

Big Data

Programming Assignment - 3

COEN 242 - Spring 2023

Group Members :

Anand Kulkarni (W1638929)

Pralhad Kulkarni (W1649360)

Configuration used:

Windows - i5 11th gen, 16 GB RAM

MacOS - M2, 8GB RAM

PART 1 : [SPARK]

Data partitioning Vs Total execution time in Apache Spark :

1. Parallel Processing:

- Enables parallel processing on multiple partitions simultaneously.
- Each partition can be processed independently by separate executors or worker nodes.

2. Reduced Data Shuffling:

- Minimizes data shuffling during operations like joins or aggregations.
- Local operations within each partition reduce the need for transferring data across the network.

3. Efficient Resource Utilization:

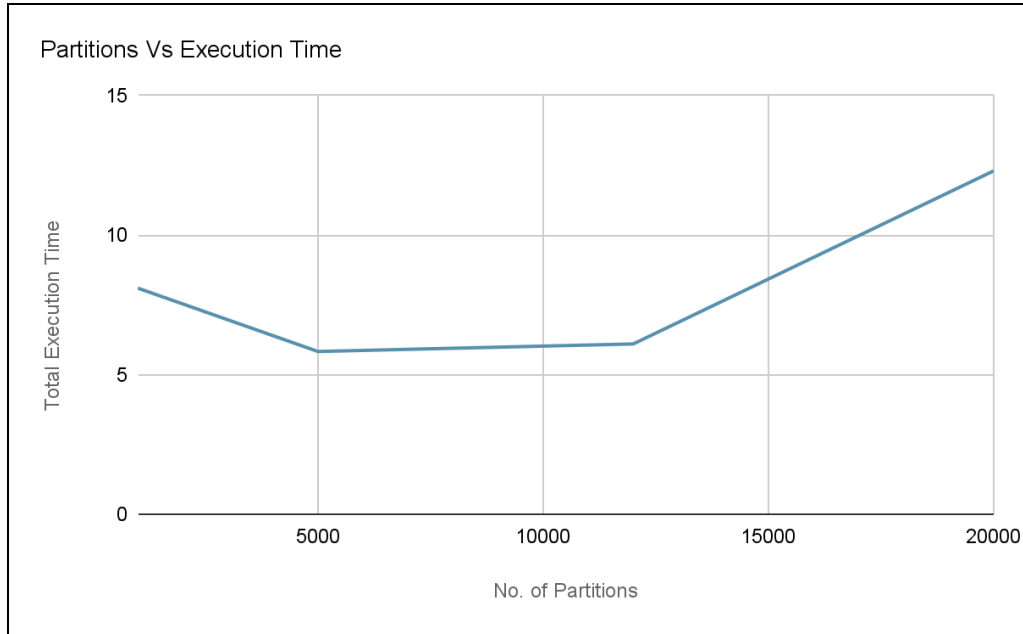
- Assigns partitions to different executors or worker nodes for optimal resource utilization.
- Each executor can process its assigned partitions independently, utilizing available resources efficiently.

4. Load Balancing:

- Achieves load balancing across the cluster by evenly distributing partitions.
- Prevents overwhelming individual executors or nodes with disproportionate amounts of data.

5. Scalability:

- Facilitates scalability by horizontally scaling the cluster.
- Each new node can process a subset of partitions, maintaining performance as data volume increases.



Execution time on 16GB dataset using **Apache spark** = **5.8 mins**

Execution time on 16GB dataset using **Hadoop** = **14 mins**

How over data partitioning affects the total execution time (same proved in above graph)?

1. Increased overhead:

- Excessive partitioning in Spark leads to increased memory usage, task scheduling, and data serialization overhead.
- Managing and processing a large number of partitions requires additional coordination and communication between tasks, leading to longer execution times (This same can be seen in above graph).

2. Task scheduling and coordination:

- Spark's task scheduling and coordination mechanisms are influenced by the number of partitions.
- Over partitioning introduces a higher number of tasks, which increases the scheduling overhead and can cause delays in task execution.

3. Data skew and load imbalance:

- Over partitioning can worsen data skew and load imbalance issues.
- Uneven data distribution across partitions may result in some partitions having significantly more data than others.
- This leads to stragglers and reduced parallelism, resulting in longer execution times as the workload is not evenly distributed among the available resources.

Why is Apache Spark technique faster than MapReduce in processing large datasets of 16 GB text files ?

