Group 01

Group Members:

Name	BITSID	Weightage
Ajay Saxena	2021sc04160	**100%**
MOVVA MANASWI	2021sc04180	**100%**
PRARITA ARORA	2021sc04049	**100%**
ADITI	2021sc04050	**100%**

Hadoop Configuration Files:

CORE-SITE

fs.defaultFS hdfs://localhost:9000 hadoop.tmp.dir /C:/bigdata/hadoop-3.2.2/dfs/tempdir fs.trash.interval 1440

HDFS-SITE

dfs.replication 1 dfs.namenode.name.dir /C:/bigdata/hadoop-3.2.2/dfs/namenode dfs.datanode.data.dir /C:/bigdata/hadoop-3.2.2/dfs/datanode

Environment Variables

Variable Name : HADOOP_HOME
Variable Value: C:\bigdata\hadoop-3.2.2
Path: %HADOOP_HOME%\bin
Path: %HADOOP_HOME%\sbin
Path: %HADOOP_CONF_DIR%\bin
Path: %HADOOP_CONF_DIR%\sbin

Spark

Spark Environment Variables

Variable Name: SPARK_HOME
Variable Value: C:\bigdata\spark-3.1.2-bin-hadoop3.2
Path: %SPARK_HOME%\bin
Path: %SPARK_HOME%\sbin

Java

Java Environment variables

Variable Name: JAVA_HOME
Variable Value : C:\bigdata\Java\jdk-11.0.12
Path : %JAVA_HOME%\bin

Directory Formation and dataset upload

I have setup single node HDFS on my Windows 11 machine and start HDFS and YARN .

C:\Users\datat>start-dfs

C:\Users\datat>start-yarn

C:\Users\datat>jps

24272 DataNode

22980 Jps

24488 NameNode

8392 ResourceManager

This command creates a directory structure in HDFS. The -p flag is used to create parent directories. The directory structure /user/datat/input/ is being created in HDFS.

hdfs dfs -mkdir -p /user/datat/input/

Upload the local files from local file system (on local machine) to HDFS. The file "C:\Users\datat\OneDrive - Data Tinker\M.Tech\3rd_Sem\BDS\BDS_Assignment_2 \yellow_tripdata_2020-06.csv" and "C:\Users\datat\OneDrive - Data Tinker\M.Tech\3rd_Sem\BDS\BDS_Assignment_2 \taxi+_zone_lookup.csv" are being uploaded to the /user/datat/input/ directory in HDFS.

hdfs dfs -put "C:\Users\datat\OneDrive - Data Tinker\M.Tech\3rd_Sem\BDS\BDS_Assignment_2\yellow_tripdata_2020-06.csv" /user/datat/input/
hdfs dfs -put "C:\Users\datat\OneDrive - Data Tinker\M.Tech\3rd_Sem\BDS\BDS_Assignment_2\taxi+_zone_lookup.csv" /user/datat/input/

Check if the dataset has been copied to HDFS location

hdfs dfs -ls /user/datat/input/

o/p: Found 2 items
-rw-r--r-- 1 datat supergroup 10724 2023-09-05 06:33 /user/datat/input/taxi+_zone_lookup.csv
-rw-r--r-- 1 datat supergroup 41661852 2023-09-04 21:51 /user/datat/input/yellow_tripdata_2020-06.csv

Step 1: Setting up PySpark in Jupyter Notebook

Before we begin let's ensure that PySpark is installed and is accessible via Jupyter Notebook. If it's not installed, we can typically do so via pip:

In [1]:
```
#pip install pyspark
```

Step 2: Start Jupyter Notebook

Launch Jupyter Notebook from Anaconda environment. Set up an environment in Anaconda for PySpark Check spark version.

In [17]:
```
import pyspark
print("PySpark version:", pyspark.__version__)
```

```
PySpark version: 3.1.1
```

Step 3: Create a New Notebook & Setup Spark Session

Now, in the Jupyter notebook:

In [18]:
```
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .appName("HDFS Data Exploration") \
    .getOrCreate()
```

This will start a local Spark session.

Step 4: Read the Data from HDFS

Question no. 01. Read the data in yellow_tripdata_2020-06.csv file into a dataframe created in spark.

