Coats Assignment

Analysis of **Yellow Taxi Trip Data** of New York City

Using **Spark** and **Hadoop** Setup

Submitted By:

Logeshwaran R - 19S016,

M.Sc Data Science,

Department of AMCS,

Thiagarajar College of Engineering.

Understanding the Data:

Dataset Name:

yellow_tripdata_2020-06.csv

Description:

- The dataset contains taxi trip data in the city of New York for the month of June 2020.
- This dataset contains specifically yellow zone trips.
- Yellow zone refers to the areas in NY city where only yellow taxis are allowed to pickup customers without any reservations as well as street-hailing.

Columns:

tpep_pickup_datetime: Pickup date and time

tpep_dropoff_datetime: Dropoff date and time

Passenger_count: The number of passengers in the vehicle.

Trip_distance : The trip distance in miles reported by the taximeter.

PULocationID : Pickup TLC Taxi Zone

DOLocationID : Dropoff TLC Taxi Zone

Payment_type A numeric code signifying how the passenger paid for the trip.

---- 1= Credit card

---- 2= Cash

---- 3= No charge

---- 4= Dispute

---- 5= Unknown

---- 6= Voided trip

Fare_amount : The time-and-distance fare calculated by the meter.

Extra : Miscellaneous extras and surcharges. Currently, this only includes the $0.50 and $1 rush hour and overnight charges.

MTA_tax : $0.50 MTA tax that is automatically triggered based on the metered rate in use.

Improvement_surcharge : $0.30 improvement surcharge assessed trips at the flag drop. The improvement surcharge began being levied in 2015.

Tip_amount : Tip amount – This field is automatically populated for credit card tips. Cash tips are not included.

Tolls_amount : Total amount of all tolls paid in trip.

Total_amount : The total amount charged to passengers. Does not include cash tips

Preprocessing:

- Initial Data consisted of 549760 rows.
- Null value Removal
```
df = df.dropna()
```
- Removal of unwanted columns
```
df=df.drop(columns=["VendorID","store_and_fwd_flag",
                    "congestion_surcharge","RatecodeID"])
```
- Creating two new columns hour and day_of_week
```
df["tpep_pickup_datetime"]=pd.to_datetime(
        df["tpep_pickup_datetime"], format="%d-%m-%Y %H:%M")
df["hour"] = df["tpep_pickup_datetime"].dt.hour
df["day_of_week"] = df["tpep_pickup_datetime"].dt.day_name()
```
- Negative fare_amount is being assumed as penalty/money returned to passenger, hence it is not removed.
- Final data consists of 499043 rows.

Why Spark and Hadoop:

Apache Spark and Apache Hadoop are widely used in big data processing and analytics for several reasons:

1. Scalability
2. Data Processing Frameworks
3. In-Memory Processing (Spark)
4. Ease of Use (Spark)
5. Libraries and Ecosystem (Spark)
6. Real-Time Processing (Spark)
7. Fault Tolerance
8. Cost-Effective (Commodity hardware)
9. Compatibility
10. Batch and Stream Processing (Spark)

Analysis and Query Results:

1. **Reading *yellow_tripdata_2020-06.csv* file into a DataFrame created in spark:**

```
taxi = spark.read.format(“csv”).option(“header”, True).\
                                    option(“separator”, ”,”).\
                                    option(“inferSchema”, True).\
                                    load(“hdfs:///yellow_tripdata_2020-06.csv”)
taxi.createOrReplaceTempView("taxitb")
```

2. **Count the number of taxi trips for each hour:**

```
query = "select hour, count(hour) as num_of_taxi_trips from taxitb group by
        hour order by num_of_taxi_trips desc"
```

   Inference: This query allows us to do temporal analysis like finding peak hour. The peak hour for June 2020 is 15:00 which is 3 pm.

3. **Average fare amount collected by hour of the day:**

```
query = "select hour, avg(fare_amount) as avg_fare from taxitb group by hour
        order by avg_fare"
```

   Inference: This query tells whether time has an effect on Taxi’s fare. The hour with highest average fare is 4 with $18.23.

4. **Average fare amount compared to the average trip distance:**

```
query = "select avg(fare_amount) as avg_fare, avg(trip_distance) as
        avg_trip_dist, avg_fare/avg_trip_dist as ratio from taxitb"
```

   Inference: This query tells the average fare per unit distance covered by the taxi. The average fare per unit distance is $4.13.

5. **Average fare amount and average trip distance by day of the week:**

```
query = "select day_of_week, avg(fare_amount) as avg_fare,
        avg(trip_distance) as avg_trip_dist from taxitb group by day_of_week
        order by day_of_week"
```

   Inference: This query gives us the average fare and average distance travelled in a taxi for all the 7 days of the week.

6. **In the month of June 2020, find the zone which had maximum number of pickups:**

   query = "select looktb.zone, count(taxitb.PULocationID) as no_of_pickups
   from looktb inner join taxitb on looktb.LocatioID =
   taxitb.PULocationID group by looktb.zone order by
   no_of_pickups desc limit 1”

   Inference: This gives us the location with highest no.of customer pickups which is here the Upper East Side North.

