Logic For First Submission

Step1: Writing of job to consume clickstream data from Kafka and ingest to Hadoop 1

1.1 Created a python file for the code which will ingest the relevant data from Kafka into hadoop.

vi spark_kafka_to_local.py

1.2 Explaining the code in spark_kafka_to_local.py

- Necessary imports

```
import os
import sys

//Environment setup
os.environ["PYSPARK_PYTHON"] = "/opt/cloudera/parcels/Anaconda/bin/python"
os.environ["JAVA_HOME"] = "/usr/java/jdk1.8.0_161/jre"
os.environ["SPARK_HOME"] =
"/opt/cloudera/parcels/SPARK2-2.3.0.cloudera2-1.cdh5.13.3.p0.316101/lib/spark2/"
os.environ["PYLIB"] = os.environ["SPARK_HOME"] + "/python/lib"
sys.path.insert(0, os.environ["PYLIB"] + "/py4j-0.10.6-src.zip")
sys.path.insert(0, os.environ["PYLIB"] + "/pyspark.zip")

//Selective Imports
from pyspark.sql import SparkSession
from pyspark.sql.functions import *
```

- Creating a Spark Session for performing actions like reading of data.

```
spark = SparkSession \
    .builder \
    .appName("Kafka-to-local") \
    .getOrCreate()
```

- Reading of data from Kafka topic into a data frame to make an action ready data.

```
df = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "18.211.252.152:9092") \
    .option("startingOffsets", "earliest") \
    .option("subscribe", "de-capstone3") \
    .load()
```

- Dropping irrelevant columns from the data frame and make data more relevant to the use case.

```
df= df \
    .withColumn('value_str',df['value'].cast('string').alias('key_str')).drop('value') \
    .drop('key','topic','partition','offset','timestamp','timestampType')
```

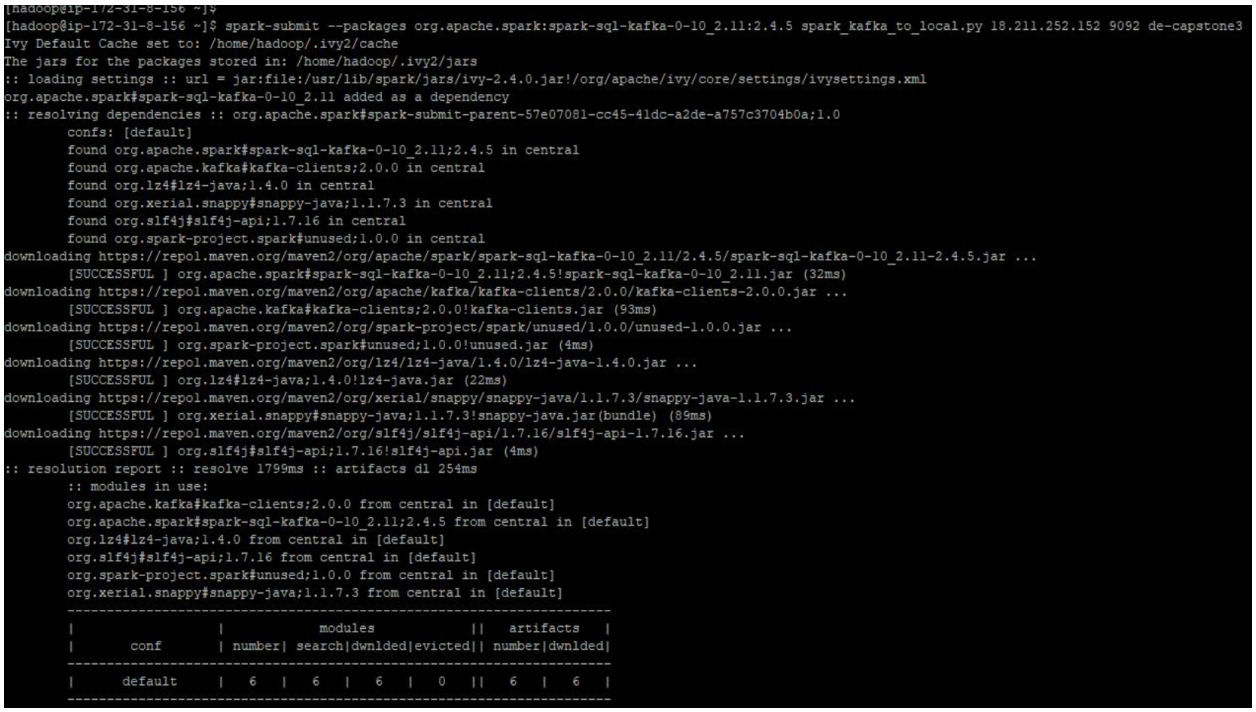
- Writing data from Kafka topic to HDFS for more actions to be performed, using awaitTermination to protect and keep running until terminated.

```
df.writeStream \
  .format("json") \
  .outputMode("append") \
  .option("path", "/user/root/clickstream_dump_op") \
  .option("checkpointLocation", "/user/root/clickstream_dump_cp") \
  .start() \
  .awaitTermination()
```