1. **In-Memory Processing:** Spark leverages in-memory processing, which means it keeps the intermediate data in memory rather than writing it to disk after each stage. This significantly reduces disk I/O and provides faster access to data, resulting in improved processing speed.
2. **Data Partitioning:** Spark partitions data across the cluster and performs operations on each partition in parallel. This allows for parallel processing of data, enabling faster execution compared to MapReduce, which relies on disk-based processing.
3. **DAG Execution Model:** Spark uses a **Directed acyclic graph (DAG)** execution model, where it optimizes the execution plan by chaining together multiple operations and minimizing data shuffling. This optimization reduces the overall execution time and improves performance.
4. **Caching and Persistence:** Spark allows for caching and persisting intermediate data in memory or on disk. This enables reusing the data across multiple iterations or stages, eliminating the need to recompute the same data repeatedly and providing faster access to the cached data.
5. **Efficient Data Structures:** Spark introduces **Resilient Distributed Datasets (RDDs)**, which are fault-tolerant and immutable distributed collections of data. RDDs provide efficient operations for data transformations and actions, allowing for faster data processing.
6. **Machine Learning and Graph Processing Libraries:** Spark provides built-in libraries for machine learning (MLlib) and graph processing (GraphX), which are optimized for speed and efficiency. These libraries leverage Spark's in-memory computing capabilities, resulting in faster processing of large-scale ML and graph algorithms.

PART 2 :

Section A : NASA Log analytics

Q 3.1



day_of_week	endpoint	invocations
Friday	/images/NASA-logo...	29139
Monday	/images/NASA-logo...	30380
Saturday	/images/NASA-logo...	16168
Sunday	/images/KSC-logos...	15218
Thursday	/images/NASA-logo...	46920
Tuesday	/images/NASA-logo...	33685
Wednesday	/images/NASA-logo...	37598

Q 3.2

Only the year 1995 is present in the dataset, so there will be only one row present in output as shown below :

year	count
1995	20899

Section B : Apache Kafka , Spark and HDFS

Why store log data in Parquet format?

1. **Columnar Storage** : Parquet is a columnar storage format, which means that the data is organized and stored column by column rather than row by row. This structure allows for efficient compression and encoding techniques specific to each column, resulting in reduced storage space and improved query performance.
2. **Compression** : Parquet uses various compression algorithms, such as Snappy, Gzip, and LZO, to compress the data at a column level. This compression significantly reduces the storage footprint, enabling cost-effective storage of large volumes of log data.
3. **Predicate Pushdown** : Parquet supports predicate pushdown, which means that it can push down filters to the storage layer during query execution. This capability allows for skipping irrelevant data blocks, resulting in faster query processing and improved performance.
4. **Schema Evolution** : Parquet provides schema evolution capabilities, allowing for changes in the data schema over time without requiring rewriting the entire dataset. This flexibility is crucial in scenarios where log data schemas may evolve or change periodically.
5. **Compatibility with Big Data Ecosystem** : Parquet is widely adopted in the big data ecosystem, and it is compatible with various data processing frameworks such as Apache Spark, Apache Hive, and Apache Impala. This compatibility enables seamless integration with existing big data workflows and allows for efficient data processing and analytics.

What are different methods to analyze log data in parquet format ?

1. **Batch Processing using Spark** : In this approach, the entire Parquet file or a subset of it is read into Spark as a DataFrame or RDD, and various transformations and analyses are performed using Spark's DataFrame or RDD API. Batch processing is suitable when the data can be processed in batches and near **real-time analysis** is **not** required.
2. **Interactive Querying with SQL Tools** : Parquet files can be loaded into interactive querying tools like Apache Drill, Apache Impala, or Presto. These tools provide SQL-based querying and analysis capabilities on large datasets, including log data stored in Parquet format. They offer fast query performance and support ad-hoc analysis.
3. **Machine Learning with Spark MLlib** : Parquet files can be used as input for building machine learning models using Spark's MLlib library. The log data can be preprocessed, transformed, and used to train models for various purposes such as anomaly detection, classification, or predictive analytics.

4. **Real-time Analysis with Spark Streaming :** If real-time analysis of streaming log data is required, Spark Streaming can be used. It enables processing and analysis of data as it arrives, allowing for immediate insights, alerts, or actions based on the log data. Spark Streaming integrates seamlessly with streaming sources like Apache Kafka, making it suitable for continuous processing and analysis of log data in real time.

Why did we choose to use Spark Streaming for analyzing NASA log data stored in parquet format instead of other ways mentioned earlier ?

1. **Real-time Analysis of Log Events :** NASA log data contains records of various events and activities, such as server requests, user interactions, or system statuses. Analyzing these log events in real-time can provide valuable insights for monitoring, troubleshooting, and anomaly detection. Spark Streaming enables continuous processing and analysis of streaming data, allowing us to perform real-time monitoring and analysis of the log events as they occur.
2. **Handling Streaming Nature of Log Data :** NASA log data is often generated in a continuous and streaming fashion. Spark Streaming integrates seamlessly with streaming sources like Apache Kafka. By leveraging Spark Streaming, we can efficiently consume and process the streaming log data, ensuring that we capture and analyze the events in a timely manner.
3. **Scalability for Large Volumes of Data :** NASA log data can grow significantly in volume. Spark Streaming's inherent scalability and distributed processing capabilities enable it to handle large volumes of log data with ease. It can distribute the processing across a cluster of machines, allowing for parallel execution and efficient utilization of computing resources.
4. **Fault Tolerance and Data Integrity :** Spark Streaming provides built-in fault tolerance mechanisms that ensure the processing of log data is resilient to failures. It automatically recovers from failures and maintains data integrity, allowing us to process the log events reliably without losing any important information. This is crucial for analyzing NASA log data, as system disruptions or restarts should not result in data loss or inconsistencies.
5. **Integration with Spark Ecosystem for Advanced Analysis :** Spark Streaming seamlessly integrates with the wider Spark ecosystem, including Spark SQL, Spark MLlib, and Spark GraphX. This integration enables you to perform advanced analysis on the NASA log data. For example, you can leverage Spark SQL to query and aggregate log events, utilize MLlib for anomaly detection or predictive analytics, or apply graph algorithms using Spark GraphX to discover patterns or relationships within the log data.

