**USED CAR DATA ANALYSIS**

A PROJECT REPORT

***Submitted by***

BL.EN.U4AIE19007 APOORVA.M

BL.EN.U4AIE19041 TANUJ.M

BL.EN.U4AIE19068 AISHWARYA.V

**19AIE304 BIG DATA AND DATABASE MANAGEMENT**



AMRITA SCHOOL OF ENGINEERING, BANGALORE

 AMRITA VISHWA VIDYAPEETHAM

**BANGALORE 560 035**

#### **TABLE OF CONTENTS**

1. **ABSTRACT………………………………………………...3**
2. **INTRODUCTION………………………………………….4**
3. **PROPOSED SYTEM ARCHITECTURE……………….12**
4. **RESULTS………………………………………………….13**
5. **CONCLUSION…………………………………….………28**
6. **REFERENCES…………………………………………….28**
7. **ANNEXURE……………………………………………….29**

**ABSTRACT**

The auto industry is changing rapidly and car prices are only going up. So, to speak, new cars are getting costlier each year, making them a very high value purchase for the common man. And quite ironically, the average life span of a car is going down despite the steady rise in prices, which brings in good news for potential used car buyers.

A used car in fact makes more sense for first time buyers upgrading from two-wheelers or public transportation, or for that matter, someone looking to buy a second set of wheels in the family. This Project is focused on filtering out cars for people from different walks of life depending on their requirements, be it for personal use or professional use. The project is implemented using Spark SQL data frame, structed streaming and Visualizations have been performed using spark.

**Key Words**: Pyspark, Spark Streaming, Spark SQL, DataFrame, Visualization, Big Data, Database

**INTRODUCTION**

### ABOUT THE DATASET

### Context

### Kaggle Dataset: <https://www.kaggle.com/ananaymital/us-used-cars-dataset>

### Dataset size: 9.98 GB

### 

The dataset contains 3 million real world used cars details.

This data was obtained by running a self-made crawler on CarGurus inventory in September 2020.

### Acknowledgements

This data is for academic, research and individual experimentation only and is not intended for commercial purposes.

### MOTIVATION

Imagine a web app that can estimate the listing price of a vehicle. What features of the vehicle should be used to build a price prediction regression model?

A used car in fact makes more sense for first time buyers upgrading from two-wheelers or public transportation, or for that matter, someone looking to buy a second set of wheels in the family. This Project is focused on filtering out cars for people from different walks of life depending on their requirements, be it for personal use or professional use. The project is implemented using Spark SQL data frame, structed streaming and Visualizations have been performed using spark.

Pros And Cons of Buying A Certified Pre Owned Car | CarTrade Blog

Read an auto guide on Pros and Cons of Buying a Certified Pre-Owned Car on CarTrade Blog.

**DATASET ATTRIBUTES:**

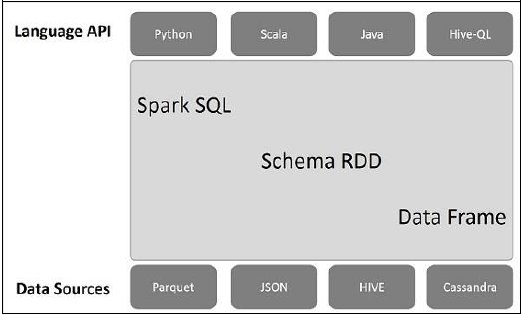
1. vin: Type String. Vehicle Identification Number is a unique encoded string for every vehicle.
2. back\_legroom: Type String. Legroom in the rear seat.
3. bed: Type String. Category of bed size (open cargo area) in pickup truck. Null usually means the vehicle isn't a pickup truck
4. bed\_height: Type String. Height of bed in inches
5. bed\_length: Type String. Length of bed in inches
6. body\_type: Type String. Body Type of the vehicle. Like Convertible, Hatchback, Sedan, etc.
7. cabin: Type String. Category of cabin size (open cargo area) in pickup truck. Eg: Crew Cab, Extended Cab, etc.
8. city: Type String. city where the car is listed. Eg: Houston, San Antonio, etc.
9. cityfueleconomy: Type Float. Fuel economy in city traffic in km per litre
10. combinefueleconomy: Type Float. Combined fuel economy is a weighted average of City and Highway fuel economy in km per litre
11. daysonmarket: Type Integer. Days since the vehicle was first listed on the website.
12. dealer\_zip: Type Integer. Zipcode of the dealer
13. description: Type String. Vehicle description on the vehicle's listing page
14. engine\_cylinders: Type String. The engine configuration. Eg: I4, V6, etc.
15. enginedisplacement: Type Float. enginedisplacement is the measure of the cylinder volume swept by all of the pistons of a piston engine, excluding the combustion chambers.
16. engine\_type: Type String. The engine configuration. Eg: I4, V6, etc.
17. exterior\_color: Type String. Exterior color of the vehicle, usually a fancy one same as the brochure.
18. fleet: Type Boolean. Whether the vehicle was previously part of a fleet.
19. frame\_damaged: Type Boolean. Whether the vehicle has a damaged frame.
20. franchise\_dealer: Type Boolean. Whether the dealer is a franchise dealer.
21. franchise\_make: Type String. The company that owns the franchise.
22. front\_legroom: Type String. The legroom in inches for the passenger seat
23. fueltankvolume: Type String. Fuel tank's filling capacity in gallons
24. fuel\_type: Type String. Dominant type of fuel ingested by the vehicle.
25. has\_accidents: Type Boolean. Whether the vin has any accidents registered.
26. height: Type String. Height of the vehicle in inches
27. highwayfueleconomy: Type Float. Fuel economy in highway traffic in km per litre
28. horsepower: Type Float. Horsepower is the power produced by an engine.
29. interior\_color: Type String. Interior color of the vehicle, usually a fancy one same as the brochure.
30. isCab: Type Boolean. Whether the vehicle was previously taxi/cab.
31. is\_certified: Type Boolean. Whether the vehicle is certified. Certified cars are covered through warranty period
32. is\_cpo: Type Boolean. Pre-owned cars certified by the dealer. Certified vehicles come with a manufacturer warranty for free repairs for a certain time period.
33. is\_new: Type Boolean. If True means the vehicle was launched less than 2 years ago.
34. is\_oemcpo: Type Boolean. Pre-owned cars certified by the manufacturer.
35. latitude: Type Float. Latitude from the geolocation of the dealership.
36. length: Type String. Length of the vehicle in inches
37. listeddate: Type String. The date the vehicle was listed on the website. Does not make daysonmarket obsolete. The prices is dayson\_market days after the listed date.
38. listing\_color: Type String. Dominant color group from the exterior color.
39. listing\_id: Unique Type Integer. Listing id from the website
40. longitude: Type Float. Longitude from the geolocation of the dealership.
41. mainpictureurl: Type String.