In [19]:
```
#read your data from HDFS into a DataFrame.The default HDFS port is 9000, but if the HDFS has been configured differently, localhost
data_path = "hdfs://localhost:9000/user/datat/input/yellow_tripdata_2020-06.csv"

df = spark.read.csv(data_path, header=True, inferSchema=True)

# Show the first few rows of the DataFrame to ensure it's loaded correctly
df.show(5)
```

```
+--------------------+--------------------+---------------+-------------+------------+------------+------------+-----------+-----+---
|tpep_pickup_datetime|tpep_dropoff_datetime|passenger_count|trip_distance|PULocationID|DOLocationID|payment_type|fare_amount|extra|mta
+--------------------+--------------------+---------------+-------------+------------+------------+------------+-----------+-----+---
|    01-06-2020 00:31|    01-06-2020 00:49|              1|          3.6|         140|          68|           1|       15.5|  3.0|
|    01-06-2020 00:42|    01-06-2020 01:04|              1|          5.6|          79|         226|           1|       19.5|  3.0|
|    01-06-2020 00:39|    01-06-2020 00:49|              1|          2.3|         238|         116|           2|       10.0|  0.5|
|    01-06-2020 00:56|    01-06-2020 01:11|              1|          5.3|         141|         116|           2|       17.5|  3.0|
|    01-06-2020 00:16|    01-06-2020 00:29|              1|          4.4|         186|          75|           1|       14.5|  3.0|
+--------------------+--------------------+---------------+-------------+------------+------------+------------+-----------+-----+---
only showing top 5 rows
```

Step 5: Explore the Data

Now we can use the PySpark DataFrame API to work with our data

In [20]:
```
#print the schema
df.printSchema()
```

```
root
 |-- tpep_pickup_datetime: string (nullable = true)
 |-- tpep_dropoff_datetime: string (nullable = true)
 |-- passenger_count: integer (nullable = true)
 |-- trip_distance: double (nullable = true)
 |-- PULocationID: integer (nullable = true)
 |-- DOLocationID: integer (nullable = true)
 |-- payment_type: integer (nullable = true)
 |-- fare_amount: double (nullable = true)
 |-- extra: double (nullable = true)
 |-- mta_tax: double (nullable = true)
 |-- tip_amount: double (nullable = true)
 |-- tolls_amount: double (nullable = true)
 |-- improvement_surcharge: double (nullable = true)
 |-- total_amount: double (nullable = true)
```

In [21]:
```
#count the number of rows
df.count()
```

Out[21]: 549760

Question-2. Count the number of taxi trips for each hour

In [27]:
```
# Convert pickup datetime string to timestamp
from pyspark.sql.functions import from_unixtime, unix_timestamp

df = df.withColumn("pickup_datetime", from_unixtime(unix_timestamp("tpep_pickup_datetime", 'dd-MM-yyyy HH:mm')).cast("timestamp"))
```

In [28]:
```
# To count the number of taxi trips for each hour, we need to extract the hour from the tpep_pickup_datetime column and then group by
hourly_counts = df.groupBy(hour("pickup_datetime").alias("Hour")).agg(count("*").alias("Number of Trips")).orderBy("Hour")

# Show the hourly counts
hourly_counts.show(24)
```

```
+----+---------------+
|Hour|Number of Trips|
+----+---------------+
|   0|           8122|
|   1|           6643|
|   2|           5111|
|   3|           5124|
|   4|           7136|
|   5|           6955|
|   6|          14907|
|   7|          19957|
|   8|          24824|
|   9|          28408|
|  10|          31948|
|  11|          35190|
|  12|          38083|
|  13|          39475|
|  14|          40525|
|  15|          40971|
|  16|          38627|
|  17|          38225|
|  18|          34181|
|  19|          26477|
|  20|          18518|
|  21|          15020|
|  22|          13238|
|  23|          12095|
+----+---------------+
```

Question-2.2. Create a table view of the data frame created in step 1 above and write SparkSQL queries to find out the following:

Question no.03. Average fare amount collected by hour of the day

In [29]:
```
from pyspark.sql.functions import from_unixtime, unix_timestamp, hour

# Convert the string column to a timestamp column
df_timestamp = df.withColumn("pickup_timestamp", from_unixtime(unix_timestamp("tpep_pickup_datetime", "dd-MM-yyyy HH:mm")))

# Extract the hour from the new timestamp column
df_with_hour = df_timestamp.withColumn("hour_of_day", hour("pickup_timestamp"))

# Create a new temp view with the extracted hour
df_with_hour.createOrReplaceTempView("taxi_data_with_hour")

# Now compute the average fare amount by hour of the day using SparkSQL
avg_fare_by_hour = spark.sql("""
    SELECT
        hour_of_day,
        AVG(fare_amount) as average_fare
    FROM
        taxi_data_with_hour
    WHERE
        hour_of_day IS NOT NULL
    GROUP BY
        hour_of_day
    ORDER BY
        hour_of_day
""")

avg_fare_by_hour.show(24)  # to display averages for all 24 hours
```

```
+-----------+------------------+
|hour_of_day|      average_fare|
+-----------+------------------+
|          0|18.880695641467593|
|          1|27.534966129760434|
|          2| 30.12912737233388|
|          3|  35.2012607338016|
|          4|40.932777466367995|
|          5|20.005923795830288|
|          6| 11.69149459985242|
|          7|11.342811043744058|
|          8|11.184160087012579|
|          9|11.276688256829068|
|         10|11.909430324276952|
|         11|11.991532821824373|
|         12|11.864507260457405|
|         13| 11.58097226092462|
|         14|12.048847378161607|
|         15|12.755428717873595|
|         16|12.926983198280979|
|         17|13.052417266187044|
|         18|12.484358269213894|
|         19|12.241942818295133|
|         20| 13.67892590992549|
|         21| 14.87007723035953|
|         22|16.231789545248535|
|         23|17.937012815212864|
+-----------+------------------+
```

Question-4. Average fare amount compared to the average trip distance.

We want to calculate the average fare amount and average trip distance, and then determine the ratio of the average fare to the average distance for each hour of the day.

In [30]:
```
# Compute the average fare amount and average trip distance by hour of the day using SparkSQL
avg_fare_and_distance_by_hour = spark.sql("""
    SELECT
        hour_of_day,
        AVG(fare_amount) as average_fare,
        AVG(trip_distance) as average_distance,
        CASE
            WHEN AVG(trip_distance) != 0 THEN AVG(fare_amount) / AVG(trip_distance)
            ELSE NULL
        END as fare_to_distance_ratio
    FROM
        taxi_data_with_hour
    WHERE
        hour_of_day IS NOT NULL
    GROUP BY
        hour_of_day
    ORDER BY
        hour_of_day
""")

avg_fare_and_distance_by_hour.show(24)  # to display averages for all 24 hours
```

hour_of_day	average_fare	average_distance	fare_to_distance_ratio
0	18.880695641467593	5.3101957645899995	3.55555547826877
1	27.534966129760434	6.998961312659949	3.934150354561082
2	30.12912737233388	7.810856975151618	3.8573395298598507
3	35.2012607338016	28.23991608118659	1.2465072712185803
4	40.932777466367995	11.320184977578476	3.6159106540610617
5	20.005923795830288	5.961084112149523	3.3560881576986
6	11.69149459985242	3.082606829006503	3.792729740892868
7	11.342811043744058	3.5050804229092534	3.2361057879320803
8	11.184160087012579	2.5761774895262635	4.341377926203854
9	11.276688256829068	2.5608205435088713	4.403544904938007
10	11.909430324276952	2.781062038312256	4.282331771176321
11	11.991532821824373	2.738920716112535	4.378196400969379
12	11.864507260457405	8.445872173935877	1.4047699297500027
13	11.58097226092462	2.5870150728309036	4.476577033720898
14	12.048847378161607	4.129674768661324	2.9176262183153376
15	12.755428717873595	2.9969097654438523	4.256193784995216
16	12.926983198280979	3.640493437233023	3.5508876533248754
17	13.052417266187044	3.1995482014388466	4.079456362094289
18	12.484358269213894	3.1398695181533567	3.9760755015566023
19	12.241942818295133	3.167968425425843	3.8642881412713423
20	13.67892590992549	3.6439890916945665	3.7538328369595555
21	14.87007723035953	4.0594121171770965	3.6631110124143147
22	16.231789545248535	4.657468650853604	3.485109780024742
23	17.937012815212864	5.283533691608104	3.3948894550823856

Question no.5. Average fare amount and average trip distance by day of the week

To calculate the average fare amount and average trip distance by day of the week, we would need to extract the day of the week from the tpep_pickup_datetime (or pickup_timestamp we've converted it to a timestamp format).