Factors that affect Apache kafka producer and consumer :

- **Network latency** between producers, Kafka brokers, and consumers can impact message delivery and processing time.
- **Message size and compression** affect the performance, with larger sizes requiring more resources and compression reducing transfer time.
- **Serialization** and deserialization mechanisms influence processing time and efficiency.
- Producer and consumer configurations, such as batch size and linger time, impact throughput and latency.
- **Message acknowledgment** and delivery semantics affect reliability and latency.
- The consumer group and partition distribution influence parallelism and scalability.
- **Hardware resources**, including CPU, memory, and disk I/O, impact performance.
- **Optimization**, monitoring, and performance tuning are important for identifying and addressing bottlenecks and limitations.
- We have selected a batch size as 1000 and also tried with 200 and 300 as analysis while running the producer.

EDA analysis on parquet files :

(Rest of EDA analysis is shown in the Log_analysis_parquet.ipynb)

```
paths_df = (logs_df
            .groupBy('endpoint')
            .count()
            .sort('count', ascending=False).limit(20))
paths_df.show()
```

```
+-----+-----+
| endpoint | count |
+-----+-----+
| /Archives/edgar/d... | 863 |
| /Archives/edgar/d... | 861 |
| /Archives/edgar/d... | 853 |
| /Archives/edgar/d... | 853 |
| /Archives/edgar/d... | 852 |
| /Archives/edgar/d... | 851 |
| /Archives/edgar/d... | 850 |
| /Archives/edgar/d... | 850 |
| /Archives/edgar/d... | 847 |
| /Archives/edgar/d... | 846 |
| /Archives/edgar/d... | 842 |
| /Archives/edgar/d... | 842 |
| /Archives/edgar/d... | 840 |
| /Archives/edgar/d... | 839 |
| /Archives/edgar/d... | 839 |
| /Archives/edgar/d... | 835 |
| /Archives/edgar/d... | 834 |
| /Archives/edgar/d... | 833 |
| /Archives/edgar/d... | 833 |
| /Archives/edgar/d... | 831 |
+-----+-----+
```

Weekly 404 response code error :




















```
from pyspark.sql.functions import col

weekly_404 = not_found_df.groupBy('day_of_week').count().orderBy(col('count').asc())
weekly_404.show()
```

```
+-----+-----+
|day_of_week|count|
+-----+-----+
|      Monday|17173|
+-----+-----+
```

Final output screenshot of parquet files uploaded to HDFS :

Show entries Search:

<input type="checkbox"/>	 Permission	 Owner	 Group	 Size	 Last Modified	 Replication	 Block Size	 Name	
<input type="checkbox"/>	-rw-r--r--	pralhad	supergroup	8 KB	Jun 08 23:01	1	128 MB	.DS_Store	
<input type="checkbox"/>	drwxr-xr-x	pralhad	supergroup	0 B	Jun 08 23:01	0	0 B	_spark_metadata	
<input type="checkbox"/>	-rw-r--r--	pralhad	supergroup	1.22 MB	Jun 08 23:01	1	128 MB	part-00000-0c822a6e-970f-4e1a-9a1d-cd598193bce5-c000.snappy.parquet	
<input type="checkbox"/>	-rw-r--r--	pralhad	supergroup	55.32 KB	Jun 08 23:01	1	128 MB	part-00000-6f96cad8-1d37-46b4-b67e-76d0c9ea81a7-c000.snappy.parquet	
<input type="checkbox"/>	-rw-r--r--	pralhad	supergroup	549.65 KB	Jun 08 23:01	1	128 MB	part-00000-88388fb2-d737-474b-abed-dee2a47e0856-c000.snappy.parquet	
<input type="checkbox"/>	-rw-r--r--	pralhad	supergroup	163.2 KB	Jun 08 23:01	1	128 MB	part-00000-9415283f-c125-4c12-8be2-bae552aa8192-c000.snappy.parquet	
<input type="checkbox"/>	-rw-r--r--	pralhad	supergroup	3.38 KB	Jun 08 23:01	1	128 MB	part-00000-d3acb701-a68a-4c88-82dc-9d7dd25d76f2-c000.snappy.parquet	
<input type="checkbox"/>	-rw-r--r--	pralhad	supergroup	20.05 KB	Jun 08 23:01	1	128 MB	part-00000-d794c4f4-4ec8-435d-81d9-c29062bc1a5b-c000.snappy.parquet	
<input type="checkbox"/>	-rw-r--r--	pralhad	supergroup	30.26 KB	Jun 08 23:01	1	128 MB	part-00000-d9a79b41-c904-4722-a1b1-2b5f0b55eb68-c000.snappy.parquet	
<input type="checkbox"/>	-rw-r--r--	pralhad	supergroup	1.17 KB	Jun 08 23:01	1	128 MB	part-00000-ef9610f0-d588-4f59-9331-bd52e4f17869-c000.snappy.parquet	