## **Spark SQL Architecture**

The following illustration explains the architecture of Spark SQL −

This architecture contains three layers namely, Language API, Schema RDD, and Data Sources.

* **Language API** − Spark is compatible with different languages and Spark SQL. It is also, supported by these languages- API (python, scala, java, HiveQL).
* **Schema RDD** − Spark Core is designed with special data structure called RDD. Generally, Spark SQL works on schemas, tables, and records. Therefore, we can use the Schema RDD as temporary table. We can call this Schema RDD as Data Frame.
* **Data Sources** − Usually the Data source for spark-core is a text file, Avro file, etc. However, the Data Sources for Spark SQL is different. Those are Parquet file, JSON document, HIVE tables, and Cassandra database.



**Fig: Spark SQL Architecture**

**Python Spark (pySpark)**

We are using the Python programming interface to Spark (pySpark) .It provides an easy-to-use programming abstraction and parallel runtime. RDDs are the key concept.

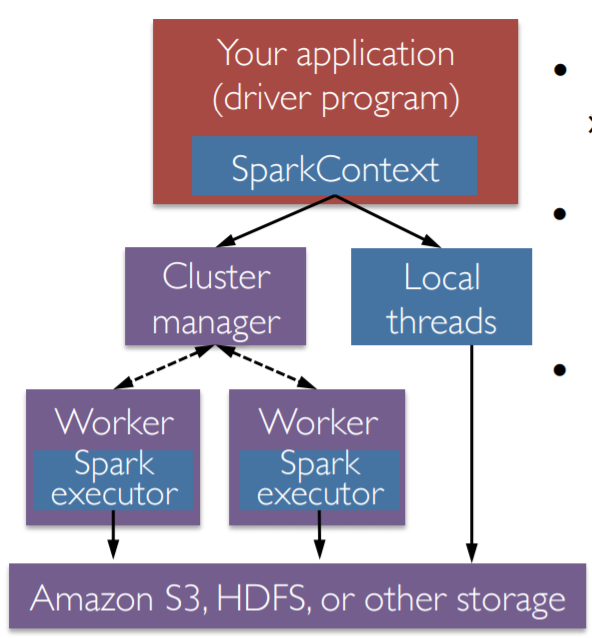
PySpark is an interface for Apache Spark in Python. It not only allows you to write Spark applications using Python APIs, but also provides the PySpark shell for interactively analyzing your data in a distributed environment. PySpark supports most of Spark’s features such as Spark SQL, DataFrame, Streaming, MLlib (Machine Learning) and Spark Core

**Spark Driver and Workers:**

A Spark program is two programs: A driver program and a workers program

Worker programs run on cluster nodes or in local threads.RDDs are distributed

across workers.



**Fig: Spark Context**

**Spark Context**

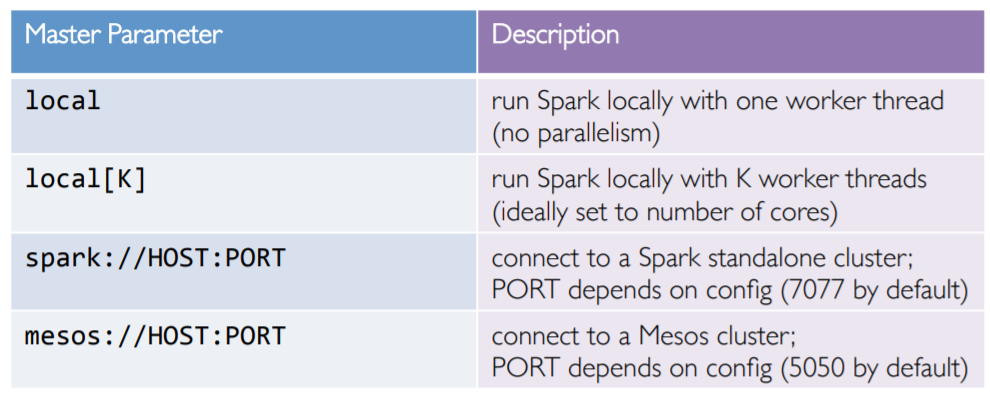
A Spark program first creates a SparkContext object

* » Tells Spark how and where to access a cluster
* » pySpark shell and Databricks Cloud automatically create the sc variable
* » iPython and programs must use a constructor to create a new SparkContext

• Use SparkContext to create RDDs

**Spark Essentials: Master**

The master parameter for a SparkContext determines which type and size of cluster to use.



**Fig: Table**

**STRUCTURED STREAMING**

Structured Streaming is a high-level API for stream processing that became production-ready in Spark 2.2. Structured Streaming allows you to take the same operations that you perform in batch mode using Spark’s structured APIs, and run them in a streaming fashion. This can reduce latency and allow for incremental processing. The best thing about Structured Streaming is that it allows you to get value out rapidly and quickly of streaming systems with virtually no code changes. It also makes it easy to reason about because you can write your batch job to prototype it and then you can convert it to a streaming job. The way all this works is by incrementally processing that data.

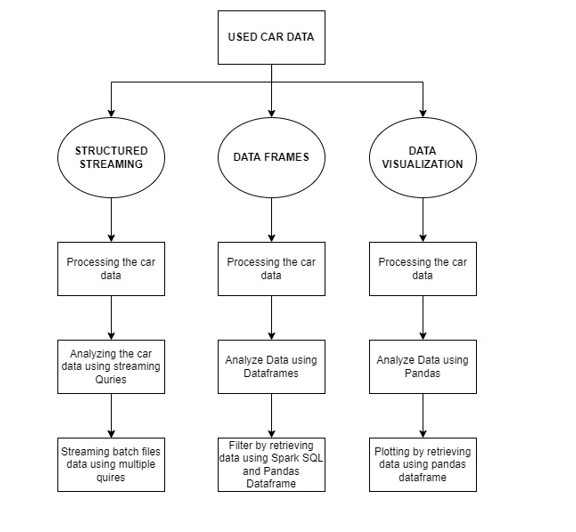
Structured Streaming is a scalable and fault-tolerant stream processing engine built on the Spark SQL engine. You can express your streaming computation the same way you would express a batch computation on static data. The Spark SQL engine will take care of running it incrementally and continuously and updating the final result as streaming data continues to arrive. You can use the Dataset/DataFrame API in Scala, Java, Python or R to express streaming aggregations, event-time windows, stream-to-batch joins, etc. The computation is executed on the same optimized Spark SQL engine. Finally, the system ensures end-to-end exactly-once fault-tolerance guarantees through checkpointing and Write-Ahead Logs. In short, Structured Streaming provides fast, scalable, fault-tolerant, end-to-end exactly-once stream processing without the user having to reason about streaming.

