## Logic For First Submission

**<Properly explain the code, list the steps to run the code provided by you and attach screenshots of code execution>**

**Note:** Be as descriptive as possible.

**Task 1: Write a job to consume clickstream data from Kafka and ingest to Hadoop**

create a python file (spark\_kafka\_to\_local.py) the code which will ingest the relevant data from Kafka into hadoop. vi spark\_kafka\_to\_local.py

**spark\_kafka\_to\_local.py**

1. **Importing the files:**

from pyspark.sql import SparkSession

from pyspark.sql.functions import \*

from pyspark.sql.types import \*

1. **Establishing Spark Session**

Here, spark = SparkSession.builder.appName("KafkaRead").getOrCreate() creates a SparkSession and gives it a name of "KafkaRead". If a SparkSession with that name already exists, it will be reused.

spark.sparkContext.setLogLevel('ERROR') sets the logging level for the SparkContext to ERROR.This means that only log messages with a severity of ERROR or higher will be recorded. By setting the logging level to ERROR, only error messages will be recorded, which can help reduce the amount of output produced by the application and make it easier to identify and troubleshoot issues.

spark = SparkSession.builder.appName("KafkaRead").getOrCreate()

spark.sparkContext.setLogLevel('ERROR')

1. **Reading data from Kafka Server & Topic given**

lines = spark.readStream.format("kafka") \

.option("kafka.bootstrap.servers","18.211.252.152:9092") \

.option("subscribe","de-capstone3") \

.option("failOnDataLoss","false") \

.option("startingOffsets", "earliest") \

.load()

This PySpark code reads a data stream from a Kafka topic. The code uses the readStream method of the SparkSession object, spark, and sets the format to "kafka" to indicate that the data source is a Kafka topic. The options set for reading the data are:

* kafka.bootstrap.servers: This sets the address and port of the Kafka bootstrap server, in this case 18.211.252.152:9092
* subscribe: This sets the name of the Kafka topic to subscribe to, in this case de-capstone3
* failOnDataLoss: This sets whether the stream processing should fail if data loss is detected, in this case false
* startingOffsets: This sets the starting offset position for reading the data, in this case earliest

Finally, the load() method is called to load the data stream into Spark.

1. **Casting raw data as string and aliasing**

kafkaDF = lines.selectExpr("cast(key as string)","cast(value as string)")

A new DataFrame, kafkaDF, from the input DataFrame lines, which was loaded from the Kafka topic in the previous code. The selectExpr method is used to cast the "key" and "value" columns of the lines DataFrame to strings and select them as new columns in the kafkaDF DataFrame.

The resulting kafkaDF DataFrame will have two columns: "key" as string and "value" as string

1. **Wrting kafka data into json file**

output = kafkaDF \

.writeStream \

.outputMode("append") \

.format("json") \

.option("truncate", "false") \

.option("path","/user/hadoop/clickStreamData/") \

.option("checkpointLocation", "/user/hadoop/clickstream\_checkpoint/") \

.start()

This line of code writes the kafkaDF DataFrame to an output location. The code uses the writeStream method of the kafkaDF DataFrame and sets various options for writing the data

outputMode: This sets the output mode for the data, in this case "append

format: This sets the format of the output data, in this case "json

option("truncate", "false"): This sets whether the output files

option("path","/user/hadoop/clickStreamData/"): This sets the path to the output location, in this case "/user/hadoop/clickStreamData/

option("checkpointLocation,

/user/hadoop/clickstream\_checkpoint/"): This sets the location of the checkpoint data, which is used to keep track of the progress of the write stream and recover from failures, in this case /user/hadoop/clickstream\_checkpoint/

