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. Make sure to put this file in Hadoop local file system from your computer using scp -I commands. We will need do the same for all Spark jobs python files in the upcoming sections

spark_kafka_to_local.py

1. **Importing the files:**

```
from pyspark.sql import SparkSession

from pyspark.sql.functions import *

from pyspark.sql.types import *
```

2. **Establishing Spark Session**

The statement spark = SparkSession.builder.appName("KafkaRead").getOrCreate() creates a new SparkSession and assigns it a name of "KafkaRead". If a SparkSession with the same name already exists, the existing one will be reused instead of creating a new one.

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

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

spark.sparkContext.setLogLevel('ERROR')
```

3. 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()
```

The PySpark code presented here uses the readStream method of the SparkSession object, spark, to read data from a Kafka topic. The format is set to "kafka" to indicate that the data source is a Kafka topic. Various options are set to read the data, such as the address and port of the Kafka bootstrap server, the name of the Kafka topic to subscribe to, whether stream processing should fail if data loss is detected, and the starting offset position for reading the data. The load() method is then used to load the data stream into Spark.

4. Casting raw data as string and aliasing

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

In this PySpark code, a new DataFrame named kafkaDF is created based on the input DataFrame called "lines". The lines DataFrame was previously loaded from a Kafka topic using the readStream method. The selectExpr method is applied to the lines DataFrame to cast the "key" and "value" columns as strings and select them as new columns in the kafkaDF DataFrame.

As a result, the kafkaDF DataFrame will contain two columns - "key" and "value" - both of which will be of string data type.

5. 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()
```

The following code writes the kafkaDF DataFrame to a specified output location. It uses the writeStream method of the kafkaDF DataFrame and sets various options for writing the data, including the output mode, output format, and the path to the output location. It also sets the checkpoint location, which is used to track the progress of the write stream and recover from failures. Finally, the start() method is called to start the write stream and write the data to the specified output location.

The outputMode option specifies the output mode for the data, which in this case is "append". The format option sets the format of the output data to "json". The option("truncate", "false") sets whether the output files should be truncated if they already exist. The option("path","/user/hadoop/clickStreamData/") sets the path to the output location, which is "/user/hadoop/clickStreamData/". The option("checkpointLocation", "/user/hadoop/clickstream_checkpoint/") sets the location of the checkpoint data, which is used to keep track of the progress of the write stream and recover from failures. Finally, the start() method is called to start the write stream and begin writing the data to the specified output location.

6. **output.awaitTermination()**

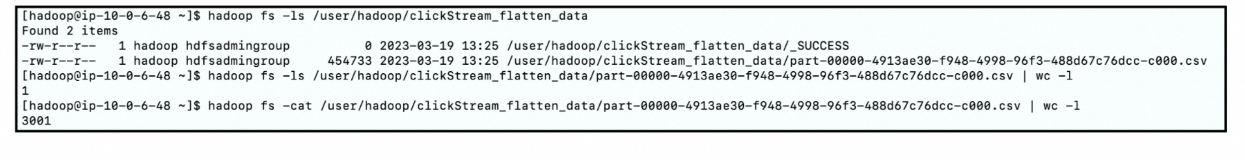
   This line of code blocks the execution of the code until the write stream, represented by the output object, terminates. The awaitTermination method waits indefinitely until the write stream is finished. This is used to ensure that all the data has been written before the program exits.

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
```

For checking the output use the below command

```
hadoop fs -cat /user/hadoop/clickStream_flatten_data/part-00000-4913ae30-
f948-4998-96f3-488d67c76dcc-c000.csv | wc -l
```



```
[hadoop@ip-10-0-6-48 ~]$ hadoop fs -ls /user/hadoop/clickStream_flatten_data
Found 2 items
-rw-r--r--   1 hadoop hdfsadmingroup          0 2023-03-19 13:25 /user/hadoop/clickStream_flatten_data/_SUCCESS
-rw-r--r--   1 hadoop hdfsadmingroup     454733 2023-03-19 13:25 /user/hadoop/clickStream_flatten_data/part-00000-4913ae30-f948-4998-96f3-488d67c76dcc-c000.csv
[hadoop@ip-10-0-6-48 ~]$ hadoop fs -ls /user/hadoop/clickStream_flatten_data/part-00000-4913ae30-f948-4998-96f3-488d67c76dcc-c000.csv | wc -l
1
[hadoop@ip-10-0-6-48 ~]$ hadoop fs -cat /user/hadoop/clickStream_flatten_data/part-00000-4913ae30-f948-4998-96f3-488d67c76dcc-c000.csv | wc -l
3001
```