**Diagram

Description automatically generated**Internally, by default, Structured Streaming queries are processed using a micro-batch processing engine, which processes data streams as a series of small batch jobs thereby achieving end-to-end latencies as low as 100 milliseconds and exactly-once fault-tolerance guarantees. However, since Spark 2.3, we have introduced a new low-latency processing mode called Continuous Processing, which can achieve end-to-end latencies as low as 1 millisecond with at-least-once guarantees. Without changing the Dataset/DataFrame operations in your queries, you will be able to choose the mode based on your application requirements.

**Fig: Structured Streaming Processing Model**

**PROPOSED SYSTEM ARCHITECTURE**

****

**Fig: Flowchart describing our proposed system architecture**

**RESULTS**

**SPARK STREAMING**:

SPLITTING THE DATA

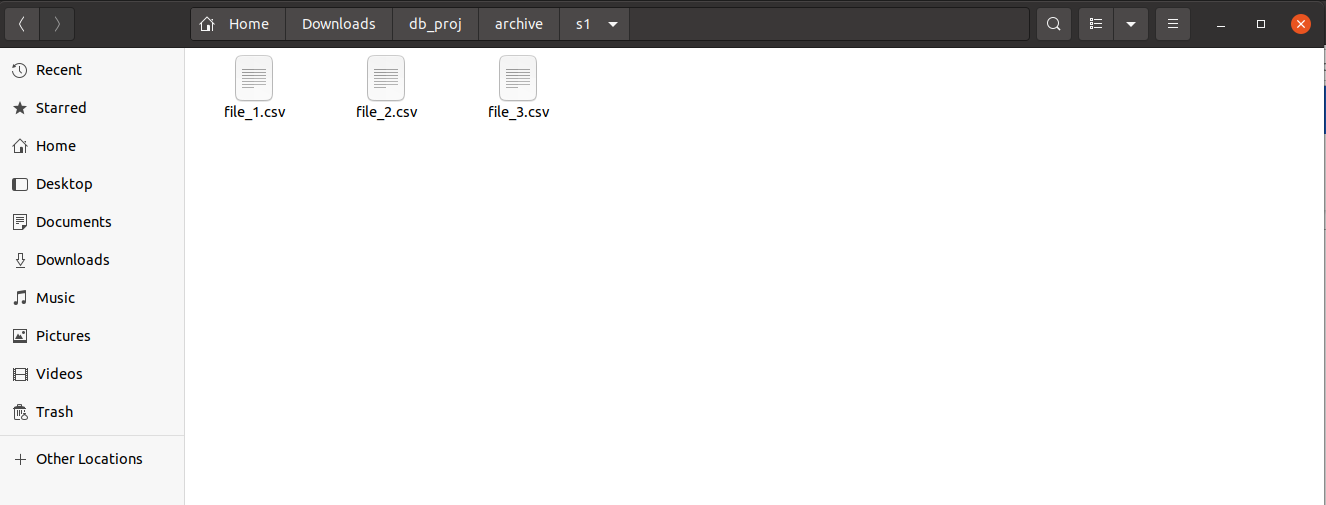
The UsedCar.csv data set file size 9.4GB. As this Size is large, it is split into 61 batches and stored in a Folder S2 . Out of which Four of the batches are moved to another folder S1. A python program has been written to move the remaining batches in S2 to S1 in a time interval of 5 milli seconds.

STREAMING

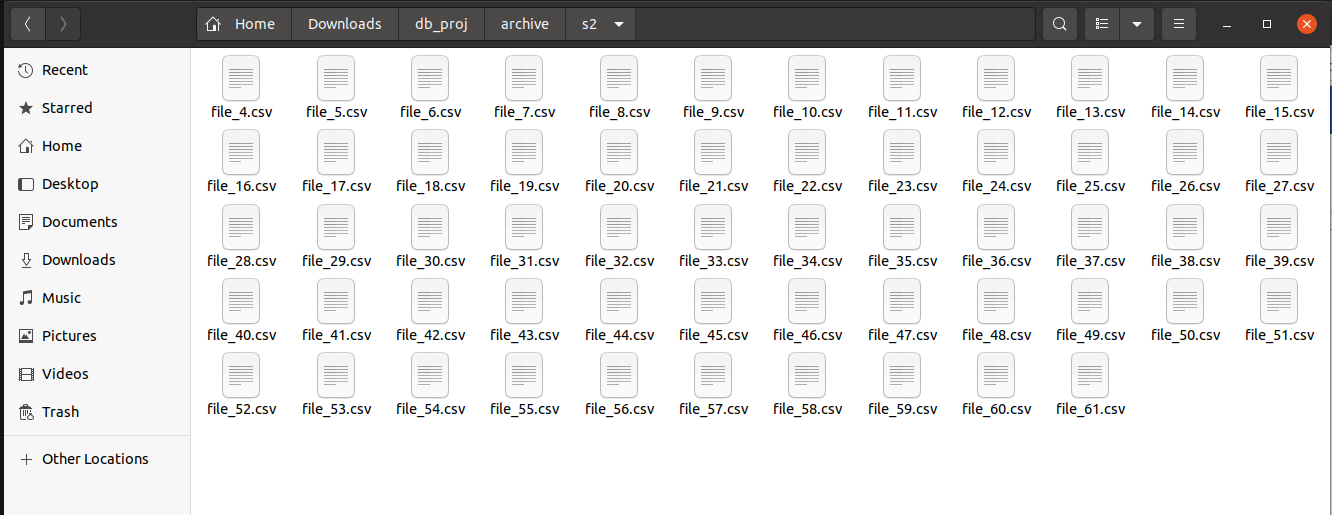
Spark shell is started along with running the python file. Required attributes for Streaming are initialized in spark.  files moving from Folder S2 to S1 can be observed. Required Queries are executed and batches of data being printed with top 20 rows will be observed.