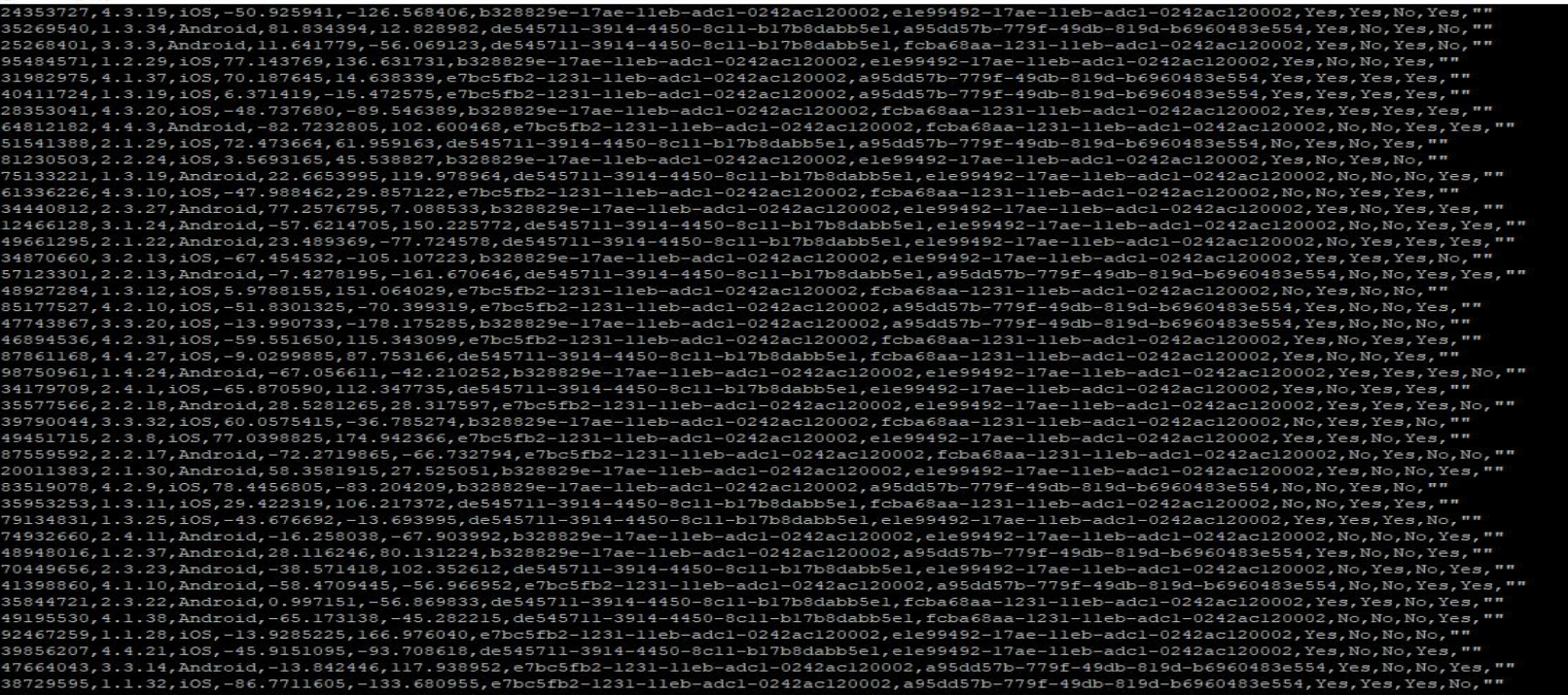
The start() method at the end starts the write stream and begins writing the data to the specified output location

1. **output.awaitTermination()**

This line of code blocks the execution of the code until the write stream, represented by the output object, terminates. The awaitTermination method waits indefinitely until the write stream is finished. This is used to ensure that all the data has been written before the program exits.