Convert tpep_pickup_datetime to a timestamp format.
Extract the day of the week from the timestamp.
Group by the day of the week to compute the average fare and trip distance.

In [32]:
```
from pyspark.sql.functions import dayofweek, date_format

# Convert the string column to a timestamp column
df_timestamp = df.withColumn("pickup_timestamp", from_unixtime(unix_timestamp("tpep_pickup_datetime", "dd-MM-yyyy HH:mm")))

# Extract the day of the week from the new timestamp column
# we can use date_format to get the name of the day, or dayofweek to get the numeric representation (1 = Sunday, 2 = Monday, etc.)
df_with_day = df_timestamp.withColumn("day_of_week", date_format("pickup_timestamp", "EEEE"))

# Create a new temp view with the day of the week
df_with_day.createOrReplaceTempView("taxi_data_with_day")

# Now compute the average fare amount and average trip distance by day of the week using SparkSQL
avg_fare_and_distance_by_day = spark.sql("""
    SELECT
        day_of_week,
        AVG(fare_amount) as average_fare,
        AVG(trip_distance) as average_distance
    FROM
        taxi_data_with_day
    GROUP BY
        day_of_week
    ORDER BY
        CASE day_of_week
            WHEN 'Sunday' THEN 1
            WHEN 'Monday' THEN 2
            WHEN 'Tuesday' THEN 3
            WHEN 'Wednesday' THEN 4
            WHEN 'Thursday' THEN 5
            WHEN 'Friday' THEN 6
            WHEN 'Saturday' THEN 7
        END
""")

avg_fare_and_distance_by_day.show(7)  # to display averages for all days of the week
```

```
+-----------+-----------------+------------------+
|day_of_week|     average_fare|  average_distance|
+-----------+-----------------+------------------+
|     Sunday|14.686292601998977| 3.910443634278223|
|     Monday|13.409943486081076| 4.262469750235043|
|    Tuesday| 13.44987940367479| 3.202572362689536|
|  Wednesday|13.279588696533812|6.0476238081182645|
|   Thursday|13.404982738924733|3.9995484502692804|
|     Friday|13.702733958318719| 3.778188952327255|
|   Saturday|13.880926298071149|3.6022067638824393|
+-----------+-----------------+------------------+
```

Q 6 In the month of June 2020, find the zone which had maximum number of pick ups.

To solve this, we need the PULocationID from the yellow_tripdata_2020-06.csv file to determine the pick-up locations and corresponding zone from taxi+_zone_lookup.csv data which need to be read from HDFS into another dataframe. Join this new dataframe with the previously created dataframe on the PULocationID column (PULocationID in yellow_tripdata_2020-06.csv matches LocationID in taxi+_zone_lookup.csv).
Group by the zone column and count the occurrences for each zone.
Order the result in descending order and pick the first row.

In [33]:
```
# Step 1: Read the taxi+_zone_lookup.csv data
zone_lookup_path = "hdfs://localhost:9000/user/datat/input/taxi+_zone_lookup.csv"
zone_df = spark.read.csv(zone_lookup_path, header=True, inferSchema=True)

# Check the schema to confirm the structure
zone_df.printSchema()

# Step 2: Join the two dataframes
joined_df = df.join(zone_df, df.PULocationID == zone_df.LocationID, how='left')

# Step 3: Group by zone and count
zone_counts = joined_df.groupBy("Zone").count()

# Step 4: Order by count in descending order and display the first row
top_zone = zone_counts.orderBy("count", ascending=False).first()

print(f"The zone with the maximum number of pickups in June 2020 is: {top_zone['Zone']} with {top_zone['count']} pickups.")
```

```
root
 |-- LocationID: integer (nullable = true)
 |-- Borough: string (nullable = true)
 |-- Zone: string (nullable = true)
 |-- service_zone: string (nullable = true)

The zone with the maximum number of pickups in June 2020 is: Upper East Side North with 23098 pickups.
```

Q.07. In the month of June 2020, find the zone which had maximum number of drops.

To find the zone with the maximum number of drops for June 2020, we would follow a similar approach to the previous answer. However, this time we would be focusing on the DOLocationID column (DOLocationID represents the drop-off location in the yellow_tripdata_2020-06.csv dataset).