****

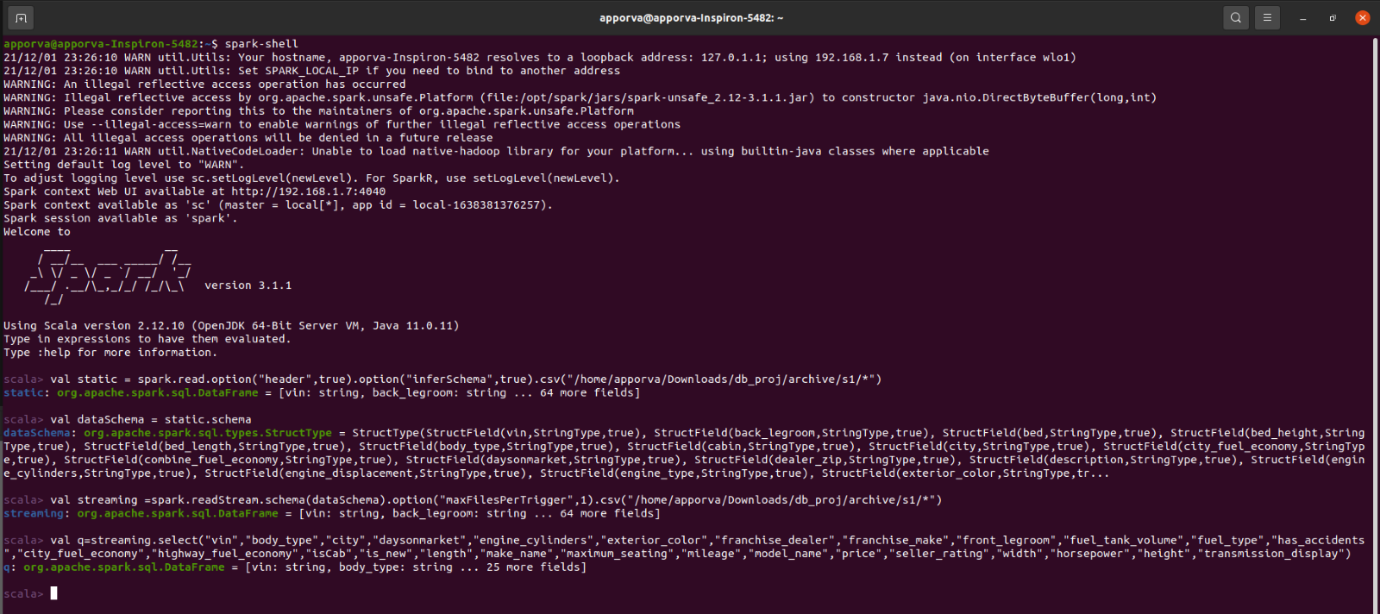
**Fig: Python code for moving files from folder S2 to S1**

****

**Fig: Folder S1**

****

**Fig: Folder S2**

****

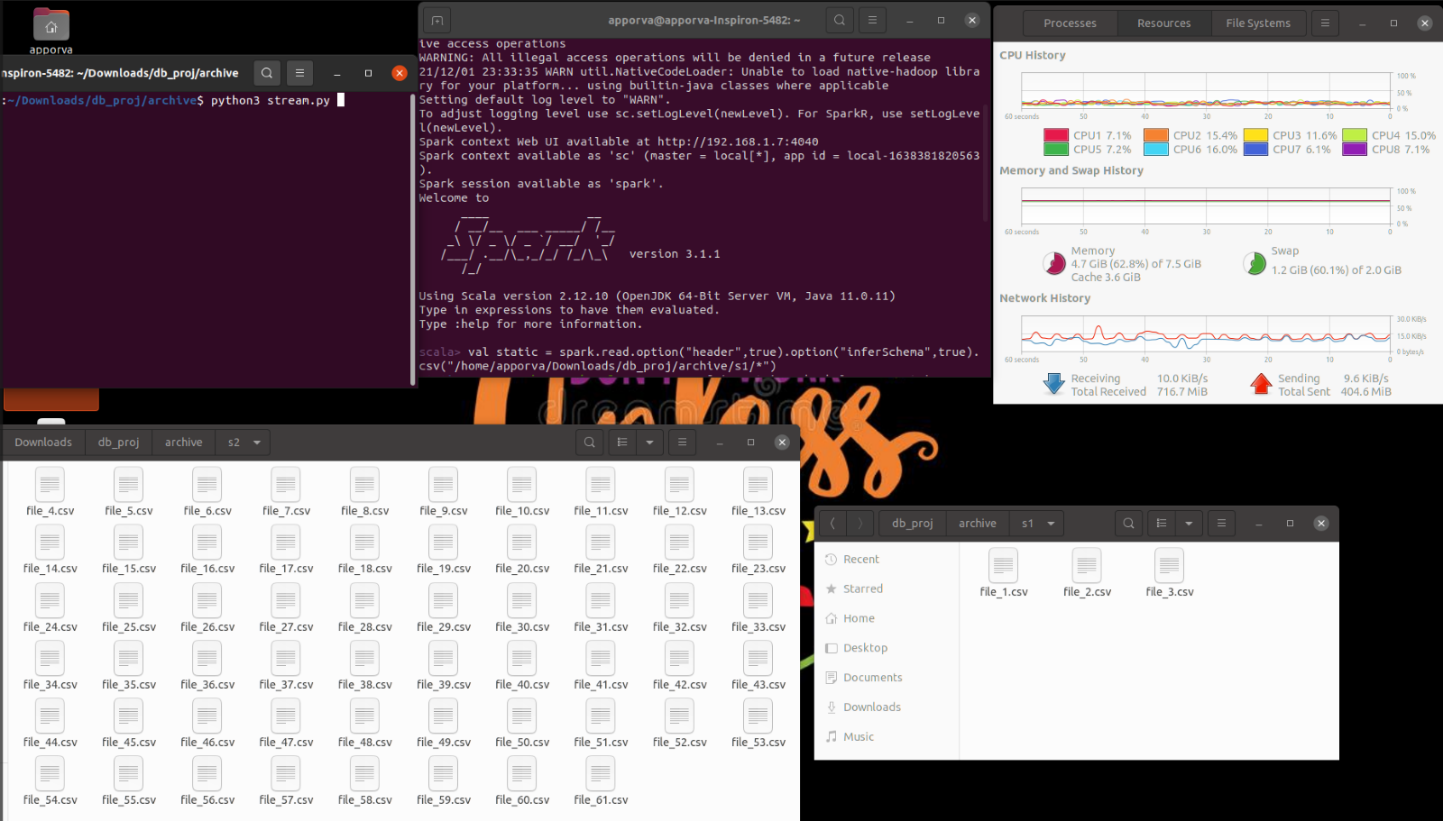
**Fig: Initializing spark session and the required variables for streaming the dataset**

**PARALLELISM**

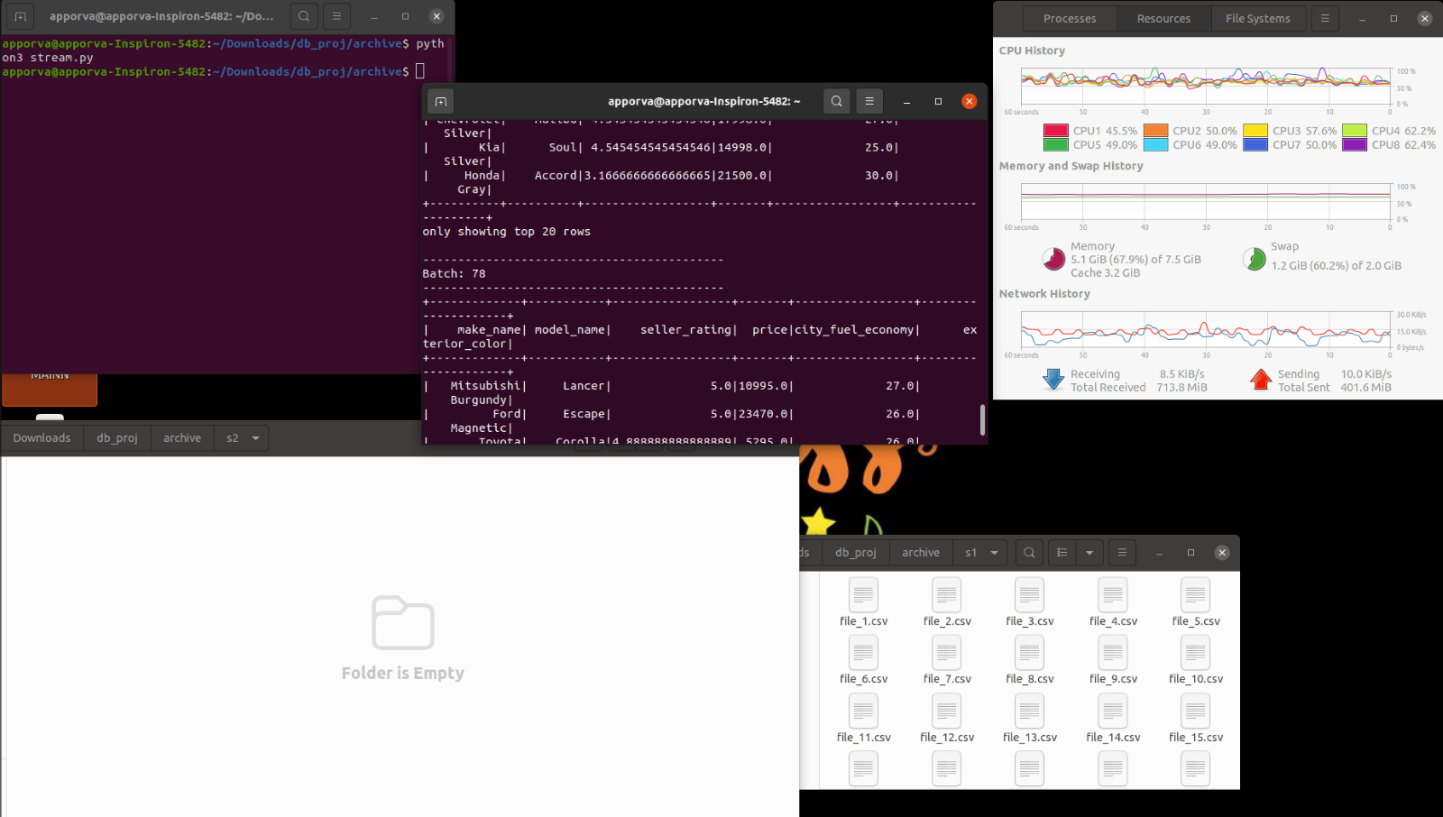
Here the figures below display all the required components for executing Streaming. Fig (1) is a snippet of the components before streaming is commenced and Fig (2) is a snippet of the components taken during streaming.

It can be observed from Fig (I) that the CPU cores are not running at a high percentage whereas when Fig (II) is observed it is notable that all the 8 cores are being used in high percentage.

This shows that load has been equally distributed among all the 8 cores. From this it can be inferred that parallelism is observed.

****

**Fig: Snippet of the components before streaming is commenced**

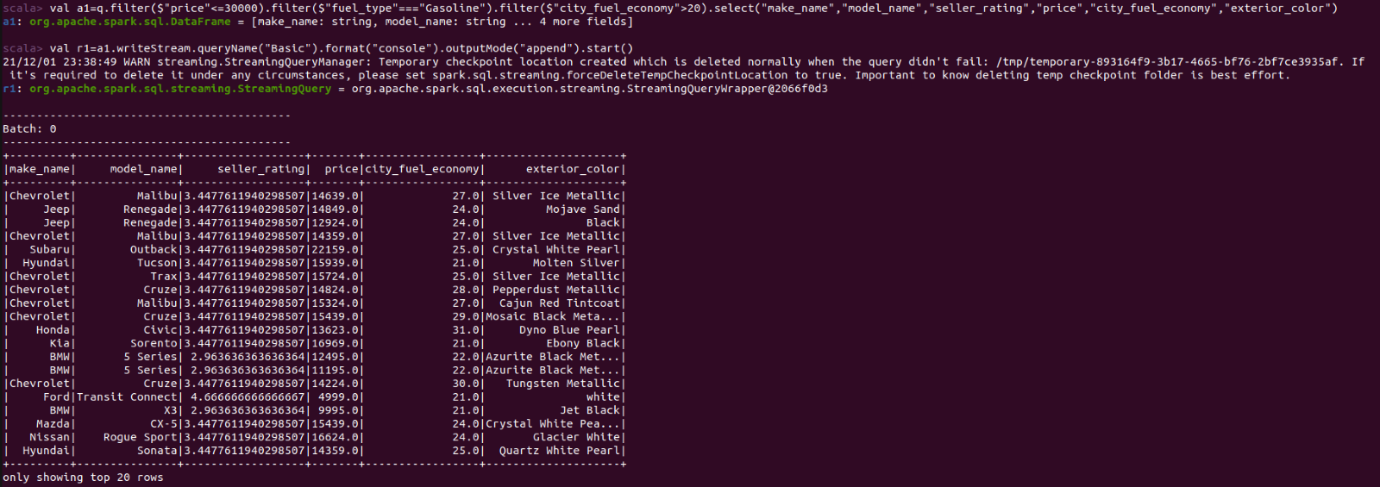
****

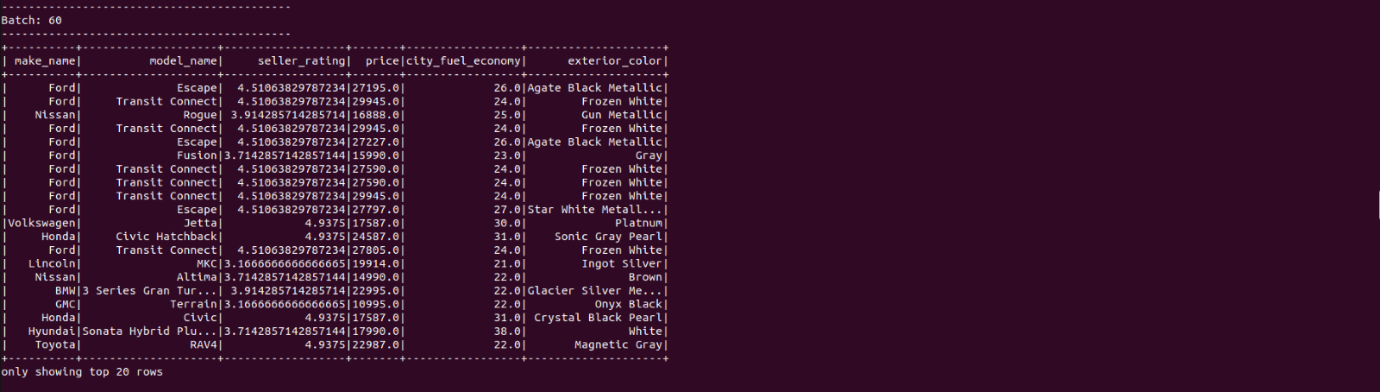
**Fig: Snippet of the components taken during streaming spark session**

**QUERIES:**

1. **Query: Basic**

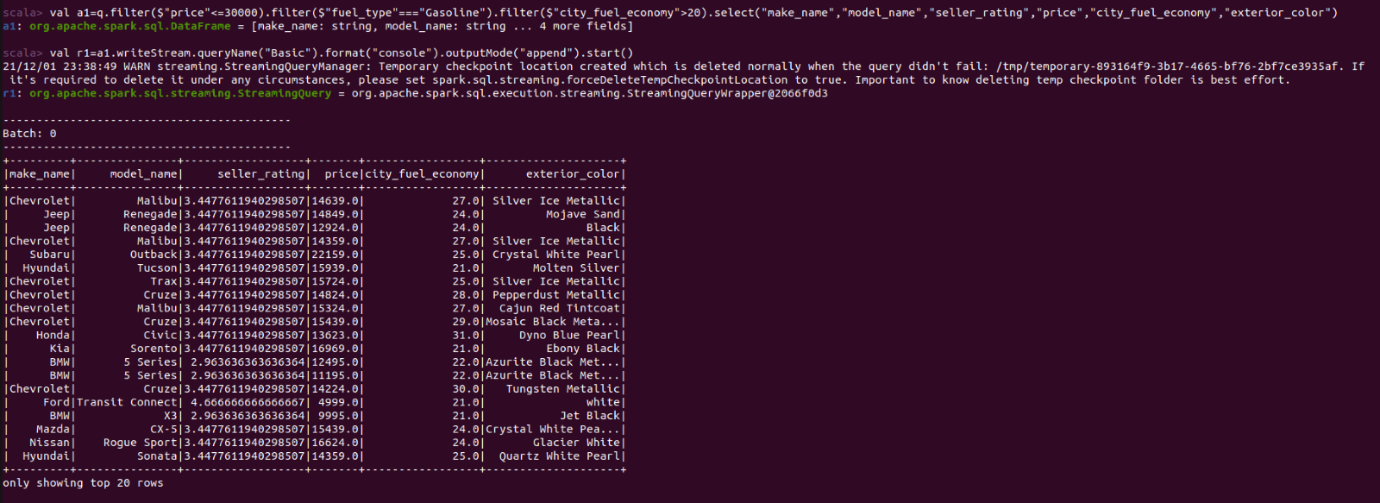
This query is based on analyzing all the basic attributes that an average working person would consider for buying a second-hand car for daily use. The attributes that have been taken into consideration are price, fuel type and city fuel economy.

****

****

1. **Query: Comfort**

This query is designed for the customers who would buy cars with comfort as priority. Keeping Comfort in mind the attributes considered here are front leg room, back leg room and maximum seating.

****

****

**3) Query: Traveler**

This query filters out cars that are apt for Travelers who are always on the go. The attributes that are considered here are high way fuel economy, fuel type and fuel tank volume.

****

****

**4)Query: Cab Driver**

This is created for helping cab driver find suitable cars for their work. The attributes considered here are city fuel economy, fuel type and fuel tank volume.

****

****

**5)Query: Packers and Movers**

This is for Packing agencies who want to get big Vehicles. The attribute that is considered here are length and city fuel economy.

****

****

Importing Dataset

Table

Description automatically generated

Selecting required columns and filling null values

Graphical user interface, application

Description automatically generated

Query1

Table

Description automatically generated

Query2

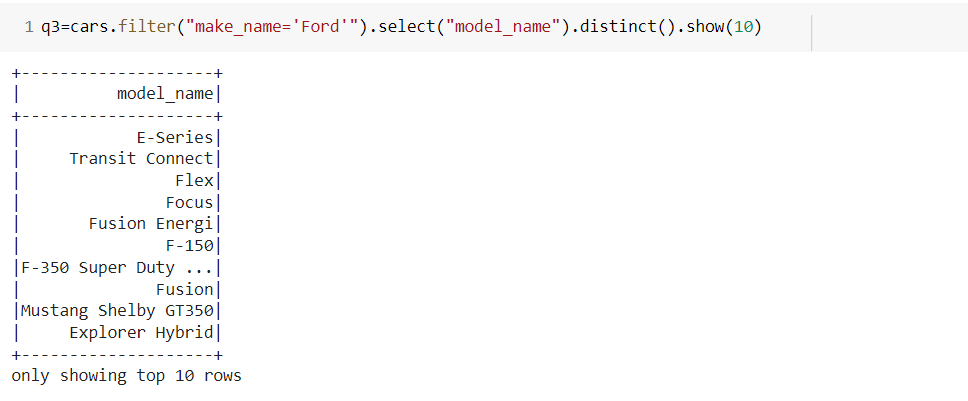
Graphical user interface, text, application, table

Description automatically generated with medium confidence

Query3:

Graphical user interface, text, application, chat or text message

Description automatically generated

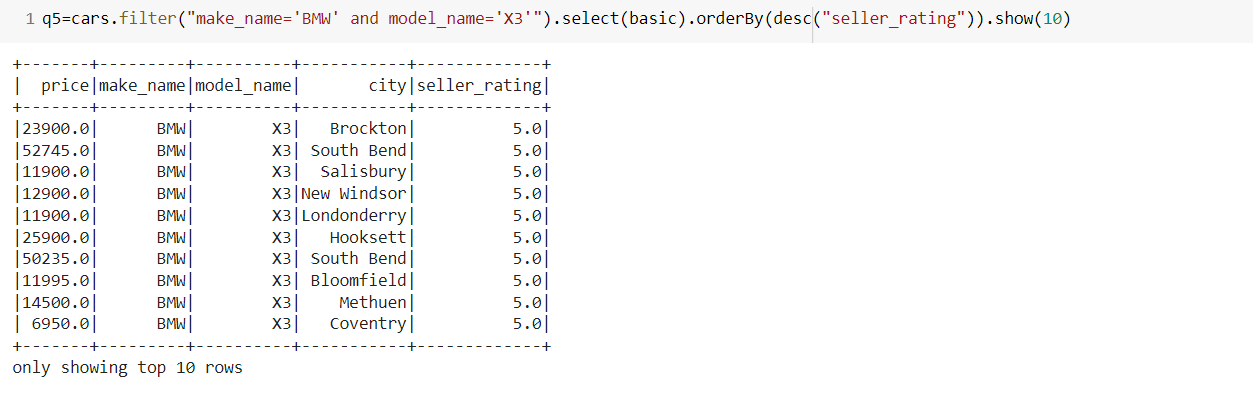
****

Query4

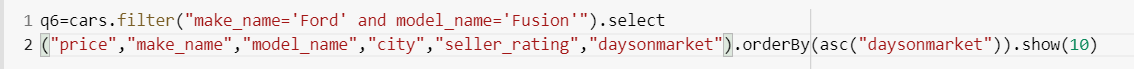
Table

Description automatically generated

Query5



Query6



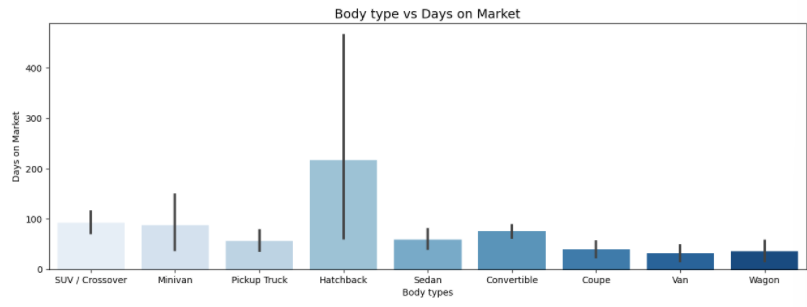


Data Visualization

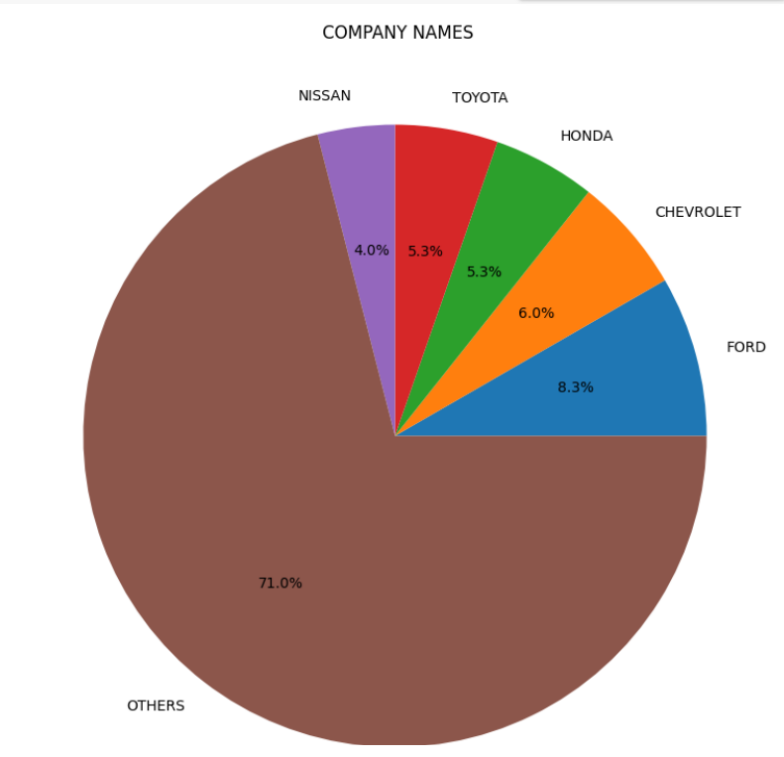
Graphical user interface, text, application

Description automatically generated

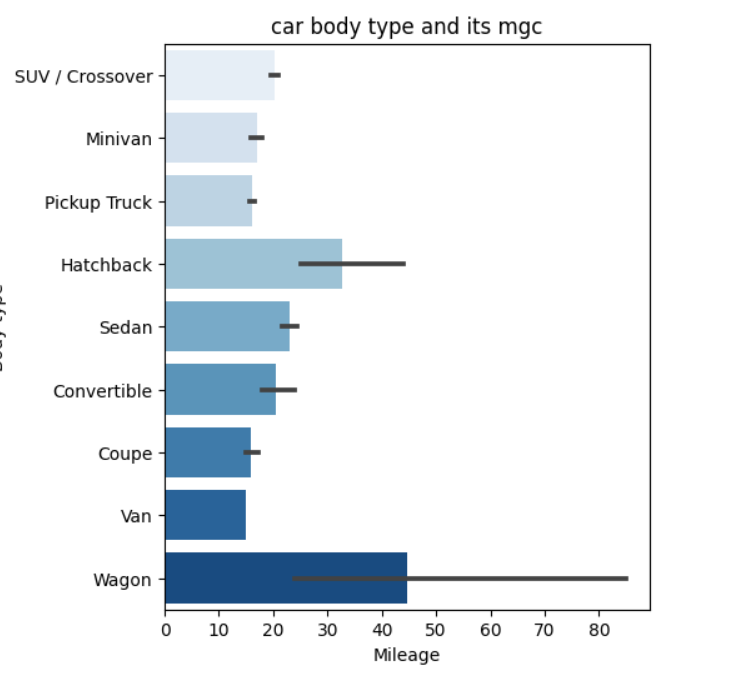
Bar plot for car model and the number of days it’s available on market



Pie chart for car company names and their available car count

****

Bar plot for car body type and it’s milage given in city.

****

**CONCLUSION**

Opting for used cars these days has become a more wise option keeping in mind maintenance, the ever rising prices and their life span. This project has focused on filtering out cars for different kinds of people according to their uses and requirements. Various aspects have been considered to accomplish maximum satisaction of the buyers. Enquiries for both personal and professional vehicle needs have been taken care off. Using Structured Streaming, Sql Dataframes and Visualizations satisfactory results were obtained.

**REFERENCES**

1. *https://www.researchgate.net/publication/319306871\_Predicting\_the\_Price\_of\_Used\_Cars\_using\_Machine\_Learning\_Techniques*
2. *https://www.irjet.net/archives/V8/i4/IRJET-V8I4278.pdf*
3. [*https://www.temjournal.com/content/81/TEMJournalFebruary2019\_113\_118.pdf*](https://www.temjournal.com/content/81/TEMJournalFebruary2019_113_118.pdf)
4. [*http://ripublication.com/irph/ijict\_spl/ijictv4n7spl\_17.pdf*](http://ripublication.com/irph/ijict_spl/ijictv4n7spl_17.pdf)

**ANNEXURE**

**Structured Streaming:**

spark-shell

val static = spark.read.option("header",true).option("inferSchema",true).csv("/home/apporva/Downloads/db\_proj/archive/s1/\*")

val dataSchema = static.schema

val streaming =spark.readStream.schema(dataSchema).option("maxFilesPerTrigger",1).csv("/home/apporva/Downloads/db\_proj/archive/s1/\*")

val q=streaming.select("vin","body\_type","city","daysonmarket","engine\_cylinders","exterior\_color","franchise\_dealer","franchise\_make","front\_legroom","fuel\_tank\_volume","fuel\_type","has\_accidents","city\_fuel\_economy","highway\_fuel\_economy","isCab","is\_new","length","make\_name","maximum\_seating","mileage","model\_name","price","seller\_rating","width","horsepower","height","transmission\_display")

