995-07-01T09:30:01.000+0530	GET	/history/apollo/	HTTP/1.0	200	6245
2	unicomp6.unicomp.net	1995-07-01T09:30:06.000+0530	GET	/shuttle/countdown/	HTTP/1.0	200	3985
3	199.120.110.21	1995-07-01T09:30:09.000+0530	GET	/shuttle/missions/sts-73/mission-sts-73.html	HTTP/1.0	200	4085
4	burger.letters.com	1995-07-01T09:30:11.000+0530	GET	/shuttle/countdown/liftoff.html	HTTP/1.0	304	0
5	199.120.110.21	1995-07-01T09:30:11.000+0530	GET	/shuttle/missions/sts-73/sts-73-patch-small.gif	HTTP/1.0	200	4179
6	burger.letters.com	1995-07-01T09:30:12.000+0530	GET	/images/NASA-logosmall.gif	HTTP/1.0	304	0
7	burger.letters.com	1995-07-01T09:30:12.000+0530	GET	/shuttle/countdown/video/livevideo.gif	HTTP/1.0	200	0
8	205.212.115.106	1995-07-01T09:30:12.000+0530	GET	/shuttle/countdown/countdown.html	HTTP/1.0	200	3985
9	d104.aa.net	1995-07-01T09:30:13.000+0530	GET	/shuttle/countdown/	HTTP/1.0	200	3985
10	129.94.144.152	1995-07-01T09:30:13.000+0530	GET	/	HTTP/1.0	200	7074

Now that we have parsed the log events into separate columns, including the 'timestamp' column, we can proceed with the analysis to address Q3.1. In order to determine the day of the week for each log entry, we employ the `date_format` function from the `pyspark.sql.functions` module.

By applying the `date_format` function, we extract the day of the week from the 'timestamp' column, enabling us to group the data based on this information. This step is essential for identifying the endpoint that received the highest number of invocations on a specific day of the week.With the day of the week extracted as a separate column, we can now proceed to the next steps of the analysis and compute the count of invocations for each endpoint and day of the week.

```
def with_highest_request(df_logs):
    day_df = (
       df_logs
       .withColumn(
          "day",
          F.date_format(F.col("timestamp"), 'EEEE')
       )
       .select('timestamp', 'day', 'endpoint')
        .groupby('day', 'endpoint')
        .count()
        .sort('count', ascending=False)
        .limit(1)
    )

    return day_df
```

	day	endpoint	count
1	Thursday	/images/NASA-logosmall.gif	25496
2	Wednesday	/images/NASA-logosmall.gif	19269
3	Friday	/images/NASA-logosmall.gif	18245
4	Thursday	/images/KSC-logosmall.gif	16785
5	Tuesday	/images/NASA-logosmall.gif	15920

Based on the analysis conducted, it has been determined that Thursday is the day of the week with the highest number of invocations. Additionally, the endpoint '/images/NASA-logosmall.gif' received the most invocations on this particular day.

To address the second question regarding the frequent occurrence of 404 status codes, a similar approach is employed. By extracting the 'year' from the timestamp column using the `date_format` function, we can isolate the year component of each log entry. Subsequently, the data is grouped by year, and the count of 404 status codes is computed.

By examining the resulting dataset, it is possible to identify the years in which the least number of 404 status codes occurred. The top 10 years with the lowest counts of 404 status codes can be determined by sorting the data in ascending order and selecting the corresponding entries.

```
def with_freq_404_year(df_logs):
    error_df = (
            df_logs
            .filter(df_logs['status'] == 404)
        )

    year_df = (
        error_df
        .withColumn(
            "year",
            F.year(error_df['timestamp'])
         )
        .select('year', 'status')
    )

    http_sum_df = (
        year_df
        .groupby('year','status')
        .count()
        .sort('count', ascending=True)
        .limit(10)
    )

    return http_sum_df
```

2. Section B

In this section, our focus shifts towards Kafka producer and consumer operations, specifically in conjunction with Apache Spark. The objective is to explore how these technologies can be integrated to enable real-time data processing and analysis.