7. **In the month of June 2020, find the zone which had maximum number of drops:**

   query = "select looktb.zone, count(taxitb.DOLocationID) as no_of_dropoffs
   from looktb inner join taxitb on looktb.LocatioID =
   taxitb.DOLocationID group by looktb.zone order by
   no_of_dropoffs desc limit 1”

   Inference: This gives us the location with highest no.of customer dropoffs which is here the Upper East Side North.

8. **Average no of passengers by hour of the day:**

   query = "select hour, avg(passenger_count) as avg_passenger_count from
   taxitb group by hour order by hour"

   Inference: This tells us the average no.of passengers that travel by taxi in a particular hour of the day.

9. **Total number of payments made by different type for the month:**

   query = "select payment_type, count(payment_type) as payment_type_count
   from taxitb group by payment_type order by payment_type_count
   desc"

   Inference: This query allows us to find the most used payment method. Using this we can improve our fare collection and safety of it by encouraging people to use cashless transactions.

10. **Configuring Hadoop cluster and Spark Installation on the cluster:**

    10.1 Installing a Linux OS:

    Installed Ubuntu 22.04.3 in Virtual Box 7.0 on Windows 10

    10.2 Creating user in Ubuntu:

    Created a user ‘hadoop’

10.3 Downloading and Installing Hadoop:

Downloaded ‘hadoop-3.3.0’ using the command

wget”https://archive.apache.org/dist/hadoop/common/hadoop3.3.0/ hadoop-3.3.0.tar.gz”

Extract this tar file using the command

tar -xzf Hadoop-3.3.0.tar.gz

10.4 Setting Password less SSH between nodes:

Created a localhost inside the user ‘hadoop’ and set up password less SSH between both. This helps in sharing and accessing files while executing a job.

10.5 Setting up HDFS Configuration files:

Configured core-site.xml file and hdfs-site.xml file in the */hadoop-3.3.0/etc/hadoop* location.

10.6 Creating Directories for hdfs:

Created directory *hdfs/namenode* and *hdfs/datanode* in the home directory where the hadoop-3.3.0 lies.

10.7 Installing Java:

Installed java-11-openjdk and stored its location in the */hadoop-3.3.0/etc/hadoop/hadoop-env.sh* as *JAVA_HOME*.

10.8 Formatting hdfs:

Formatted hdfs file system with the command

./hdfs namenode -format This command formats the hdfs file system makes it ready to store files.

Now if we use the command *start-dfs.sh* the namenode and the datanode should start. We can see the nodes and its configurations at *localhost:9870*. To stop the nodes use *stop-dfs.sh* .

10.9 Setting up YARN Configuration files:

Configured yarn-env.sh, yarn-site.xml, capacity-scheduler.xml files in the /hadoop-3.3.0/etc/hadoop location. This allocates disk memory to the job scheduler, sets queue capacities, application lifetime, job settings etc and other necessary configurations

10.10 Adding files into the HDFS:

To add files to the Hadoop File System this command is used

./hdfs dfs -put /actual/file/path /

The last / denotes the HDFS directory. This command stores the files inside HDFS. The Hadoop nodes can access only the files inside HDFS. The files can also be uploaded to HDFS through the *localhost:9870*.

10.11 Downloading and Installing Spark:

Using the following command spark can be downloaded

wget"https://www.apache.org/dyn/closer.lua/spark/spark-3.4.1/spark-3.4.1-bin-hadoop3.tgz"

Extract the spark using the command

tar -xzf spark-3.4.1-bin-hadoop3.tgz

10.12 Setting up Spark Configuring files:

Configured spark-env.sh file to setup *HADOOP_CONF_DIR* and *YARN_CONF_DIR* as we will be using Hadoop YARN for job scheduling in Spark.

10.13 Running Tasks and Queries on Spark:

To run tasks and queries on spark we can use pyspark, scala or java. We can run tasks in scala using both an interactive environment or executing a scala file containing queries and tasks.

Pyspark also has an interactive environment and it uses python for coding tasks. Queries can be executed using SparkSQL which can be imported in all the above languages.