**Run Spark Submit command, to ingest the relevant data from Kafka into hadoop**

spark-submit --packages org.apache.spark:spark-sql-kafka-0-10\_2.11:2.4.5 spark\_kafka\_to\_local.py 18.211.252.152 9092 de-capstone3



**Create another file spark\_local\_flatter**

**spark\_local\_flatten.py**

1. **Importing the data**

from pyspark.sql import SparkSession

from pyspark.sql import functions as F

from pyspark.sql.functions import col

from pyspark.sql.types import \*

1. **Establishing a spark connection**

spark=SparkSession \

.builder \

.appName('transformKafkaData') \

.master('yarn') \

.getOrCreate()

This code creates a Spark session with the name "transformKafkaData" and runs it on a YARN cluster. The SparkSession is the entry point to using Spark SQL and creating DataFrames, and it is used to manage the connection to a Spark cluster.

1. **Reading a data from json file extracted from kafka server**

df=spark.read.json('/user/hadoop/clickStreamData/')

This code reads a JSON file located at "/user/hadoop/clickStreamData/" and creates a Spark DataFrame object named "df". The DataFrame represents the data in a tabular form, similar to a table in a relational database, and provides a way to perform operations on the data using Spark's APIs

1. **To flatten the raw data store into respective columns in a dataframe**

flatten\_df=df.withColumn("value", F.split(F.regexp\_replace(F.regexp\_replace((F.regexp\_replace("value",'\{|}',"")),'\:',','),'\"|"',"").cast("string"),','))\

.withColumn("customer\_id", F.element\_at("value",2))\

.withColumn("app\_version", F.element\_at("value",4))\

.withColumn("OS\_version",F.element\_at("value",6))\

.withColumn("lat",F.element\_at("value",8))\

.withColumn("lon", F.element\_at("value",10))\

.withColumn("page\_id", F.element\_at("value",12))\

.withColumn("button\_id",F.element\_at("value",14))\

.withColumn("is\_button\_click",F.element\_at("value",16))\

.withColumn("is\_page\_view",F.element\_at("value",18))\

.withColumn("is\_scroll\_up",F.element\_at("value",20))\

.withColumn("is\_scroll\_down",F.element\_at("value",22))\

.withColumn("date\_hour",F.element\_at("value",24))\

.withColumn("minutes",F.element\_at("value",25))\

.withColumn("seconds",F.element\_at("value",26))\

.drop("value")

This code performs several transformations on the input DataFrame "df" and creates a new DataFrame "flatten\_df". The following steps are performed:

* The "value" column is transformed using a combination of functions including F.split, F.regexp\_replace, and F.element\_at to extract various values such as "customer\_id", "app\_version", "OS\_version", "lat", "lon", etc
* New columns are created based on the extracted values and are named as "customer\_id", "app\_version", "OS\_version", "lat", "lon", etc
* The original "value" column is dropped from the DataFrame

This operation results in a flattened DataFrame with individual columns for each extracted value

1. **To concatenate date\_hour, minutes and seconds column to make it into timestamp format**

flatten\_df=flatten\_df.select("\*",F.concat(col("date\_hour"),F.lit(":"),col("minutes"),F.lit(":"),col("seconds")).alias("timestamp"))

This code modifies the "flatten\_df" DataFrame by adding a new column named "timestamp". The new column is created using the F.concat function to concatenate the values of the existing "date\_hour", "minutes", and "seconds" columns with a ":" separator. The resulting string is stored in the new "timestamp" column