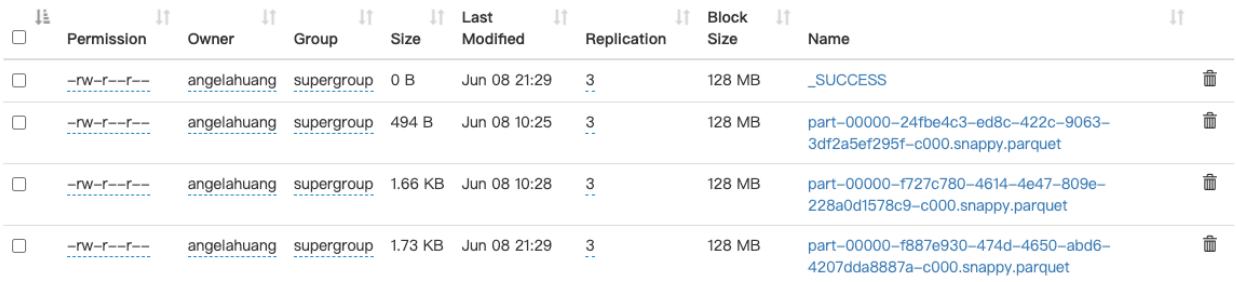
The presented code establishes a comprehensive pipeline for log file processing and analysis using a Kafka producer-consumer model integrated with Spark. This enables real-time ingestion, transformation, and storage of log data for valuable insights. The Kafka producer efficiently sends log file lines to a designated Kafka topic, ensuring seamless streaming of data. Subsequently, a Kafka consumer coupled with Spark allows for advanced transformations and analytics. The transformed data can be converted to Parquet format, a high-performance storage format, and seamlessly stored in HDFS for fault tolerance and scalability. This integrated workflow enables organizations to extract real-time insights from log data, facilitating timely decision-making and anomaly detection. The approach is highly efficient, scalable, and well-suited for log analytics use cases.

In the consumer code, we utilize the create_read_stream function to establish a connection with the Kafka cluster and subscribe to the specified topic (17gb). The logs are read in as a streaming DataFrame, enabling real-time processing of data as it arrives. To extract meaningful insights from the log data,

we employ the with_parsed_fields function, which utilizes regular expressions and Spark's string manipulation functions to parse the log lines into separate columns. This allows us to extract useful information such as the host, timestamp, method, endpoint, protocol, status, and content size.

```
Angela-MacBook:PA3 angelahuang$ python3 test_kafka.py
23/06/08 23:06:17 WARN Utils: Your hostname, Angela-MacBook.local resolves to a loopback address: 127.0.0.
0)
23/06/08 23:06:17 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
23/06/08 23:06:18 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using b
+-------------+--------------------------------------------------------------------------------------+------+
|Day in a week|endpoint                                                                              |Count |
+-------------+--------------------------------------------------------------------------------------+------+
|Monday       |/Archives/edgar/data/0001724128/000121390021017151/s131120_10k.htm                    |798006|
|Monday       |/Archives/edgar/data/0000205007/000114554923015369/0001145549-23-015369.txt           |797868|
|Monday       |/Archives/edgar/data/0001330427/000093247112005362/0000932471-12-005362.txt           |797757|
|Monday       |/Archives/edgar/data/1826671/000121390023023294/0001213900-23-023294.txt              |797720|
|Monday       |/Archives/edgar/data/0001866295/000110465921080403/tm2117388d5_ex3-32g008.jpg         |797628|
|Monday       |/Archives/edgar/data/0001704304/000092963819000386/0000929638-19-000386.txt           |797608|
|Monday       |/Archives/edgar/data/0000101382/000119312517053195/d317189d10k.htm                    |797516|
|Monday       |/Archives/edgar/data/1319616/000120919119019599/0001209191-19-019599-index.htm        |797459|
|Monday       |/Archives/edgar/data/0001203957/000119312510223947/0001193125-10-223947-index.htm     |797457|
|Monday       |/Archives/edgar/data/0001326380/000132638023000019/gme-20230128.htm                   |797412|
+-------------+--------------------------------------------------------------------------------------+------+
only showing top 10 rows
```

