Log Analytics: A Big Data Approach

By: Adit Shrimal and Varun Hande

Introduction

In the digital era, the enormous volume of data being generated daily has given rise to the concept of Big Data. Processing this vast amount of data requires robust technologies capable of handling both batch and stream processing. Batch processing deals with data collected over time and processed all at once, while stream processing involves the real-time analysis of continuous data streams.

One crucial application of stream processing is Log Analytics. Log files, produced by various network devices and applications, hold a record of runtime events and transactions. Analyzing these logs helps uncover valuable insights, aiding in decision-making, improving system performance, and enhancing security protocols.

To manage the challenges of Big Data and Log Analytics, technologies such as Apache Spark, Kafka, and Hadoop Distributed File System (HDFS) form a powerful stack. Apache Spark is known for efficiently processing large-scale data workloads, Kafka allows real-time data pipelining, and HDFS provides a reliable, scalable storage system.

In this project, we leverage these technologies to address our problem statement: performing Log Analytics on vast amounts of log data. We implement a data pipeline where log files are fed into a Kafka producer, then consumed and processed by Spark Streaming, and finally stored as Parquet files in HDFS. This project demonstrates the potential of combining Kafka, Spark, and HDFS for real-time data processing and analytics.

Objective

Our project is divided into two sections, each having a distinct objective and approach.

Section A

In this section, our goal is to understand and implement log analytics, taking guidance from a comprehensive case study. The problem involves exploratory data analysis (EDA) of log data to extract two specific sets of information:

1. The endpoint that received the highest number of invocations on each day of the week throughout the dataset.
2. The total number of 404 status codes generated on each day of the week throughout the dataset.

Our high-level approach involves processing and transforming the raw log data using Apache Spark. We will then group and aggregate the data to generate insights on the highest invoked endpoints and frequency of 404 status codes for each day of the week.

Section B

In Section B, we venture into the realm of Big Data technologies, combining Kafka, Spark Streaming, Parquet files, and HDFS. The problem statement revolves around establishing a data flow from log files to HDFS through a Kafka producer, a Kafka consumer, and Spark Streaming, with the final results stored as Parquet files in HDFS.

Our approach to tackle this task is as follows:

1. First, we create a Kafka producer that reads from the log file and publishes the data to a Kafka topic.
2. Next, we implement a Kafka consumer subscribed to the same topic. The consumer checks for new data periodically, ingests the data, and transforms it using Spark.
3. We then perform EDA on the transformed data, similar to the techniques used in Section A.
4. Finally, we convert the results from the analysis into Parquet file format and store these files in HDFS.

This pipeline allows us to harness the benefits of real-time data streaming with Kafka, large-scale data processing with Spark, and reliable storage with HDFS.

Methodology

Section A

For Section A, we focused on the processing and transformation of raw log data using Apache Spark. Here's a brief overview of the steps taken:

Data Processing

1. **Data Extraction**: We loaded the data directly from the log files. The data was unstructured, and hence, regular expressions (regexp_extract) were used to extract relevant information and structure the data into columns. For example, to extract the 'host' from the log, we used:

```
regexp_extract('value', r'^([^\s]+\s)', 1).alias('host')
```

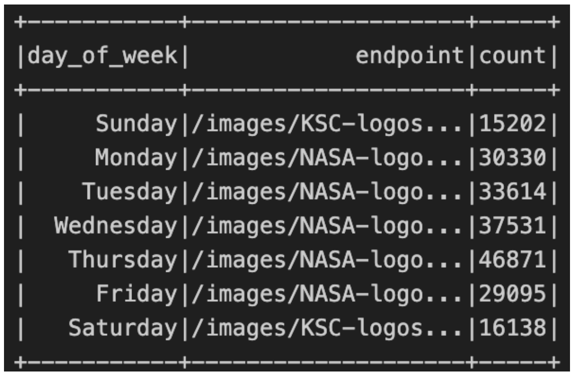
2. **Null Handling**: We checked for null values in our data. We found that the **content_size** column contained null values. Considering the nature of our data, we replaced these null values with 0.

3. **Timestamp Parsing**: The **timestamp** field in our data was a string, including timezone information. To make it usable, we converted this string into a timestamp format. We used a User Defined Function (UDF) in PySpark, which maps the month name to a month number and formats the timestamp string.