Create another file spark_local_flatter
spark_local_flatten.py (Make sure the load this file from your local computer to Hadoop using scp -I command)

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 *
```

2. **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.

3. **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

4. **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

5. **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

6. To remove extra characters \n from timestamp column

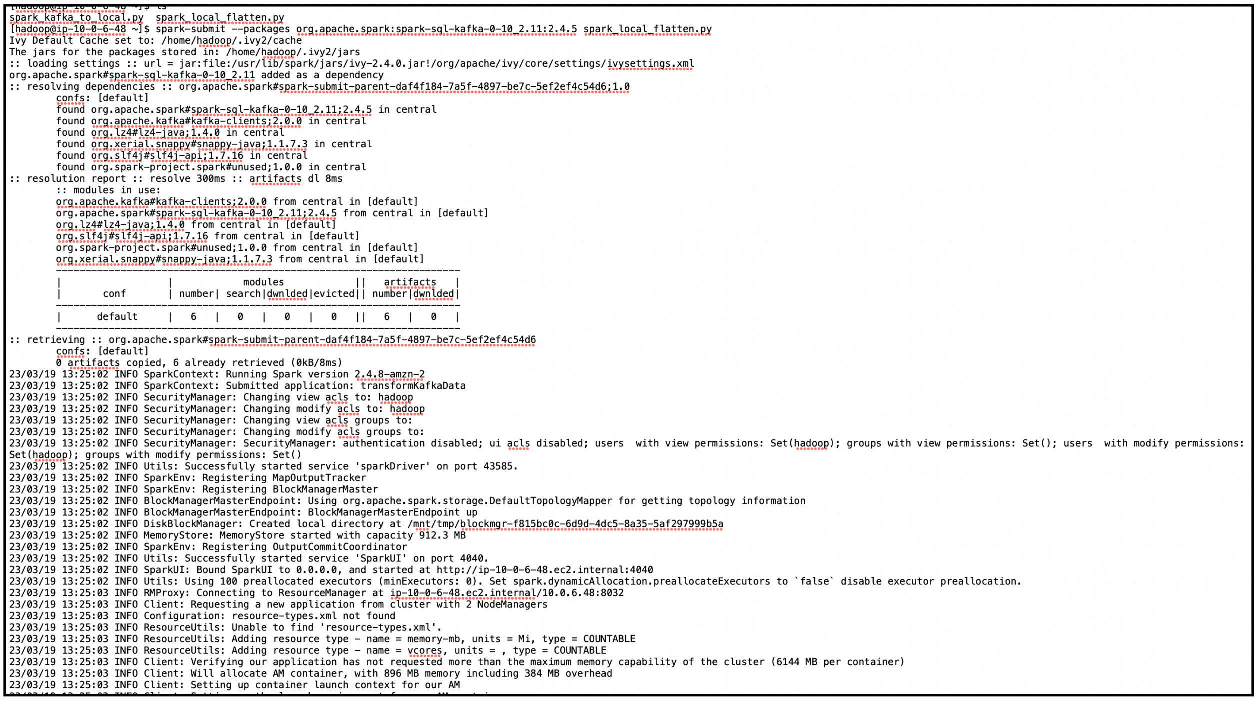
```
flatten_df = flatten_df.select("*").withColumn("timestamp",F.expr("substring(timestamp, 1, length(timestamp)-2)")).drop("date_hour").drop("minutes").drop("seconds")
```

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

7. 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