Furthermore, we implement two specific analyses within the process_micro_batch function. The first analysis aims to identify the highest requests day by grouping the data based on the day of the week and endpoint, and then counting the occurrences. The second analysis focuses on finding the top 10 years with the least 404 error responses. This is achieved by filtering the log data for 404 status codes, extracting the year from the timestamp, and aggregating the counts for each year.



Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
-rw-r--r--	angelahuang	supergroup	0 B	Jun 08 21:29	3	128 MB	_SUCCESS
-rw-r--r--	angelahuang	supergroup	494 B	Jun 08 10:25	3	128 MB	part-00000-24fbe4c3-ed8c-422c-9063-3df2a5ef295f-c000.snappy.parquet
-rw-r--r--	angelahuang	supergroup	1.66 KB	Jun 08 10:28	3	128 MB	part-00000-f727c780-4614-4e47-809e-228a0d1578c9-c000.snappy.parquet
-rw-r--r--	angelahuang	supergroup	1.73 KB	Jun 08 21:29	3	128 MB	part-00000-f887e930-474d-4650-abd6-4207dda8887a-c000.snappy.parquet

In order to process Kafka data concurrently, we tried to demonstrate the usage of threading in Python. By creating a separate thread, the code enables the parallel execution of the process_kafka_data function, which is responsible for consuming and handling Kafka messages.

In addition, to improve overall performance, we add the thread function in kafka_consumer. The efficiency gained from using a daemon thread is particularly beneficial in scenarios where the background task, such as processing Kafka data,

may take an extended period or could potentially block the program's execution. By allowing the program to exit without waiting for the daemon thread, system resources can be efficiently managed and utilized for other essential tasks.

By combining Kafka and Spark, we gain several advantages for log analysis. Firstly, Kafka provides a distributed and fault-tolerant messaging system that ensures the reliable ingestion of log data in real-time. It acts as a buffer between the producer and consumer, allowing for decoupling and scalability. Secondly, Spark offers powerful data processing capabilities, enabling efficient transformations, aggregations, and analytics on large-scale datasets. Spark's ability to process streaming data in micro-batches ensures continuous and near-real-time analysis of log events. Last but not least, Spark seamlessly integrates with various data sources, including Kafka and HDFS, facilitating data ingestion, processing, and storage in a unified pipeline.

We can see that the integration of Kafka and Spark in log analysis workflows provides several benefits. It enables real-time streaming analytics, allowing organizations to gain insights and react promptly to emerging patterns or issues. The distributed nature of Kafka and Spark allows for horizontal scalability, ensuring efficient processing of high-volume log data. Moreover, the ability to store the transformed data in the Parquet file format enhances query performance and facilitates further analysis. That is, the combination of Kafka and Spark provides a robust and scalable solution for log analysis, enabling organizations to extract valuable insights from large and continuously evolving log datasets.

IV. Conclusion

Throughout this project, we extensively utilized various Big Data technologies to meet our requirements. Our primary focus was on leveraging the capabilities of Spark DataFrame API, Spark SQL, Spark Streaming, Parquet files, and Hadoop Distributed File Systems. We explored different approaches and techniques to determine the most efficient ones that align with our project objectives.

One of our initial tasks involved conducting word count analysis using the Spark DataFrame API. This allowed us to gain a deep understanding of the API's powerful features for distributed data processing. We gained valuable insights into how different Spark languages effectively distribute data and optimize transformations and actions.

In addition to word count analysis, we harnessed the power of the Spark DataFrame API for log data analytics transformations. By leveraging PySpark's extensive library of built-in functions and operations, we were able to perform comprehensive exploratory analysis on the log data. Spark's ability to handle large-scale datasets and perform distributed processing proved invaluable in efficiently processing and analyzing the log data.