Exploratory Data Analysis

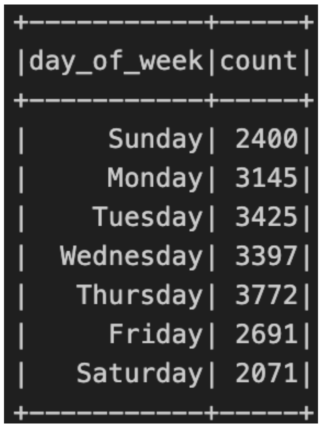
We performed various EDA steps on the processed data to generate insights. Although we conducted several analyses, we focus on the following two requirements:

1. **Highest Invoked Endpoints per Day**: To find the most frequently accessed endpoint for each day of the week, we added a new column **day_of_week** to our dataframe and grouped the data by this column and **endpoint**. We then counted the number of instances and ordered them in descending order for each day of the week, selecting only the first row (most frequently accessed endpoint).



```
+-----------+--------------------+-----+
|day_of_week|            endpoint|count|
+-----------+--------------------+-----+
|     Sunday|/images/KSC-logos...|15202|
|     Monday|/images/NASA-logo...|30330|
|    Tuesday|/images/NASA-logo...|33614|
|  Wednesday|/images/NASA-logo...|37531|
|   Thursday|/images/NASA-logo...|46871|
|     Friday|/images/NASA-logo...|29095|
|   Saturday|/images/KSC-logos...|16138|
+-----------+--------------------+-----+
```

2. **Total 404 Status Codes per Day:** We calculated the total number of 404 status codes generated on each day of the week for the entire dataset.



```
+-----------+-----+
|day_of_week|count|
+-----------+-----+
|     Sunday| 2400|
|     Monday| 3145|
|    Tuesday| 3425|
|  Wednesday| 3397|
|   Thursday| 3772|
|     Friday| 2691|
|   Saturday| 2071|
+-----------+-----+
```

These steps provided us with an overview of the log data and allowed us to gather specific insights as required by the problem statement.

Section B

1. Environment Setup

The environment setup included the installation of HDFS (Hadoop Distributed File System), Apache Kafka, and a standalone Apache Spark cluster.

- HDFS is a distributed file system designed to handle large data sets running on commodity hardware. It is used here as a distributed storage system to store our resulting parquet files.
- Apache Kafka is an open-source, distributed event streaming platform. We chose Kafka for its high throughput, built-in partitioning, replication, and fault-tolerance. In Kafka, a 'topic' is a category to which the data records are published. We created a topic named "log_topic".
- Apache Spark is a cluster computing system that offers comprehensive libraries and APIs for big data processing and analytics. We set up a standalone Spark cluster for the sake of simplicity, but in a production environment, it is generally integrated with cluster management solutions like Mesos or YARN.

2. Kafka Publisher

The Kafka Publisher plays a crucial role in reading data from log files and publishing it to Kafka. We encapsulated this functionality in the **`` `section-b/kafka/publisher.py` ``** script. The script accepts three arguments: the log file path, the batch size for data to be sent, and the Kafka topic name. The key parts of the script are:

- **KafkaProducer**: This class is used to produce messages to Kafka topics. Here, it's instantiated with a list of broker addresses (in our case, **`localhost:9092`**).
- **Producer loop**: The script reads each line from the log file, sends it to the Kafka topic using **`producer.send(topic_name, line.encode('utf-8'))`**, and tracks how many messages have been sent. Once the count reaches the specified batch size, the producer sends any remaining messages and pauses to prevent Kafka from being overwhelmed (**`producer.flush()`**).

  In Kafka, **`linger_ms`** and **`batch_size`** are important configuration parameters that control how the producer batches messages for sending to the Kafka broker. The **`linger_ms`** setting is a time delay parameter that the producer uses to wait for more messages to arrive before sending the current batch. This can help increase the efficiency of sending messages by reducing the number of requests sent. On the other hand, **`batch_size`** is a size limit that determines how many messages the producer can accumulate into one batch before it needs to be sent.