Showing 1 to 10 of 10 entries Previous **1** Next

Kafka producer :

```
C:\spark-3.4.0-bin-hadoop3>bin\spark-submit --packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.4.0 "D:\Big Data\Log Analysis\kafka_consumer.py"
:: loading settings :: url = jar:file:/C:/spark-3.4.0-bin-hadoop3/jars/ivy-2.5.1.jar!/org/apache/ivy/core/settings/ivysettings.xml
Ivy Default Cache set to: C:\Users\kulka\.ivy2\cache
The jars for the packages stored in: C:\Users\kulka\.ivy2\jars
org.apache.spark:spark-sql-kafka-0-10_2.12 added as a dependency
:: resolving dependencies :: org.apache.spark#spark-submit-parent-0eff8695-df6f-49de-9b55-e730120f5e5e;1.0
  confs: [default]
    found org.apache.spark#spark-sql-kafka-0-10_2.12:3.4.0 in central
    found org.apache.spark#spark-token-provider-kafka-0-10_2.12:3.4.0 in central
    found org.apache.kafka#kafka-clients;3.3.2 in central
    found org.lz4#lz4-java;1.8.0 in central
    found org.xerial.snappy#snappy-java;1.1.9.1 in central
    found org.slf4j#slf4j-api;2.0.6 in central
    found org.apache.hadoop#hadoop-client-runtime;3.3.4 in central
    found org.apache.hadoop#hadoop-client-api;3.3.4 in central
    found commons-logging#commons-logging;1.1.3 in central
    found com.google.code.findbugs#jsr305;3.0.0 in central
    found org.apache.commons#commons-pool2;2.11.1 in central
  :: resolution report :: resolve 652ms :: artifacts dl 93ms
  :: modules in use:
    com.google.code.findbugs#jsr305;3.0.0 from central in [default]
    commons-logging#commons-logging;1.1.3 from central in [default]
    org.apache.commons#commons-pool2;2.11.1 from central in [default]
    org.apache.hadoop#hadoop-client-api;3.3.4 from central in [default]
    org.apache.hadoop#hadoop-client-runtime;3.3.4 from central in [default]
    org.apache.kafka#kafka-clients;3.3.2 from central in [default]
    org.apache.spark#spark-sql-kafka-0-10_2.12:3.4.0 from central in [default]
    org.apache.spark#spark-token-provider-kafka-0-10_2.12:3.4.0 from central in [default]
    org.lz4#lz4-java;1.8.0 from central in [default]
    org.slf4j#slf4j-api;2.0.6 from central in [default]
    org.xerial.snappy#snappy-java;1.1.9.1 from central in [default]
  |-----|
  |   conf   |   number | modules | artifacts | | |
|---|---|---|---|---|---|
  |   default |    11    |    0    |    0    | 11 | 0 |
  |-----|-----|-----|-----|
  :: retrieving :: org.apache.spark#spark-submit-parent-0eff8695-df6f-49de-9b55-e730120f5e5e
    confs: [default]
    0 artifacts copied, 11 already retrieved (0kB/10ms)
23/06/08 23:06:33 INFO SparkContext: Running Spark version 3.4.0
23/06/08 23:06:33 INFO ResourceUtils: =====
23/06/08 23:06:33 INFO ResourceUtils: No custom resources configured for spark.driver.
23/06/08 23:06:33 INFO ResourceUtils: =====
23/06/08 23:06:33 INFO SparkContext: Submitted application: KafkaConsumerSparkStreaming
23/06/08 23:06:33 INFO ResourceProfile: Default ResourceProfile created, executor resources: Map(cores -> name: cores, amount: 1, script: , vendor: , memory -> name: memory, amount: 1024, script: , vendor: ,
Heap -> name: offheap, amount: 0, script: , vendor: ), task resources: Map(cpus -> name: cpus, amount: 1.0)
23/06/08 23:06:33 INFO ResourceProfile: Limiting resource is cpu
```

