WILFRID LAURIER UNIVERSITY

LAURIER

Inspiring Lives.

CP 422 – Programming for Big Data

Fall 2024

Group 23

Assignment #1

Member 1 Name: Nicholas Cai	ID: 210555500
Member 2 Name: Isha Pabla	ID: 190296510
Member 3 Name: Propa Roy	ID: 190453380
Member 4 Name: Saihaan Baig	ID: 211725230
Member 5 Name: Rajan Dosanjh	ID: 200829900
Member 6 Name: Akif Rahman	ID: 210290530

Submission Date: October 9, 2024

All listed members have contributed, read, and approved this submission. All listed members have acknowledged each team member's equal and reasonable contribution to this submission.

Member 1 Name: Nicholas Cai	Signature: NC
Member 2 Name: Isha Pabla	Signature: IP
Member 3 Name: Propa Roy	Signature: PR
Member 4 Name: Saihaan Baig	Signature: SB
Member 5 Name: Rajan Dosanjh	Signature: RD
Member 6 Name: Akif Rahman	Signature: AR

CP422 Assignment One

Data Ingestion and Data Exploration

In [ ]:
```
# load data and display schema
df1 = spark.read.format("csv").option("header", "true").load("dbfs:/FileStore/share
#display first 5 rows
display(df1.head(5))
```

VendorID	tpep_pickup_datetime	tpep_dropoff_datetime	passenger_count	trip_distance
2	2015-01-15 19:05:39	2015-01-15 19:23:42	1	1.59
1	2015-01-10 20:33:38	2015-01-10 20:53:28	1	3.30
1	2015-01-10 20:33:38	2015-01-10 20:43:41	1	1.80
1	2015-01-10 20:33:39	2015-01-10 20:35:31	1	.50
1	2015-01-10 20:33:39	2015-01-10 20:52:58	1	3.00

In [ ]:
```
#statistics for all numerical columns
columns = ['passenger_count', 'trip_distance', 'fare_amount']
df1.describe(columns).show()
```

```
+-------+------------------+------------------+------------------+
|summary|   passenger_count|     trip_distance|       fare_amount|
+-------+------------------+------------------+------------------+
|  count|          12748986|          12748986|          12748986|
|   mean|1.6814908260154964|13.459129611562718|11.905659425776989|
| stddev|1.3379235172874737| 9844.094218468374|10.302537135952232|
|    min|                 0|               .00|             -0.01|
|    max|                 9|             99.90|            999.99|
+-------+------------------+------------------+------------------+
```

Step 4: Data Cleansing

In [ ]:
```
# a) handle missing values
df_cleaned = df1.na.drop(subset=["fare_amount", "trip_distance", "passenger_count"]

# b) Filter out rows with invalid data
df_cleaned = df_cleaned.filter((df_cleaned["fare_amount"] > 0.0) & (df_cleaned["tri


from pyspark.sql.functions import to_timestamp, col
# c) Converting pickup_datetime and dropoff_datetime to timestamp
df_cleaned = df_cleaned.withColumn("tpep_pickup_datetime", to_timestamp(df_cleaned[
df_cleaned = df_cleaned.withColumn("tpep_dropoff_datetime", to_timestamp(df_cleaned
# d) Create new columns, such as trip duration (in minutes) and trip speed (in mile
df_cleaned = df_cleaned.withColumn("trip_duration", (col("tpep_dropoff_datetime").c
df_cleaned = df_cleaned.withColumn("trip_speed",col("trip_distance") / (col("trip_d

### remove hash to display dataframe

display(df_cleaned.head(10))
```

VendorID	tpep_pickup_datetime	tpep_dropoff_datetime	passenger_count	trip_distance
2	2015-01-15T19:05:39.000+0000	2015-01-15T19:23:42.000+0000	1	1.59
1	2015-01-10T20:33:38.000+0000	2015-01-10T20:53:28.000+0000	1	3.30
1	2015-01-10T20:33:38.000+0000	2015-01-10T20:43:41.000+0000	1	1.80
1	2015-01-10T20:33:39.000+0000	2015-01-10T20:35:31.000+0000	1	.50
1	2015-01-10T20:33:39.000+0000	2015-01-10T20:52:58.000+0000	1	3.00
	2015-01-	2015-01-		

Step 5: Explorator Data Analysis

In [ ]:

```
from pyspark.sql.functions import avg

passanger_stats = df_cleaned.groupBy("passenger_count") \
    .agg(avg("fare_amount").alias("Average_fare"), avg("trip_distance").alias("Aver
    .orderBy("passenger_count", ascending=True)

display(passanger_stats)
```

passenger_count	Average_fare	Average_TripDistance
0	10.656507448186531	2.2684520725388597
1	11.745547515606345	14.937032342159762
2	12.365831817242697	18.128457929665586
3	12.075318298731478	2.8640130524626892
4	12.145093869519004	2.8824759573421357
5	11.96483540565141	2.8649327401331903
6	11.79932281080413	2.797320617139272
7	12.428571428571429	3.3000000000000003
8	33.5	7.263333333333333

In [ ]:
```
#find busies time of day -- pickup hours with higher counts are busier hours -- thu
from pyspark.sql.functions import hour

#new_df is the dataframe which also contains a hours feature where it takes only th

new_df = df_cleaned.withColumn("hours", hour(df_cleaned["tpep_pickup_datetime"]))

busiest_time = new_df.groupBy("hours").count()
busiest_time = busiest_time.orderBy("count", ascending=False)
display(busiest_time)
```

hours	count
19	800917
18	795071
20	730056
21	707590
22	683067
17	664322
14	654800
15	644313
12	633759

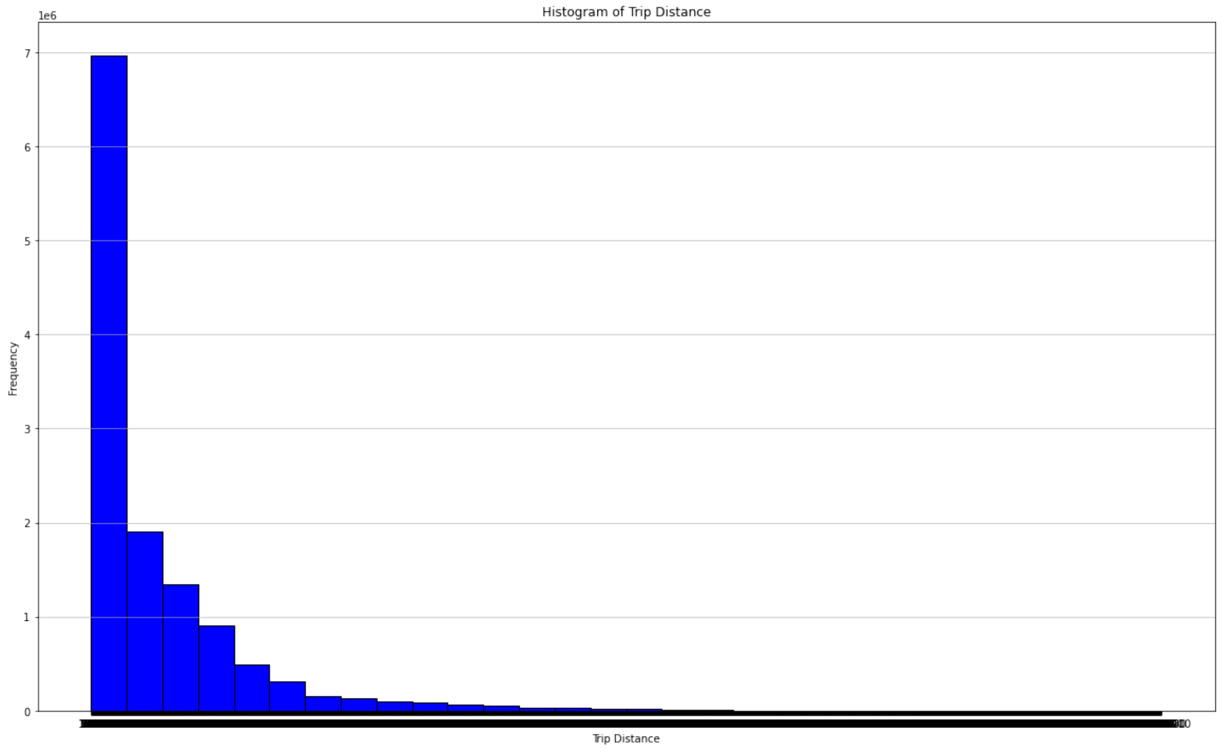
In [ ]:
```
#Which neighbourhoods have the highest average fare amount
"""
split nyc into east and west and determine which is busier
get center longitutude if a pickup longitude is < than the center it is west nyc
if greator it is east nyc


"""
from pyspark.sql.functions import when

#make a location row with
df_with_location = df_cleaned.withColumn(
    "neighbourhood",
    when(df_cleaned.pickup_longitude < -74.0060, "West").otherwise("East")
)
#calculate average fare for pickup location at west nyc and east nyc
neighbourhood_avg = df_with_location.groupBy("neighbourhood").agg(avg("fare_amount"

display(neighbourhood_avg)

neighbourhood_avg.show()
```

neighbourhood	average_fare_amount
East	11.784060709455977
West	13.31388753868902

```
+-------------+-------------------+
|neighbourhood|average_fare_amount|
+-------------+-------------------+
|         East| 11.784060709455977|
|         West|  13.31388753868902|
+-------------+-------------------+
```

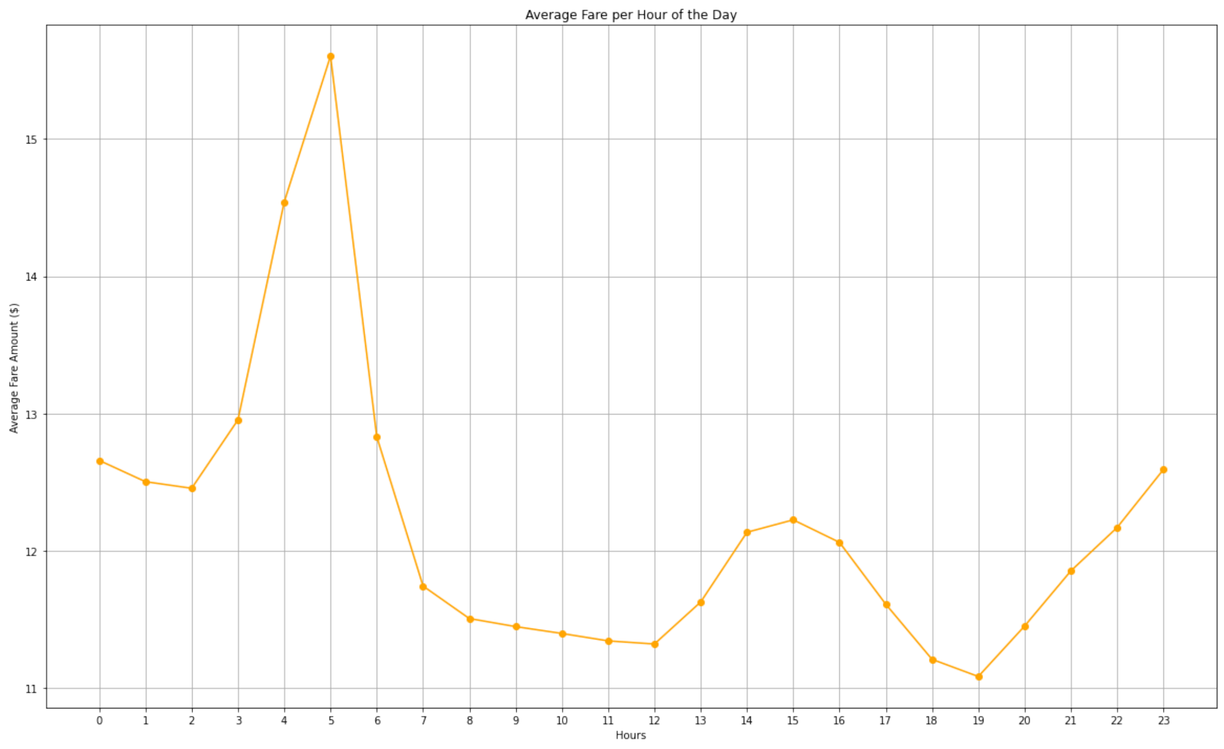
Step 6: Plotting Trip Distances

In [ ]:
```
import pandas as pd
import matplotlib.pyplot as plt

df_cleaned = df_cleaned.toPandas()

# Create histogram
plt.figure(figsize=(20, 12))
plt.hist(df_cleaned['trip_distance'], bins=30, color='blue', edgecolor='black')
plt.title('Histogram of Trip Distance')
plt.xlabel('Trip Distance')
plt.ylabel('Frequency')
plt.grid(axis='y', alpha=0.75)

# Show the plot
plt.show()
```



Visualize average fares by hour

In [ ]:
```
#make a dataset of averagefare for every hour of the day and copnvert to pandas dat
average_fare_graph = new_df.groupBy("hours") \
    .agg(avg("fare_amount").alias("Average_fare")) \
    .orderBy("hours", ascending=True)

average_fare_graph = average_fare_graph.toPandas()

plt.figure(figsize=(20, 12))
plt.plot(average_fare_graph['hours'], average_fare_graph['Average_fare'], marker='o
plt.xlabel('Hours')
plt.ylabel('Average Fare Amount ($)')
plt.title('Average Fare per Hour of the Day')
plt.grid(True)
plt.xticks(range(0, 24))
plt.show()
```



Summary and Insights

Busiest time of the day: Looking at our analysis we can see a table where there is a count of yellow taxi trips at each hour of the day where it can be seen that 17th-22nd hour of the day are the busiest for taxis. The demand for taxis in later times of the day could be due to an increase in alcohol consumption at night as people go out for drinks and dinner.

In the Average Fare per Hour of The Day graph above, it can be seen that the fare prices increase rapidly between midnight and early morning. This could be due to the increase in demand as there are less nightshift workers. The cheapest time is mid day when there is likely the most amount of yellow taxis readily available

Part B: Advanced Prescriptive Analytics:

Feature Engineering: 1. Trip Duration:

In [ ]:
```
from pyspark.sql.functions import col, to_timestamp, floor, lpad, concat, lit

# Convert columns to timestamp if they aren't already
df1 = df1.withColumn("tpep_pickup_datetime", to_timestamp("tpep_pickup_datetime"))
df1 = df1.withColumn("tpep_dropoff_datetime", to_timestamp("tpep_dropoff_datetime")

# Calculate trip duration in seconds
df1 = df1.withColumn("trip_duration_seconds", (col("tpep_dropoff_datetime").cast("l

# Extract hours, minutes, and seconds from trip duration
df1 = df1.withColumn("hours", floor(col("trip_duration_seconds") / 3600)) \
         .withColumn("minutes", floor((col("trip_duration_seconds") % 3600) / 60))
         .withColumn("seconds", col("trip_duration_seconds") % 60)

# Format the duration as HH:mm:ss
df1 = df1.withColumn("trip_duration_formatted",
                     concat(
                         lpad(col("hours").cast("string"), 2, "0"), lit(":"),
                         lpad(col("minutes").cast("string"), 2, "0"), lit(":"),
                         lpad(col("seconds").cast("string"), 2, "0")
                     ))

# Display result to verify
df1.select("tpep_pickup_datetime", "tpep_dropoff_datetime", "trip_duration_formatte
+--------------------+--------------------+-----------------------+
|tpep_pickup_datetime|tpep_dropoff_datetime|trip_duration_formatted|
+--------------------+--------------------+-----------------------+
| 2015-01-15 19:05:39|  2015-01-15 19:23:42|               00:18:03|
| 2015-01-10 20:33:38|  2015-01-10 20:53:28|               00:19:50|
| 2015-01-10 20:33:38|  2015-01-10 20:43:41|               00:10:03|
| 2015-01-10 20:33:39|  2015-01-10 20:35:31|               00:01:52|
| 2015-01-10 20:33:39|  2015-01-10 20:52:58|               00:19:19|
| 2015-01-10 20:33:39|  2015-01-10 20:53:52|               00:20:13|
| 2015-01-10 20:33:39|  2015-01-10 20:58:31|               00:24:52|
| 2015-01-10 20:33:39|  2015-01-10 20:42:20|               00:08:41|
| 2015-01-10 20:33:39|  2015-01-10 21:11:35|               00:37:56|
| 2015-01-10 20:33:40|  2015-01-10 20:40:44|               00:07:04|
+--------------------+--------------------+-----------------------+
only showing top 10 rows
```

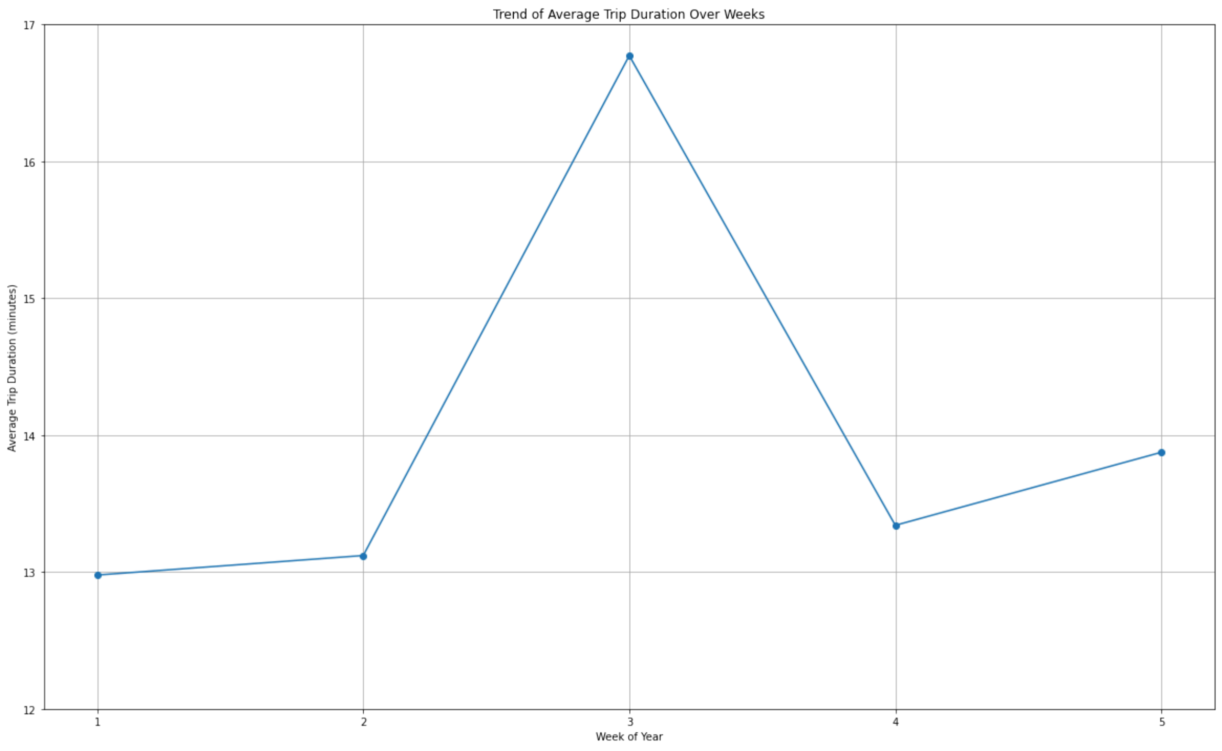
Feature Engineering: 2. Hour and Day Extraction:

In [ ]:
```python
from pyspark.sql.functions import hour, dayofweek

# Extract hour and day of the week from pickup_datetime
df1 = df1.withColumn("pickup_hour", hour("tpep_pickup_datetime")) \
         .withColumn("pickup_day", dayofweek("tpep_pickup_datetime"))

df1.select("tpep_pickup_datetime","pickup_hour", "pickup_day").show(5)
```

```
+--------------------+-----------+----------+
|tpep_pickup_datetime|pickup_hour|pickup_day|
+--------------------+-----------+----------+
| 2015-01-15 19:05:39|         19|         5|
| 2015-01-10 20:33:38|         20|         7|
| 2015-01-10 20:33:38|         20|         7|
| 2015-01-10 20:33:39|         20|         7|
| 2015-01-10 20:33:39|         20|         7|
+--------------------+-----------+----------+
only showing top 5 rows
```

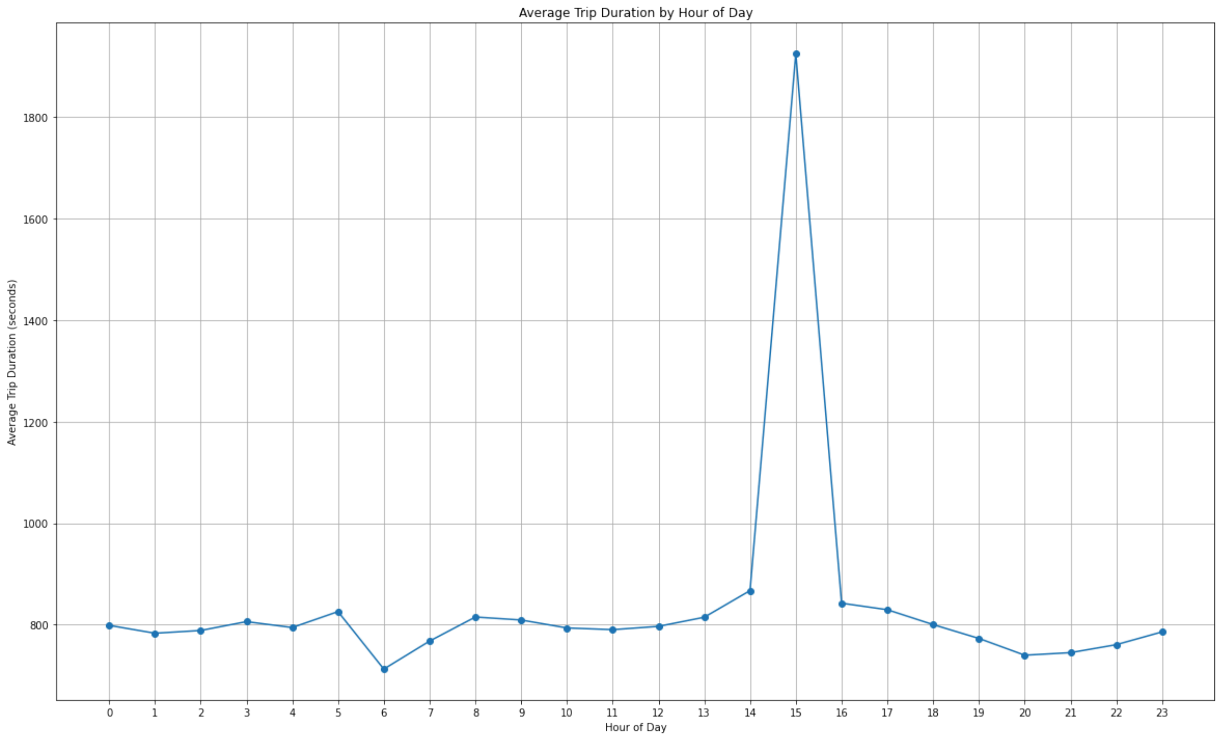
Feature Engineering: 3. Trend Over Month:

In [ ]:
```python
from pyspark.sql.functions import weekofyear

# Extract the week of the year from pickup_datetime
df1 = df1.withColumn("pickup_week", weekofyear("tpep_pickup_datetime"))

# Calculate the average trip duration in minutes per week and sort by week
weekly_trends = df1.groupBy("pickup_week") \
    .agg((avg("trip_duration_seconds") / 60).alias("avg_trip_duration_minutes")) \
    .orderBy("pickup_week") \
    .toPandas()

# Plotting the weekly trend in minutes
import matplotlib.pyplot as plt

if not weekly_trends.empty:
    plt.figure(figsize=(20,12))
    plt.plot(weekly_trends["pickup_week"], weekly_trends["avg_trip_duration_minutes
    plt.xlabel("Week of Year")
    plt.ylabel("Average Trip Duration (minutes)")
    plt.title("Trend of Average Trip Duration Over Weeks")
    plt.xticks(range(1, weekly_trends["pickup_week"].max() + 1))  # Show relevant w
    plt.yticks(range(12,18))
    plt.grid()
    plt.show()
else:
    print("No data available for weekly trends.")
```



Feature Engineering: 4. Hourly Analysis:

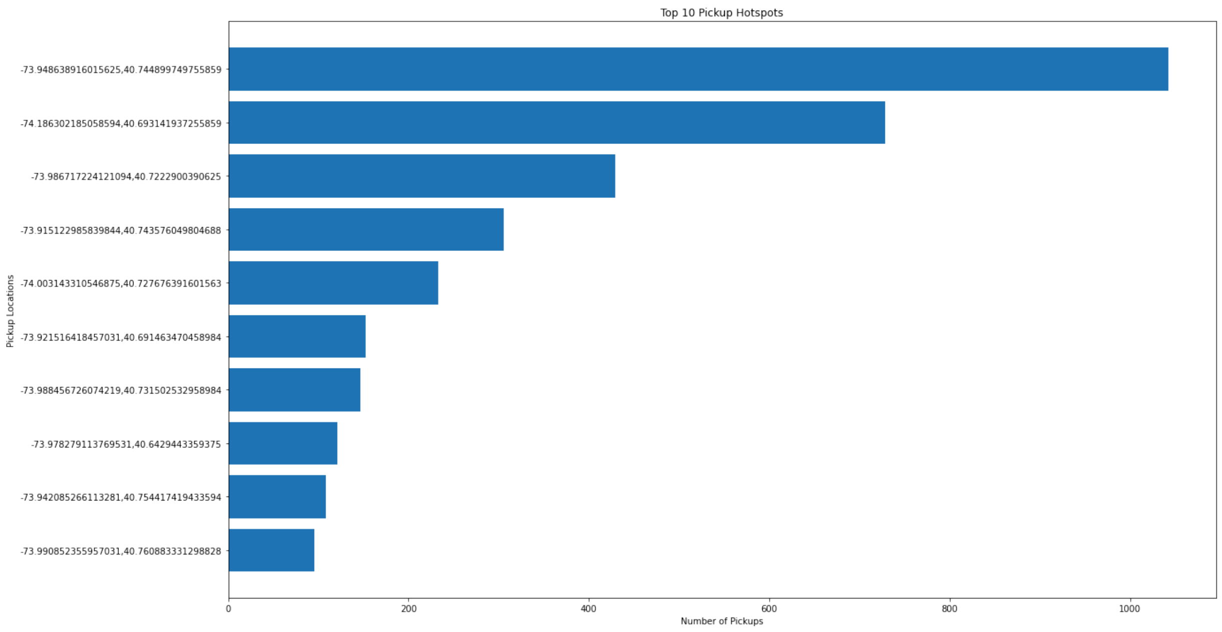
In [ ]:
```
from pyspark.sql.functions import hour

# Extract the hour from pickup_datetime
df1 = df1.withColumn("pickup_hour", hour("tpep_pickup_datetime"))

# Calculate the average trip duration per hour and sort by hour
hourly_analysis = df1.groupBy("pickup_hour").avg("trip_duration_seconds").orderBy("

# Plotting the hourly analysis
import matplotlib.pyplot as plt

if not hourly_analysis.empty:
    plt.figure(figsize=(20,12))
    plt.plot(hourly_analysis["pickup_hour"], hourly_analysis["avg(trip_duration_sec
    plt.xlabel("Hour of Day")
    plt.ylabel("Average Trip Duration (seconds)")
    plt.title("Average Trip Duration by Hour of Day")
    plt.xticks(range(0, 24))  # Set x-axis to show all 24 hours
    plt.grid()
    plt.show()
else:
    print("No data available for hourly analysis.")
```



Feature Engineering: 5. Identify Hotspots:

In [ ]:
```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col

# Initialize Spark session if not already done
spark = SparkSession.builder.appName("TaxiData").getOrCreate()

# Load the data into a DataFrame
df1 = spark.read.format("csv").option("header", "true").load("dbfs:/FileStore/share

# Filter out rows where pickup_longitude or pickup_latitude is 0
df_filtered = df1.filter((col("pickup_longitude") != 0) & (col("pickup_latitude") !

# Group by pickup location to find hotspots and sort by count
pickup_hotspots = df_filtered.groupBy("pickup_longitude", "pickup_latitude").count(

# Plot pickup hotspots
import matplotlib.pyplot as plt

if not pickup_hotspots.empty:
    plt.figure(figsize=(20,12))
    plt.barh(pickup_hotspots["pickup_longitude"].astype(str) + "," + pickup_hotspot
    plt.xlabel("Number of Pickups")
    plt.ylabel("Pickup Locations")
    plt.title("Top 10 Pickup Hotspots")
    plt.gca().invert_yaxis()  # To display highest at the top
    plt.show()
```

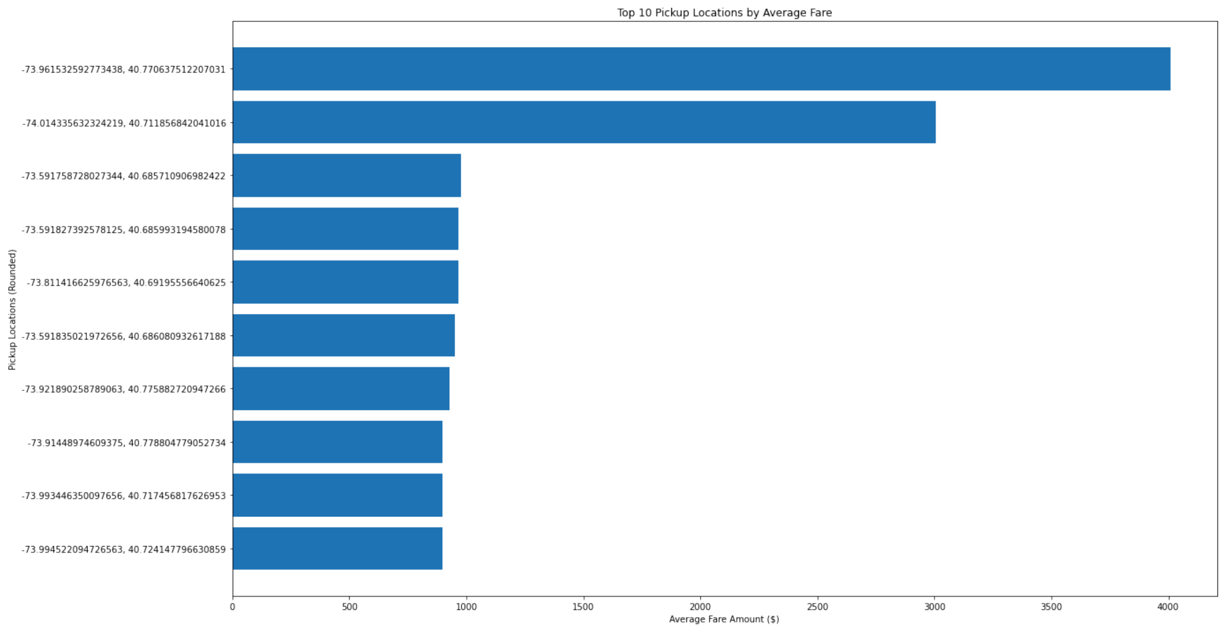
```
else:
    print("No data available for pickup hotspots.")
```



Feature Engineering: 6. Average Fare by Pickup Location:

In [ ]:
```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, avg, round, concat, lit
import matplotlib.pyplot as plt

# Initialize Spark session if not already done
spark = SparkSession.builder.appName("TaxiData").getOrCreate()

# Load the data into a DataFrame
df1 = spark.read.format("csv").option("header", "true").load("dbfs:/FileStore/share

# Filter out rows where pickup_longitude, pickup_latitude, or fare_amount is invali
df_filtered = df1.filter(
    (col("pickup_longitude") != 0) & (col("pickup_latitude") != 0) & (col("fare_amo
)

# Round longitude and latitude and create a pickup_location identifier
df_filtered = df_filtered.withColumn("pickup_location",
                                     concat(col("pickup_longitude"), lit(", "), col

# Group by pickup location to calculate average fare and sort by highest average fa
top_fare_locations = df_filtered.groupBy("pickup_location").agg(avg("fare_amount").

# Plot top fare locations
if not top_fare_locations.empty:
    plt.figure(figsize=(20,12))
    plt.barh(top_fare_locations["pickup_location"], top_fare_locations["average_far
    plt.xlabel("Average Fare Amount ($)")
    plt.ylabel("Pickup Locations (Rounded)")
    plt.title("Top 10 Pickup Locations by Average Fare")
    plt.gca().invert_yaxis()  # To display highest fare at the top
    plt.show()
```

```
else:
    print("No data available for top fare locations.")
```



Feature Engineering: 7. Correlation Analysis:

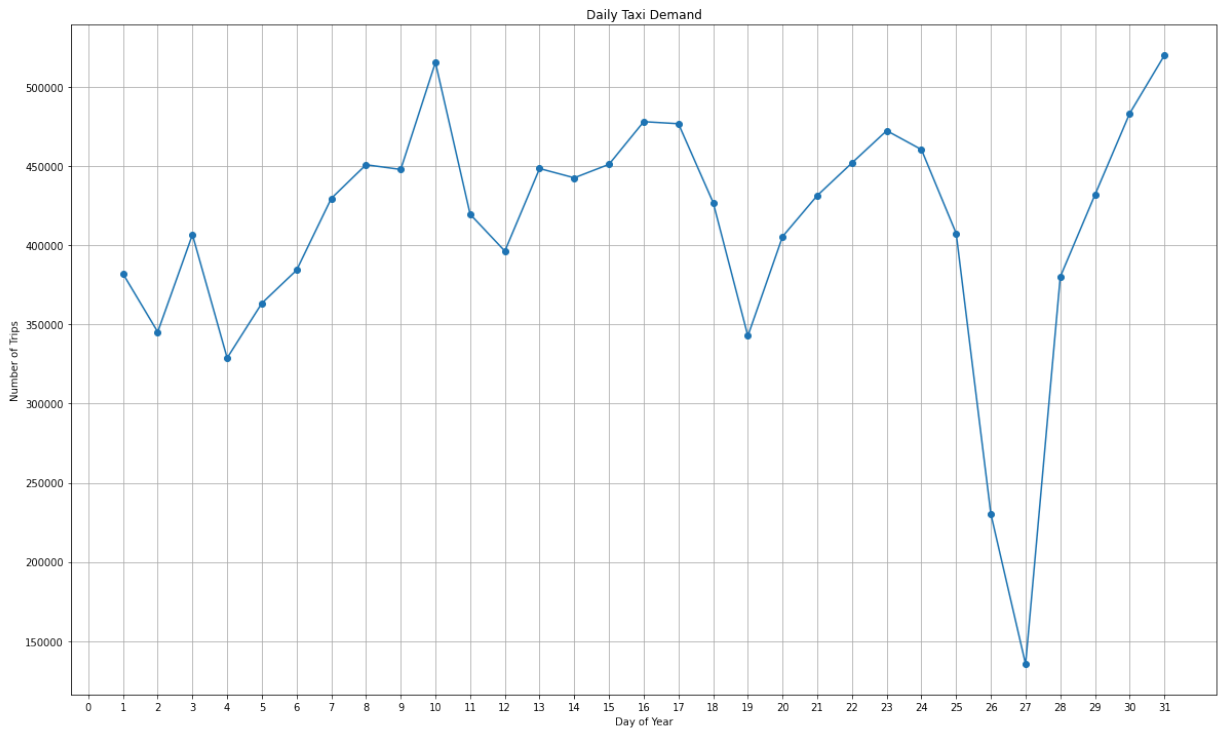
In [ ]:
```
from pyspark.sql.functions import col, to_timestamp

# Convert columns to timestamp
df1 = df1.withColumn("tpep_pickup_datetime", to_timestamp("tpep_pickup_datetime"))
df1 = df1.withColumn("tpep_dropoff_datetime", to_timestamp("tpep_dropoff_datetime")

# Calculate trip duration in seconds
df1 = df1.withColumn("trip_duration_seconds", (col("tpep_dropoff_datetime").cast("l

# Cast columns to float if they are in string format
df1 = df1.withColumn("trip_duration_seconds", col("trip_duration_seconds").cast("fl
df1 = df1.withColumn("trip_distance", col("trip_distance").cast("float"))
df1 = df1.withColumn("fare_amount", col("fare_amount").cast("float"))

# Calculate correlations
trip_duration_distance_corr = df1.stat.corr("trip_duration_seconds", "trip_distance
trip_duration_fare_corr = df1.stat.corr("trip_duration_seconds", "fare_amount")
trip_distance_fare_corr = df1.stat.corr("trip_distance", "fare_amount")

# Display correlation results
print("Correlation between Trip Duration and Trip Distance:", trip_duration_distanc
print("Correlation between Trip Duration and Fare Amount:", trip_duration_fare_corr
print("Correlation between Trip Distance and Fare Amount:", trip_distance_fare_corr
```

```
Correlation between Trip Duration and Trip Distance: 1.5306018836784782e-05
Correlation between Trip Duration and Fare Amount: 0.011440083925224673
Correlation between Trip Distance and Fare Amount: 0.00044221178580128485
```

Feature Engineering: 8. Monthly Taxi Demand Analysis:

In [ ]:
```
from pyspark.sql.functions import dayofyear
```

```
# Extract day of the year from pickup datetime
df1 = df1.withColumn("pickup_day", dayofyear("tpep_pickup_datetime"))

# Group by day to calculate the number of trips
daily_demand = df1.groupBy("pickup_day").count().orderBy("pickup_day").toPandas()

# Plotting daily demand
import matplotlib.pyplot as plt

if not daily_demand.empty:
    plt.figure(figsize=(20,12))
    plt.plot(daily_demand["pickup_day"], daily_demand["count"], marker='o', linesty
    plt.xlabel("Day of Year")
    plt.ylabel("Number of Trips")
    plt.title("Daily Taxi Demand")
    plt.xticks(range(0, 32))
    plt.grid()
    plt.show()
else:
    print("No data available for daily demand.")
```



Part C: Practice questions

1. **Define Big Data and explain why traditional data processing methods are insufficient for managing it.**

Big Data refers to data that is high in volume, variety, and velocity. This means it comes from diverse sources, is rapidly changing, and needs to have an intrinsic value. Traditional data processing methods are insufficient for managing big data as they are not capable of storing and analyzing large amounts of data and cannot organize and process it rapidly. Data is generated daily and needs to be managed efficiently, which is why new data processing tools and methods are required to take stored data and clean it in real time to make it relevant and meaningful.

2. **Describe the role of Hadoop in solving big data challenges. What are the core strengths of Hadoop compared to traditional relational databases?**

Hadoop is an open-source software framework that allows us to store, analyze, and process a huge volume of datasets across a cluster of computers. Hadoop helps us solve big data challenges by providing us with a scalable and cost-efficient method for storing and processing data. Hadoop compared to traditional relational databases has built-in fault tolerance, processes both structured and unstructured data, and is more scalable and cost-effective which makes Hadoop an ideal choice for big data challenges.

3. **How do Hadoop's scalability and fault-tolerance features contribute to its popularity in big data processing?**

Hadoop's scalability feature allows it to handle large amounts of data by adding more nodes in a cluster or increasing the data storage capacity all while increasing processing power and improving efficiency. Hadoop's fault tolerance feature replicates pieces of data across 3 different nodes as a way of avoiding data loss if a component fails. These features contribute to Hadoop's popularity in big data processing, making it reliable and high-performing.

4. **What are the key characteristics (the 4Vs) of Big Data?**

The 4 key characteristics of Big Data are volume, variety, velocity, and value. The first key characteristic of big data is the high volume of data. Massive amounts of data are being stored and generated. The second characteristic is variety, which means data comes from several diverse sources. The third characteristic is velocity which refers to the unprecedented speed at which data is generated and processed. The last characteristic is value which refers to Big data's potential of generating some positive value or utility.

5. **Differentiate between structured, semi-structured, and unstructured data with examples. Explain how Hadoop handles these different types of data.**

Structured data refers to data that is organized, has a data type, and contains a predefined format. An example of structured data is a SQL database table which contains student first names, last names, identification numbers, email, program name, and birth city. Semi-structured data does not have a defined data model that a machine can completely understand. Still, it uses metadata such as tags to allow the data to be analyzed and managed more easily than unstructured data. An example of semi-structured data is emails as it contains some structured information in terms of a sender address, recipient address, and date sent, and unstructured data in terms of the body of the message sent which are not processed by a machine. Unstructured data refers to data that does not have a predefined data model which means it is difficult to process and analyze. An example of unstructured data includes images or videos. Hadoop stores all three of these data types but processes them in different ways. Structured data is processed using HBase or Apache Hive using MapReduce or Spark, semi-structured data uses frameworks such as Apache Hive or Apache Pig to extract patterns and query data, and unstructured data is processed using MapReduce or Apache Spark or machine learning libraries.

6. **Define the following key components of the HDFS and describe their functions:**

a) **NameNode** - The NameNode is a master node that stores the directory tree of the file system file metadata, and the locations of each file in the cluster.

b) **DataNode** - The DataNode is a worker node that stores and manages HDFS blocks on the local disk. It also reports the health and status of individual data stores back to the NameNode.

c) **Secondary NameNode** - The Secondary NameNode is a master node. It performs housekeeping tasks and checkpointing on behalf of the NameNode. It is not a backup NameNode.

d) **Block Replication** - Block Replication is the process in which files are stored in HDFS as fixed-size blocks, which are then replicated across many nodes in a cluster.

7. **Explain the MapReduce programming model. What are the main phases of a MapReduce job, and what are their purposes?**

MapReduce is a programming model within Hadoop that is used to process big data. It is composed of three main phases. The first phase is the Map function which takes input as large datasets and transforms it into key-value pairs. The second is the Shuffle and Sort which sorts and groups the intermediate key-value pairs and ensures the keys from the mappers are going to the correct reducer. The final phase is the Reduce function which takes the sorted key-value pairs and aggregates them into a final key-value pair.

8. **Describe the concept of data locality in MapReduce and its importance for efficient processing in distributed systems.**

Data locality in MapReduce refers to the principle of moving computation closer to the data rather than moving data across the network to the computation. In distributed systems, the data is often stored across multiple nodes. MapReduce takes advantage of this by trying to execute the map tasks on the nodes where the data resides (or nearby), reducing the need for large data transfers over the network. The importance of data locality is in its ability to improve processing efficiency and performance. Transferring large amounts of data over the network can be costly in terms of time and bandwidth. By ensuring that the computation happens closer to the data, MapReduce minimizes network congestion, reduces latency, and speeds up the overall processing. This makes distributed data processing more scalable and cost-effective.

9. **Explain what a combiner is in the MapReduce framework and how it optimizes the performance of a MapReduce job.**

A combiner in the MapReduce framework is an optional class used after the map function and before the shuffle and sort phase to reduce the volume of data transferred between them. The combiner may not always be optimal and effective to use as it can increase the amount of data being transferred and can produce incorrect results based on the aggregation function being used.

It optimizes the performance of a MapReduce by minimizing the intermediate key-value pairs produced by the mapper and summarizing or combining them using the combiner before sending them to the reducer. This limits network congestion, reduces the size of the intermediate key-value pairs thereby reducing the time to transfer data between the map and reduce function, and optimizing the reducer by preprocessing its input data.

10. **Explain what Hadoop Streaming is and how it allows Python programs to interact with Hadoop's MapReduce framework.**

Hadoop Streaming is a utility that lets you use any executable or script (like Python) as a mapper or reducer in Hadoop's MapReduce framework. It works by piping input data through standard input (stdin) to your script, which processes the data and outputs key-value pairs to standard output (stdout). This allows for language flexibility, making it easy to write MapReduce jobs in Python or other languages instead of Java.

11. **Discuss the concept of standard input and standard output in the context of Hadoop Streaming. Why are these important for connecting Python programs to Hadoop?**

In Hadoop Streaming, standard input (stdin) and standard output (stdout) are used to pass data to programs. The program reads its input and parses them line by line from standard input and the program writes the output through standard output after it is processed. These are important for connecting Python programs to Hadoop as they allow Python scripts to interact with MapReduce without special libraries. Through standard input, Hadoop can send data directly to the Python program and standard output allows the Python program to send its results to Hadoop. This means Hadoop will accept and handle data without being concerned with where the data originates from and where it is going.

12. **Provide an example of a real-world use case where Hadoop Streaming with Python would be preferable over other methods.**

A real-world example of using Hadoop Streaming with Python is in telecommunications companies. They can use Hadoop to analyze large amounts of data for many purposes. For instance, they can use data for predictive maintenance to prevent equipment failures in their infrastructure, analyze data to plan efficient network routes and find the best locations for new cell towers, and examine customer behavior and billing information to create better service plans and offers. Python makes these tasks easier to implement quickly, offering flexibility and faster development compared to other languages like Java.