```
[hadoop@ip-10-0-6-48 ~]$ ls
spark_kafka_to_local.py  spark_local_flatten.py
[hadoop@ip-10-0-6-48 ~]$ spark-submit --packages org.apache.spark:spark-sql-kafka-0-10_2.11:2.4.5 spark_local_flatten.py
Ivy Default Cache set to: /home/hadoop/.ivy2/cache
The jars for the packages stored in: /home/hadoop/.ivy2/jars
:: loading settings :: url = jar:file:/usr/lib/spark/jars/ivy-2.4.0.jar!/org/apache/ivy/core/settings/ivysettings.xml
org.apache.spark#spark-sql-kafka-0-10_2.11 added as a dependency
:: resolving dependencies :: org.apache.spark#spark-submit-parent-daf4f184-7a5f-4897-be7c-5ef2ef4c54d6;1.0
	confs: [default]
	found org.apache.spark#spark-sql-kafka-0-10_2.11;2.4.5 in central
	found org.apache.kafka#kafka-clients;2.0.0 in central
	found org.lz4#lz4-java;1.4.0 in central
	found org.xerial.snappy#snappy-java;1.1.7.3 in central
	found org.slf4j#slf4j-api;1.7.16 in central
	found org.spark-project.spark#unused;1.0.0 in central
:: resolution report :: resolve 300ms :: artifacts dl 8ms
	:: modules in use:
	org.apache.kafka#kafka-clients;2.0.0 from central in [default]
	org.apache.spark#spark-sql-kafka-0-10_2.11;2.4.5 from central in [default]
	org.lz4#lz4-java;1.4.0 from central in [default]
	org.slf4j#slf4j-api;1.7.16 from central in [default]
	org.spark-project.spark#unused;1.0.0 from central in [default]
	org.xerial.snappy#snappy-java;1.1.7.3 from central in [default]
	---------------------------------------------------------------------
	|                  |            modules            ||   artifacts   |
	|       conf       | number| search|dwnlded|evicted|| number|dwnlded|
	---------------------------------------------------------------------
	|      default     |   6   |   0   |   0   |   0   ||   6   |   0   |
	---------------------------------------------------------------------
:: retrieving :: org.apache.spark#spark-submit-parent-daf4f184-7a5f-4897-be7c-5ef2ef4c54d6
	confs: [default]
	0 artifacts copied, 6 already retrieved (0kB/8ms)
23/03/19 13:25:02 INFO SparkContext: Running Spark version 2.4.8-amzn-2
23/03/19 13:25:02 INFO SparkContext: Submitted application: transformKafkaData
23/03/19 13:25:02 INFO SecurityManager: Changing view acls to: hadoop
23/03/19 13:25:02 INFO SecurityManager: Changing modify acls to: hadoop
23/03/19 13:25:02 INFO SecurityManager: Changing view acls groups to:
23/03/19 13:25:02 INFO SecurityManager: Changing modify acls groups to:
23/03/19 13:25:02 INFO SecurityManager: SecurityManager: authentication disabled; ui acls disabled; users  with view permissions: Set(hadoop); groups with view permissions: Set(); users  with modify permissions: Set(hadoop); groups with modify permissions: Set()
23/03/19 13:25:02 INFO Utils: Successfully started service 'sparkDriver' on port 43585.
23/03/19 13:25:02 INFO SparkEnv: Registering MapOutputTracker
23/03/19 13:25:02 INFO SparkEnv: Registering BlockManagerMaster
23/03/19 13:25:02 INFO BlockManagerMasterEndpoint: Using org.apache.spark.storage.DefaultTopologyMapper for getting topology information
23/03/19 13:25:02 INFO BlockManagerMasterEndpoint: BlockManagerMasterEndpoint up
23/03/19 13:25:02 INFO DiskBlockManager: Created local directory at /mnt/tmp/blockmgr-f815bc0c-6d9d-4dc5-8a35-5af297999b5a
23/03/19 13:25:02 INFO MemoryStore: MemoryStore started with capacity 912.3 MB
23/03/19 13:25:02 INFO SparkEnv: Registering OutputCommitCoordinator
23/03/19 13:25:02 INFO Utils: Successfully started service 'SparkUI' on port 4040.
23/03/19 13:25:02 INFO SparkUI: Bound SparkUI to 0.0.0.0, and started at http://ip-10-0-6-48.ec2.internal:4040
23/03/19 13:25:02 INFO Utils: Using 100 preallocated executors (minExecutors: 0). Set spark.dynamicAllocation.preallocateExecutors to `false` disable executor preallocation.
23/03/19 13:25:03 INFO RMProxy: Connecting to ResourceManager at ip-10-0-6-48.ec2.internal/10.0.6.48:8032
23/03/19 13:25:03 INFO Client: Requesting a new application from cluster with 2 NodeManagers
23/03/19 13:25:03 INFO Configuration: resource-types.xml not found
23/03/19 13:25:03 INFO ResourceUtils: Unable to find 'resource-types.xml'.
23/03/19 13:25:03 INFO ResourceUtils: Adding resource type - name = memory-mb, units = Mi, type = COUNTABLE
23/03/19 13:25:03 INFO ResourceUtils: Adding resource type - name = vcores, units = , type = COUNTABLE
23/03/19 13:25:03 INFO Client: Verifying our application has not requested more than the maximum memory capability of the cluster (6144 MB per container)
23/03/19 13:25:03 INFO Client: Will allocate AM container, with 896 MB memory including 384 MB overhead
23/03/19 13:25:03 INFO Client: Setting up container launch context for our AM
```

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
```