Kafka consumer :

```
C:\spark-3.4.0-bin-hadoop3>bin\spark-submit --packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.4.0 "D:\Big Data\Log Analysis\kafka_consumer.py"
:: loading settings :: url = jar:file:/C:/spark-3.4.0-bin-hadoop3/jars/ivy-2.5.1.jar!/org/apache/ivy/core/settings/ivysettings.xml
Ivy Default Cache set to: C:\Users\kulka\.ivy2\cache
The jars for the packages stored in: C:\Users\kulka\.ivy2\jars
org.apache.spark:spark-sql-kafka-0-10_2.12 added as a dependency
:: resolving dependencies :: org.apache.spark#spark-submit-parent-0eff8695-df6f-49de-9b55-e730120f5e5e;1.0
  confs: [default]
    found org.apache.spark#spark-sql-kafka-0-10_2.12:3.4.0 in central
    found org.apache.spark#spark-token-provider-kafka-0-10_2.12:3.4.0 in central
    found org.apache.kafka#kafka-clients;3.3.2 in central
    found org.lz4#lz4-java;1.8.0 in central
    found org.xerial.snappy#snappy-java;1.1.9.1 in central
    found org.slf4j#slf4j-api;2.0.6 in central
    found org.apache.hadoop#hadoop-client-runtime;3.3.4 in central
    found org.apache.hadoop#hadoop-client-api;3.3.4 in central
    found commons-logging#commons-logging;1.1.3 in central
    found com.google.code.findbugs#jsr305;3.0.0 in central
    found org.apache.commons#commons-pool2;2.11.1 in central
  :: resolution report :: resolve 652ms :: artifacts dl 93ms
  :: modules in use:
    com.google.code.findbugs#jsr305;3.0.0 from central in [default]
    commons-logging#commons-logging;1.1.3 from central in [default]
    org.apache.commons#commons-pool2;2.11.1 from central in [default]
    org.apache.hadoop#hadoop-client-api;3.3.4 from central in [default]
    org.apache.hadoop#hadoop-client-runtime;3.3.4 from central in [default]
    org.apache.kafka#kafka-clients;3.3.2 from central in [default]
    org.apache.spark#spark-sql-kafka-0-10_2.12:3.4.0 from central in [default]
    org.apache.spark#spark-token-provider-kafka-0-10_2.12:3.4.0 from central in [default]
    org.lz4#lz4-java;1.8.0 from central in [default]
    org.slf4j#slf4j-api;2.0.6 from central in [default]
    org.xerial.snappy#snappy-java;1.1.9.1 from central in [default]
  |-----|
  |   conf   |   number | modules | artifacts | | |
|---|---|---|---|---|---|
  |   default |    11    |    0    |    0    | 11 | 0 |
  |-----|-----|-----|-----|
  :: retrieving :: org.apache.spark#spark-submit-parent-0eff8695-df6f-49de-9b55-e730120f5e5e
    confs: [default]
    0 artifacts copied, 11 already retrieved (0kB/10ms)
23/06/08 23:06:33 INFO SparkContext: Running Spark version 3.4.0
23/06/08 23:06:33 INFO ResourceUtils: =====
23/06/08 23:06:33 INFO ResourceUtils: No custom resources configured for spark.driver.
23/06/08 23:06:33 INFO ResourceUtils: =====
23/06/08 23:06:33 INFO SparkContext: Submitted application: KafkaConsumerSparkStreaming
23/06/08 23:06:33 INFO ResourceProfile: Default ResourceProfile created, executor resources: Map(cores -> name: cores, amount: 1, script: , vendor: , memory -> name: memory, amount: 1024, script: , vendor: ,
Heap -> name: offheap, amount: 0, script: , vendor: ), task resources: Map(cpus -> name: cpus, amount: 1.0)
23/06/08 23:06:33 INFO ResourceProfile: Limiting resource is cpu
```

Section C :

Overview of K-means clustering using PySpark :

By using **data parallelism**, the workload is divided among multiple processors, allowing them to process subsets of the data simultaneously. Result parallelism enables the exchange of intermediate results between the processors and the master process, facilitating the computation and update of centroids.

1. The dataset D , consisting of records X_1, X_2, \dots, X_n , is partitioned into smaller subsets and distributed across P processors. This partitioning is done at the beginning to create parallelism from the start.
2. Each processor works on its own subset of records and computes the distances between the records and the current centroids C .
3. Based on the distances, each record is assigned to the closest centroid, forming local clusters within each processor.
4. The processors independently compute the sum and count of records for each local cluster and share this information with a master process.
5. The master process aggregates the local cluster information from all processors to obtain a global view of the clusters.

Result parallelism :

1. Once one iteration of the algorithm is completed, the updated centroid values are computed by the master process using the aggregated information from the local clusters.
2. The updated centroid values are shared back with the processors, allowing them to update their local copies of the centroids.
3. The iteration process continues until a convergence criterion is met, such as a maximum number of iterations or a small change in centroid positions.
4. Finally, when the algorithm converges, the master process collects the local clusters from all processors and combines them into a global clustering result.

This parallelization technique of K-means clustering helps in distributing the computational load, reducing the execution time, and enabling the processing of large-scale datasets in a distributed environment.

Approach to K-mean clustering explained below :

How did we convert Numerical Value of log data into features ?

