Isaac Opiyo

Using Spark and Streamlite to perform analytics on Apache logs data:

1. Acquire the cluster.

Basic information

Name	cluster-0a2f-opiyo-m
Instance ID	8955466269432340287
Description	None
Type	Instance
Status	Running
Creation time	Oct 15, 2023, 6:04:01 pm UTC-04:00
Zone	us-east1-d
Instance template	None
In use by	None
Reservations	Automatically choose (default)
Labels	goog-datap... : enabled, goog-datap... : cluster-0a..., goog-datap... : d4297372-7..., goog-datap... : us-east1
Tags	–

2. Load the data into the master, move the data into HDFS.

I first downloaded the data from the website to the shell using:
wget https://andrewwinslow.com/access.log -O access.log

And then created a hadoop file directory using:

hadoop fs -mkdir /user/isaac/data

Then copied the downloaded file to the hadoop directory using:

hadoop fs -copyFromLocal access.log /user/isaac/data/access.log

Here is the results:

```
isaacopiyo_wabwire@cluster-d151-m:~$ hadoop fs -mkdir /user/isaac/data
isaacopiyo_wabwire@cluster-d151-m:~$ hadoop fs -copyFromLocal access.log /user/isaac/data/access.log
isaacopiyo_wabwire@cluster-d151-m:~$ hadoop fs -ls /user/isaac/data/access.log
-rw-r--r--   1 isaacopiyo_wabwire hadoop   16129865 2023-10-14 20:46 /user/isaac/data/access.log
```

3. Upload to GCP Cluster

I first uploaded the file on my bucket

gsutil cp gs://my-bucket-opiyo/access.log gs://my-bucket-opiyo/script/

Then I submitted a pyspark job as

gcloud dataproc jobs submit pyspark gs://my-bucket-opiyo/script/apache.log
--cluster=cluster-aec1 --region=us-east1
Here are the results:

```
isaacopiyo_wabwire@cluster-0a2f-opiyo-m:~$ gsutil cp gs://my-bucket-opiyo/HelloWolrd.py gs://my-bucket-opiyo/script/
Copying gs://my-bucket-opiyo/HelloWolrd.py [Content-Type=text/x-python-script]...
/ [1 files][  220.0 B/  220.0 B]
Operation completed over 1 objects/220.0 B.
isaacopiyo_wabwire@cluster-0a2f-opiyo-m:~$ gcloud dataproc jobs submit pyspark gs://my-bucket-opiyo/script/HelloWolr
Job [9a6cd0f1197e4aeb8a189a94bc8f9407] submitted.
Waiting for job output...
Hello, World!
Job [9a6cd0f1197e4aeb8a189a94bc8f9407] finished successfully.
done: true
```

4. Extracting 5 IP addresses that generate the most client errors

```
+--------------+-----+
|     client_ip|count|
+--------------+-----+
| 173.255.176.5| 2059|
|  212.9.160.24|  126|
|  13.77.204.88|   78|
|51.210.243.185|   58|
|193.106.30.100|   53|
+--------------+-----+
```

Solution

I relied on google bucket to store the data for future data access. So I started by loading the data from the bucket to Spark.

```
>>> log_data = spark.read.text("gs://my-bucket-opiyo/data/access.log")

>>>
>>> log_data.show(5)
+--------------------+
|               value|
+--------------------+
|                    |
|13.66.139.0 - - [...|
|157.48.153.185 - ...|
|157.48.153.185 - ...|
|216.244.66.230 - ...|
+--------------------+
only showing top 5 rows
```

Then I had to call the following spark packages to start a spark session and transform the dataframe into rows and columns respectively:
from pyspark.sql import SparkSession
from pyspark.sql.functions import regexp_extract, to_timestamp

I created a Spark session as follows:
spark = SparkSession.builder.appName("AccessLogAnalysis").getOrCreate()

Then I defined the apache log pattern for extracting data as:
log_pattern = r'^(\S+) (\S+) (\S+) \[([\w:/]+\s[+\-]\d{4})\] "(\S+) (\S+) (\S+)" (\d{3}) (\d+) "(\S+)" "([^"]+)" "([^"]+)"'

Then I applied the log pattern to the dataframe to extract values for each specified columns:

```
df = log_data
df = df.select(
   regexp_extract("value", log_pattern, 1).alias("client_ip"),
   regexp_extract("value", log_pattern, 2).alias("remote_log_name"),
   regexp_extract("value", log_pattern, 3).alias("user_id"),
   to_timestamp(regexp_extract("value", log_pattern, 4), "dd/MMM/yyyy:HH:mm:ss
Z").alias("date_time"),
   regexp_extract("value", log_pattern, 5).alias("request_type"),
   regexp_extract("value", log_pattern, 6).alias("api"),
   regexp_extract("value", log_pattern, 7).alias("protocol_version"),
   regexp_extract("value", log_pattern, 8).alias("status_code").cast("int"),
   regexp_extract("value", log_pattern, 9).alias("byte").cast("int"),
   regexp_extract("value", log_pattern, 10).alias("referrer"),
   regexp_extract("value", log_pattern, 11).alias("user_agent"),
   regexp_extract("value", log_pattern, 12).alias("response_time")
)
```