For our analysis, we have kept **`linger_ms`** and **`batch_size`** at their default values. The default **`linger_ms`** is 0 milliseconds, meaning the producer will not artificially delay sending messages, and the batch is sent as soon as it is ready. The default **`batch_size`** is 16384 bytes (16KB), which is the maximum size of the batch that can be created. These defaults provide a balance between latency and throughput under most operating conditions. However, they could be tweaked based on specific requirements or to optimize for either lower latency or higher throughput.

3. Spark Streaming Job

This part of the pipeline is implemented in the **`section-b/spark-streaming/consumer.py`** script and performs real-time processing on the streaming data fetched from Kafka. The script reads log data from Kafka, parses it, performs window-based counting, and writes the output to HDFS.

- **Reading from Kafka**: Spark's structured streaming API provides the **`readStream`** function to read data in real-time from various sources, including Kafka. The data read from Kafka is in binary format, which we cast to STRING.
- **Parsing the log data**: We use Spark's built-in **`regexp_extract`** function to parse the log line into separate columns based on the log format.
- **Timestamp conversion**: The timestamp string in the log is in a specific format, which needs to be converted to a timestamp data type. This is done using a combination of the **`unix_timestamp`** and **`to_timestamp`** functions.
- **Adding a watermark**: We added a watermark to the timestamp column to handle late data and manage the amount of stateful data that Spark needs to remember while processing the stream.
- **EDA and writing to HDFS**: We grouped the data based on different columns and a window on the timestamp, counted the number of requests, and wrote the results to different directories in HDFS. Writing to HDFS provides a resilient and distributed storage for our output data, which can then be used by downstream applications or services.

A key consideration in real-time processing is the size of the window and the sliding interval, which must be carefully chosen to balance latency and computation overhead. In our case, we chose a window size of 3 hours.

4. Writing data to HDFS

The final step is to store the processed data for future use or further analysis. HDFS was chosen as it offers a reliable and scalable storage for large datasets, which is a typical requirement in log analysis applications. Each DataFrame was written into a different directory in HDFS to keep the results separate and organized.

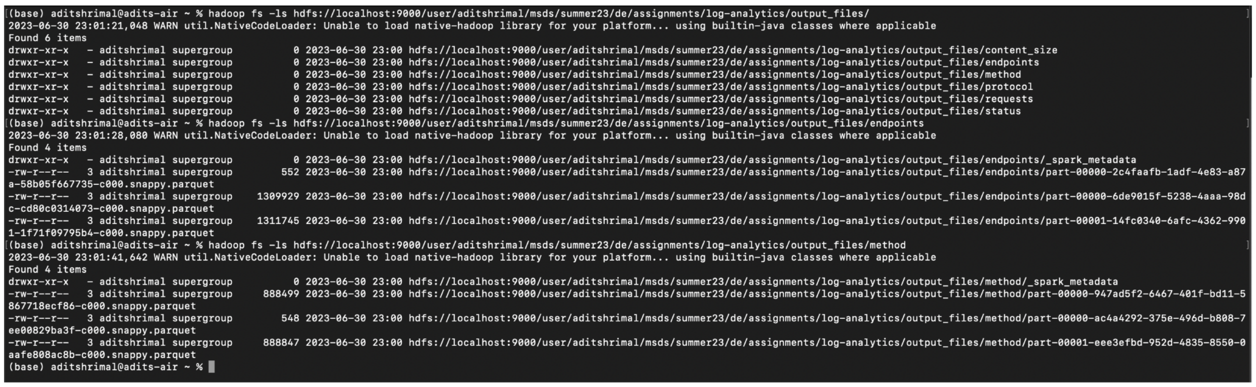
The data is written into Parquet format which is a columnar storage file format that is optimized for use with big data processing frameworks. Parquet provides efficient data compression and encoding schemes to save storage space and to support complex data types.

We used Spark's built-in functionality to write the data into Parquet files in HDFS. The **`writeStream`** method of the **`DataFrameWriter`** API is used to write streaming data in a tabular format. The 'append' mode is used to add new records to existing files.

In terms of best practices, we used checkpointing to provide fault-tolerance for the streaming queries. A checkpoint directory is specified where Spark will write all the checkpoint information. This is crucial to ensure the recovery of the application in case of any failures.

For managing and controlling the rate at which the data is processed, we used **`trigger`** in the **`writeStream`** method. This helps control the frequency with which the streaming data is processed. We set the processing time to '**`10 seconds`**', meaning Spark will check for new data every 10 seconds.

Finally, the **`awaitAnyTermination`** function is called to wait for any of the queries to terminate in order to stop the streaming application. It is essential to call this function at the end to keep the application running and processing the incoming data.



```
(base) aditshrimal@adits-air ~ % hadoop fs -ls hdfs://localhost:9000/user/aditshrimal/msds/summer23/de/assignments/log-analytics/output_files/
2023-06-30 23:01:21,048 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Found 6 items
drwxr-xr-x   - aditshrimal supergroup          0 2023-06-30 23:00 hdfs://localhost:9000/user/aditshrimal/msds/summer23/de/assignments/log-analytics/output_files/content_size
drwxr-xr-x   - aditshrimal supergroup          0 2023-06-30 23:00 hdfs://localhost:9000/user/aditshrimal/msds/summer23/de/assignments/log-analytics/output_files/endpoints
drwxr-xr-x   - aditshrimal supergroup          0 2023-06-30 23:00 hdfs://localhost:9000/user/aditshrimal/msds/summer23/de/assignments/log-analytics/output_files/method
drwxr-xr-x   - aditshrimal supergroup          0 2023-06-30 23:00 hdfs://localhost:9000/user/aditshrimal/msds/summer23/de/assignments/log-analytics/output_files/protocol
drwxr-xr-x   - aditshrimal supergroup          0 2023-06-30 23:00 hdfs://localhost:9000/user/aditshrimal/msds/summer23/de/assignments/log-analytics/output_files/requests
drwxr-xr-x   - aditshrimal supergroup          0 2023-06-30 23:00 hdfs://localhost:9000/user/aditshrimal/msds/summer23/de/assignments/log-analytics/output_files/status
(base) aditshrimal@adits-air ~ % hadoop fs -ls hdfs://localhost:9000/user/aditshrimal/msds/summer23/de/assignments/log-analytics/output_files/endpoints
2023-06-30 23:01:28,080 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Found 4 items
drwxr-xr-x   - aditshrimal supergroup          0 2023-06-30 23:00 hdfs://localhost:9000/user/aditshrimal/msds/summer23/de/assignments/log-analytics/output_files/endpoints/_spark_metadata
-rw-r--r--   3 aditshrimal supergroup        552 2023-06-30 23:00 hdfs://localhost:9000/user/aditshrimal/msds/summer23/de/assignments/log-analytics/output_files/endpoints/part-00000-2c4faafb-1adf-4e83-a87
a-58b05f667735-c000.snappy.parquet
-rw-r--r--   3 aditshrimal supergroup    1309929 2023-06-30 23:00 hdfs://localhost:9000/user/aditshrimal/msds/summer23/de/assignments/log-analytics/output_files/endpoints/part-00000-6de9015f-5238-4aaa-98d
c-cd80c0314073-c000.snappy.parquet
-rw-r--r--   3 aditshrimal supergroup    1311745 2023-06-30 23:00 hdfs://localhost:9000/user/aditshrimal/msds/summer23/de/assignments/log-analytics/output_files/endpoints/part-00001-14fc0340-6afc-4362-990
1-1f71f09795b4-c000.snappy.parquet
(base) aditshrimal@adits-air ~ % hadoop fs -ls hdfs://localhost:9000/user/aditshrimal/msds/summer23/de/assignments/log-analytics/output_files/method
2023-06-30 23:01:41,642 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Found 4 items
drwxr-xr-x   - aditshrimal supergroup          0 2023-06-30 23:00 hdfs://localhost:9000/user/aditshrimal/msds/summer23/de/assignments/log-analytics/output_files/method/_spark_metadata
-rw-r--r--   3 aditshrimal supergroup     888499 2023-06-30 23:00 hdfs://localhost:9000/user/aditshrimal/msds/summer23/de/assignments/log-analytics/output_files/method/part-00000-947ad5f2-6467-401f-bd11-5
867718ecf86-c000.snappy.parquet
-rw-r--r--   3 aditshrimal supergroup        548 2023-06-30 23:00 hdfs://localhost:9000/user/aditshrimal/msds/summer23/de/assignments/log-analytics/output_files/method/part-00000-ac4a4292-375e-496d-b808-7
ee00829ba3f-c000.snappy.parquet
-rw-r--r--   3 aditshrimal supergroup     888847 2023-06-30 23:00 hdfs://localhost:9000/user/aditshrimal/msds/summer23/de/assignments/log-analytics/output_files/method/part-00001-eee3efbd-952d-4835-8550-0
aafe808ac8b-c000.snappy.parquet
(base) aditshrimal@adits-air ~ %
```

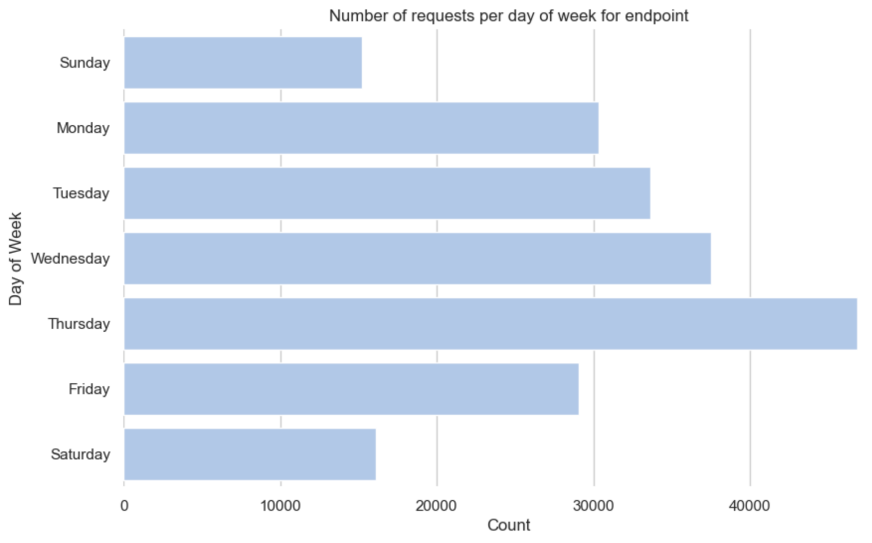
In summary, this methodology showcases a standard pipeline for real-time log analysis using Apache Kafka and Spark Streaming, and storing the processed data in HDFS. It involves practices such as batching in Kafka, windowing, watermarking, and checkpointing in Spark Streaming, and using Parquet as a storage format for large datasets. The pipeline is scalable and can be adjusted based on the specific requirements of the data and application.

Data Analysis and Insights

Section A: Batch Processing Insights

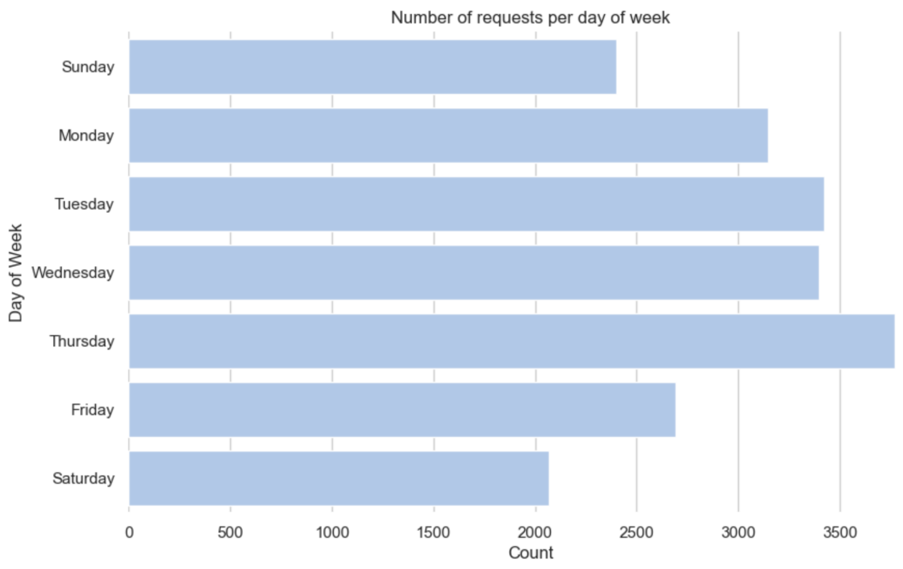
1. Highest Number of Invocations for a Specific Day of the Week

   In this analysis, we focus on finding out the endpoint that received the highest number of invocations on a specific day of the week, along with the corresponding count of invocations. This information is crucial as it helps us identify peak periods of activity on the server and can guide decisions such as when to carry out server maintenance or when to expect heavy loads.