Vector Assembler :

All attributes under consideration are numerical or discrete numeric, hence we need to convert them into features using a Vector Assembler. A vector assembler is a transformer that converts a set of features into a single vector column often referred to as an array of features (as shown in below screenshot - in last column).

```
from pyspark.ml.feature import VectorAssembler
from pyspark.sql.types import DoubleType

# Type cast 'response_code' and 'bytes' to double as other columns are of same type
transformed_df = transformed_df.withColumn('response_code', col('response_code').cast(DoubleType()))
transformed_df = transformed_df.withColumn('bytes', col('bytes').cast(DoubleType()))

assemble = VectorAssembler(inputCols=['response_code', 'bytes', 'ip_address_index', 'endpoint_index', 'method_index'], outputCol='features')
assembled_df = assemble.transform(transformed_df)
assembled_df.show()
```

ip_address	method	endpoint	response_code	bytes	ip_address_index	endpoint_index	method_index	features
75.77.74.150	GET	/Archives/edgar/d...	200.0	5302.0	106035.0	58.0	2.0	[200.0,5302.0,106...
176.120.174.1	POST	/Archives/edgar/d...	500.0	34727.0	45226.0	78.0	1.0	[500.0,34727.0,45...
162.200.34.120	POST	/Archives/edgar/d...	403.0	25428.0	37252.0	36.0	1.0	[403.0,25428.0,37...
133.224.84.239	GET	/Archives/edgar/d...	304.0	27248.0	20067.0	95.0	2.0	[304.0,27248.0,20...
211.239.196.64	GET	/Archives/edgar/d...	200.0	32034.0	67278.0	80.0	2.0	[200.0,32034.0,67...
152.154.66.105	GET	/Archives/edgar/d...	200.0	43782.0	31196.0	33.0	2.0	[200.0,43782.0,31...
41.76.166.189	PUT	/Archives/edgar/d...	304.0	53533.0	85888.0	113.0	3.0	[304.0,53533.0,85...
134.27.187.18	POST	/Archives/edgar/d...	404.0	15445.0	20656.0	97.0	1.0	[404.0,15445.0,20...
152.77.55.45	PUT	/Archives/edgar/d...	200.0	44416.0	31567.0	127.0	3.0	[200.0,44416.0,31...
115.252.123.132	POST	/Archives/edgar/d...	500.0	11453.0	9896.0	96.0	1.0	[500.0,11453.0,98...
155.58.151.29	GET	/Archives/edgar/d...	502.0	7442.0	33151.0	142.0	2.0	[502.0,7442.0,331...
13.79.113.61	POST	/Archives/edgar/d...	404.0	10734.0	18086.0	99.0	1.0	[404.0,10734.0,18...
88.193.180.43	POST	/Archives/edgar/d...	304.0	52678.0	113229.0	71.0	1.0	[304.0,52678.0,11...
179.250.21.134	DELETE	/Archives/edgar/d...	500.0	5462.0	47157.0	17.0	0.0	[500.0,5462.0,471...
189.142.75.183	PUT	/Archives/edgar/d...	500.0	37548.0	52896.0	39.0	3.0	[500.0,37548.0,52...
17.173.32.136	GET	/Archives/edgar/d...	404.0	53177.0	41570.0	27.0	2.0	[404.0,53177.0,41...
40.243.18.254	PUT	/Archives/edgar/d...	500.0	23317.0	85233.0	57.0	3.0	[500.0,23317.0,85...
28.163.76.134	POST	/Archives/edgar/d...	304.0	16386.0	77425.0	94.0	1.0	[304.0,16386.0,77...
68.8.175.30	PUT	/Archives/edgar/d...	403.0	38299.0	101755.0	50.0	3.0	[403.0,38299.0,10...
27.79.71.172	DELETE	/Archives/edgar/d...	403.0	15051.0	77223.0	5.0	0.0	[403.0,15051.0,77...

How feature scaling was done to apply k -means clustering ?

1. Standard Scaler and Normalizer :

```
from pyspark.ml.feature import StandardScaler

scale = StandardScaler(inputCol='features', outputCol='standardized')
data_scale = scale.fit(assembled_df)
data_scale_output = data_scale.transform(assembled_df)
data_scale_output.show()
```

```
from pyspark.ml.feature import Normalizer

l1_norm = Normalizer().setP(1).setInputCol("features").setOutputCol("l1_norm")
normalized_df = l1_norm.transform(assembled_df)
normalized_df.show()
```

Used Standard Scaler and Normalizer for following reasons :

1. **Comparable Scale:** By standardizing the data using StandardScaler, all the columns in the feature vector are brought to a comparable scale. This ensures that each feature contributes equally to the distance calculations during clustering. Without scaling, features with larger magnitudes may dominate the clustering process, leading to biased results.
2. **Normalization:** Normalizer not only standardizes the scale but also normalizes the data by subtracting the mean and dividing by the standard deviation. Normalization removes any inherent biases caused by the mean and variance of the data. It helps to center the data around zero and adjusts the spread of the data, making it more suitable for clustering algorithms.

Methods used to find optimal K value after feature scaling and Why :

1. Silhouette Score :

The Silhouette score is a popular evaluation metric used to assess the quality of clustering results. It measures how well each sample in a cluster is matched to other samples in the same cluster compared to samples in other clusters.

- **Quantitative Evaluation:** The silhouette score provides a quantitative measure of the clustering quality. It assigns a score to each data point, ranging from -1 to 1, where higher scores indicate better clustering. By calculating the average silhouette score for a range of k values, you can assess the overall quality of clustering and identify the k value that yields the highest average score.
- **Interpretability :** The silhouette score is intuitive and easy to interpret. A positive score indicates that the data point is well-clustered, as it is closer to points within its own cluster compared to points in other clusters. A negative score suggests that the data point may have been assigned to the wrong cluster.
- **Optimal Cluster Number Selection :** The Silhouette score can be used in combination with the elbow method to determine the optimal number of clusters for k-means clustering. By computing the Silhouette score for different values of k and plotting it against the number of clusters, you can identify the value of k that maximizes the Silhouette score. This helps in choosing the most appropriate number of clusters.

```

Silhouette Score: 0.764989644333913
Silhouette Score: 0.7197068203717747
Silhouette Score: 0.7182825477911833
Silhouette Score: 0.698355445191042
Silhouette Score: 0.6995737686035479
Silhouette Score: 0.6976980204583906
Silhouette Score: 0.6878771293964626
Silhouette Score: 0.6858259154273475

output.agg({"prediction": "avg"}).collect()[0]

Row(avg(prediction)=4.118508333333334)

```

The optimal value of $k = 4$

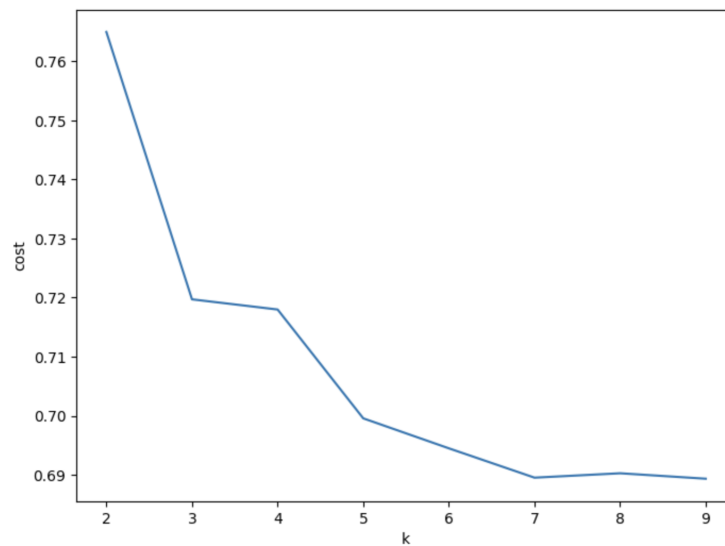


Fig. Graph - Silhouette Score Vs K cluster

2. Elbow method :

The elbow method is used for finding the optimal number of clusters (k) in K-means clustering. It is a visual evaluation method that helps in determining the appropriate value of k based on the distortion or inertia of the clusters.

1. **Sum of Squared Distances** : The elbow method calculates the sum of squared distances between each data point and its nearest centroid within a cluster. It provides a measure of how compact the clusters are. The objective of K-means clustering is to minimize this sum of squared distances.
2. **Visual Assessment** : The elbow method plots the sum of squared distances against the number of clusters. The plot typically forms an "elbow" shape, where the rate of decrease

in the sum of squared distances slows down significantly after a certain number of clusters. The location of the elbow point on the plot indicates the optimal value of k .

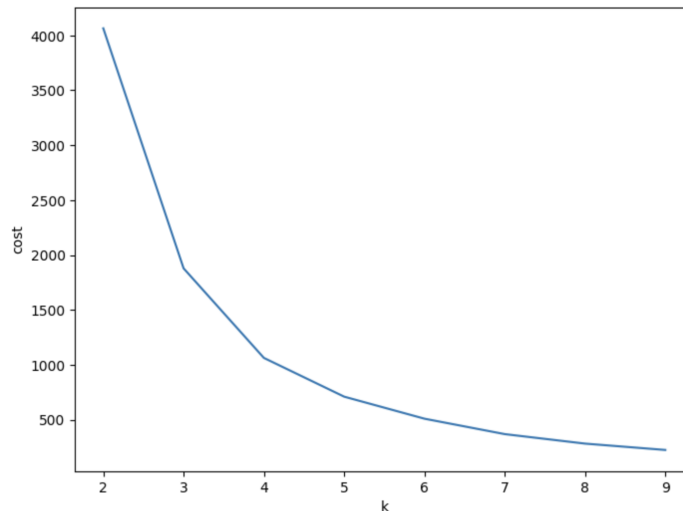


Fig. Graph - Sum of squares (Inertia) Vs K cluster

After finding the optimal value of $K = 4$, we ran the K Means algorithm again and found cluster centers as shown below :

```
cluster_centers = KMeans_fit.clusterCenters()
print("Cluster Centers:")
for center in cluster_centers:
    print(center)
```

Cluster Centers:

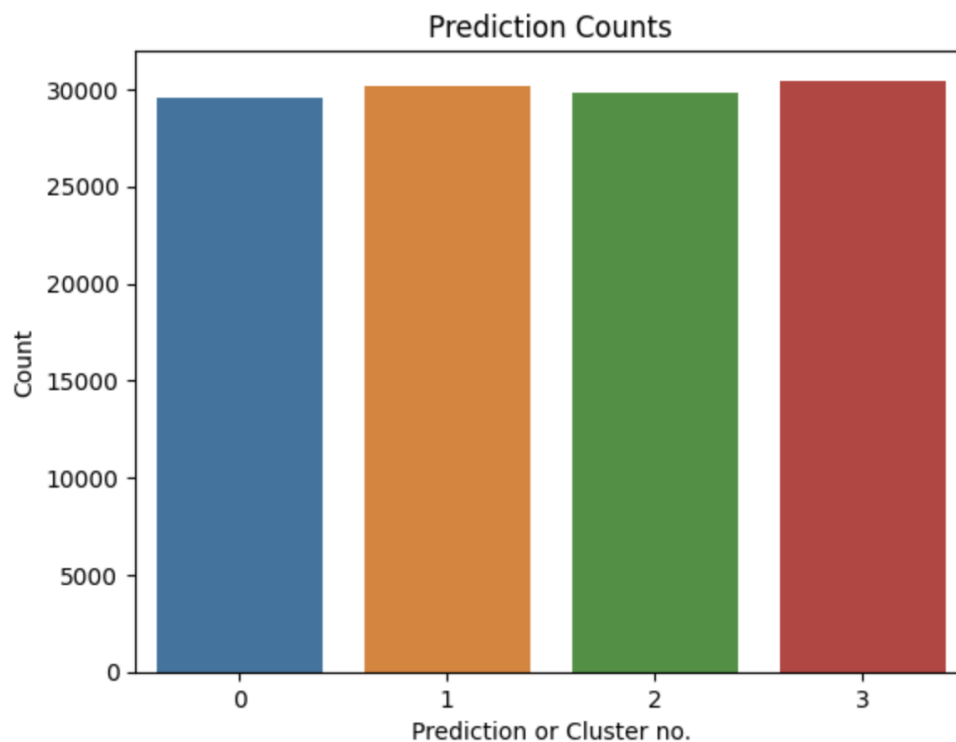
```
[3.5988179  1.765699   2.67834275  1.61208535  2.17780021]
[3.65357431  1.68108226  0.78019812  1.73638456  2.16011274]
[3.61320607  1.75229789  1.79454545  2.65790807  0.52871809]
[3.62779075  1.71889663  1.69123562  0.76372223  0.50598116]
```

Count for all K cluster is as follows :

```
grouped_df = output.groupBy("prediction").count()  
grouped_df.toPandas()
```

	prediction	count
0	1	30141
1	3	30456
2	2	29809
3	0	29594

Graph :



Grouping together predictions of clusters and API methods :

```
pred_method = output.groupBy('prediction', 'method').count().orderBy(col('method')).orderBy(col('prediction'))
pred_method.show()
```

prediction	method	count
0	PUT	15010
0	POST	2133
0	GET	12451
1	POST	2405
1	PUT	14892
1	GET	12844
2	POST	12618
2	GET	2573
2	DELETE	14618
3	DELETE	15456
3	POST	12911
3	GET	2089

Grouping together predictions of clusters and API response codes :

```
pred_res = output.groupBy('prediction', 'response_code').count().orderBy(col('response_code')).orderBy(col('prediction'))
pred_res.show()
```

prediction	response_code	count
0	304.0	4315
0	500.0	4120
0	404.0	4175
0	303.0	4270
0	403.0	4104
0	200.0	4416
0	502.0	4194
1	303.0	4223
1	502.0	4367
1	404.0	4341
1	200.0	4163
1	403.0	4345
1	304.0	4215
1	500.0	4487
2	500.0	4176
2	200.0	4346
2	303.0	4254
2	404.0	4286
2	304.0	4320
2	403.0	4322

only showing top 20 rows

Conclusion :

From the exploratory data analysis (EDA) and K-means clustering performed on the parquet files, we have determined that the optimal number of clusters is $K=4$. This conclusion was drawn using both the Silhouette method and the Elbow method.

Furthermore, each cluster (0, 1, 2, 3) contains a balanced number of data points, as evident from the graphical representation.

Based on the clustering output, we can conclude that clusters 0 and 1 predominantly consist of API methods such as PUSH, POST, and GET. On the other hand, clusters 2 and 3 primarily include API methods such as POST, GET, and DELETE.

Overall, the logs data has been grouped based on the response code and API method using the K-means algorithm, providing insights into the different clusters formed.