In [34]:

```
# The zone lookup data is already loaded in the previous step as zone_df

# Join the dataframe using DOLocationID for drop-offs
joined_drop_df = df.join(zone_df, df.DOLocationID == zone_df.LocationID, how='left')

# Group by Zone and count for drop-offs
drop_zone_counts = joined_drop_df.groupBy("Zone").count()

# Order by count in descending order and get the top zone
top_drop_zone = drop_zone_counts.orderBy("count", ascending=False).first()

print(f"The zone with the maximum number of drop-offs in June 2020 is: {top_drop_zone['Zone']} with {top_drop_zone['count']} drop-of
```

```
The zone with the maximum number of drop-offs in June 2020 is: Upper East Side North with 22254 drop-offs.
```

Q.08.Average no of passengers by hour of the day

To compute the average number of passengers by hour of the day, we'll use the df_with_hour dataframe from our previously provided code, which already has the hour_of_day extracted from the pickup_timestamp.

In [35]:
```
from pyspark.sql import functions as F

# Group by hour_of_day and compute average passengers
avg_passengers_per_hour = df_with_hour.groupBy("hour_of_day") \
                                       .agg(F.avg("passenger_count").alias("avg_passenger_count"))

# Order the results by hour_of_day for better readability
ordered_avg_passengers = avg_passengers_per_hour.orderBy("hour_of_day")

# Display the results
ordered_avg_passengers.show(24)
```

```
+-----------+------------------+
|hour_of_day|avg_passenger_count|
+-----------+------------------+
|          0| 1.3300081766148815|
|          1| 1.2963541666666667|
|          2|  1.308466051969824|
|          3| 1.3093270365997638|
|          4| 1.2861205915813425|
|          5| 1.3718697829716193|
|          6| 1.3303137428192664|
|          7| 1.3431017976810977|
|          8|  1.359296915388592|
|          9| 1.3521955975550306|
|         10|  1.361447777998543|
|         11| 1.3555843529624496|
|         12| 1.3567879870492352|
|         13| 1.3521634939012896|
|         14| 1.3638007863695938|
|         15| 1.3572915863345416|
|         16|   1.35842077865147|
|         17| 1.3650200560470356|
|         18|  1.376865328634901|
|         19| 1.3628078030060762|
|         20| 1.3558048103607772|
|         21| 1.3686376434914447|
|         22| 1.3518518518518519|
|         23| 1.3064166486017776|
+-----------+------------------+
```

Q.09. Total number of payments made by different type for the month.

To determine the total number of payments made by different payment types for the month, we would group by month and payment_type column and count the number of occurrences for each type.

In [37]:
```
from pyspark.sql import functions as F

# Convert the string column to a timestamp column
df_timestamp = df.withColumn("pickup_timestamp", F.from_unixtime(F.unix_timestamp("tpep_pickup_datetime", "dd-MM-yyyy HH:mm")))

# Extract the month from the timestamp column
df_month = df_timestamp.withColumn("Month", F.month("pickup_timestamp"))

# Generate the PaymentDescription column
payment_description = F.when(df_month["payment_type"] == '1', 'Credit card') \
                       .when(df_month["payment_type"] == '2', 'Cash') \
                       .when(df_month["payment_type"] == '3', 'No charge') \
                       .when(df_month["payment_type"] == '4', 'Dispute') \
                       .when(df_month["payment_type"] == '5', 'Unknown') \
                       .when(df_month["payment_type"] == '6', 'Voided trip') \
                       .otherwise('Other').alias('PaymentDescription')

# Group by month and payment_type, then aggregate
agg_df = df_month.groupBy("Month", "payment_type") \
                 .agg(F.count("*").alias("TotalCount"))

# Add the PaymentDescription column to the result
result_df = agg_df.withColumn("PaymentDescription", payment_description)

# Order the result by Month and payment_type
ordered_result = result_df.orderBy("Month", "payment_type")

ordered_result.show()
```

```
+-----+------------+----------+------------------+
|Month|payment_type|TotalCount|PaymentDescription|
+-----+------------+----------+------------------+
|    1|           2|         3|              Cash|
|    5|           1|         1|       Credit card|
|    5|           2|         3|              Cash|
|    6|        null|     50717|             Other|
|    6|           1|    322565|       Credit card|
|    6|           2|    168937|              Cash|
|    6|           3|      5245|         No charge|
|    6|           4|      2275|           Dispute|
|    6|           5|        12|           Unknown|
|    7|           1|         2|       Credit card|
+-----+------------+----------+------------------+
```