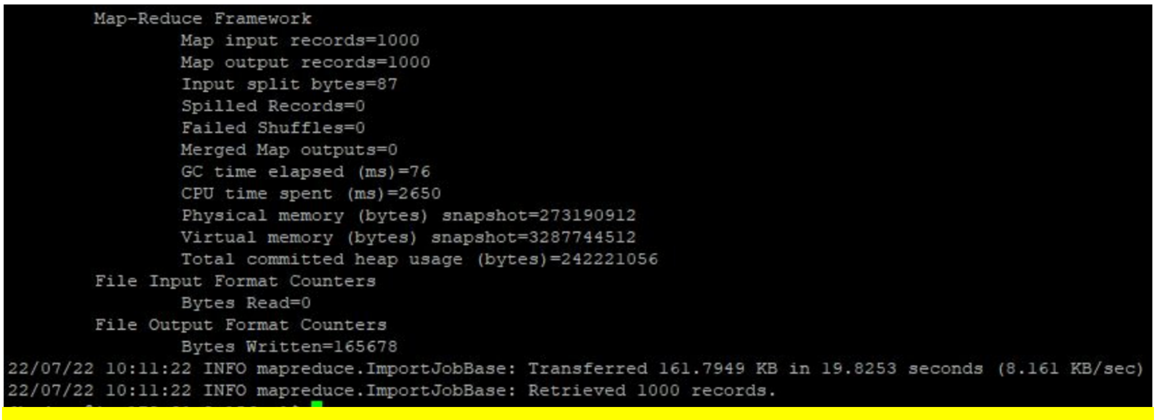
   c. go to the MySQL Connector directory and then copy it to the Sqoop library to complete the installation.

```
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-east-1.rds.amazonaws.com/
testdatabase \
> --table bookings \
> --username student \
> --password STUDENT123 \
> --target-dir /user/root/project/bookings \
> -m1
```



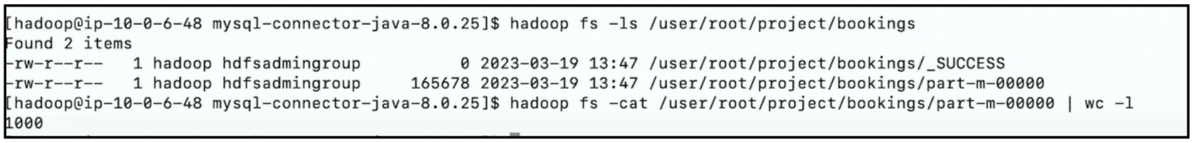
```
        Map-Reduce Framework
                Map input records=1000
                Map output records=1000
                Input split bytes=87
                Spilled Records=0
                Failed Shuffles=0
                Merged Map outputs=0
                GC time elapsed (ms)=76
                CPU time spent (ms)=2650
                Physical memory (bytes) snapshot=273190912
                Virtual memory (bytes) snapshot=3287744512
                Total committed heap usage (bytes)=242221056
        File Input Format Counters
                Bytes Read=0
        File Output Format Counters
                Bytes Written=165678
22/07/22 10:11:22 INFO mapreduce.ImportJobBase: Transferred 161.7949 KB in 19.8253 seconds (8.161 KB/sec)
22/07/22 10:11:22 INFO mapreduce.ImportJobBase: Retrieved 1000 records.
```