#Q1

val a1=q.filter($"price"<=30000).filter($"fuel\_type"==="Gasoline").filter($"city\_fuel\_economy">20).select("make\_name","model\_name","seller\_rating","price","city\_fuel\_economy","exterior\_color")

val r1=a1.writeStream.queryName("Basic").format("console").outputMode("append").start()

#Q2

val c1=q.filter($"maximum\_seating"==="7 seats").filter($"back\_legroom"==="38.3 in").select("body\_type","seller\_rating","price")

val r3=c1.writeStream.queryName("Comfort").format("console").outputMode("append").start()

#Q3

val d1=q.filter($"highway\_fuel\_economy">15).filter($"fuel\_type"==="Gasoline").filter($"fuel\_tank\_volume"==="26 gal").select("seller\_rating","price","highway\_fuel\_economy","is\_new")

val r4=d1.writeStream.queryName("Traveller").format("console").outputMode("append").start()

#Q4

val e1=q.filter($"city\_fuel\_economy">10).filter($"fuel\_type"==="Gasoline").filter($"fuel\_tank\_volume"==="26 gal").filter($"is\_new"==="False").filter($"has\_accidents"==="False").select("seller\_rating","price","city\_fuel\_economy")

val r5=e1.writeStream.queryName("Cab\_driver").format("console").outputMode("append").start()

#Q5

val bq=q.filter($"length"==="231.9 in").filter($"city\_fuel\_economy">10).select("make\_name","seller\_rating","price","has\_accidents")

val rq=bq.writeStream.queryName("Packers and Movers").format("console").outputMode("append").start()

**Pyspark DataFrame Queries:**

import traceback

import findspark

import os

import sys

from pyspark.sql import SparkSession

import pandas as pd

import numpy as np

import matplotlib

import matplotlib.pyplot as plt

import matplotlib.pylab as pylab

import seaborn as sns

import subprocess

from pyspark.sql.functions import \*

from functools import reduce

from pyspark.ml.feature import StringIndexer

from pyspark.sql.types import DoubleType

from pyspark.ml import Pipeline

spark = SparkSession \

    .builder \

    .master("local[\*]") \

    .appName("PySpark Craigslist") \

    .config("spark.some.config.option", "some-value") \

    .getOrCreate()

sc = spark.sparkContext

vehicle\_listings = spark.read.format("csv").option("header", "true").load("/content/drive/MyDrive/unzipped\_dbms/used\_cars\_data.csv")

type(vehicle\_listings)

vehicle\_listings.show()

print(vehicle\_listings.columns)

print(vehicle\_listings.count(),len(vehicle\_listings.columns))

vehicle\_listings\_clean=vehicle\_listings.drop\_duplicates(['vin', 'back\_legroom', 'bed', 'bed\_height', 'bed\_length', 'body\_type', 'cabin', 'city', 'city\_fuel\_economy', 'combine\_fuel\_economy', 'daysonmarket', 'dealer\_zip', 'description', 'engine\_cylinders', 'engine\_displacement', 'engine\_type', 'exterior\_color', 'fleet', 'frame\_damaged', 'franchise\_dealer', 'franchise\_make', 'front\_legroom', 'fuel\_tank\_volume', 'fuel\_type', 'has\_accidents', 'height', 'highway\_fuel\_economy', 'horsepower', 'interior\_color', 'isCab', 'is\_certified', 'is\_cpo', 'is\_new', 'is\_oemcpo', 'latitude', 'length', 'listed\_date', 'listing\_color', 'listing\_id', 'longitude', 'main\_picture\_url', 'major\_options', 'make\_name', 'maximum\_seating', 'mileage', 'model\_name', 'owner\_count', 'power', 'price', 'salvage', 'savings\_amount', 'seller\_rating', 'sp\_id', 'sp\_name', 'theft\_title', 'torque', 'transmission', 'transmission\_display', 'trimId', 'trim\_name', 'vehicle\_damage\_category', 'wheel\_system', 'wheel\_system\_display', 'wheelbase', 'width', 'year'])

cars=df.na.fill({'body\_type':'Sedan','city':'all','daysonmarket':220,'engine\_cylinders':'unknown','exterior\_color':'unknown','franchise\_dealer':False,'franchise\_make':'unknown','front\_legroom':39,'fuel\_tank\_volume': 13.2,'fuel\_type':'unknown','has\_accidents':False,'city\_fuel\_economy':15,'highway\_fuel\_economy':12,'isCab':False,'is\_new':False,'length':230,'make\_name':'unknown','maximum\_seating':5,'mileage':0,'model\_name':'unknown','price':'negotiable','seller\_rating':2,'width':73.2,'horsepower':145.0,'height':61})

cars.show(5)

basic=["price","make\_name","model\_name","city","seller\_rating"]

q1=cars.where("price<=100000 and city='Howell'").select(basic).orderBy(desc("seller\_rating")).show(10)

q2=cars.where("make\_name='BMW'or make\_name='Ford'").select("make\_name","model\_name","body\_type","is\_new","price","seller\_rating").show(20)

q3=cars.select("body\_type").distinct().show(5)

q3=cars.filter("make\_name='Ford'").select("model\_name").distinct().show(10)

q4=cars.filter("transmission\_display='Automatic' and seller\_rating=5.0").select("make\_name","model\_name","body\_type","transmission\_display","horsepower","price","seller\_rating").sort(desc("seller\_rating")).show()

q5=cars.filter("make\_name='BMW' and model\_name='X3'").select(basic).orderBy(desc("seller\_rating")).show(10)

q6=cars.filter("make\_name='Ford' and model\_name='Fusion'").select("price","make\_name","model\_name","city","seller\_rating","daysonmarket").orderBy(asc("daysonmarket")).show(10)

**Pyspark Data Visualization code:**

df\_heat=cars.sample(False,0.0001,63).toPandas()

df\_heat.head(2)

df\_heat['daysonmarket']= pd.to\_numeric(df\_heat['daysonmarket'])

import seaborn as sns

plt.figure(figsize=(15,5), dpi=100)

sns.barplot(x = 'body\_type',y = 'daysonmarket',data = df\_heat,palette = "Blues")

plt.title('Body type vs Days on Market', fontsize = 14)

plt.xlabel('Body types')

plt.ylabel('Days on Market')

plt.show()

import seaborn as sns

cars = ['FORD', 'CHEVROLET', 'HONDA', 'TOYOTA', 'NISSAN', 'OTHERS']

data = [25, 18, 16, 16, 12, 213]

plt.figure(figsize=(15,10), dpi=100)

plt.pie(data, labels = cars,autopct='%1.1f%%')

plt.title(' COMPANY NAMES')

plt.show()

import seaborn as sns

plt.figure(figsize=(5,6), dpi=100)

sns.barplot(x ='city\_fuel\_economy',y='body\_type',data = df\_heat,palette = "Blues")

plt.title('car body type and its mgc')

plt.xlabel('Mileage')

plt.ylabel('Body type')

plt.show()