Upon completing the log data analysis, we established a Kafka producer-consumer workflow, enabling seamless data flow between the producer and consumer. This architecture allowed us to subscribe to the topic and periodically check for new data without disrupting the original data. This asynchronous communication mechanism proved highly effective in our project.

The knowledge and experience gained from this project have equipped us with valuable skills in working with Big Data technologies, enabling us to tackle similar data analysis challenges in the future.

V. Reference

Pusukuri, Kishore. COEN 242 Class Slides. Santa Clara University
https://spark.apache.org/docs/latest/monitoring.html
https://towardsdatascience.com/scalable-log-analytics-with-apache-spark-a-comprehensive-case-study-2be3eb3be977
https://spark.apache.org/docs/latest/sql-programming-guide.html

Top100 Words for 16GB.txt

said 16983038
would 5829109
one 5827854
new 5619251
also 4618231
us 4602724
people 4302078
last 4096906
year 4011803
two 3964144
first 3839807
mr 3746747
years 3637652
time 3635760
could 3374880
like 3058690
government 2520240
says 2417351
may 2393105
get 2346949
many 2344542
back 2295703
million 2255486
three 2250200
president 2193692
even 2175400
made 2130243
make 2104945
told 2092903
since 2041708
world 2039952
percent 2022481
police 2014250
well 1965773
still 1956831
state 1948101
way 1936600
much 1878812
say 1866770
day 1831556
week 1829329
home 1827482
take 1785403
per 1779050
work 1770896
going 1744159
think 1670557
company 1665814
good 1635409
dont 1629480
next 1605312
including 1594887
see 1561688
states 1543254
another 1536907
group 1521302
go 1508945
public 1506218
de 1499434
house 1494732
around 1482330
city 1474866
former 1461398
part 1446451
second 1433375
national 1415826
obama 1405174
know 1386991
want 1379621
game 1371927
four 1371032
news 1368259
found 1366151
end 1353926
right 1336449
court 1330434
team 1329924
united 1310698
need 1307171
report 1299343
country 1295488
help 1289802
according 1281617
business 1275145
market 1267854
life 1256423
long 1242001
set 1232190
months 1227463
man 1226580
best 1211261
come 1208192
cent 1203960
officials 1195387
family 1186525
left 1178734
high 1177794
show 1172154
health 1163668
york 1161016

Top100 Words(>6) for 16GB.txt

government	2520240	companies	869826	possible	629503
million	2255486	washington	866772	thought	628703
president	2193692	members	861953	control	628021
percent	2022481	service	853089	current	624576
company	1665814	federal	852384	program	624140
including	1594887	campaign	820329	getting	623774
another	1536907	statement	817540	interest	612373
national	1415826	economy	816287	outside	605728
country	1295488	british	813789	england	599716
according	1281617	university	806441	minutes	599437
business	1275145	services	799299	spokesman	596504
officials	1195387	director	788859	continue	594807
american	1129214	important	773502	increase	588451
international	1116052	working	766645	private	586730
financial	1097729	players	755612	authorities	584908
support	1080825	countries	740881	meeting	578137
children	1072801	decision	732901	history	576197
however	1072387	earlier	732872	together	575985
security	1058065	different	722896	results	575269
billion	1049368	general	722268	community	571257
without	1039688	research	716089	reports	565968
political	1018915	capital	705038	leaders	565284
european	1018667	saturday	702016	problem	562464
whether	1014215	believe	701746	hospital	559119
yearold	981085	industry	700059	process	556804
tuesday	969122	department	699764	markets	555948
already	960627	executive	696821		
minister	942536	quarter	696623		
information	927619	following	665860		
wednesday	920376	although	662413		
reported	917579	foreign	654706		
expected	916793	despite	652919		
economic	910616	official	643514		
thursday	910329	council	639609		
something	896500	election	635236		
military	893360	announced	632756		
several	877379	looking	630482		