Step2: Executing 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

Screenshot showing spark command run



```
[hadoop@ip-172-31-8-156 ~]$
[hadoop@ip-172-31-8-156 ~]$ 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
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-57e07081-cc45-41dc-a2de-a757c3704b0a;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
downloading https://repo1.maven.org/maven2/org/apache/spark/spark-sql-kafka-0-10_2.11/2.4.5/spark-sql-kafka-0-10_2.11-2.4.5.jar ...
        [SUCCESSFUL ] org.apache.spark#spark-sql-kafka-0-10_2.11;2.4.5!spark-sql-kafka-0-10_2.11.jar (32ms)
downloading https://repo1.maven.org/maven2/org/apache/kafka/kafka-clients/2.0.0/kafka-clients-2.0.0.jar ...
        [SUCCESSFUL ] org.apache.kafka#kafka-clients;2.0.0!kafka-clients.jar (93ms)
downloading https://repo1.maven.org/maven2/org/spark-project/spark/unused/1.0.0/unused-1.0.0.jar ...
        [SUCCESSFUL ] org.spark-project.spark#unused;1.0.0!unused.jar (4ms)
downloading https://repo1.maven.org/maven2/org/lz4/lz4-java/1.4.0/lz4-java-1.4.0.jar ...
        [SUCCESSFUL ] org.lz4#lz4-java;1.4.0!lz4-java.jar (22ms)
downloading https://repo1.maven.org/maven2/org/xerial/snappy/snappy-java/1.1.7.3/snappy-java-1.1.7.3.jar ...
        [SUCCESSFUL ] org.xerial.snappy#snappy-java;1.1.7.3!snappy-java.jar(bundle) (89ms)
downloading https://repo1.maven.org/maven2/org/slf4j/slf4j-api/1.7.16/slf4j-api-1.7.16.jar ...
        [SUCCESSFUL ] org.slf4j#slf4j-api;1.7.16!slf4j-api.jar (4ms)
:: resolution report :: resolve 1799ms :: artifacts dl 254ms
        :: 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   |   6   |   6   |   0   ||   6   |   6   |
        ---------------------------------------------------------------------
```

Screenshot showing file data in structured format

Step3: Created a python file to flatten and clean up the data.(spark_local_flatten.py)

vi spark_local_flatten.py

Explanation of code for spark_local_flatten.py

- Necessary imports
  import os
  import sys

- Environment setup
  os.environ["PYSPARK_PYTHON"] = "/opt/cloudera/parcels/Anaconda/bin/python"
  os.environ["JAVA_HOME"] = "/usr/java/jdk1.8.0_232-cloudera/jre"
  os.environ["SPARK_HOME"]="/opt/cloudera/parcels/SPARK2-2.3.0.cloudera2-1.cdh5.13.3.p0.316101/lib/spark2/"
  os.environ["PYLIB"] = os.environ["SPARK_HOME"] + "/python/lib"
  sys.path.insert(0, os.environ["PYLIB"] +"/py4j-0.10.6-src.zip")
  sys.path.insert(0, os.environ["PYLIB"] +"/pyspark.zip")

- Selected imports
  from pyspark.sql import SparkSession
  from pyspark.sql.functions import *

- Creating a spark session for reading operation

  spark=SparkSession.builder.appName("Kafka-to-HDFS").master("local").getOrCreate()

- Reading data from HDFS into a data frame for actions to be performed

  df=spark.read.json("/user/root/clickstream_dump_op/part-00000-10aa7bd4-2598-4e95-a35d-607379ab389f-c000.json")

- Selecting the columns from the clickstream data set from json format to data frame.

```
df=df.select(get_json_object(df['value_str'],"$.customer_id").alias("customer_id"),
        get_json_object(df['value_str'],"$.app_version").alias("app_version"),
        get_json_object(df['value_str'],"$.OS_version").alias("OS_version"),
        get_json_object(df['value_str'],"$.lat").alias("lat"),
        get_json_object(df['value_str'],"$.lon").alias("lon"),
        get_json_object(df['value_str'],"$.page_id").alias("page_id"),
        get_json_object(df['value_str'],"$.button_id").alias("button_id"),
        get_json_object(df['value_str'],"$.is_button_click").alias("is_button_click"),
        get_json_object(df['value_str'],"$.is_page_view").alias("is_page_view"),
        get_json_object(df['value_str'],"$.is_scroll_up").alias("is_scroll_up"),
        get_json_object(df['value_str'],"$.is_scroll_down").alias("is_scroll_down"),
        get_json_object(df['value_str'],"$.timestamp").alias("timestamp")
         )
```