<Command to view the imported data>

hadoop fs -ls /user/root/project/bookings

hadoop fs -cat /user/root/project/bookings/part-m-00000 | wc -l

<Screenshot of the data>



```
[hadoop@ip-10-0-6-48 mysql-connector-java-8.0.25]$ hadoop fs -ls /user/root/project/bookings
Found 2 items
-rw-r--r--   1 hadoop hdfsadmingroup          0 2023-03-19 13:47 /user/root/project/bookings/_SUCCESS
-rw-r--r--   1 hadoop hdfsadmingroup     165678 2023-03-19 13:47 /user/root/project/bookings/part-m-00000
[hadoop@ip-10-0-6-48 mysql-connector-java-8.0.25]$ hadoop fs -cat /user/root/project/bookings/part-m-00000 | wc -l
1000
```

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 *
```

2. **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

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

```
df=spark.read.csv('/user/root/project/bookings/part-m-00000',header=False,inferSchema = True)
```

The code is reading a CSV file located at "/user/root/project/bookings/part-m-00000'" 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/')
```

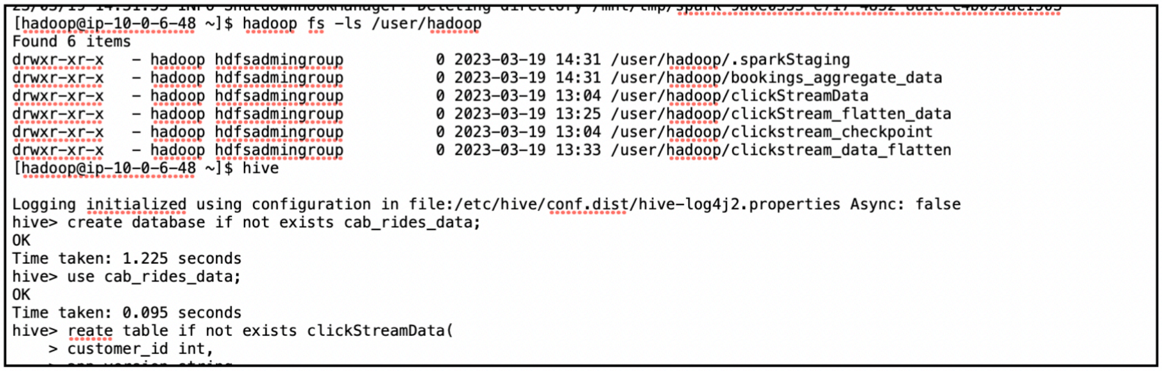
The code writes the data in the "aggregate_df" dataframe to a CSV file located at "/user/hadoop/bookings_aggregate_data/". The write method is used to write the data to the file and the csv method is used to specify that the file format is CSV. This will create a new CSV file with the data in the "aggregate_df" 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**

3. Checking the data file in hadoop

hadoop fs -ls /user/hadoop

<Screenshot of the file in HDFS>



```
[hadoop@ip-10-0-6-48 ~]$ hadoop fs -ls /user/hadoop
Found 6 items
drwxr-xr-x   - hadoop hdfsadmingroup          0 2023-03-19 14:31 /user/hadoop/.sparkStaging
drwxr-xr-x   - hadoop hdfsadmingroup          0 2023-03-19 14:31 /user/hadoop/bookings_aggregate_data
drwxr-xr-x   - hadoop hdfsadmingroup          0 2023-03-19 13:04 /user/hadoop/clickStreamData
drwxr-xr-x   - hadoop hdfsadmingroup          0 2023-03-19 13:25 /user/hadoop/clickStream_flatten_data
drwxr-xr-x   - hadoop hdfsadmingroup          0 2023-03-19 13:04 /user/hadoop/clickstream_checkpoint
drwxr-xr-x   - hadoop hdfsadmingroup          0 2023-03-19 13:33 /user/hadoop/clickstream_data_flatten
[hadoop@ip-10-0-6-48 ~]$ hive

Logging initialized using configuration in file:/etc/hive/conf.dist/hive-log4j2.properties Async: false
hive> create database if not exists cab_rides_data;
OK
Time taken: 1.225 seconds
hive> use cab_rides_data;
OK
Time taken: 0.095 seconds
hive> reate table if not exists clickStreamData(
    > customer_id int,
```

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");
```

4. 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;
```

5. 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;

After this follow the Run the HIVE queries as explained in Queries.pdf file