1. **To remove extra characters \n from timestamp column**

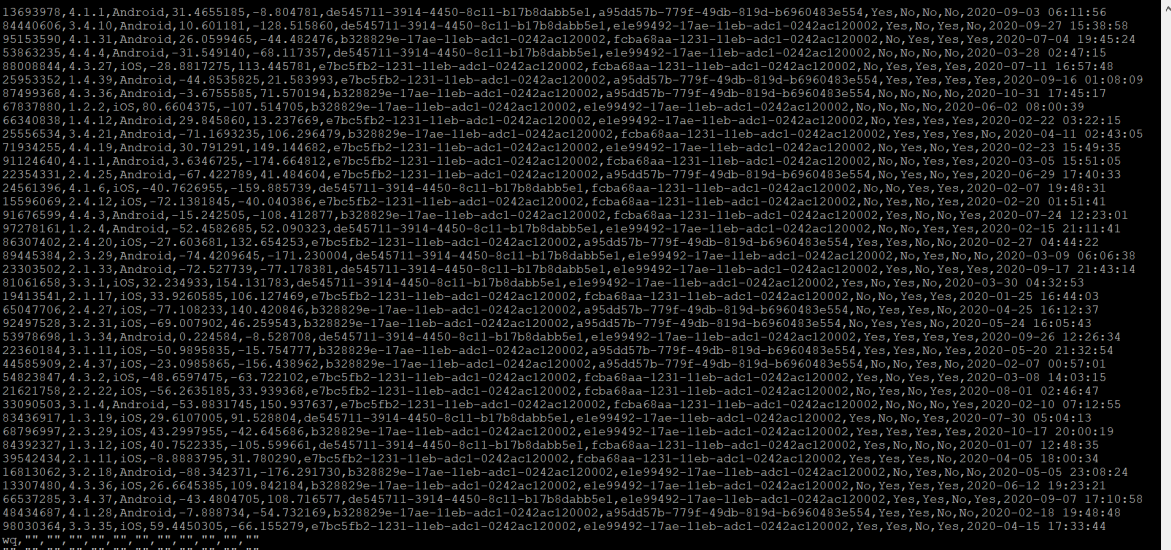
flatten\_df = flatten\_df.select("\*").withColumn("timestamp",F.expr("substring(timestamp, 1, length(timestamp)-2)")).drop("date\_hour").drop("minutes").drop("seconds")

The code is using the select method to select all columns of the flatten\_df DataFrame. Then, it's using the withColumn method to add a new column called "timestamp" with the value obtained by applying the expression "substring(timestamp, 1, length(timestamp)-2)". Finally, it's using the drop method twice to remove the "date\_hour" and "minutes" columns

1. **To write the flattened dataframe in csv file**

flatten\_df.write.option("header","true").csv('/user/hadoop/clickStream\_flatten\_data/')

This code writes the flatten\_df DataFrame to a CSV file located at "/user/hadoop/clickStream\_flatten\_data/". It uses the write method and sets the option "header" to "true", indicating that the first row of the file should contain the column names



Task 2: To write a script to ingest the relevant bookings data from AWS RDS to Hadoop

1. we need to setup MySQL Connector on AWS EMR

a. Run the following command

wget https://de-mysql-connector.s3.amazonaws.com/mysql

b. Run the following step to extract the MySQL connector tar file

tar -xvf mysql-connector-java

c. go to the MySQL Connector directory and then copy it to the Sqoop library to complete the installation.

cd mysql-connector-java-8.0.25/

sudo cp mysql-connector-java \

Imported data from AWS RDS to Hadoop using command:

sqoop import \

--connect jdbc:mysql://upgraddetest.cyaielc9bmnf.us

--table bookings \

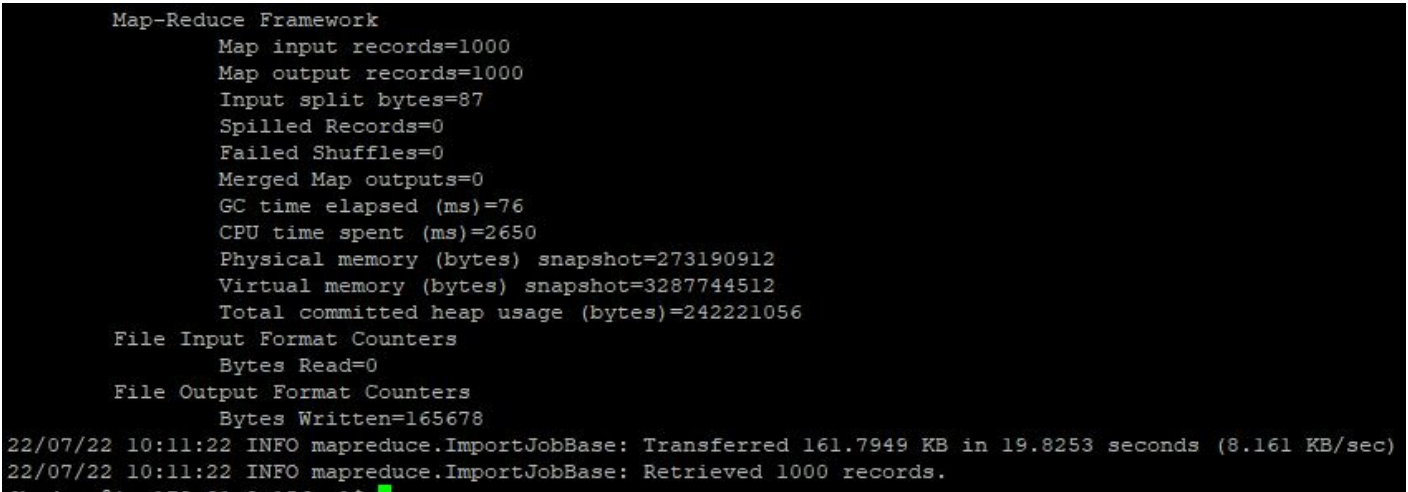
--username student

--password STUDENT123

--null-string '\\N' --null-non-string '\\N'

--target-dir bookings \

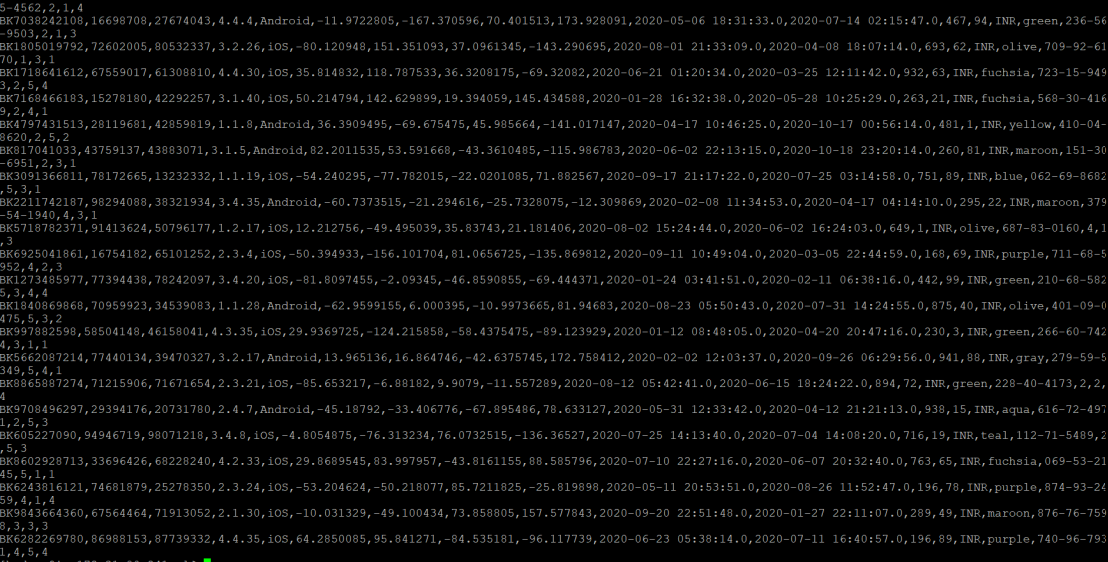
-m 1



3. Use this command to check the imported data

hadoop fs -ls bookings

hadoop fs -cat bookings/part-m-00000



**Task 3: To create aggregates for finding date-wise total bookings using the Spark script**

**Creating a python file.**

1. **Importing the libraries**

from pyspark.sql import SparkSession

from pyspark.sql import functions as F

from pyspark.sql.functions import col

from pyspark.sql.types import \*

1. **Establishing spark connection**

spark = SparkSession \

.builder \

.appName('aggregateBatchData') \

.master('yarn') \

.getOrCreate()