- Writing the dataset back to hdfs in csv format for further use case.

  df.coalesce(1).write.format('csv').mode('overwrite').save('/user/root/clickstream_flattened',header='true')

- Run Spark Submit command
  spark-submit --packages org.apache.spark:spark-sql-kafka-0-10_2.11:2.4.5 spark_local_flatten.py

Step4: Writing script to ingest the relevant bookings data from AWS RDS to Hadoop

- Need to install MySQL connector on EMR

  wget https://de-mysql-connector.s3.amazonaws.com/mysql-connector-java-8.0.25.tar.gz
  tar -xvf mysql-connector-java-8.0.25.tar.gz

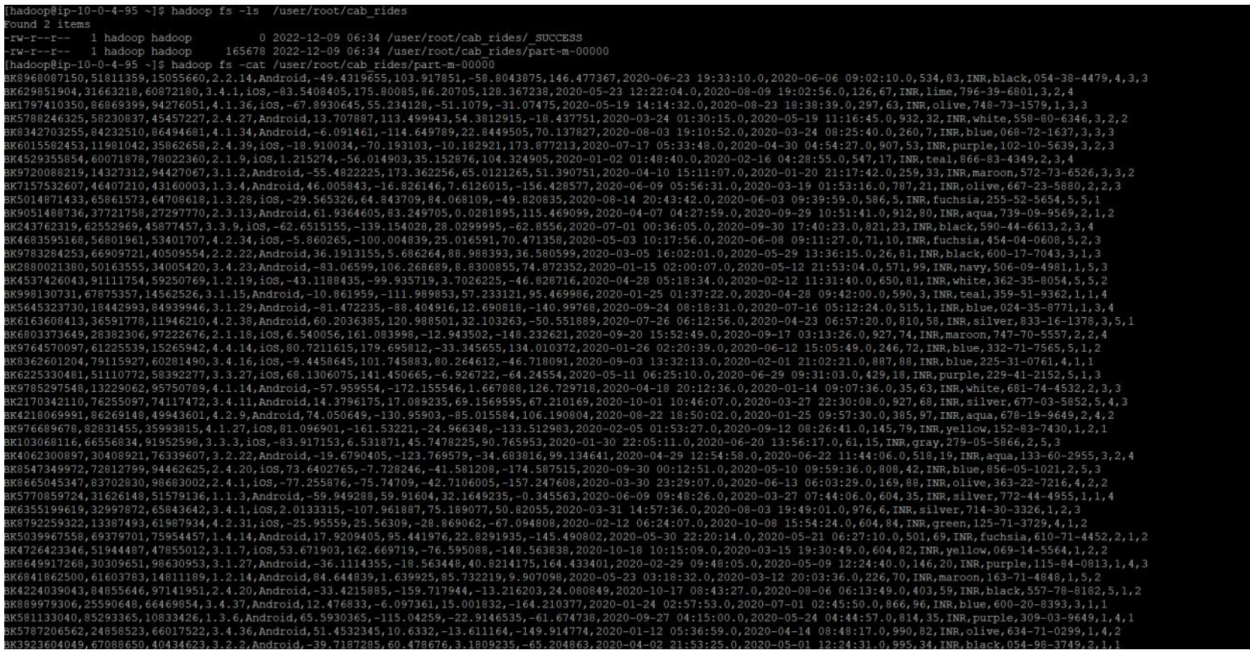
- Running the Sqoop import command to import data from AWS RDS to Hadoop

  sqoop import \
  --connect
  jdbc:mysql://upgraddetest.cyaielc9bmnf.us-east-1.rds.amazonaws.com/testdatabase \
  --table bookings \
  --username student --password STUDENT123 \
  --target-dir /user/root/cab_rides \
  -m 1

- Command to view the imported data

  hadoop fs –cat /user/root/cab_rides/part-m-00000

  Screenshot showing file contains



Step5: Creating aggregates for finding date-wise total bookings using the Spark script

- Created a python file which will contain the code to aggregate the booking data for finding date wise total bookings.

  vi datewise_bookings_aggregates_spark.py

Explanation of code for datewise_bookings_aggregates_spark.py

- Necessary imports
  import os

import sys

- Environment setup

```
os.environ["PYSPARK_PYTHON"] = "/opt/cloudera/parcels/Anaconda/bin/python"
os.environ["JAVA_HOME"] = "/usr/java/jdk1.8.0_232-cloudera/jre"
os.environ["SPARK_HOME"]="/opt/cloudera/parcels/SPARK2-2.3.0.cloudera2-1.cdh5.13
.3.p0.316101/lib/spark2/"
os.environ["PYLIB"] = os.environ["SPARK_HOME"] + "/python/lib"
sys.path.insert(0, os.environ["PYLIB"] +"/py4j-0.10.6-src.zip")
sys.path.insert(0, os.environ["PYLIB"] +"/pyspark.zip")
```

- Selective imports

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import *
```

- Creating spark session for reading operation of data into data frame

```
spark=SparkSession.builder.appName("datewise_bookings_aggregates_spark").master(
"local").getOrCreate()
```

- Reading data from HDFS to data frame for actions to be performed

```
df=spark.read.csv("/user/root/cab_rides/part-m-00000")
```

- Getting The count of the dataset

```
df.count()
```

- Renaming some columns for better readability and understanding

```
new_col =
["booking_id","customer_id","driver_id","customer_app_version","customer_phone_os_v
ersion","pickup_lat","pickup_lon","drop_lat",

"drop_lon","pickup_timestamp","drop_timestamp","trip_fare","tip_amount","currency_cod
e","cab_color","cab_registration_no","customer_rating_by_driver",
        "rating_by_customer","passenger_count"]


new_df = df.toDF(*new_col)
```

- Converting pickup_timestamp to date by extracting date from pickup_timestamp for aggregation

new_df=new_df.select("booking_id","customer_id","driver_id","customer_app_version","customer_phone_os_version","pickup_lat","pickup_lon","drop_lat",

"drop_lon",to_date(col('pickup_timestamp')).alias('pickup_date').cast("date"),"drop_timestamp","trip_fare","tip_amount","currency_code","cab_color","cab_registration_no","customer_rating_by_driver",
"rating_by_customer","passenger_count")

- Aggregating data on pickup_date

  agg_df=new_df.groupBy("pickup_date").count().orderBy("pickup_date")

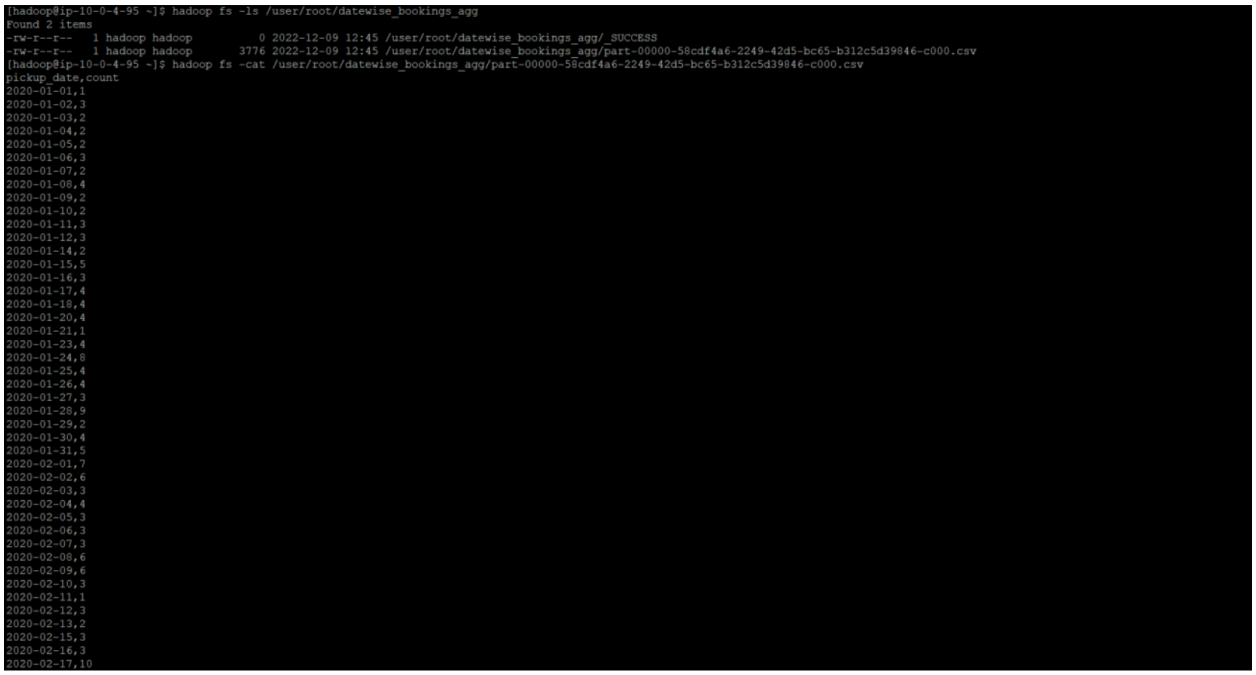
- The count of Bookings aggregates_table

  agg_df.count()

- Writing aggregated data to HDFS into CSV file format

  agg_df.coalesce(1).write.format('csv').mode('overwrite').save('/user/root/datewise_bookings_agg',header='true')

Screenshot showing files created in HDFS



Step6: Creating database and tables in hive and performing the tasks.

- **Creating Hive Managed Tables to perform use case queries.**

First create a database

create database if not exists cab_booking_data ;
use cab_booking_data;

Create **Clickstream** table

```
CREATE TABLE IF NOT EXISTS clickstream
  (
     customer_id      INT,
     app_version      STRING,
     os_version       STRING,
     lat              DOUBLE,
     lon              DOUBLE,
     page_id          VARCHAR(100),
     button_id        VARCHAR(100),
     is_button_click  STRING,
     is_page_view     STRING,
     is_scroll_up     STRING,
     is_scroll_down   STRING,
     `timestamp`      TIMESTAMP
  ) row format delimited fields TERMINATED BY ',' stored AS textfile;
```

Screenshot showing clickstream table created

```
hive> create table if not exists clickstream(
    >        customer_id int,
    >        app_version string,
    >        os_version string,
    >        lat double,
    >        lon double,
    >        page_id varchar(100),
    >        button_id varchar(100),
    >        is_button_click string,
    >        is_page_view string,
    >        is_scroll_up string,
    >        is_scroll_down string,
    >        `timestamp` timestamp)
    > ROW FORMAT DELIMITED
    > FIELDS TERMINATED BY ','
    > stored as textfile;
OK
Time taken: 0.547 seconds
```

Create **Booking** table

```
CREATE TABLE IF NOT EXISTS booking
  (
     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                 INT,
     tip_amount                INT,
     currency_code             STRING,
     cab_color                 STRING,
     cab_registration_no       INT,
     customer_rating_by_driver VARCHAR(100),
     rating_by_customer        INT,
     passenger_count           INT
  ) row format delimited fields TERMINATED BY ',' stored AS textfile;
```

Screenshot showing booking table created

```
Time taken: 0.347 seconds
hive> create table if not exists booking (
    >       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 int,
    >       tip_amount int,
    >       currency_code string,
    >       cab_color string,
    >       cab_registration_no int,
    >       customer_rating_by_driver varchar(100),
    >       rating_by_customer int,
    >       passenger_count int)
    > ROW FORMAT DELIMITED
    > FIELDS TERMINATED BY ','
    > stored as textfile;
OK
Time taken: 0.105 seconds
```

Create **Aggregated date wise** table

```
CREATE TABLE IF NOT EXISTS aggregate_datewise
  (
     pickup_date      DATE,
     booking_id_count INT
```

```
) row format delimited fields TERMINATED BY ',' stored AS textfile;
```

Screenshot showing aggregated date wise table created

- Commands to load the data into Hive tables

  1. Loading data into clickstream table

  load data local inpath '/home/hadoop/clickstream/clickstream_flattened.csv' into table clickstream;

  2. Loading data into booking table

  load data local inpath '/home/hadoop/cab_rides/part-m-00000' into table booking;

  3. Loading data into aggregate_datewise table

  load data local inpath '/home/hadoop/datewise_bookings_agg/datewise_agg' into table aggregate_datewise;

  Screenshot showing loading of data into different tables

```
Time taken: 1.857 seconds, Fetched: 1000 row(s)
hive> load data local inpath '/home/hadoop/clickstream/clickstream_flattened.csv' into table clickstream;
Loading data to table capstone.clickstream
OK
Time taken: 0.442 seconds
```

```
hive> load data local inpath '/home/hadoop/cab_rides/part-m-00000' into table booking;
Loading data to table capstone.booking
OK
Time taken: 1.566 seconds
hive> select * from capstone.booking;
OK
```

```
hive> load data local inpath '/home/hadoop/datewise_bookings_agg/datewise_agg.csv' into table aggregate_datewise;
Loading data to table capstone.aggregate_datewise
OK
Time taken: 0.426 seconds
```