Then I printed the resulting dataframe as:

```
df.show(truncate=False)
```

Data Cleaning:

The next step was to clean up the data. So I first identified which columns had nulls,and by how much and filled the nulls with 0 and NaN as:

```
from pyspark.sql.functions import col, sum as spark_sum
```

I created a list of columns as

```
columns = df.columns
```

I then created a list of counts for null values in each column as

```
null_counts = [spark_sum(col(column).isNull().cast("int")).alias(column) for column in
columns]
```

Then I transformed the list to a DataFrame as

```
null_counts_df = df.select(null_counts)
```

Then I printed the result

```
null_counts_df.show()
```

Once I identified columns with nulls, I then filled nulls in those columns with 0 and NaN as:

```
from pyspark.sql.functions import coalesce
```

```
filled_values = {
   "date_time": "N/A",
   "status_code": 0,
   "byte": 0
}

filled_df = df.fillna(filled_values)
```

Data Analysis

At this point my dataframe was ready for analysis. I used the following code to generate the 5 IP addresses with the most client errors:

```
from pyspark.sql import functions as F
```

I used string search in the status_code column to select rows with client errors only. 4 represents the status code for client error.

```
client_errors_df = df.filter(df['status_code'].like("4%"))
```

Then I grouped the rows by client_IP column and counted the rows for each group.

```
ip_error_counts = client_errors_df.groupBy('client_ip').count()
```

I then ordered the results in descending order.

```
top_5_ips_with_errors = ip_error_counts.orderBy(F.desc('count')).limit(5)
```

Then I printed the results

```
top_5_ips_with_errors.show()
```

5. Extracting percentage of each request type (GET, PUT, POST, etc.)

```
>>> # Show the result
>>> request_type_percentages.show()
+------------+-----+-------------------+
|request_type|count|         percentage|
+------------+-----+-------------------+
|        POST|44574| 56.962122373869036|
|        HEAD|  252|0.32203649746971325|
|         GET|33233| 42.469202065122936|
|            |  193|0.24663906353831214|
+------------+-----+-------------------+
```

Here is the implementation

```
from pyspark.sql import functions as F
```

I started by grouping the data frame by request type and counting the number of rows per each group

```
request_type_counts = df.groupBy('request_type').count()
```

Then counted the total number of requests

```
total_requests = df.count()
```

Lastly, I calculated the percentage of each request type

```
request_type_percentages = request_type_counts.withColumn(
    'percentage',
    (F.col('count') / total_requests) * 100
)
```

Then I printed the results:

```
request_type_percentages.show()
```

6. Extracting percent of the responses types

```
>>> response_percentages_df.show()
+--------------------+-----+-----------------+
|       Response Type|Count|       Percentage|
+--------------------+-----+-----------------+
|Informational (10...|    0|              0.0|
|Successful (200-299)|70607|90.23028165414303|
|Redirection (300-...| 2814|3.596074221745131|
|Client Error (400...| 4638|5.927005060573531|
|Server Error (500...|    0|              0.0|
+--------------------+-----+-----------------+
```

Here is the implementation

```
from pyspark.sql import functions as F
```

I started by filtering the DataFrame into different response types using the status_code column

```
informational = df.filter((df['status_code'] >= 100) & (df['status_code'] < 200))
successful = df.filter((df['status_code'] >= 200) & (df['status_code'] < 300))
redirection = df.filter((df['status_code'] >= 300) & (df['status_code'] < 400))
client_error = df.filter((df['status_code'] >= 400) & (df['status_code'] < 500))
server_error= df.filter((df['status_code'] >= 500) & (df['status_code'] < 600))
```

Then I counted the total responses

```
total_responses = df.count()
```

Then I defined a function to generate the percentage for each response type

```
def calculate_percentage(response_df, response_type):
    count = response_df.count()
    percentage = (count / total_responses) * 100
    return (response_type, count, percentage)
```

Then I changed the naming of the response types as

```
response_types = [
    ("Informational (100-199)", informational),
    ("Successful (200-299)", successful),
    ("Redirection (300-399)", redirection),
    ("Client Error (400-499)", client_error),
    ("Server Error (500-599)", server_error)
]
```

Then I called my function

```
response_percentages = [calculate_percentage(df, response_type) for response_type, df in
response_types]
```

Lastly, I printed the results

```
response_percentages_df = spark.createDataFrame(response_percentages, ["Response
Type", "Count", "Percentage"])
response_percentages_df.show()
```