This code creates a Spark session object for a Spark application named "aggregateBatchData". The .master('yarn') argument specifies that the Spark application should run on a YARN cluster. The Spark session object is used to interact with Spark and perform various Spark operations such as reading data, transforming data, and saving data. The .getOrCreate() method at the end retrieves an existing Spark session with the same configuration if it exists, otherwise it creates a new Spark session

1. **To read data from csv file extracted from AWS RDS and stored in HDFS**

df=spark.read.csv('/user/hadoop/bookings-data/',header=False,inferSchema = True)

The code is reading a CSV file located at "/user/hadoop/bookings-data/" using Apache Spark's read method and storing the data in a dataframe named "df". The header parameter is set to False, meaning the first row of the file is not treated as a header row. The inferSchema parameter is set to True, which means Spark will attempt to infer the schema of the data based on the data itself.

**4.To add column headers according to given data**

new\_columns = ["booking\_id","customer\_id","driver\_id","customer\_app\_version","customer\_phone\_os\_version","pickup\_lat",

"pickup\_lon","drop\_lat","drop\_lon","pickup\_timestamp","drop\_timestamp","trip\_fare","tip\_amount","currency\_code",

"cab\_color","cab\_registration\_no","customer\_rating\_by\_driver","rating\_by\_customer","passenger\_count"]

new\_df = df.toDF(\*new\_columns)

The code is defining a new list of column names "new\_columns" which contains the names for the columns in the dataframe "df". The "toDF" method of "df" is then used to create a new dataframe "new\_df" with the specified column names. The "\*new\_columns" syntax passes the elements of the "new\_columns" list as separate arguments to the "toDF" method, allowing the column names to be dynamically set based on the elements in the list.

**5. To create a new column with date extracted from pickup\_timestamp column**

new\_df = new\_df.withColumn("date", F.to\_date(F.col("pickup\_timestamp")))

The code creates a new column "date" in the dataframe "new\_df" by using the withColumn method and passing a column expression using the F.to\_date method from the pyspark.sql.functions library. The F.to\_date method converts the "pickup\_timestamp" column to a date format, and the F.col method accesses the "pickup\_timestamp" column in the dataframe. The resulting dataframe will have a new column "date" with the date extracted from the "pickup\_timestamp" column

**6. To get the datewise bookings aggregate**

aggregate\_df = new\_df.groupby('date').count()

The code creates a new dataframe "aggregate\_df" by grouping the data in the "new\_df" dataframe by the "date" column and aggregating the data using the count function. The groupby method is used to group the data by the "date" column, and the count method is used to count the number of rows in each group. The resulting dataframe will have two columns: "date" and "count". The "date" column will contain the unique dates from the "date" column in the original dataframe, and the "count" column will contain the number of rows in each group (i.e., the number of rows with the same date).

**7. To write the resultant dataframe in csv files in HDFS**

aggregate\_df.write.csv('/user/hadoop/bookings\_aggregate\_data/')

The code writes the data in the "aggregate\_df" dataframe to a CSV file located at "/user/hadoop/bookings\_aggregate\_data/". The write method is used to write the data to the file and the csv method is used to specify that the file format is CSV. This will create a new CSV file with the data in the "aggregate\_df" dataframe and save it at the specified file path.