Question no.10. Configuring Hadoop cluster and Spark installation on the cluster

Before we proceed for configuration of Hadoop Cluster on windows 11 we need following softwares to be downloaded:

1.spark-3.1.2-bin-hadoop3.2.tgz

2.hadoop-3.2.2.tar.gz

3.hadoop-dependencies-3.2.2.zip (Configuration files)

4.jdk-11.0.12 if java is not there

Once the above softwares are downloaded we need to extract these files to new folder(bigdata) in C drive and follow the below steps

Step 1: Install Java Development Kit (JDK)

If Java is not already installed on our Windows 11 machine, download and install JDK 11.0.12. We can download it from the official Oracle website or use an OpenJDK distribution.

Step 2: Extract Hadoop and Spark Files

Create a new folder named "bigdata" in the C drive (C:\bigdata).
Extract the contents of the hadoop-3.2.2.tar.gz file into the C:\bigdata folder. This can be done using a tool like 7-Zip or WinRAR.
Extract the contents of the spark-3.1.2-bin-hadoop3.2.tgz file into the C:\bigdata folder as well.

Step 3: Configure Hadoop

Inside the C:\bigdata\hadoop-3.2.2 folder, the Hadoop configuration files are there. These files are used to configure the Hadoop cluster.
Copy the hadoop-dependencies-3.2.2.zip file to the C:\bigdata\hadoop-3.2.2 folder.
Extract the contents of hadoop-dependencies-3.2.2.zip into the same folder, overwriting any existing files if prompted.

Step 4: Set Environment Variables

To make Hadoop and Spark accessible from anywhere in our Windows environment, we need to set the HADOOP_HOME and SPARK_HOME environment variables. Right-click on "This PC" or "My Computer" and select "Properties."
Click on "Advanced system settings" on the left-hand side.
Click the "Environment Variables" button.
Under "System Variables," click "New" to add a new variable.
Enter "HADOOP_HOME" as the variable name and set the variable value to "C:\bigdata\hadoop-3.2.2" (the path to the Hadoop installation).
Click "OK" to save the variable.
Repeat the process to add another variable named "SPARK_HOME" with the value "C:\bigdata\spark-3.1.2-bin-hadoop3.2" (the path to your Spark installation).
Additionally, add the "bin" directories of Hadoop and Spark to your system's PATH variable.
Edit the PATH variable under "System Variables."
Add the following two entries to the PATH (make sure to replace "C:\bigdata" with the actual path to your installations):
"%HADOOP_HOME%\bin"
"%SPARK_HOME%\bin"

Step 5: Configure Hadoop and Spark

Hadoop and Spark both require configuration files to run. We can find sample configuration files in the respective installation directories (C:\bigdata \hadoop-3.2.2\etc\hadoop and C:\bigdata\spark-3.1.2-bin-hadoop3.2\conf).
We need to edit these files to match our cluster's configuration. Key files to configure include:
Hadoop: core-site.xml, hdfs-site.xml
Spark: spark-defaults.conf (for Spark properties).

Step 6: Start Hadoop Services

Open a command prompt and navigate to the Hadoop bin directory (C:\bigdata\hadoop-3.2.2\bin).
Start Hadoop services by running the following command:
start-dfs
start-yarn

Step 7: Verify Hadoop Cluster

Open a web browser and visit the Hadoop Resource Manager's web interface at http://localhost:8088/ to verify that the Hadoop cluster is running correctly.

Step 8: Start Spark

Open a command prompt and navigate to the Spark bin directory (C:\bigdata\spark-3.1.2-bin-hadoop3.2\bin).
Start Spark by running the following command:
spark-shell

Note: I am running pyspark in jupyter anaconda

Step 9: Verify Spark

We can run Spark commands and applications to verify that Spark is working correctly.

C:\Users\datat>jps
12432 SparkSubmit
24272 DataNode
26208 Jps
24488 NameNode
8392 ResourceManager

In [ ]: