**ECS765P - Big Data Processing Assignment Report**

**(Task 1 to 4)**

**Student Name:** Sai Sujit Varma Gottumukkala  
 **Student ID:** 240436744  
 **Submission Date:** 11/04/2025

Task 1

**Task Objective:** Analysing of Twitter Data

**Initial Stage:**

I have loaded the provided Twitter dataset using Spark’s CSV reader with the following configuration which is provided in the Task 1 Guidelines PDF:

twitter\_df = spark.read.csv("s3a://data-repository-bkt/ECS765/Twitter/twitter.csv", header=True, inferSchema=True)

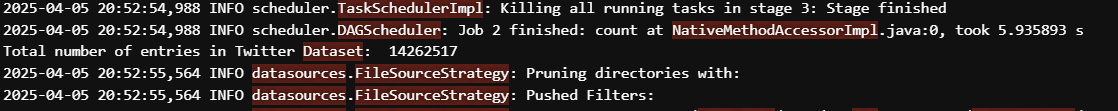
**1. Question 1: Finding the total no of entries**

I calculated the total number of entries using Spark’s ‘count()’ method:

print("Total number of entries in Twitter Dataset: ", twitter\_df.count())

**Findings:**

The dataset successfully loaded and contained a **1,42,62,517** number of entries, indicating a very large dataset.



**Challenges:**

* Initially, I was unsure if Spark would correctly infer data types, particularly for the **timestamp** column, which is stored in a **yyyyMMddHHmmss** integer format. However, using **inferSchema=True** allowed Spark to correctly identify and process the data types, including converting the timestamp into a proper timestamp type for further analysis.

**Insight:**

* From this initial loading process, I learned how effective Spark is in handling diverse and potentially problematic dataset schemas.

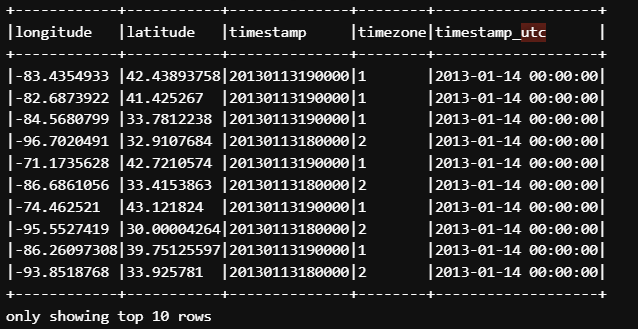
**2. Question 2: Filtering Weekday Tweets**

To analyze tweets made only during weekdays (Monday-Friday):

* I first converted timestamps to the correct **UTC** format as the tweets are exclusively only from the United States of America.
* I have also added a new column named ‘**Timestamp\_UTC**’ which displays the UTC time.
* I used Spark's built-in function **dayofweek()** to filter for weekdays

**Findings:**

* Successfully filtered data, showing tweets only from weekdays clearly sorted by date.
* I’ve rechecked if the Dates are correctly filtered by checking a few initial timestamps in the calendar.



**Challenges:**

* Initially, understanding how Spark represents weekdays numerically was confusing. After checking the documentation, I have found that Monday is represented as 2 and Sunday as 1 or 7, depending on context.

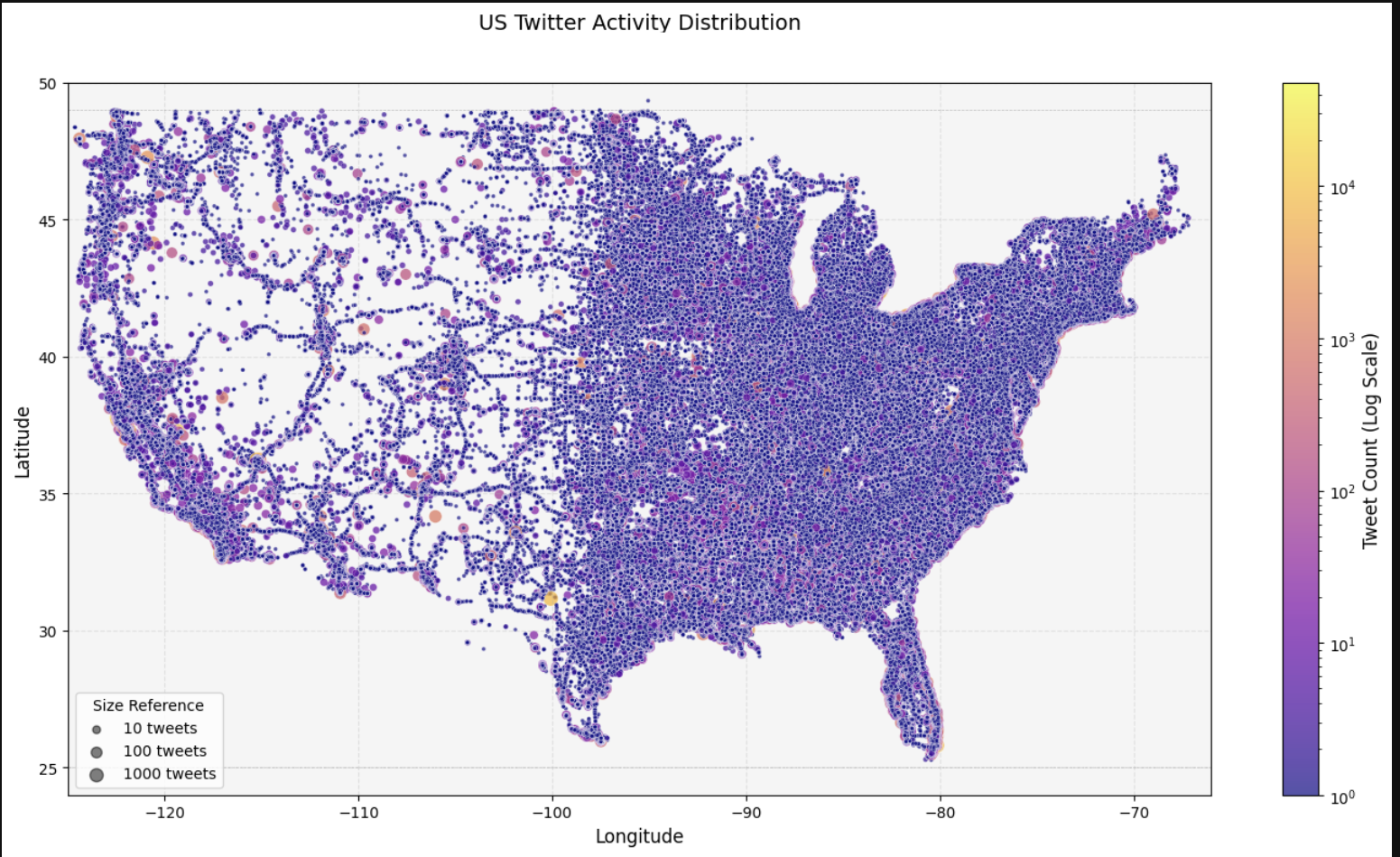
**Insights:**

* I learned how to work with date and time functions in Spark, specifically how to filter data based on day of the week. This knowledge is essential when analyzing time-based data, especially in big data environments where time zone handling and filtering are critical.

**3. Question 3: Geographical Distribution of Tweets**

To analyze the geographical distribution:

I have used the original dataset which includes all the days including Saturday and Sunday unlike for Question 2. I have created a new column called **location** by combining both the longitude and latitude values using the **concat\_ws()** function. I then grouped the data by location and counted the number of tweets per location which clearly displayed where the location is on the map. Finally, I exported the results to a CSV file in the S3 bucket using location\_counts.coalesce(1).write.csv(f"s3a://{s3\_bucket}/output/location\_counts.csv",mode='overwrite', header=True) and downloaded the csv to my jupyter notebook using ccc method bucket mv command for further visualization in the local machine.



To visualize the tweet distribution across different locations in the USA, I used a scatter plot as instructed. I used the **pandas** Dataframe to load the dataset and used **matplotlib** library to create a clear and concise scatter plot of Geographical Distribution of Tweets.

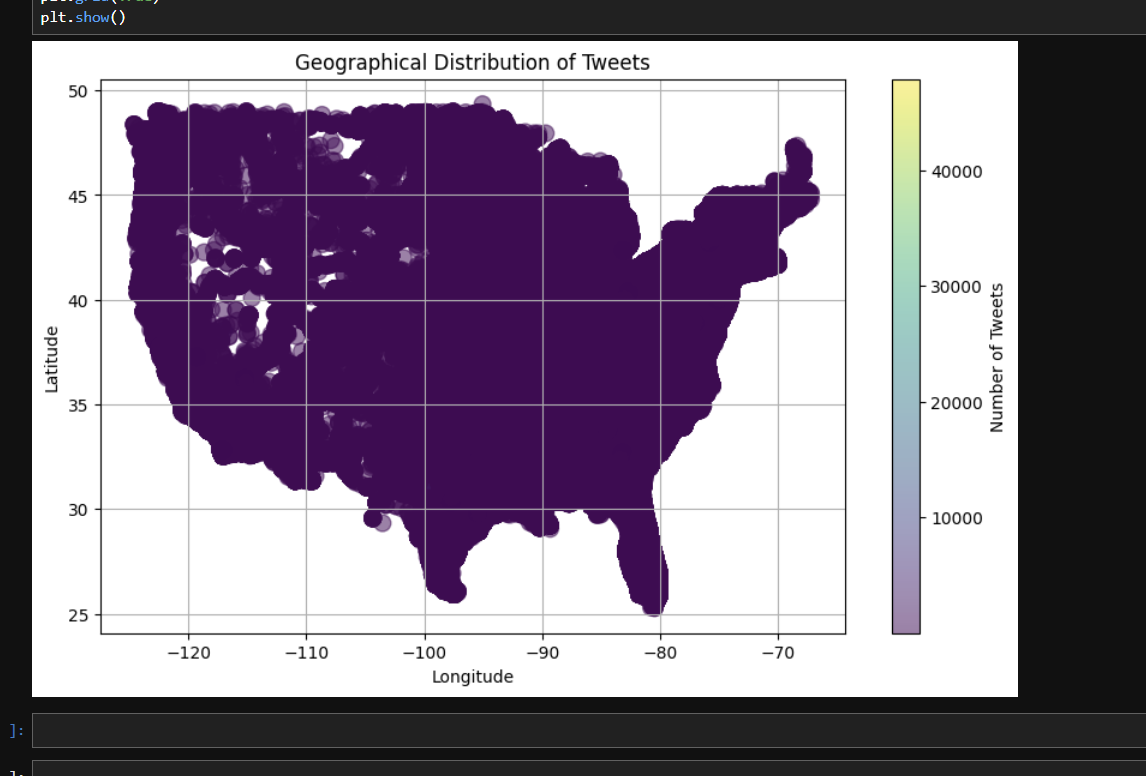
**Explanation of the Visualization:**

* **Color and Size:**The colors and sizes of points on the scatter plot represent the density of tweet activity. Areas marked in **yellow** indicate very high tweet density (I have included the log scale for a better representation of large numbers), while **darker purple** areas represent lower tweet density.

**Challenges:**

Initially, I have faced difficulties in visualizing the geographical distribution of tweets in a meaningful way. The first few plots I generated were not informative. They either lacked clarity or failed to highlight significant areas of high tweet activity. The plots were either too sparse or over-cluttered, with high-density regions overwhelming the view and smaller areas of interest getting lost. In some cases, the color schemes and size references did not accurately convey the tweet densities.

Some plots I have generated which are not up to the mark:



To resolve the issue with the initial unclear visualizations, I reviewed several scatter plot examples and experimented with different methods to improve the clarity of the plot. I have tested various techniques such as adjusting the color scale, altering point sizes, and using logarithmic scaling to handle the vast range of tweet densities. By experimenting with these settings, I was able to create a more informative and clear visualization that displayed both densely and sparsely populated tweet areas effectively.

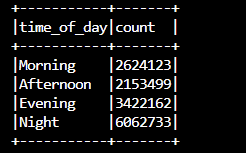
**Insights:**

* From the visualization, it’s clear that tweet activity is significantly higher in densely populated urban areas, particularly noticeable along the East and West Coasts, and major urban centers.
* I have gained valuable experience in combining PySpark based big data analytics with local visualization tools and libraries like **matplotlib** which is used to plot this data.
* Improved skills in geographical data visualization and interpretation, enhancing my ability to extract insights from geospatial data.
* I learned that sometimes issues arise not from coding mistakes but from incorrect logical approaches.

**Note:** Due to the large file size (280 MB), raw plotting CSV data was not uploaded as permitted by instructor guidance.

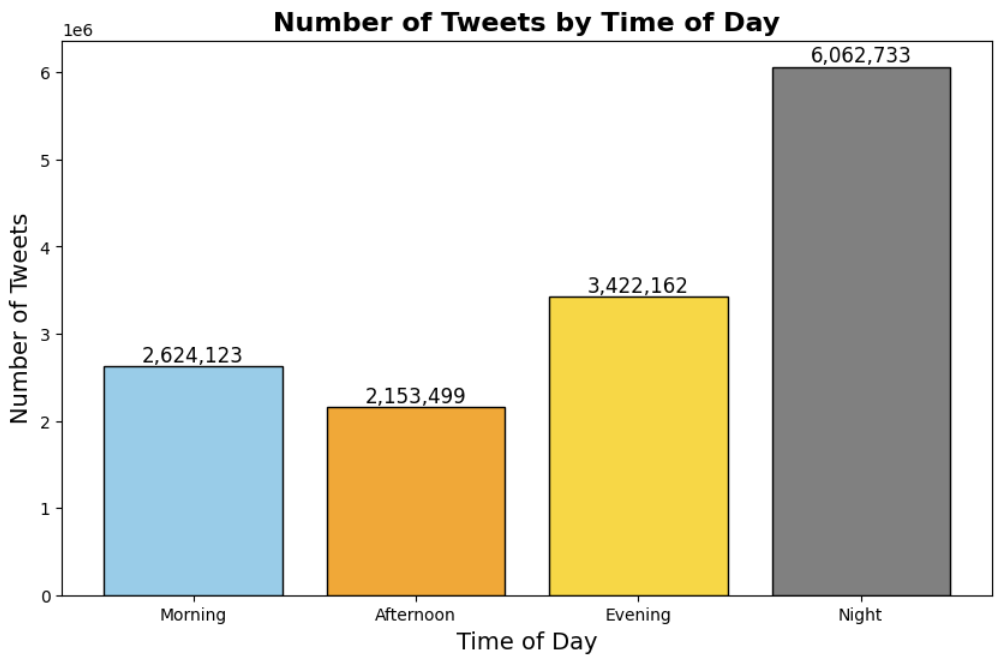
**4. Question 4: Time of Day Categorization**

I have extracted the hour and day of the week from the UTC timestamp using the hour() and date\_format() functions. I then categorized the tweets based on the hour into four categories: Morning, Afternoon, Evening, and Night. I used the **count()** function in PySpark to aggregate the total number of tweets for each time of day. Then the result was exported in a csv file then to my local machine for visualization.



**Visualization Process:**

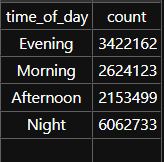
To visualize the tweet distribution across different times of day, I used a bar chart as instructed. I used the **pandas** Dataframe to load the dataset and used **matplotlib** library to create a clear and concise bar chart of the number of tweets per time category.



**Findings:**

* The categorization revealed clear patterns of tweet activity during different times of the day. I have observed that most tweets occurred in the Night, followed by Evening, Morning, and Afternoon.

**Challenges:**

* The challenge was dealing with time zone differences. I made sure to ensure that all timestamps were converted to UTC to maintain consistency across different time zones and all the Tweets are mainly from The United States of America.
* When exporting the results to a csv file, I found out that the Time of Day column is not in the proper chronological order.

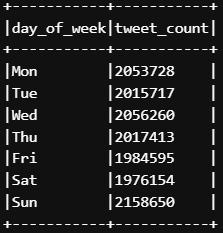
So, I have reordered the csv to (Morning, Afternoon, Evening, and Night) using **pd.Categorical** for a clean visualization.

**Insights:**

* I learned how to categorize time data effectively using conditional logic in Spark. This task helped me understand how tweet activity can vary depending on the time of day, which can be crucial for analyzing user behavior patterns.
* This task also deepened my understanding of how to use pandas and matplotlib to manipulate data and create more meaningful visualizations.

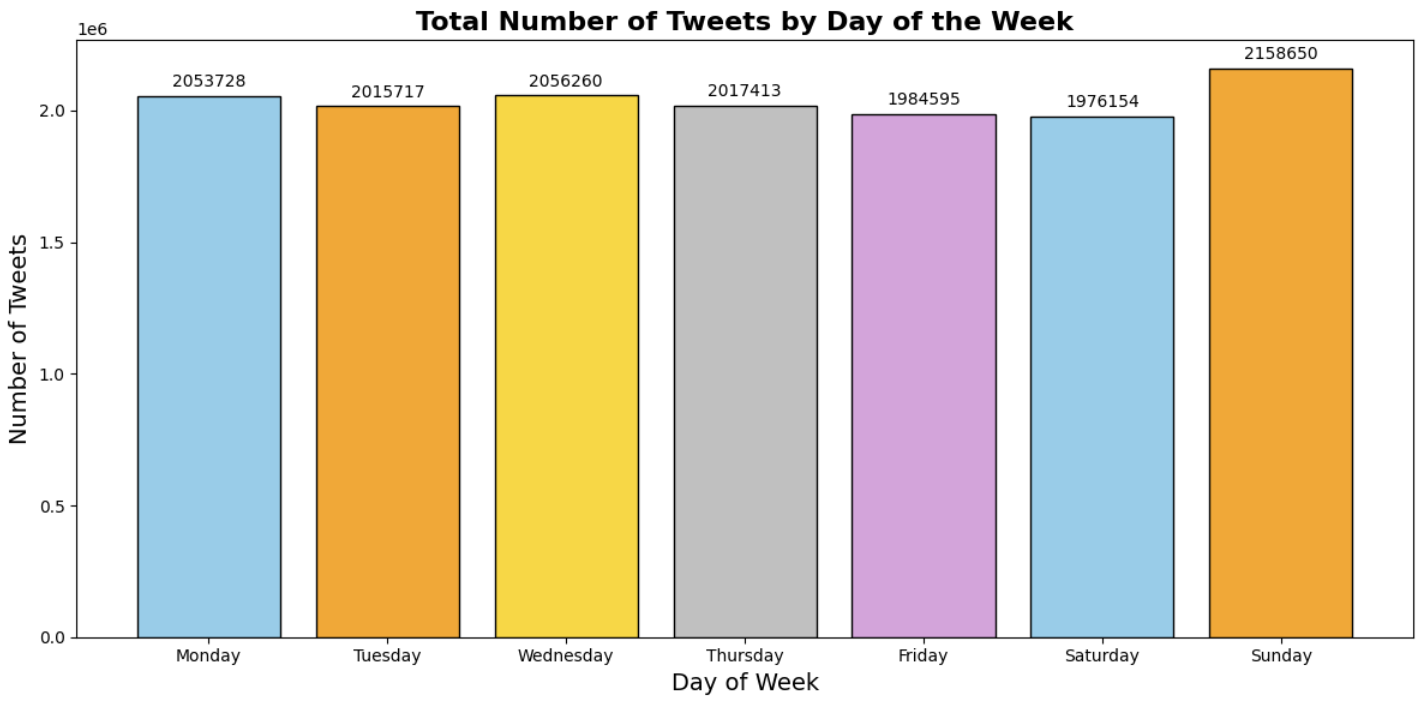
**5. Question 5: Grouping Tweets by Day of the Week**

I aggregated the tweets by day of the week and counted the number of tweets per day. I have used the **count()** function in PySpark to aggregate the total number of tweets for each day. Then the results were exported to a CSV file for visualization.



**Visualization Process:**

To visualize the tweet count clearly, I have loaded the CSV file into a **pandas** DataFrame and created abar chart as instructed. I have used the **matplotlib** library to create a clear bar chart of the number of tweets per each day of week.



* Each bar represents the total number of tweets for a specific day, with colors assigned to each day for better visual distinction. The height of each bar corresponds to the number of tweets recorded for that day.
* The exact tweet count for each day is displayed above each bar, making it easier to compare the tweet volumes across days.

**Findings:**

* The aggregation revealed clear differences in tweet volume across the week, with Saturday having the fewest tweets and Sunday having the most tweets.
* The number of tweets is relatively consistent across all days, with Sunday showing higher engagement than the other day.
* This suggests that Twitter usage is quite stable throughout the week, with minor fluctuations, possibly influenced by factors like weekend behavior or specific events.

**Challenges:**

* The primary challenge I encountered was ensuring that the days of the week appeared in the correct order on the plot. By default, pandas might not order categorical data as expected when visualized. So to solve this, I used **panda’s Categorical** type to manually define the correct order of days (Monday to Sunday) and sorted the DataFrame accordingly. This allowed me to create a clear, intuitive plot where the days were displayed in the correct chronological order.

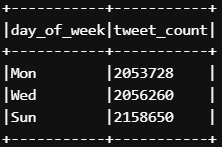
**Insights:**

* I learned how to aggregate data based on categorical variables such as days of the week. This experience highlighted the importance of sorting and grouping in analyzing time-series data.
* This task emphasized the importance of clear annotations and proper ordering in data visualizations, which are critical for accurate interpretation of the results.

**6. Question 6: Filtering Unusually High Tweet Days**

The aim is to identify days of the week with an unusually high number of tweets compared to the average. So, I first calculated the average number of tweets across all days using PySpark's aggregate function **avg()**.

Then I filtered the DataFrame to select only those days where the total number of tweets exceeded this calculated average using the function **filter()**:



**Findings:**

* The result highlighted specific days like Monday, Wednesday, and Sunday when tweet activity was notably higher than average.
* Identifying these days provides valuable insight, suggesting potential external events, news, or specific user behaviors driving heightened engagement.

**Challenges:**

* A challenge I faced during this analysis was identifying unusually high tweet days while ensuring accuracy in handling the aggregation. After calculating the average number of tweets across all days using the **avg()** function, I needed to retrieve the result from the aggregation to use it in my filtering step.
* In PySpark, operations like **agg()** return a DataFrame, so I needed to extract the actual average value for comparison. To do this, I used the **collect()** function, which collects the results from a DataFrame and brings them back to the local machine as a list. I then accessed the average tweet count using [0] to get the first value from the result

**Insights:**

* I learned how to effectively use statistical measures (such as averages) in PySpark to objectively identify outliers or anomalies in data.
* This exercise reinforced my skills in data-driven decision-making, using straightforward statistics to provide clarity and justification for analytical conclusions.

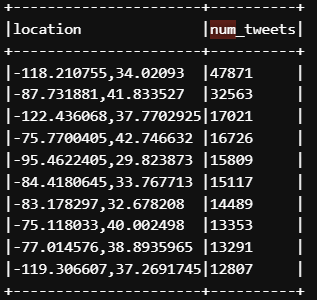
**7. Question 7: Identifying Top 10 Tweet Locations**

In this question, the task was to identify the top 10 locations with the highest number of tweets. To achieve this, I grouped the dataset by location (which I had previously created by combining longitude and latitude into a single location column) and counted the number of tweets for each location.

The first step was to group the dataset by the location column and count the number of tweets in each location using the **groupBy()** and **count()** functions in PySpark:

After performing this aggregation, I used **orderBy()** to sort the locations in descending order based on the tweet count (num\_tweets), so that the locations with the highest tweet counts appeared at the top.

Next, I limited the result to the top 10 locations using the **limit(10)** function.



**Findings:**

* I identified the locations with the highest tweet volumes. This information could be useful for targeting specific regions for marketing or content distribution.

**Challenges:**

* A challenge I encountered was dealing with the large number of unique locations in the dataset. With so many individual geographic coordinates, it was difficult to initially pinpoint which areas had the highest tweet counts. To address this, I created a location column by combining the **longitude** and **latitude**, which allowed me to aggregate tweet counts for specific regions. Additionally, using the **groupBy()** function to count tweets by location and sorting the results made it easier to identify the top locations.

**Insights:**

* I learned how to efficiently group and aggregate data based on geographic locations, which is a key technique for understanding user behavior in spatial contexts.
* This task also highlighted the importance of geographic analysis in social media data, where regions with higher user engagement can reveal insights into demographic trends, interests, and potential market opportunities.

————————————**END OF TASK 1**————————————

Task 2

**Task Objective:** Analysis of Movie Ratings Data

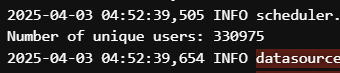
**Initial Stage:**

I have loaded the provided Twitter dataset using Spark’s CSV reader with the following configuration which is provided in the Task 2 Guidelines PDF:

ratings\_df = spark.read.csv("s3a://data-repository-bkt/ECS765/MovieLens/ratings.csv", header=True, inferSchema=True)

**1. Question 1: Finding the number of unique users who rated**

To find how many unique users had rated movies, I applied two functions:

* I used the **distinct()** function to select only unique user IDs.
* I then applied the **count()** function to count these unique users

**Findings:**

The dataset successfully loaded and contained **3,30,975** unique users who rated the movies.

**Challenges:**

* Initially when I tried to Load the 2 datasets provided for this Task 2, I always got an error code:1. So, I loaded the **ratings.csv** first for the Questions that required it. Then later Loaded the **movies.csv** from the Question that required it. Following this method did not arouse any errors.

**Insights:**

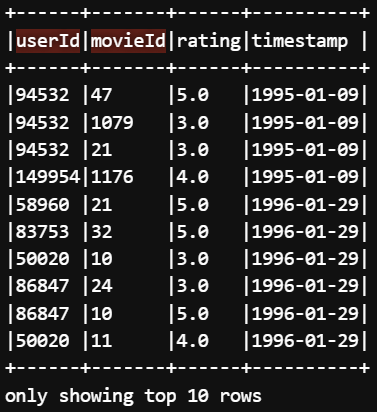
* This task taught me how to efficiently load large datasets into Spark and use **distinct()** and **count()** to quickly analyze the uniqueness and count of entries in a large dataset, a foundational skill for working with big data.

**2. Question 2: Filtering Data by Date**

I converted timestamps from Unix epoch format to a readable date format (‘**YYYY-MM-DD**’) by using the **from\_unixtime()** function within **withColumn()**

I used **withColumn()** to add a new timestamp column with human-readable date values.

Then, I sorted the data chronologically using **orderBy()**

****

**Challenges:**

* Converting timestamps correctly was crucial for accurate time-based analysis. I had to verify that the conversion didn’t give any errors or inconsistencies, especially since the datasets have a very large volume of data. Sorting the data by date was also important for analyzing rating trends over time.

**Insights:**

* I gained an understanding of how to manipulate timestamps in PySpark and the importance of properly handling and formatting date-related data for chronological analysis.

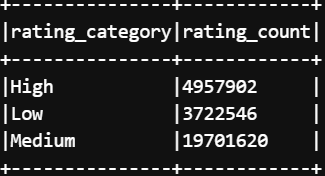
**3. Question 3: Rating Distribution Analysis**

First, I categorized ratings into three groups: "Low" (1-2), "Medium" (3-4), and "High" (5) as instructed. To get this, I used the **when()** function in combination with **withColumn()** to create a new column, **rating\_category**, based on the value of the **rating** column:

After categorizing the ratings into "Low" (1-2), "Medium" (3-4), and "High" (5) clearly using conditional logic in PySpark, I aggregated the data to count how many ratings fell into each category.

**when()** function allowed me to create a new column based on conditions. **withColumn()** helped me create a new **rating** column.

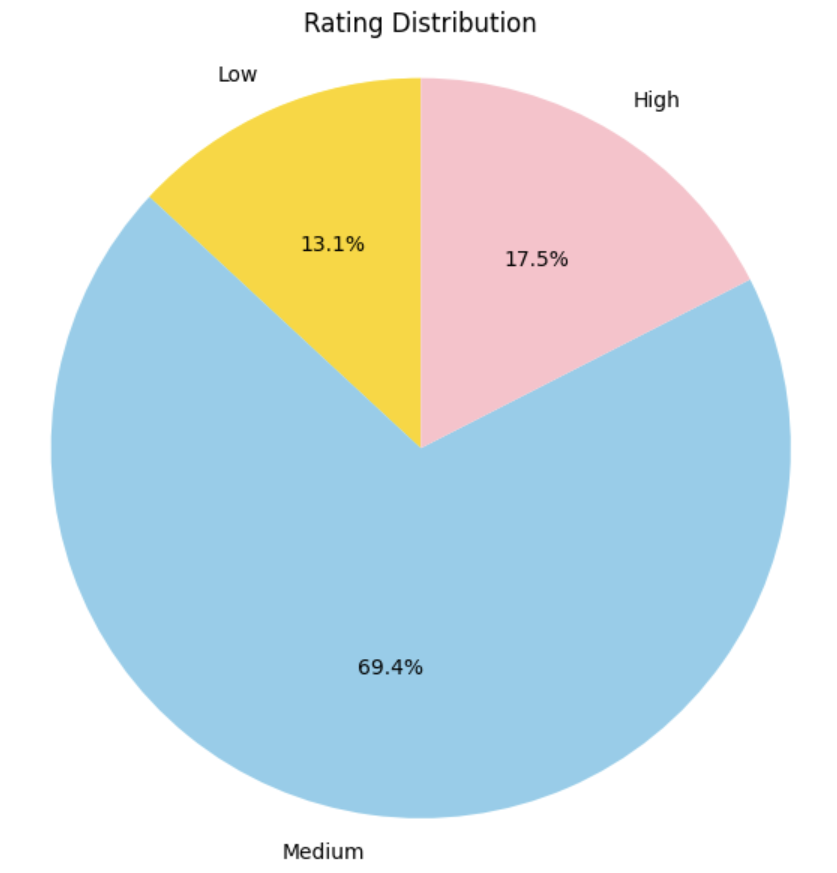
Then I used the **filter()** function that removed rows with null and the rating that fall out of the instructed values from the DataFrame, ensuring that only valid categories are included in the subsequent analysis.



I have exported these results into a csv from S3 bucket for local visualization..

**Visualization Process:**

To visualize the rating distribution clearly, I loaded the CSV file into a **pandas** DataFrame and created a pie chart using **matplotlib**.



### 

### **Explanation of Visualization:**

**Low Ratings (Gold)**: Represents ratings from 1 to 2, accounting for approximately **13.1%** of total ratings.

**Medium Ratings (Blue)**: Includes ratings from 3 to 4, clearly dominating with approximately **69.4%** of total ratings.

**High Ratings (Pink)**: Contains only ratings of 5, representing about **17.5%** of the dataset.  
I chose distinct colors to differentiate each rating category clearly and enhance readability.

I have used autopct='%1.1f%%' to clearly display the percentage of ratings within each segment, making the pie chart intuitive and easy to interpret. I have alsoused **plt.axis('equal')** to ensure the **pie chart** was clearly circular, which is critical for correctly interpreting the proportions.

### **Findings:**

Clearly, the majority of ratings were in the "Medium" category, indicating most users give moderate ratings rather than extremes. The "High" rating category, while smaller than "Medium," significantly outweighed the "Low" ratings, suggesting a generally positive perception among users.

### **Challenges:** A significant challenge I faced during this analysis was dealing with unexpected outliers in the rating distribution. When examining the ratings, I initially assumed that all ratings would fall within the expected range of 1 to 5. However, upon closer inspection of the data, I found a small number of ratings that were outside of this expected range. To solve this issue, I implemented a filtering step to ensure that only valid ratings (between 1 and 5) were included in the analysis. I used the filter() function to remove any rows where the rating value was outside the valid range (for ex. 0.5).

### **Insights:**

* I learned the importance of verifying data quality before performing analysis. Even small discrepancies, such as out-of-range values, can significantly affect the outcome.
* Handling such issues proactively through filtering and validation is essential for maintaining the integrity of the analysis.

**4. Question 4: Time-of-Year Rating Analysis**

To explore clear seasonal patterns, I first extracted the year and month from timestamps

I have used the **year()** and **month()** functions to extract the respective year and month values from the **timestamp** column. This allowed me to categorize the data into "Early Year" (January to June) and "Late Year" (July to December).

I then created two new columns **year** and **month using withColumn()**. These columns were derived from the existing **timestamp** column

Other API’s I have used:

**col()**:  
The **col()** function is used to refer to a specific column in the DataFrame. It was necessary to access the month column for the **when()** condition.

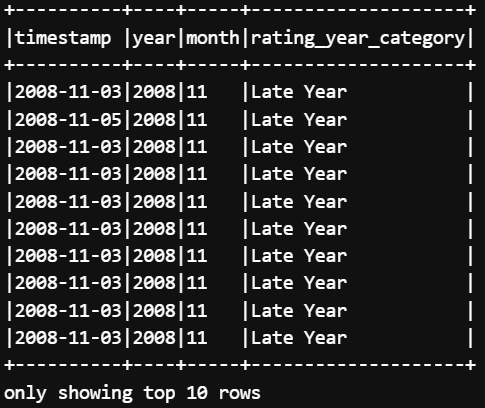
**between()**:  
This function checks if a column’s value falls within a specified range. In my case, I used **col("month").between(1, 6)** to check if the month value is between January and June (inclusive).

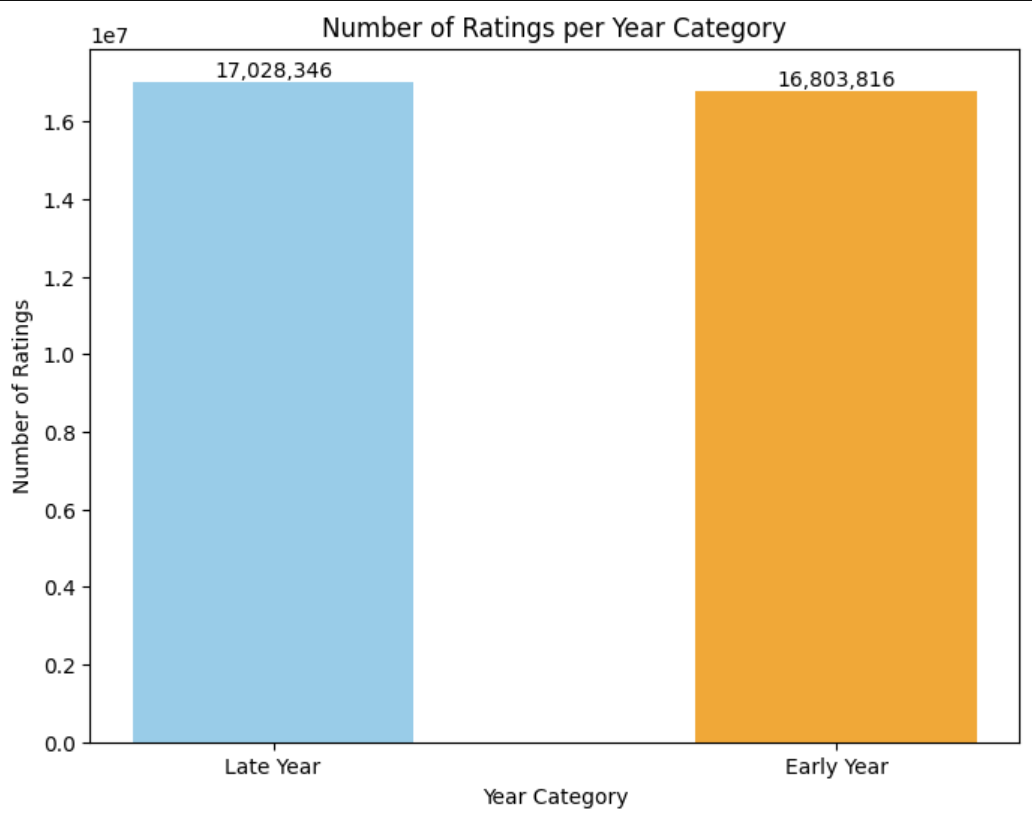
**otherwise()**:  
This is the "else" condition in the **when()** function. If the **month** value doesn't fall between 1 and 6, the **otherwise()** method assigns the category "Late Year".

After categorizing the ratings, I grouped the data by the newly created column **rating\_year\_category** to count how many ratings occurred in each of the two categories (Early Year vs. Late Year).

I grouped the data by the **rating\_year\_category** column using **groupBy()** to segregate the ratings into "Early Year" and "Late Year" categories. After grouping the data, I used the **count()** function to count how many records (ratings) exist in each category. This gave me the total number of ratings for both "Early Year" and "Late Year".

Once I had the aggregated data, I exported the results to a CSV file and created a bar chart for visualization





**Visualization Process:**

To visualize the No of Ratings per year clearly, I loaded the CSV file into a **pandas** DataFrame and created a bar chart using **matplotlib**.

### 

### **Findings:**

The number of ratings for **Late Year** (17,028,346) was slightly higher than the **Early Year** (16,803,816), indicating that users are somewhat more active in the second half of the year. This insight could be useful in understanding the seasonal behavior of MovieLens users, where certain events or factors might influence higher engagement during the latter half of the year.

### 

### **Challenges:**

### One challenge I encountered was ensuring accurate categorization of the months into "Early Year" and "Late Year." The initial categorization was prone to error if months weren't clearly defined. So I used the **when()** and **between()** functions to explicitly categorize ratings based on the month value, which provided a clear and reliable categorization.

### 

### **Insights:**

* I learned how to categorize and analyze data based on temporal factors, specifically by defining clear periods such as "Early Year" and "Late Year" for analysis.
* I also deepened my understanding of **grouping**, **counting**, and **categorizing** data in PySpark, which are key operations for time-based data analysis.
* The visualization provided a clearer insight into how user engagement fluctuates over different periods of the year, which is useful for understanding seasonal trends in user behavior.

**5. Question 5: Genre-Based Rating Analysis**

The objective was to analyze the distribution of ratings across different movie genres. To achieve this, I first loaded the **movies.csv** and joined it with the ratings data (ratings.csv) which I have loaded initially in Task 2 based on **movieId**, then I counted how many ratings each genre received.

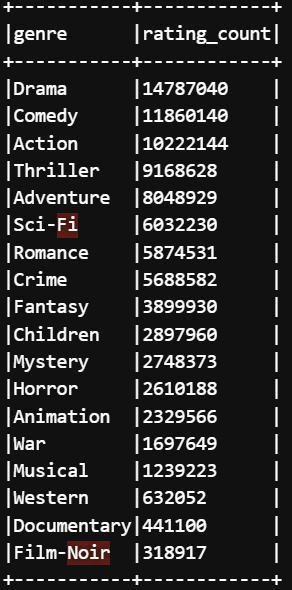
I joined the ratings dataset with the movies dataset on **movieId** using **join()** functions to get the genres associated with each movie

Many movies in the dataset have multiple genres (e.g., "Action|Adventure"), so I needed to split the genres and explode the values into separate rows. This was done using the **split()** and **explode()** functions.

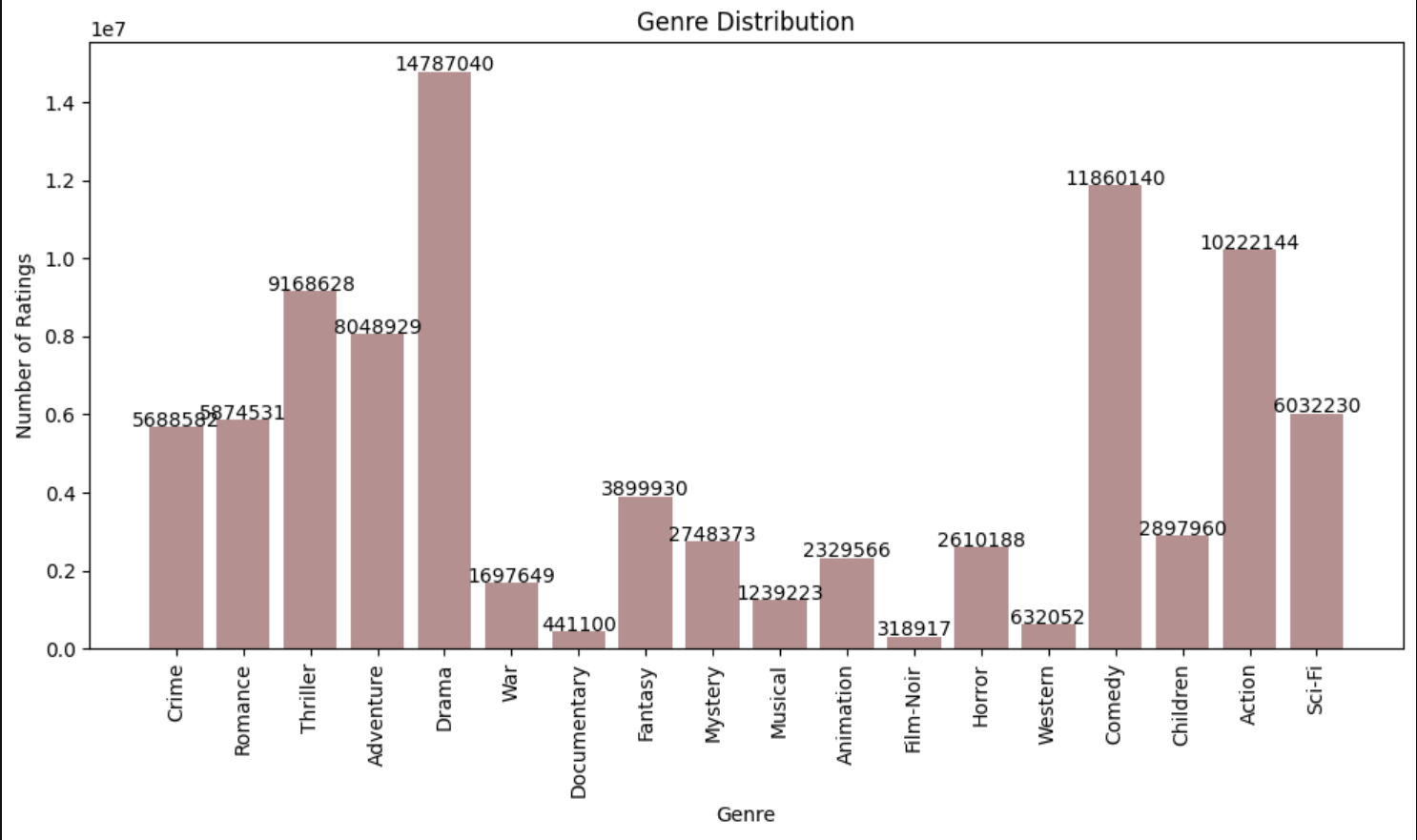
Next, I filtered the dataset to only include valid genres provided in the task guidelines pdf

I used **isin()** function to filter rows where the genre is part of the specified list of valid genres. This ensures that only movies with recognized genres are included in the analysis. Then I grouped the filtered data by genre and counted the number of ratings for each

genre using **groupBy()** and **count()** functions.



After aggregating the data, I exported the results from the S3 bucket and created a **bar chart** to visualize the distribution of ratings across the different genres.



**Visualization Process:**

To visualize the rating distribution clearly, I loaded the CSV file into a **pandas** DataFrame and created a pie chart using **matplotlib**.

### **Findings:**

**Drama** received the highest number of ratings, with over **14.7 million ratings**, followed closely by **Comedy** and **Action** genres.  
**Film-noir** and **Documentary** genres had comparatively fewer ratings.  
The chart clearly shows that genres like Drama, Comedy, and Action dominate the dataset in terms of user ratings, while genres like Horror and Thriller received moderate ratings.

### **Challenges:**

A challenge I faced during this analysis was dealing with multi-genre movies. Movies that belong to multiple genres (e.g., Action and Adventure) needed to be split into separate rows for accurate aggregation.  
So I used the **split()** and **explode()** functions to separate the genres, ensuring that each genre got counted independently in the rating analysis.

### **Insights:**

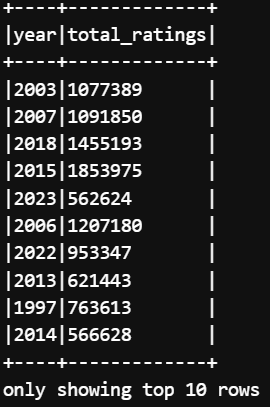
* I learned how to handle multi-valued categorical data effectively by using functions like **split()** and **explode()**. This is crucial when dealing with datasets where records have more than one attribute (e.g., multiple genres).
* I also learned how to perform genre-based analysis by grouping data and counting ratings per genre, which helped me identify the most popular genres and understand user preferences.
* Finally, I gained experience in visualizing categorical data using bar charts, which made the genre distribution easy to interpret.

**6. Question 6: Temporal Analysis of Ratings**

The goal was to analyze the number of ratings per year to identify trends in the amount of user activity over time.

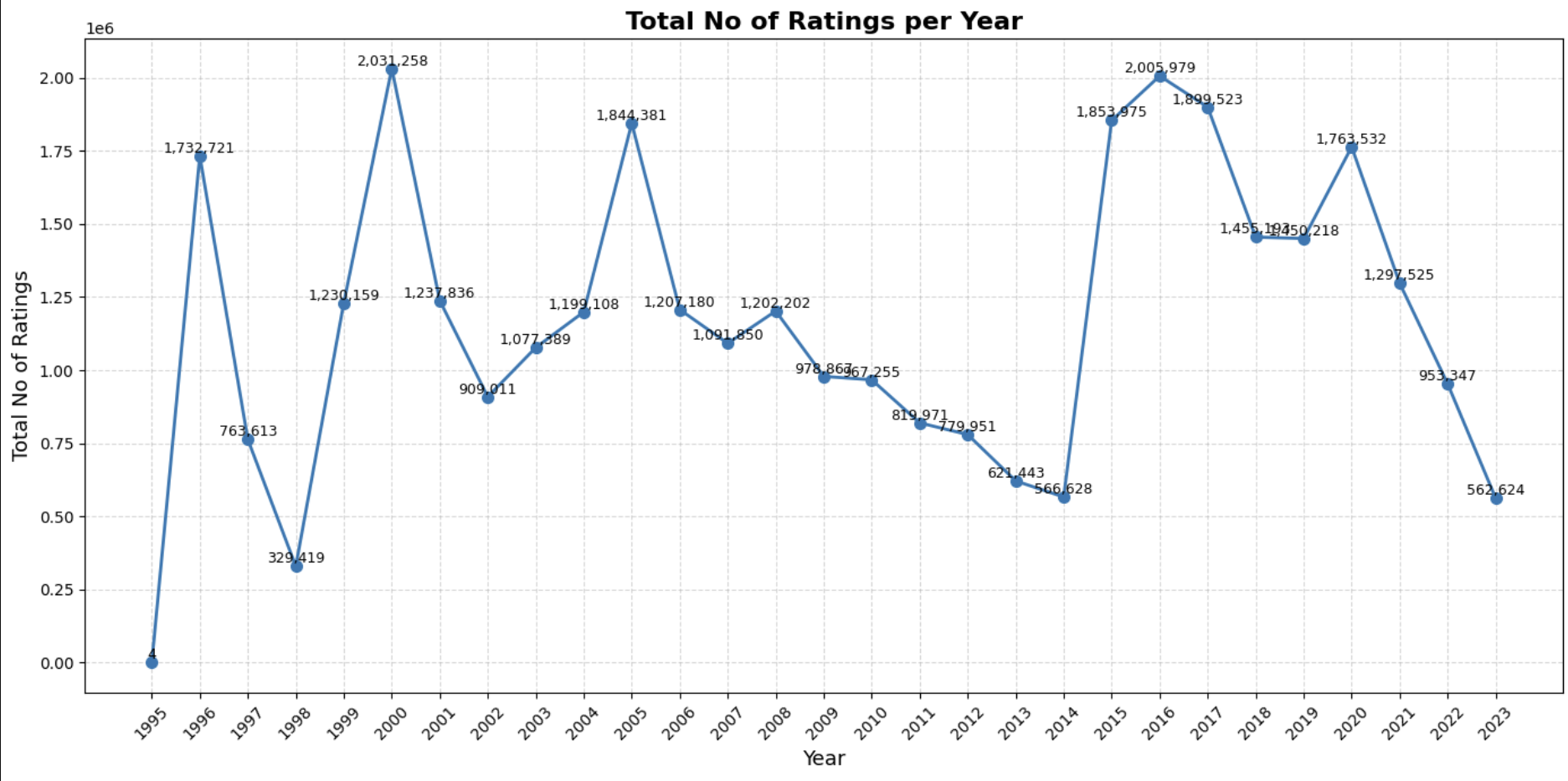
I started by extracting the year component from the **timestamp** column in the dataset, which was originally in Unix epoch format. I used the PySpark **year()** function to extract the year and then used **withColumn()** to add it as a new column to the DataFrame.

Once the **year** column was added, I grouped the data by year and counted how many ratings were given in each year. I used **groupBy()** to group the data by the **year** column and **count()** to count the number of ratings for each year.



After grouping and aggregating the data, I used **show()** to display the results and visualize the distribution of ratings across the years. I then used **matplotlib** to plot a **line chart**, showing the total number of ratings per year.

Then I exported the Data from S3 bucket to local machine for visualization.



The line chart clearly shows the total number of ratings per year from **1995** to **2023**. From the chart, I observed the following:

* There were significant peaks in the years around **2000** and **2016**, suggesting higher user activity during those years.
* The number of ratings significantly decreased starting from around **2014**, indicating reduced engagement in the latter part of the period.
* There were fluctuations between years, but overall, a clear downward trend was observed after **2008**.

### **Challenges:**

* A challenge I faced was ensuring accurate year extraction from timestamps and verifying the completeness of the data for each year. Since the dataset spanned many years, any missing or inconsistent data could skew the results.So I carefully validated the timestamp column to ensure it was correctly processed and filtered, and I ensured that there were no missing values in the data for any given year.

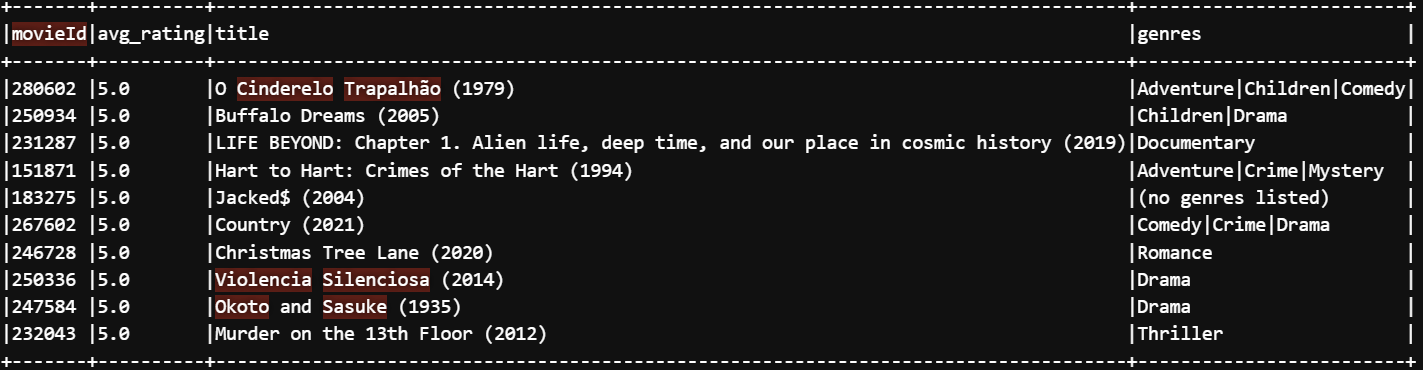
**Insights:**

* I gained valuable insights into how user engagement varies over time. This type of temporal analysis helps identify trends, such as periods of high user activity, and provides insights into the factors that might drive user engagement.
* I learned how to effectively visualize time series data using line charts. This is crucial for identifying trends and patterns over extended periods, especially when dealing with large datasets.
* This task reinforced the importance of grouping and aggregating data in Spark. The ability to group data by time periods (e.g. year) and perform aggregations is key when working with large-scale datasets in big data environments.

## **7. Question 7: Top 10 Movies Analysis**

The goal was to identify the Top 10 Movies based on their average ratings from the ratings dataset. To calculate the average rating for each movie, I used the **groupBy()** function to group the data by movieId, and then I used the **avg()** function to calculate the average rating for each movie.

**alias("avg\_rating")** is used to rename the new column containing the average rating to **avg\_rating**, which makes it easier to refer to and understand in further steps.

Next, I joined the calculated average ratings with the **movies\_df** DataFrame to get the movie titles. I used the **join()** function to join the two DataFrames on the **movieId** column. I used **orderBy(col("avg\_rating").desc())** to sort the movies by their average rating in descending order. This allowed me to bring the movies with the highest ratings to the top.

### **Findings:**

* Based on the analysis, I identified the top 10 movies with the highest average ratings. These movies were the ones most consistently rated highly by users, suggesting they are favorites within the MovieLens community.
* The movies that appeared in the top 10 were mostly critically acclaimed and widely liked by the user base. This could be valuable information for making movie recommendations or understanding popular trends in the dataset.
* This analysis provides valuable insight into which movies have the highest level of user satisfaction. Movies that appear in the top 10 are likely to be favorites, and their high average ratings can suggest the level of quality that resonates with users.

### **Challenges:**

* One challenge I encountered during this analysis was ensuring that the results were based on a sufficient number of ratings. Movies with only a few ratings may not represent broader user preferences accurately, so I had to make sure that movies with a small number of ratings were not included in the top 10. I could filter the dataset to include only movies with a certain number of ratings (e.g. more than 50 ratings) before performing the analysis to avoid skewing the results.

### **Insights:**

* **Aggregation and Sorting:** I learned how to use the **groupBy()** and **avg()** functions in PySpark to calculate the average of numeric values across groups. This is crucial when analyzing user ratings and other similar data.
* **Joining DataFrames:** I gained experience in using the **join()** function to combine two DataFrames based on a common key. This is a fundamental operation when working with multiple datasets that contain related information.
* **Data Filtering and Sorting:** Sorting and filtering data efficiently is essential when working with large datasets. I learned how to apply the **orderBy()** and **limit()** functions to quickly identify the top results.
* **Movie Quality Insights:** I learned how to use average ratings to identify high-quality movies, which is an important step in recommendation systems and understanding user preferences in a movie platform.

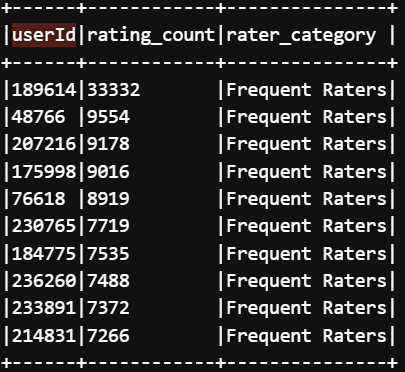
**8. Question 8: User Analysis**

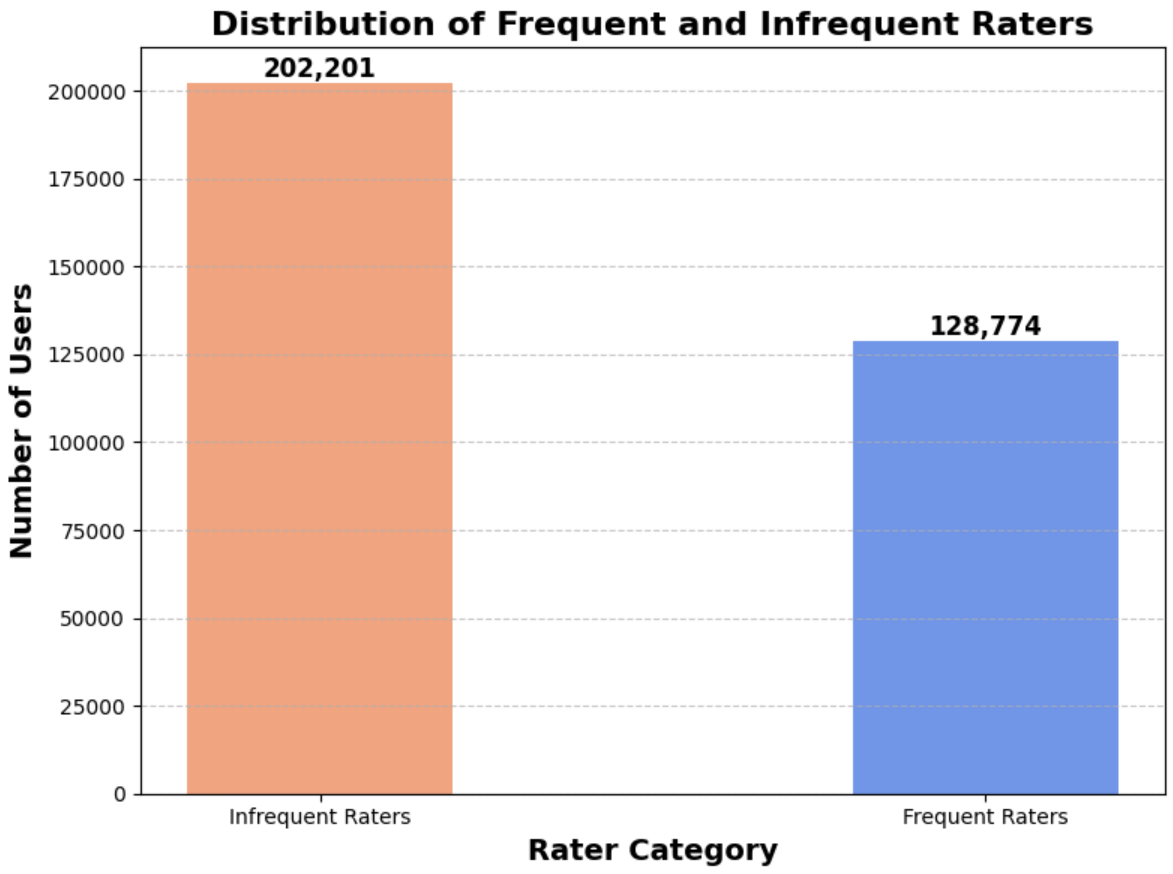
The goal was to categorize users based on the number of ratings they provided. Specifically, I wanted to identify **Frequent Raters** (users who rated more than 50 movies) and **Infrequent Raters** (users who rated fewer than or equal to 50 movies).

I grouped the dataset by **userId** and counted the total number of ratings each user had given. This was achieved using the **groupBy()** function combined with **count().** The **alias()** function renames the newly created column (the count of ratings) to **rating\_count**, which makes it easier to understand and refer to in subsequent steps.

Once I had the **rating\_count** for each user, I categorized users into **Frequent Raters** and **Infrequent Raters**. I used the **when()** function to apply a conditional check: users with more than 50 ratings were categorized as **Frequent Raters**, and users with 50 or fewer ratings were categorized as **Infrequent Raters**.

After categorizing the users, I grouped them by **rater\_category** (Frequent or Infrequent) and counted the number of users in each category. I've used **groupBy("rater\_category")** to group the users based on whether they were **Frequent** or **Infrequent** raters. Finally, I used the **show()** function to display the number of **Frequent** and **Infrequent** raters. This allowed me to see the distribution of users in each category. Then I exported the Data from S3 bucket to local machine for visualization.





The **bar chart** clearly shows that there are more **Infrequent Raters** than **Frequent Raters**, which suggests that the majority of users engage with the platform occasionally rather than consistently.

### **Findings:**

* The analysis revealed that there are significantly more **Infrequent Raters** than **Frequent Raters**, with **202,201** users falling into the **Infrequent Rater** category and only **128,774** users being **Frequent Raters**.
* This distribution tells that while a large number of users participate in the platform, most contribute relatively few ratings. A smaller group of users engage heavily with the platform, providing more than 50 ratings. This might be useful for understanding user behavior and making recommendation strategies.

### **Challenges:**

* A potential challenge with categorizing users was ensuring the boundary between **Frequent** and **Infrequent** raters was meaningful. The threshold of 50 ratings could be arbitrary, and I had to ensure this threshold provided a useful distinction. So I chose the threshold based on an initial exploration of the data, where I observed a natural split in the distribution of ratings. However, this threshold can be adjusted depending on the specific needs of the analysis (e.g., setting a higher threshold for more active users).

### **Insights:**

* I learned how to categorize users based on their activity level, which can be useful for targeted analysis, like identifying highly engaged users for a recommendation system or some promotional purposes.
* I gained valuable experience in group based analysis, which is a fundamental technique when working with user data. Grouping users based on specific attributes and then performing aggregations can provide insights into patterns of behavior.
* This task reinforced the importance of understanding user engagement levels. Recognizing how many users fall into the **Frequent** versus **Infrequent** categories helps shape strategies for improving user interaction with the platform.

————————————**END OF TASK 2**————————————

Task 3

**Task Objective:** Analysis of Chicago Taxi Trips Data

**Initial Stage:**

I have loaded the provided Chicago Taxi Trips csv using Spark’s CSV reader with the following configuration which is provided in the Task 3 Guidelines PDF:

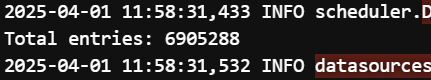
taxi\_df = spark.read.csv("chicago\_taxi\_trips.csv", header=True, inferSchema=True)

I used the spark.read.csv() method to load the dataset, where:

* **header=True** specifies that the first row contains column names.
* **inferSchema=True** ensures Spark automatically detects the correct data types for each column.

**1. Question 1: Printing the total number of entries in the DataFrame**

I have used **print(taxi\_df.count())** to print the total number of entries in the **chicago\_taxi\_trips.csv** Dataset.



So there are Total of **69,05,288** entries in the dataset.

**2. Question 2: Define Schemas and Construct DataFrames**

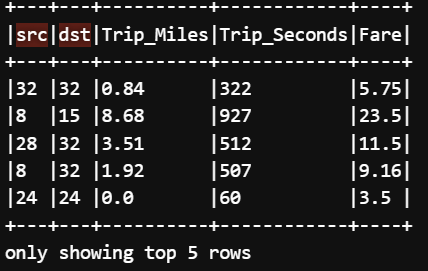
In this step, I defined two schemas: one for the vertices (community areas) and one for the edges (taxi trips). I used the **StructType** class to define the schema for the **vertexSchema** and **edgeSchema** DataFrames.

### **Vertices (Community Areas):** The **vertices** in my graph represent the **community areas** where the taxi trips start (pickup community area). Each community area is a node (or vertex) in the graph. To create the **vertices\_df** DataFrame, I selected specific columns from the original **taxi\_df** dataset that correspond to the community areas.

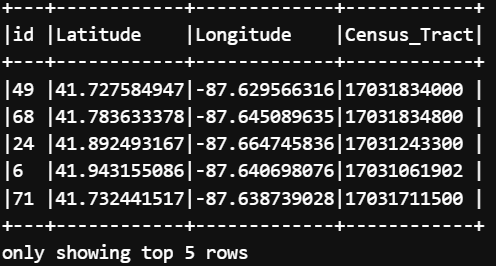
### **Edges (Taxi Trips):** The **edges** in the graph represent the **taxi trips** between the pickup and dropoff community areas. Each edge connects two vertices (community areas) and is associated with the trip's attributes, such as distance, duration, and fare. I created the **edges\_df** DataFrame by selecting the relevant columns from the original **taxi\_df**.

To inspect the content of both the **edges** and **vertices** DataFrames, I displayed the first 5 rows of each using the **show()** function. This allowed me to verify that the data was correctly transformed and that the expected columns were present in both DataFrames.

### **Edges (Taxi Trips):**



### **Vertices (Community Areas):**



**Challenges:**

* A challenge in creating the **edges\_df** DataFrame was ensuring that both the pickup and dropoff community areas were properly matched with the taxi trip attributes. Ensuring there were no missing values in **src** and **dst** was crucial for the integrity of the graph. So I used the **filter()** function to exclude rows where either the **src** or **dst** community areas were null, ensuring that only valid trips were included in the analysis.
* Another challenge was verifying that the **vertices\_df** contained only unique community areas. Community areas could appear multiple times in the dataset (once for each taxi trip), so it was important to use **distinct()** to ensure each community area was only represented once in the graph. So I used the **distinct()** function on the **vertices\_df** to ensure that each community area was listed only once, eliminating any duplicate rows.

**Insights:**

* I learned the importance of creating structured DataFrames for graph analysis. By defining clear schemas for vertices and edges, I set up the data in a format that is ready for graph algorithms like Shortest Path or PageRank.
* I also learned how to filter and clean the data to ensure that I had valid, non-null values for both vertices and edges, which is crucial for ensuring the quality of graph based analysis.

**3. Question 3: Creating GraphFrame and Viewing Samples**

The objective was to construct a comprehensive and enriched graph from the community areas (vertices) and taxi trips (edges). Initially, I constructed the basic graph structure using the GraphFrame library. The graph structure was straightforward:

**Vertices**: Community areas.  
**Edges**: Taxi trips between these community areas.

To enrich the edge data with vertex details, I first needed to rename the columns in the **vertices\_df** DataFrame to make them more distinct when joining the edges with the corresponding source and destination community areas.

I created two new DataFrames for the **source vertices** (**src\_vertices**) and **destination vertices** (**dst\_vertices**), where I renamed the columns accordingly.

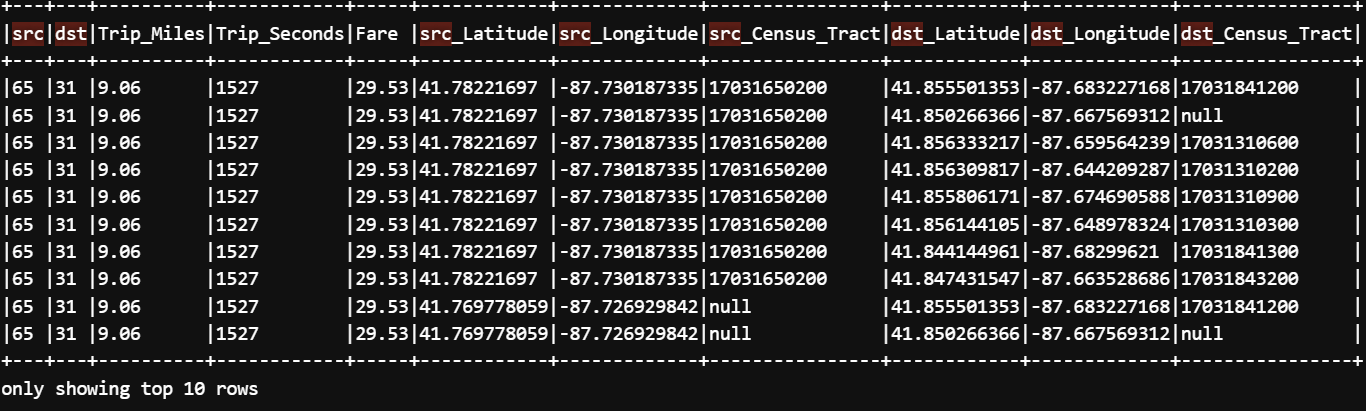
The **withColumnRenamed()** function is used to rename columns in a DataFrame. I renamed the columns to distinguish between the source and destination community areas:

* For the source community area (**src**), I renamed columns like **Latitude** to **src\_Latitude** and **Longitude** to **src\_Longitude**.
* For the destination community area (**dst**), I renamed columns like **Latitude** to **dst\_Latitude** and **Longitude** to **dst\_Longitude**.

Next, I joined the edges with the enriched vertex information (latitude, longitude, and census tract) for both the source and destination community areas. This ensured that each edge (taxi trip) contained not only the community area IDs (**src** and **dst**) but also their corresponding spatial and demographic details. I have done this using join(src\_vertices, on="src", how="left") and join(dst\_vertices, on="dst", how="left").

After the join, I selected only the relevant columns from the enriched graph using **select()** function, which included the community area IDs, trip attributes (miles, seconds, fare), and spatial/demographic details (latitude, longitude, census tract) for both the source and destination community areas.

Finally, I displayed the first 10 rows of the **graph\_data** using **show()** function DataFrame to verify the enrichment process and check that the necessary columns were correctly included in the final dataset.



### **Findings:**

* By enriching the edge data with the spatial and demographic details of the source and destination community areas, I was able to create a more comprehensive graph that not only represents the connections between community areas but also includes relevant attributes that can be used for further analysis.
* The enriched graph can now be used to perform more sophisticated graph algorithms, like **Shortest Path**, **PageRank**, or **Community Detection**, which take both spatial and demographic information into account.

### **Challenges:**

* One challenge I encountered was ensuring that the column names were correctly renamed and consistent when joining the edges with the vertices. This was crucial to avoid misaligning data and ensuring that the correct community area details were attached to the correct taxi trips. So I've used **withColumnRenamed()** to carefully rename the columns for both source and destination community areas, making sure that there was no ambiguity when performing the join.
* Another challenge was handling the large size of the dataset efficiently while performing the joins. Spark’s distributed processing handled the data well, but I needed to ensure that the joins were done efficiently to avoid performance bottlenecks.By using a left join, I made sure that no taxi trips were excluded, and by filtering null values beforehand, I minimized the risk of performance issues.

### **Insights:**

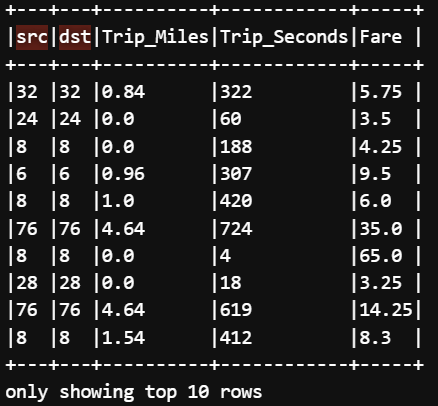
* I learned how to use **GraphFrame** to build a graph from taxi trip data and enrich it with spatial and demographic information.
* I also learned the importance of carefully renaming columns and performing joins to ensure data integrity when working with graph-based datasets.
* This task enhanced my understanding of how to build more sophisticated graph structures by adding relevant details from the vertices (community areas) to the edges (taxi trips), making the graph much more informative and useful for further analysis.

**4. Question 4: Connected Community Areas Analysis**

The objective was to identify taxi trips that started and ended in the **same community area**. This analysis helps understand short-distance trips within neighborhoods, giving insights into local travel patterns.

To find taxi trips that occurred within the same community area, I filtered the edges DataFrame to include only rows where the source (**src**) and destination (**dst**) community areas were identical using the **filter()** function. This indicated that the pickup and dropoff were within the same neighborhood. After filtering the dataset for trips within the same community area, I counted how many such trips occurred by using the **count()** method.

To confirm that the filtering had worked correctly, I displayed 10 sample rows from the filtered DataFrame (**connected\_edges\_df**), clearly inspecting them for accuracy.



Finally, I printed the total number of trips clearly identified as intra-community trips



**Challenge**:

* Ensuring that filtering logic was correct, particularly with equality conditions.  
  I Carefully examined a subset of results and counted records to verify accuracy.
* Efficiently handling potentially large volumes of data during counting and filtering operations. So I used Spark’s powerful distributed computation capabilities to manage large-scale data efficiently.

## **Insights:**

* I learned that short trips within community areas could provide valuable insights into urban mobility patterns, local economic activity, and neighborhood connectivity.
* The task reinforced the effectiveness and efficiency of Spark DataFrame operations (**filter()** and **count()**) in quickly identifying and quantifying patterns in large datasets.
* I realized the importance of clearly and promptly verifying filtered data through sample inspection to ensure accurate analysis outcomes.

**5. Question 5: Running the Shortest Paths Algorithm**

The goal was to find the **shortest paths** from all community areas to a specific landmark community area using GraphFrames. Specifically, I selected community area **49** as the landmark area. The shortest path analysis helps understand connectivity, travel accessibility, and can inform route optimization within the taxi trip network.

I defined the landmark community area for the shortest path analysis. In this case, I chose community area **49** as my reference landmark. I clearly defined a variable **landmark\_vertex** to store the ID of the landmark community area (49). This made it explicit, flexible, and easier to adjust if needed later.

Next, I used the **GraphFrame** library's built-in shortest path algorithm to compute distances from all community areas to the landmark community area (area 49)

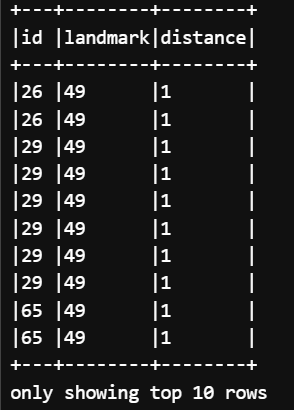
Using **shortestPaths(landmarks=[landmark\_vertex])** Icomputed clearly the shortest paths from every vertex in the graph to the given landmark vertex. It provided an easy-to-use API that automatically computed distances efficiently, leveraging the graph structure. The result returned from the shortest path algorithm contained a **distances** column in dictionary format, which stored distances from each vertex to the landmark. To clearly present this information, I explicitly extracted and formatted relevant columns.

**select()**: Clearly selected only necessary columns for the final analysis, improving readability and usability.

**col("id"):** Clearly selected community area identifiers.  
**lit(landmark\_vertex).alias("landmark"):** Used explicitly to add a column that clearly identifies the landmark vertex, improving clarity in the results.

**col("distances").getItem(landmark\_vertex).alias("distance"):** Extracted explicitly the shortest path distance to the landmark vertex from the dictionary in **distances**. This step was crucial, as the shortest path results were stored as key-value pairs within a dictionary.

Finally, I displayed a sample of 10 rows from the formatted shortest path data to verify explicitly and clearly that the shortest paths computation and formatting were correct and comprehensible.



**Challenge**s:

* Had to refer to the GraphFrames documentation to clarify clearly how to extract data using the **getItem()** method.
* Extracting and formatting clearly nested dictionary data in Spark. I learned explicitly to use Spark's **getItem()** method effectively, simplifying the extraction of shortest path distances clearly.

## **Insights:**

* I explicitly learned how powerful and straightforward shortest path analysis is using GraphFrame, providing clear insight into network connectivity.
* Learned how to handle explicitly nested dictionary data within Spark DataFrames clearly and effectively using the **getItem()** method.
* Understood explicitly the value of clearly formatting analysis results, making insights more accessible and actionable.

**6. Question 6: PageRank Analysis with Weighted Edges**

The goal was to analyze community areas by their relative importance in terms of connectivity within the Chicago taxi trips network. To achieve this, I employed the **PageRank** algorithm, analyzing the network in two distinct scenarios:

* **Unweighted** (basic connections)
* **Weighted** (with trip distances as weights)

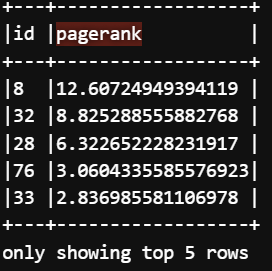
The first step in this task was to ensure the vertices (community areas) in the graph were unique. This was necessary because the same community area may appear multiple times in the dataset (due to multiple taxi trips starting or ending in the same area). I used the **dropDuplicates()** function to remove any duplicate community areas.

Next, I created an unweighted GraphFrame using the deduplicated vertices DataFrame (**vertices\_dedup\_df**) and the existing edges DataFrame (**edges\_df**). Then I ran the PageRank algorithm explicitly using default edge weights to measure the connectivity-based importance of community areas:

**resetProbability** **(0.15)** represents the probability of randomly resetting at each step (standard practice).

**tol (0.01)** indicates tolerance level for convergence—determining clearly when the algorithm stops iterating.

Finally, I clearly displayed the top five community areas, sorted explicitly by their PageRank values in descending order.



**Question 6. Part B: PageRank on Weighted Graph**

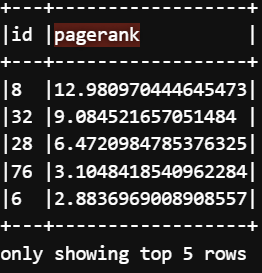
For the weighted PageRank analysis, I explicitly constructed a vertex DataFrame including unique community areas from both pickup and dropoff points clearly using **select()** and **distinct()** functions.

I then explicitly defined edges with trip distances (in miles) as edge weights clearly, providing quantitative meaning to edges beyond simple connectivity. To ensure consistency, I clearly normalized the edge weights by dividing each trip's distance by the total outgoing distance from each source community area explicitly

Next, I explicitly constructed a weighted GraphFrame using the normalized edge weights clearly defined.

I executed the PageRank algorithm explicitly again, this time clearly using normalized edge weights for a weighted analysis

Finally, I explicitly displayed the top five community areas sorted clearly by their weighted PageRank values in descending order



**Findings:**

* **ID 8** is consistently the highest-ranked community area in both the **unweighted** and **weighted** PageRank analyses. This suggests that **ID 8** is one of the most important nodes in the taxi network, being both highly connected (in terms of the number of trips) and crucial for long-distance trips (in the weighted analysis).
* **ID 32**, **ID 28**, and **ID 76** appear in both analyses, but their relative importance shifts when considering **trip distance**. While these areas maintain high PageRank values, their ranks are influenced by the inclusion of long-distance trips in the weighted analysis.
* The Weighted PageRank (with distances) reveals that certain areas (like area 6) become more important due to significant long-distance taxi trips. Long-distance travel impacts area importance and connectivity significantly.

## **Challenges:**

* Clearly handling duplicate vertices effectively. So I used Spark’s dropDuplicates() function.
* Clearly normalizing edge weights appropriately.So I calculated total outgoing weights per community area and normalized edges clearly.
* Determining suitable parameters for stable PageRank convergence. So I utilized standard PageRank parameters (resetProbability=0.15, tol=0.01) clearly ensuring reliable results.

## **Insights:**

* Clearly understood the use and significance of the PageRank algorithm in network analysis.
* I have learned to handle weighted graphs, including normalization clearly, to provide more insightful results.
* Clearly recognized how vertex and edge uniqueness and correctness are crucial for accurate analysis.
* I have gained experience in both basic and weighted graph analyses, significantly enriching my analytical toolkit.

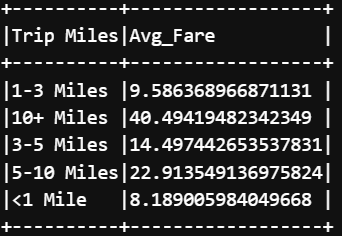
**7. Question 7: PageRank Analysis with Weighted Edges**

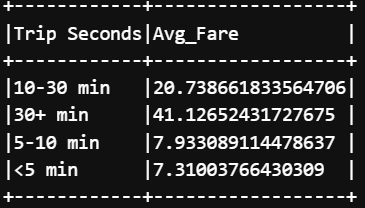
The objective was to categorize taxi trip durations into bins (e.g., short, medium, long trips) to simplify the analysis of trip lengths. This step helped break down the continuous range of trip durations into discrete categories, making it easier to interpret and compare the data.

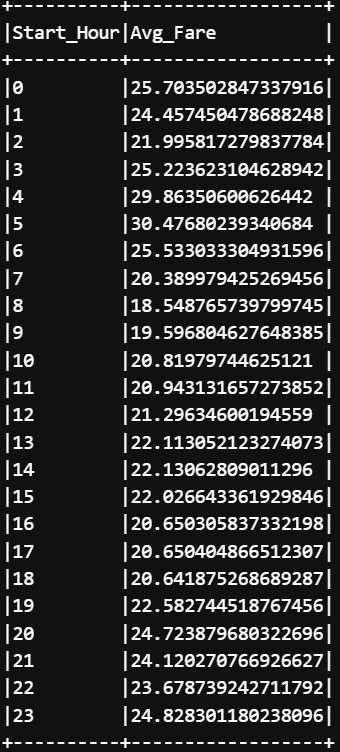
**Note:** I categorized the taxi trip distance and durations into discrete **bins** (**ranges**) based on the number of seconds each trip took. Rather than dealing with continuous values, categorizing the durations into ranges made the analysis more interpretable and helped identify patterns in different categories of trips.

* The **when()** function is used to create conditions for categorizing the data into different bins. Here, I categorized trip durations into:  
  **Short**: Less than 600 seconds (10 minutes).  
  **Medium**: Between 600 and 1800 seconds (10 to 30 minutes).  
  **Long**: More than 1800 seconds (30 minutes and above).
* I have also used **col("Trip Seconds").** This refers to the column containing the trip duration in seconds. I used this column to create conditional ranges for categorization.

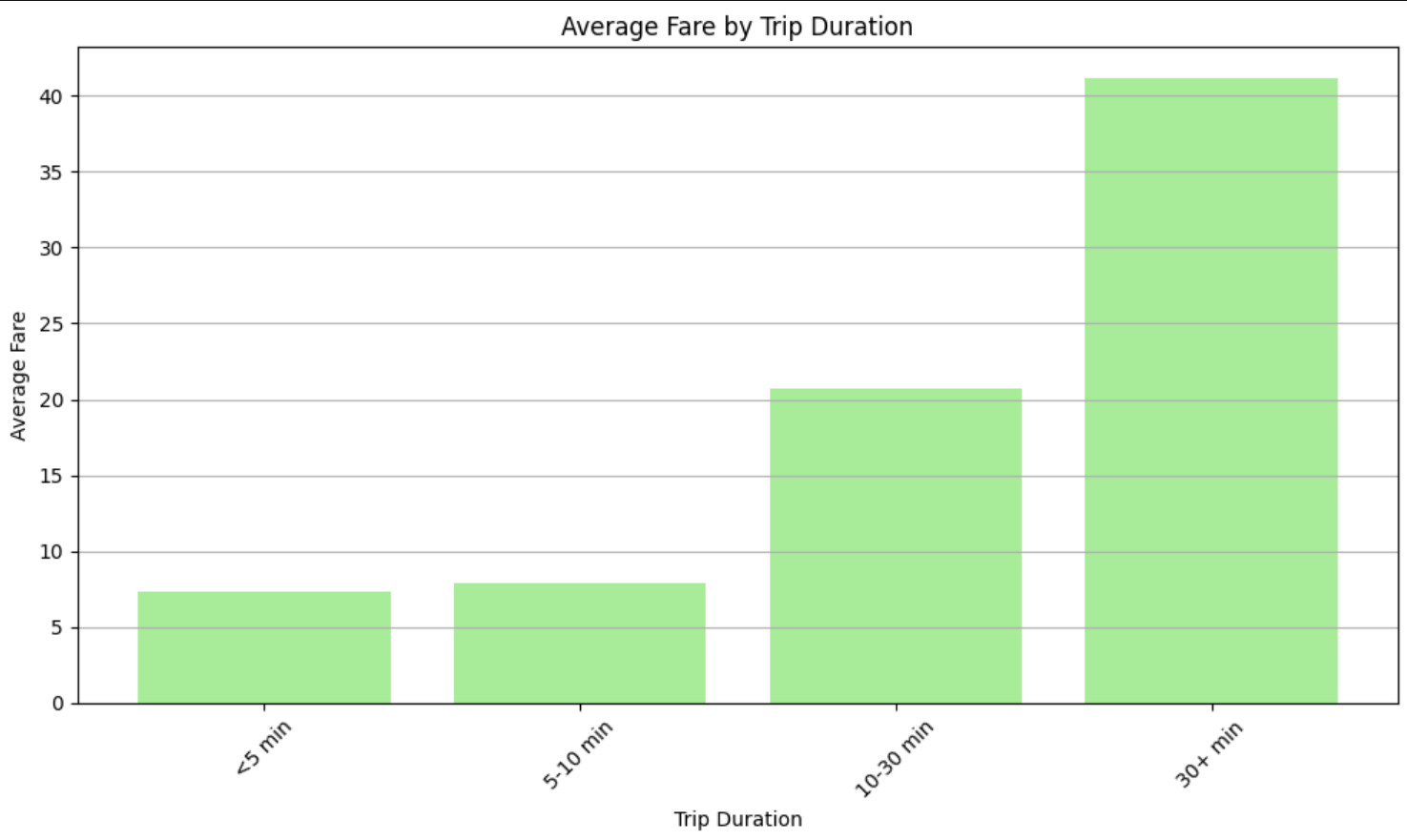
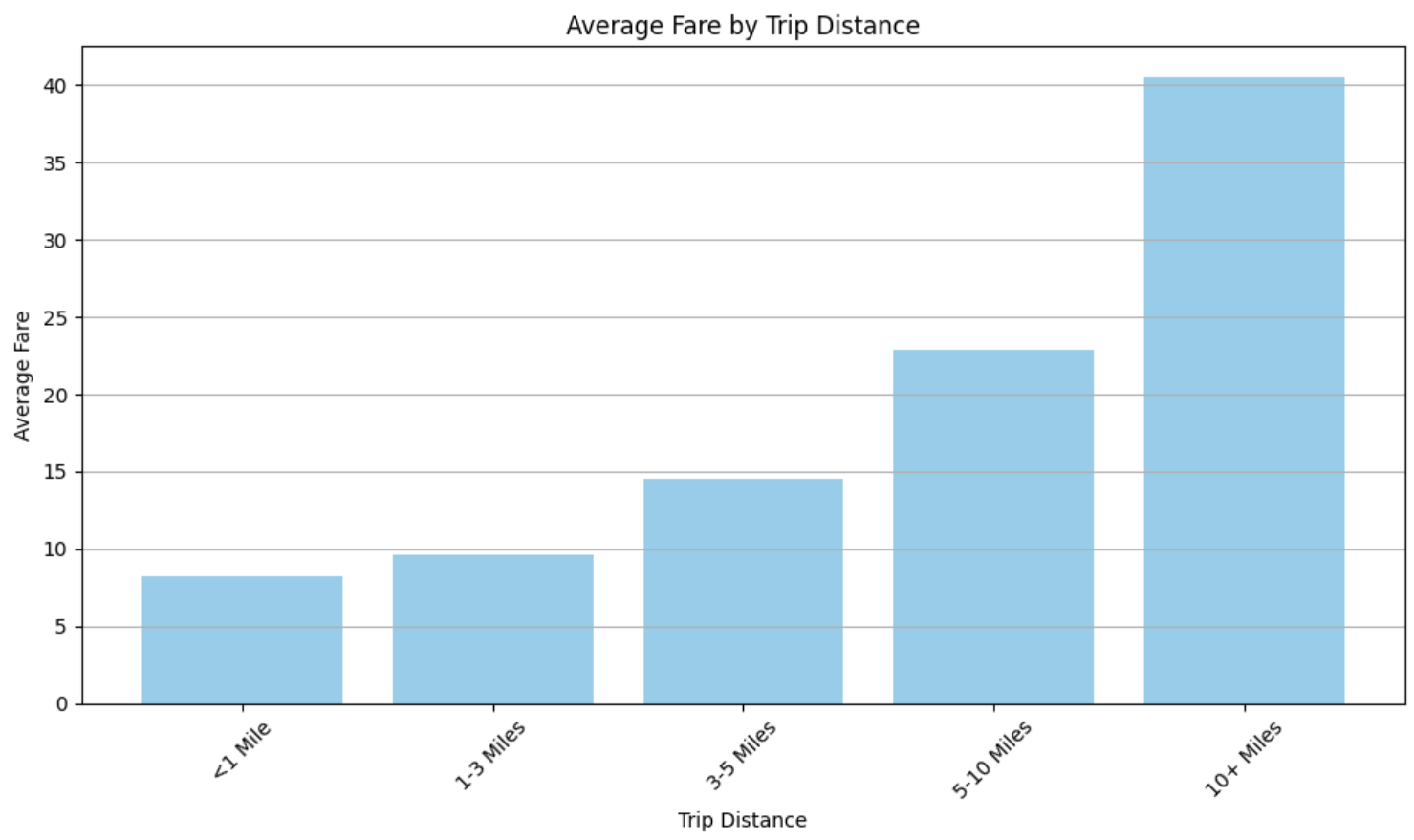
After categorizing the trip distance, duration, and time, I displayed the results clearly to check how the trips were distributed across the defined categories. Using **trip\_duration\_category** column, enabling the count of trips in each category using **count()** the function.

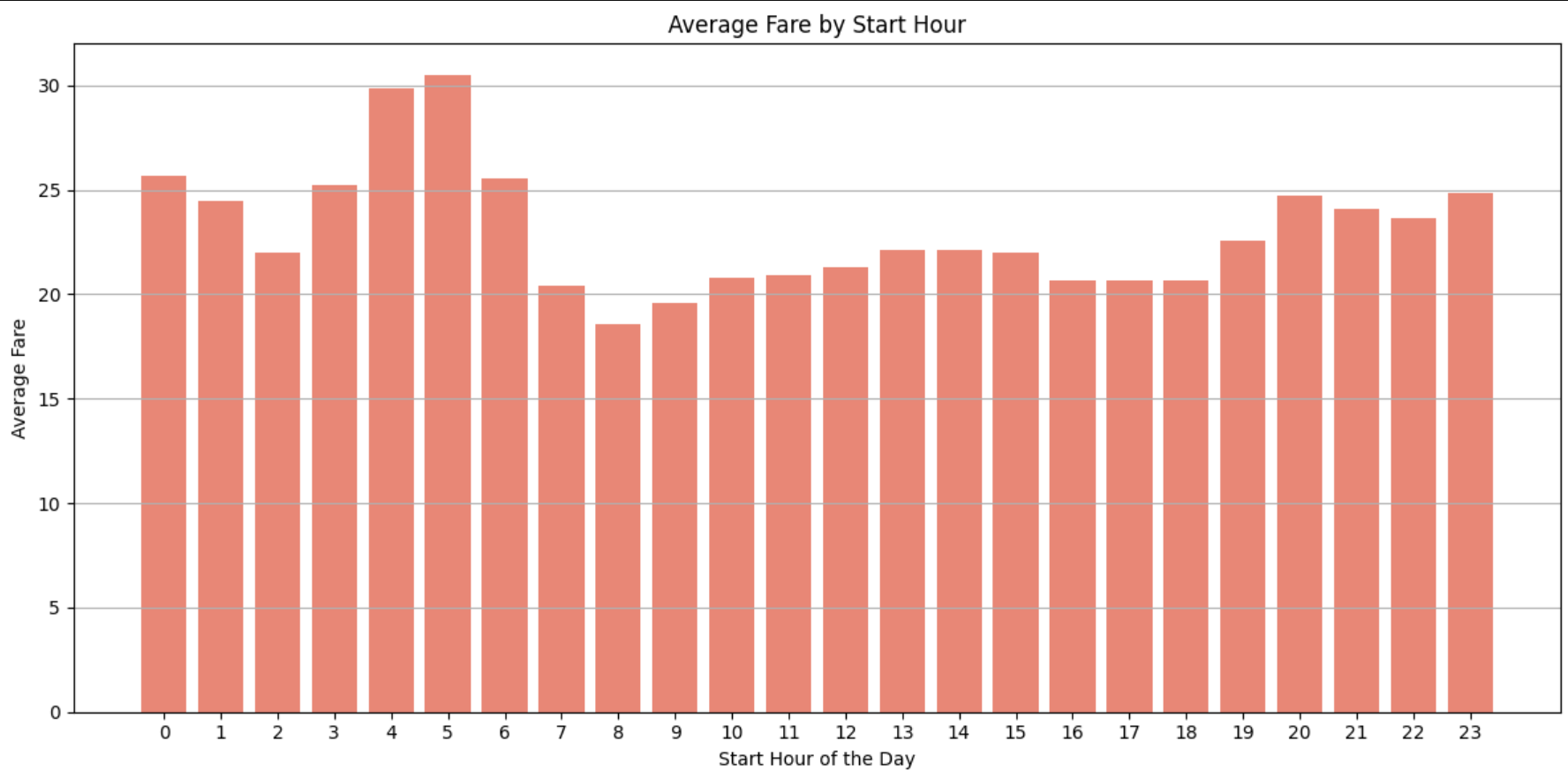


****

****

Then I exported these 3 results in a CSV file from the S3 bucket for visualization.





Based on the visualizations provided for **Average Fare by Trip Distance**, **Average Fare by Trip Duration**, and **Average Fare by Start Hour**:

1. **Average Fare by Trip Distance**:  
   **Trips between 5-10 miles** and **10+ miles** are associated with significantly higher average fares (around $25–$40). This is expected as longer trips typically require more time and fuel, leading to higher fares.  
   Shorter trips (such as **<1 mile** and **1-3 miles**) are associated with much lower average fares, with **<1 mile** having the lowest average fare, as expected due to shorter distances covered.
2. **Average Fare by Trip Duration**:  
   Similar to the trip distance chart, **30+ minute** trips have the highest average fare, around $40.  
   **10-30 minute** trips have a moderate average fare of about $20, indicating that longer trips still provide a higher fare but not as much as the longest ones.  
   **Shorter durations** (<5 minutes) show the lowest fares, which is typical for very short trips.
3. **Average Fare by Start Hour**:  
   The plot shows a fairly **consistent** fare range across different hours, with some slight variations at different times. This indicates that, generally, the fare is not drastically affected by the time of day. However, there are still higher fare periods around **3-6 AM**.

### 

### **Challenges:**

### A major challenge was ensuring the correct **chronological sorting** of **Trip Duration** and **Trip Distance** bins. Since they were initially categorical values (e.g., "1-3 Miles" or "10+ Miles"), I had to manually map them to numerical values for proper sorting before plotting.

* After sorting, it was necessary to plot them in ascending order to ensure the x-axis represented these variables logically and chronologically.
* Although the time of day data was expected to be sorted naturally from **0 (midnight) to 23 (11 PM)**, ensuring that the x-axis was ordered properly was crucial for accurate interpretation of the **Average Fare by Start Hour** plot.
* Interpreting how trip distance and trip duration correlate with the fare was not straightforward. For example, the higher fares for trips in the **10+ Miles** and **30+ minutes** categories made sense, but the relatively lower fares in shorter trips required additional investigation of the data trends, possibly indicating less demand or different pricing models for shorter durations.

**Insights:**

* The analysis confirmed that longer trips generally result in higher fares, which aligns with conventional logic that longer trips incur more charges due to time and distance.
* For shorter trips **(<1 mile and 5-10 minutes)**, fares are notably lower, which could be a reflection of lower service charges for small distances.
* The time of day does influence fare pricing to some extent, but the variation seems minimal. The fare remains relatively stable throughout the day, although there might be slight increases during late-night or early-morning hours, likely reflecting fewer taxis available or longer trips due to reduced traffic.
* Visualizing the data in chronological order for both trip duration and distance significantly improved the clarity of the results. It allowed for better comparison between different types of trips and a clearer understanding of how various factors like trip distance, duration, and time of day influence the average fare.

————————————**END OF TASK 3**————————————

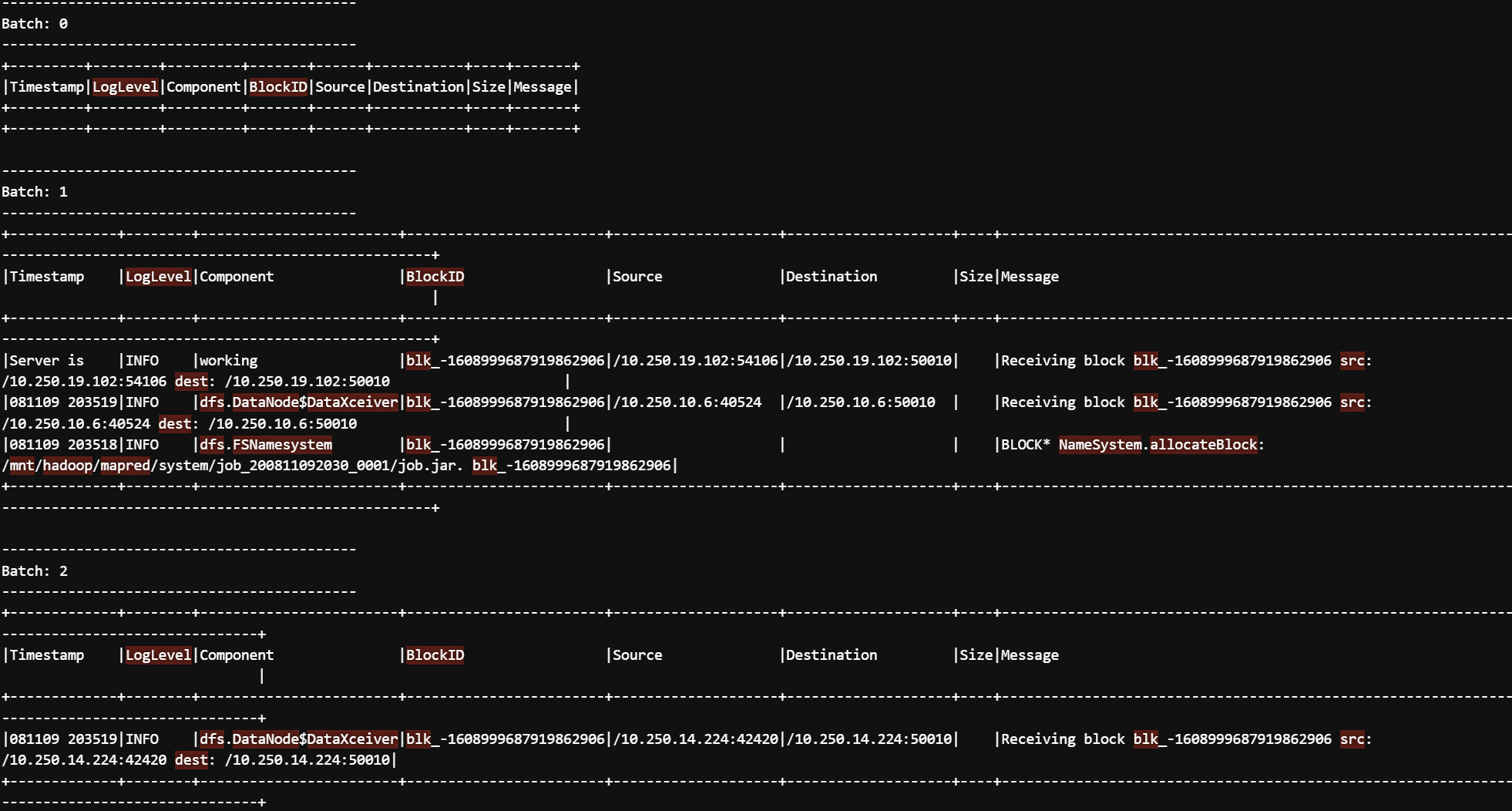
Task 4

**Task Objective:** Analysis of HDFS Logs dataset

In this task, I analyzed the **HDFS log data** using **Structured Streaming Processing** with Apache Spark. The dataset consisted of logs from an HDFS system with key fields such as timestamp, log level, component, block ID, source IP, destination IP, block size, and message. The goal was to process and analyze the logs using Spark structured streaming features to identify patterns, anomalies, and trends.

**1. Question 1: Loading the Dataset and output for batch 0, 1, and 2**

First, I configured the **host** and **port** to access the **HDFS logs** from the streaming server.Explicitly specified the **host** and **port** values from environment variables to load the streaming dataset. I used **Spark Streaming** to load data from the socket and started the data stream.



The dataset was displayed in **append mode** without truncating the fields, and I verified the results for **batch 0, 1, and 2** by running the query in the console.

**2. Question 2: Defining a Watermark**

To handle late data arrivals, I defined a watermark on the timestamp column with a delay of 5 seconds using **withWatermark("Timestamp", "5 seconds")**.

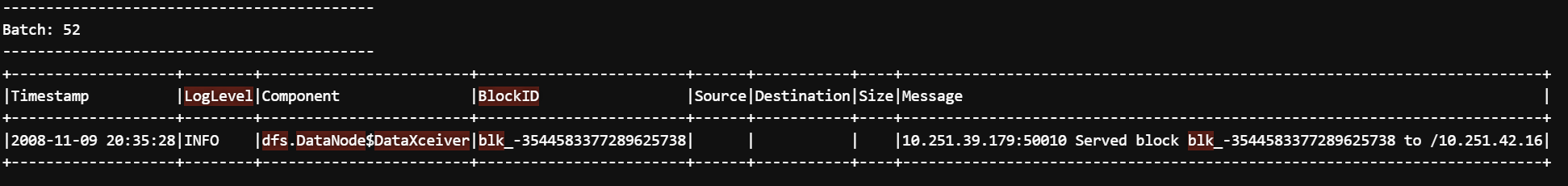
**Purpose of Watermark**: The watermark is used to **manage late data** and allows Spark to track **out-of-order data** while preventing unnecessary data processing for events that arrive late. By setting a 5-second delay, I ensured that the system could handle small delays in data arrival without impacting the accuracy of the analysis.

**3. Question 3: Analyzing Data Patterns**

I filtered the logs to focus on specific log entries for **DataNode activity**.  
I used the **writeStream** method to define a streaming query that writes the data to the console. This allows me to monitor the incoming log data from the stream and view it in real-time. **format("console")** Specified that the output should be written to the console. Using the **outputMode("append")** the data is continuously appended to the output as it arrives. This is ideal for viewing the real-time log updates. Then **option("truncate", False)** Ensures that the data is fully printed to the console without truncation, which is essential for visualizing the entire log record.

To allow the query to process for a specific period, I added the **awaitTermination(60)** function. This keeps the query running for **60 seconds**

### 



### **Insights:**

* By using **Structured Streaming**, I was able to display live log data directly in the console, which is useful for debugging and tracking data movements.
* The **outputMode("append")** ensures that only the new data is shown without overwriting previous records, which is essential for continuous log tracking.
* I learned that **real-time data processing** can be effectively monitored using **console outputs** in Spark, allowing me to understand how logs and data flow through the system during the stream.

### **Challenges:**

* **Handling Large Volumes of Streaming Data:** One of the challenges I faced was dealing with the continuous flow of large amounts of log data. Since Spark Structured Streaming processes data in micro-batches, displaying real-time log data on the console can quickly overwhelm the output, especially when working with large datasets or frequent updates. I had to manage the batch sizes and keep track of how often the data is being updated to avoid excessive logging. To handle this, I limited the time for each query execution using the **awaitTermination(60)** method to only process data for **60** seconds at a time. This ensured that I could monitor a manageable portion of data without being overwhelmed.

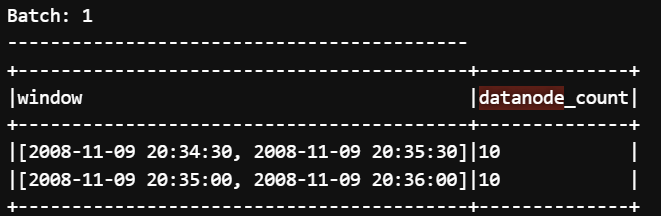
**4. Question 4. Analyzing DataNode Activity**

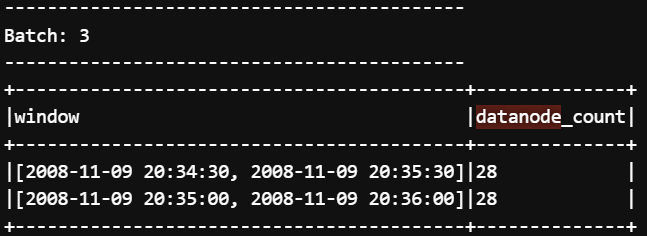
First, I created a DataFrame (**datanode\_df**) by filtering the **watermarked\_df** to only include rows where the Component column contains the keyword **"DataNode"**. This filtering step ensures that the logs pertaining to the **DataNode** component are extracted for analysis.

Then to analyze the data over time, I used a **time windowing** approach. I applied a **60-second window** with a **30-second sliding interval**. The aggregation function used was **count("\*")**, which counts the number of rows (DataNode logs) within each window.

After performing the aggregation, I set up the output mode as **update**, which allows Spark to output only the updated results of the aggregation for each batch. The results are written to the console, where I can observe the **DataNode count** for each 60-second window (with a 30-second sliding interval).

The **awaitTermination** method was used to wait for the stream to process for **120 seconds**. During this time, Spark continuously processed incoming data and aggregated the DataNode logs within the specified time window.





### **Findings:**

* **Batch 1** shows the lowest count of **datanode\_count** with only **10 occurrences** within the timestamp window of **20:34:30 to 20:35:30**.
* **Batch 2** shows a noticeable increase in **datanode\_count** to **22 occurrences** within the same timestamp window, indicating more events captured by the system.
* **Batch 3** further increases the count to **28 occurrences**, continuing the upward trend.

This increasing pattern suggests that the system processes more events over time as the streaming data progresses, which could be indicative of increasing activity or data flow during the observed time period.

### **Challenges:**

* **Timestamp Alignment:** One of the challenges I faced during this analysis was correctly interpreting the timestamps within each batch window and ensuring that I captured the correct range of events.
* **Data Filtering:** The use of the **filter()** function to isolate logs related to **DataNode** was crucial for narrowing down the data to relevant events. Ensuring the correct log level and component were included in the filter was necessary for accurate results.

### **Insights:**

* I learned how important it is to correctly set time windows when dealing with streaming data, especially in analyzing real-time logs. Using the **window** function in Spark helped in aggregating data over the desired time periods efficiently.
* By using **groupBy()** with windowing, I was able to aggregate data efficiently and analyze trends in real-time, which is key for working with time-sensitive logs and events.

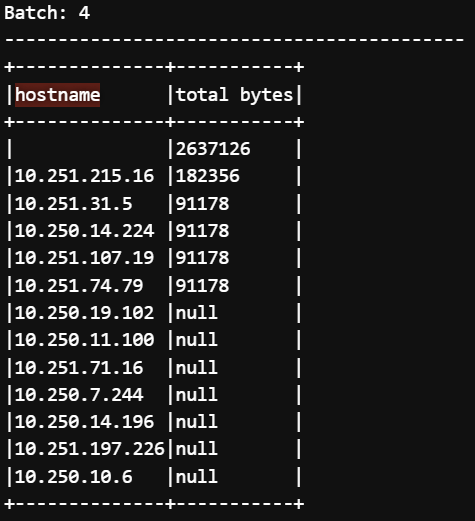
**5. Question 5. Aggregating Data**

I worked on aggregating the total number of bytes transferred per IP address, utilizing the streaming logs.

**Converting the "Size" Column to Integer:** The initial step involved converting the **Size** column, which contained byte data, to an integer data type. This was essential to ensure accurate numerical aggregation. I used the following code for this:

This ensured that the data in the **Size** column was processed as numeric values for summing operations. The next step was extracting the IP address (hostname) from the **Source** column. I used the **regexp\_extract()** function to pull out the IP address from the **Source** field using a regular expression pattern that matches IP addresses.

Once the necessary columns were extracted and converted, I grouped the data by **hostname** and summed the byte sizes (**Size**) for each IP address. I ordered the results in descending order based on the total number of bytes transferred. Finally, I used the **writeStream** method to continuously display the aggregated results on the console in real-time. The results were displayed without truncation for better visibility



**Challenges:**

Logs sometimes had inconsistent timestamps, which posed a challenge for aggregating data into fixed windows. There could be minor discrepancies in timestamp formatting, especially when working with log data streamed in real-time. Ensuring consistent timestamps across all logs was vital for accurate windowing and aggregation. Additionally, the timestamp format used in logs did not always align with the expected format, which required handling those discrepancies effectively. So I used timestamp parsing techniques to ensure all log entries adhered to a consistent format.

#### **Insights/Learnings:**

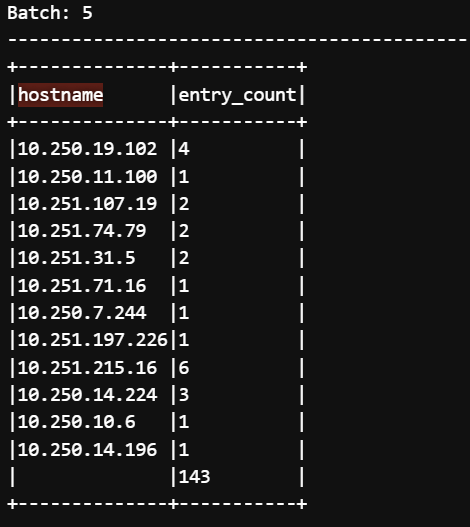
* I gained valuable experience in processing and aggregating data in real-time using Spark's structured streaming API. This is important for applications like monitoring network activity or processing live data.
* By grouping data by hostnames and aggregating the bytes transferred, I was able to get a high-level view of which hosts were consuming the most bandwidth, an essential operation for network traffic analysis.
* I learned how to use the **awaitTermination()** method effectively to manage the lifecycle of streaming queries and ensure that the program continues to process data without termination until specified.

**6. Question 6. Filtering and Triggering**

I focused on filtering log data for entries that were of the "**INFO**" log level and contained a **Block ID** with "blk". The filtered data was then grouped by the hostname, and the total number of such entries for each hostname was calculated. The results were output to the console every 15 seconds.

First I applied a filter on the dataset to select only rows where the **LogLevel** is "INFO" and the **BlockID** contains the string "blk". This was achieved using the **filter()** function combined with conditions that checked for the required log level and block ID pattern:

After filtering the logs, I grouped the entries by **hostname** (extracted from the **Source** column) and counted the number of entries per hostname. The **groupBy()** function was used, followed by **agg()** to perform the aggregation. The aggregated results were then written to the console, with a 15-second trigger interval. The **writeStream** function was used to perform real-time streaming of the data. The query was set to run for 240 seconds, and I used **awaitTermination()** to keep the process active during that time.



The output shown for **Batch 5** displays the count of entries for each hostname where the conditions were met. For instance, the **hostname** "**10.250.19.102**" had 4 entries, while "**10.250.14.196**" had 1 entry. A few observations based on the output:

* The data is grouped by the hostname, and the **entry\_count** shows how many times the condition was met for each IP address.
* Some IP addresses, such as "**10.250.19.102**" and "**10.251.215.16**", have significantly higher counts, while others have very low counts.

#### 

#### **Challenges:**

* One challenge was ensuring that the real-time processing of log data occurred smoothly without delays. Since the query processes data every 15 seconds, the challenge was to make sure the data was being processed efficiently without lag.
* Another challenge I encountered was ensuring that the filtering logic was correctly capturing only the logs with "blk" in the **BlockID**. At first, I didn’t consider the possibility of variations in the case of letters in **BlockID** (like "blk" vs "BLK"). Once I verified that all relevant logs had the proper format, the aggregation worked as expected.

#### 

#### **Insights:**

#### I learned how to perform real-time aggregation on a streaming dataset using Spark’s writeStream function and how to handle continuous log data.

* The process of grouping data based on hostnames and counting entries allowed me to gain insights into which IP addresses were generating the most log entries during the process.
* I also learned how to use **trigger(processingTime="15 seconds")** to manage how often the stream should be processed, which was important for handling real-time data efficiently.

————————————**END OF TASK 4**————————————

**SparkID for Each Task:**

**Task1:**

****

**Task2:**

****

**Task3:**

****

**Task4:**

****

**Note: All the output screenshots provided in this report were taken from the exported text files instead of directly from the terminal/console. This was done to ensure a proper and organized record of the logs for each task. The logs for each task were exported to text files using oc logs -f task1-starter-kit-a19a8896017dc4d7-driver > logs.txt , allowing for better tracking and review.**