Run this command : **spark-submit --packages org.apache.spark:spark-sql-kafka-0-10\_2.11:2.4.5 datewise\_bookings\_aggregates\_spark.py**

1. Make a directory using mkdir command

**hadoop fs -mkdir datewise\_aggregated\_data**

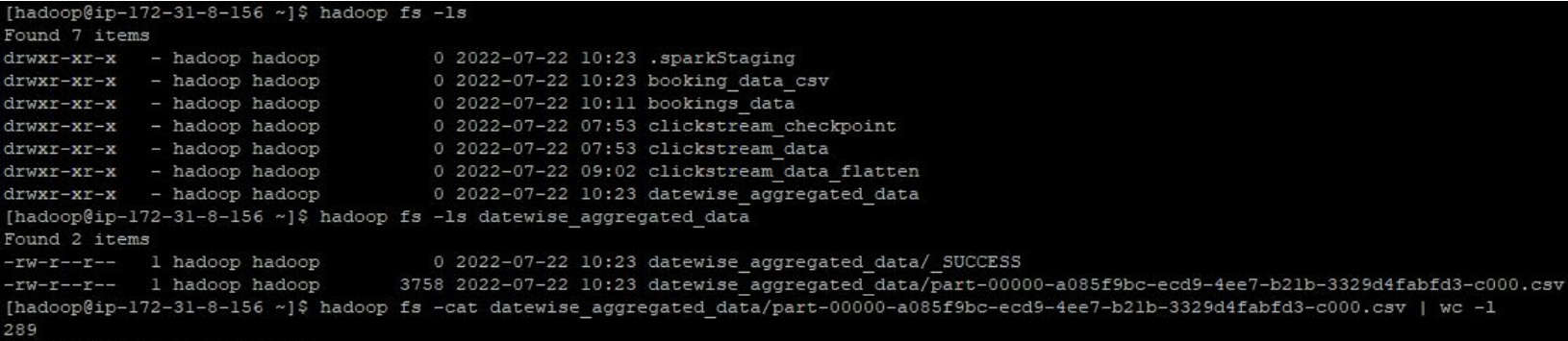
2. Loading the data from local file system to hadoop file system

**hadoop fs- put ~/ datewise\_aggregated\_data datewise\_aggregated\_data**

3. Checking the data file in hadoop

**hadoop fs -ls datewise\_aggregated\_data**

**hadoop fs -cat datewise\_aggregated\_data / part-00000-a085f9bc-ecd9-4ee7-b21b3329d4fabfd3-c000.csv | wc -l**



**Task 4: To create a Hive-managed table for clickstream data, bookings data and aggregated data:**

1. create database if not exists cab\_rides\_data;
2. use cab\_rides\_data;
3. Command to create clickStreamData table and load data from HDFS:

create table if not exists clickStreamData(

customer\_id int,

app\_version string,

os\_version string,

lat double,

lon double,

page\_id string,

button\_id string,

is\_button\_click string,

is\_page\_view string,

is\_scroll\_up string,

is\_scroll\_down string,

`timestamp` timestamp)

row format delimited fields terminated by ',' lines

terminated by '\n' stored as textfile

tblproperties("skip.header.line.count"="1");

1. create table if not exists bookingsData(

booking\_id string,

customer\_id int,

driver\_id int,

customer\_app\_version string,

customer\_phone\_os\_version string,

pickup\_lat double,

pickup\_lon double,

drop\_lat double,

drop\_lon double,

pickup\_timestamp timestamp,

drop\_timestamp timestamp,

trip\_fare double,

tip\_amount double,

currency\_code string,

cab\_color string,

cab\_registration\_no string,

customer\_rating\_by\_driver int,

rating\_by\_customer int,

passenger\_count int)

row format delimited fields terminated by ',' lines

terminated by '\n' stored as textfile;

1. create table if not exists testAggregateData( `date` string, no\_of\_bookings int)

row format delimited fields terminated by ',' lines terminated by '\n' stored as textfile;

**<Command to load the data into Hive tables>**

1.load data inpath '/user/hadoop/clickStream\_flatten\_data/' into table clickStreamData;

2. load data inpath '/user/hadoop/bookings-data/' into table bookingsData;

3. load data inpath '/user/hadoop/bookings\_aggregate\_data/' into table testAggregateData;