11. Configuration Files:

capacity-scheduler.xml

```
<configuration>
  <property>
    <name>yarn.scheduler.capacity.maximum-applications</name>
    <value>10000</value>
  </property>

  <property>
    <name>yarn.scheduler.capacity.maximum-am-resource-
percent</name>
    <value>0.1</value>
  </property>

  <property>
    <name>yarn.scheduler.capacity.resource-calculator</name>
<value>org.apache.hadoop.yarn.util.resource.DefaultResourceCalcu
lator</value>
  </property>

  <property>
    <name>yarn.scheduler.capacity.root.queues</name>
    <value>prod,dev</value>
  </property>

  <property>
    <name>yarn.scheduler.capacity.root.prod.capacity</name>
    <value>70</value>
  </property>

  <property>
    <name>yarn.scheduler.capacity.root.dev.capacity</name>
    <value>30</value>
  </property>

  <property>
    <name>yarn.scheduler.capacity.root.user-limit-factor</name>
    <value>1</value>
  </property>

  <property>
    <name>yarn.scheduler.capacity.root.maximum-capacity</name>
    <value>100</value>
  </property>

  <property>
    <name>yarn.scheduler.capacity.root.state</name>
    <value>RUNNING</value>
  </property>

  <property>

<name>yarn.scheduler.capacity.root.acl_submit_applications</name
>
    <value>*</value>
  </property>

  <property>

<name>yarn.scheduler.capacity.root.acl_administer_queue</name>
    <value>*</value>
  </property>

  <property>

<name>yarn.scheduler.capacity.root.acl_application_max_priority<
/name>
    <value>*</value>
  </property>

   <property>
     <name>yarn.scheduler.capacity.root.maximum-application-
lifetime
     </name>
     <value>-1</value>
   </property>

   <property>
     <name>yarn.scheduler.capacity.root.default-application-
lifetime
     </name>
     <value>-1</value>
   </property>

  <property>
    <name>yarn.scheduler.capacity.node-locality-delay</name>
    <value>40</value>
  </property>

  <property>
    <name>yarn.scheduler.capacity.rack-locality-additional-
delay</name>
    <value>-1</value>
  </property>

  <property>
    <name>yarn.scheduler.capacity.queue-mappings</name>
    <value></value>
  </property>

  <property>
    <name>yarn.scheduler.capacity.queue-mappings-
override.enable</name>
    <value>false</value>
  </property>

  <property>
    <name>yarn.scheduler.capacity.per-node-heartbeat.maximum-
offswitch-assignments</name>
    <value>1</value>
  </property>
  <property>
    <name>yarn.scheduler.capacity.application.fail-fast</name>
    <value>false</value>
  </property>

  <property>
    <name>yarn.scheduler.capacity.workflow-priority-
mappings</name>
    <value></value>
  </property>

  <property>
    <name>yarn.scheduler.capacity.workflow-priority-mappings-
override.enable</name>
    <value>false</value>
  </property>
</configuration>
```

core-site.xml

```
<configuration>
<property>
       <name>fs.defaultFS</name>
       <value>hdfs://localhost:9000</value>
</property>
</configuration>
```

hdfs-site.xml

```
<configuration>
<property>
       <name>dfs.replication</name>
       <value>1</value>
</property>
<property>
       <name>dfs.namenode.name.dir</name>
       <value>/home/hadoop/hdfs/namenode/</value>
</property>
<property>
       <name>dfs.datanode.data.dir</name>
       <value>/home/hadoop/hdfs/datanode</value>
</property>
</configuration>
```

spark-env.sh

```
export HADOOP_CONF_DIR=/home/hadoop/hadoop-3.3.0/etc/hadoop
export YARN_CONF_DIR=/home/hadoop/hadoop-3.3.0/etc/hadoop
export PYSPARK_PYTHON=python3
```

yarn-site.xml

```
<configuration>
<property>
       <name>yarn.resourcemanager.scheduler.class</name>

<value>org.apache.hadoop.yarn.server.resourcemanager.scheduler.c
apacity.CapacityScheduler</value>
</property>
<property>
       <name>yarn.scheduler.capacity.root.queues</name>
       <value>prod,dev</value>
</property>
<property>
       <name>yarn.scheduler.capacity.prod.capacity</name>
       <value>0.5</value>
</property>
<property>
       <name>yarn.scheduler.capacity.dev.capacity</name>
       <value>0.5</value>
</property>
<property>
       <name>yarn.scheduler.capacity.dev.maximum-capacity</name>
       <value>0.5</value>
</property>
<property>
       <name>yarn.scheduler.capacity.prod.maximum-
capacity</name>
       <value>0.7</value>
</property>
</configuration>
```

Conclusion:

In this project we have used Spark on top of the Hadoop Cluster. We made the spark to use the YARN for job scheduling as Hadoop uses commodity hardware for storing which makes it cheaper than any other setup.

Spark and Hadoop have high fault tolerance. They make multiple copies of data so that even one node fails other continues to provide the service. Both of them are highly scalable and can be scaled easily according to the project. We used this Spark to run queries over a huge dataset containing about 5 lakh rows.

SparkSQL was used for running queries. These queries provided crucial insights like

- Peak taxi demand hour which can be used to increase the taxi cout at that particular hour.
- Average fare in an hour – used to set future fare amount
- Average fare per mile distance travelled – used to set fare amount
- Zones with most pickups and drop-offs can be used to increase taxi availability in those zones
- Factors affecting the fare_amount – distance, hour, dayofweek this can be observed on the resulting query tables.

In conclusion, combination of Spark and Hadoop both harnesses the power of Parallel computing using RAM and Distributed storage using cheap hardware. This system is more efficient than many other similar systems for Big Data Analytics.