2. Total Number of 404 Status Codes per Weekday

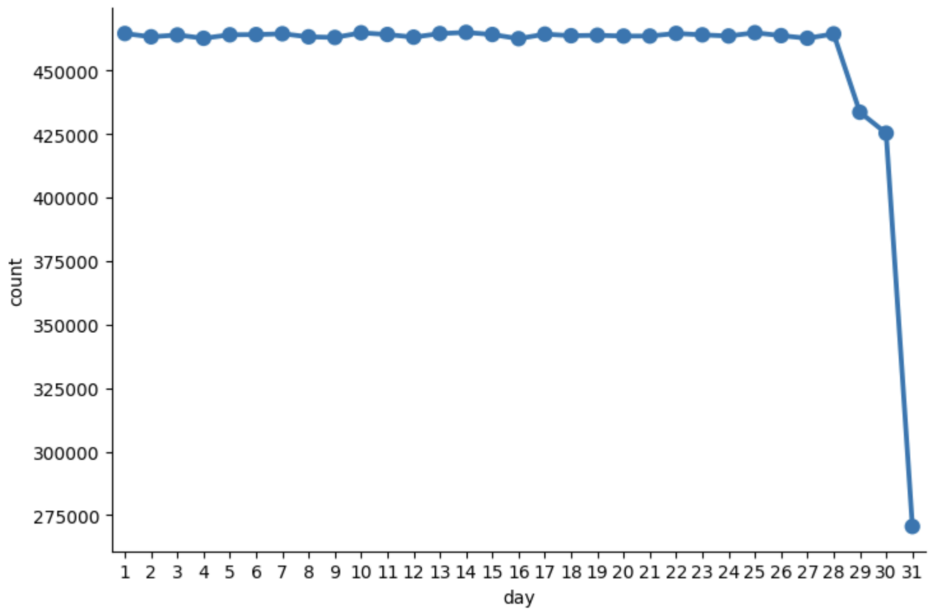
   The total number of 404 status codes generated per weekday for the entire dataset provides an overview of when the most errors are occurring. If we notice that there are consistently more 404 errors on certain days, this could indicate a regular event or process that's causing broken links. Alternatively, it could point to a day when users are more active and thus more likely to stumble upon pages that don't exist.



Section B: Real-time Streaming Insights

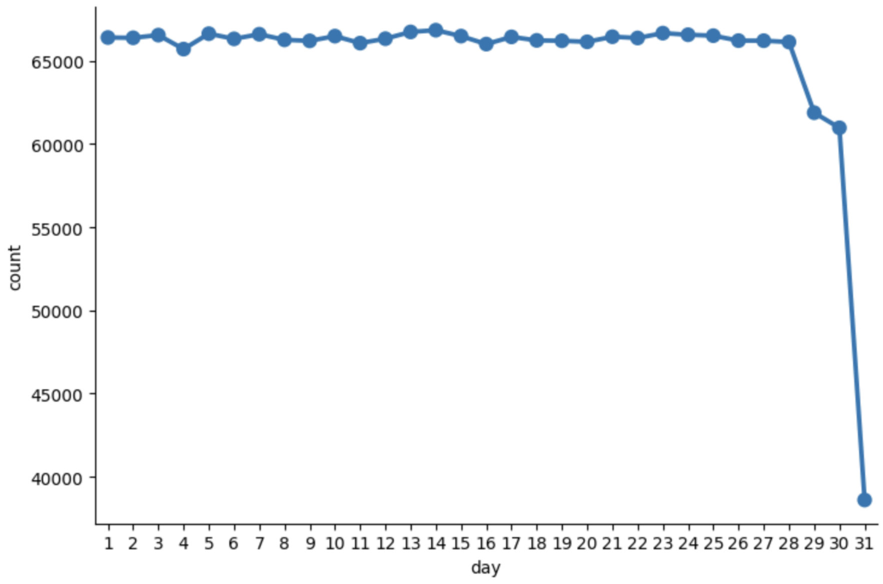
In this section, we are employing Spark structured streaming to analyze the data in real-time. The results are being stored as Parquet files in an HDFS system.

1. **Number of Unique Hosts per Day of the Month**
   Keeping track of the number of unique hosts that interact with our server each day of the month provides insight into the daily usage patterns. This can show if there is a trend in the volume of unique visitors based on the time of the month.



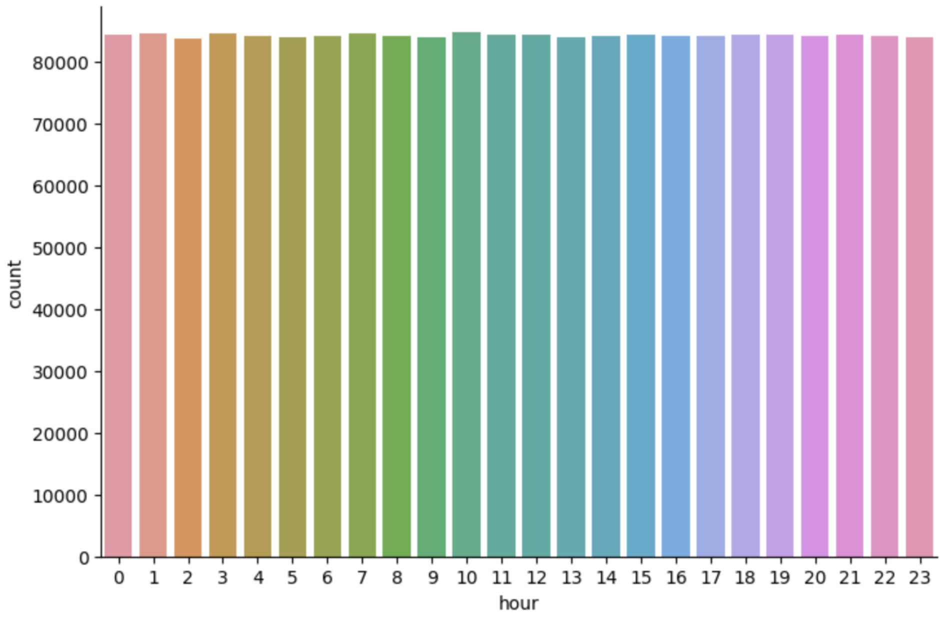
2. 404 Errors Per Day of the Month

Monitoring the number of 404 errors per day of the month allows us to detect any recurring issues based on the day of the month. This might highlight problems with specific features or content that are only available or heavily used on certain days.



3. 404 Errors By Hour of the Day

Examining the 404 errors by hour of the day offers a more granular perspective. If a specific hour consistently shows a high number of 404 errors, it might be due to scheduled tasks or server updates that cause temporary disruptions in service.



Conclusion

Our project presented a robust exploration into the realm of web server logs analysis using modern data engineering tools and methods. Leveraging both batch and stream processing capabilities of PySpark, we handled a large dataset to derive meaningful insights.

In Section A, we employed batch processing to conduct a comprehensive Exploratory Data Analysis (EDA) on the web server logs. The results were illuminating, revealing valuable insights such as the most frequently accessed endpoint on specific days and the total number of 404 status codes generated each day of the week.

In Section B, we stepped into the world of real-time data with Spark Structured Streaming. Using this approach, we performed analyses like identifying the number of unique hosts per day, tracking the occurrence of 404 errors on a daily basis, and monitoring 404 errors hour-by-hour.

Future Scope

1. **Machine Learning and Clustering Use-cases**
   A promising avenue for advancing this work lies in the integration of machine learning models for advanced log data analysis. This could involve:

   - **Clustering algorithms** to spot common patterns or anomalies in requests
   - **Predictive models** to forecast server load based on historical data

   Such enhancements could lead to sophisticated monitoring and alerting systems and provide valuable insights for optimal server management.

2. **Embracing Apache Flink**
   A potential evolution of the data processing pipeline could include the integration of Apache Flink. Known for its high processing speed, ease of use, and powerful stream processing capabilities, Apache Flink has several appealing features:

   - **Event-time processing and windowing** for complex real-time analytics
   - **Advanced fault-tolerance mechanisms** for reliable and accurate results

   Transitioning to Flink could open doors to additional capabilities and efficiencies, marking it as an exciting avenue for future project development.