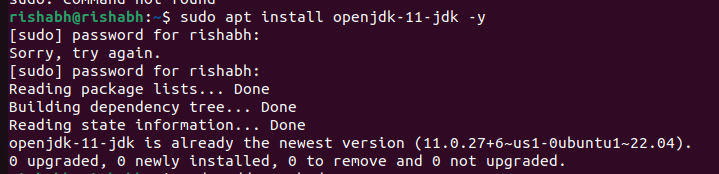
# **Using PySpark for Memory-Efficient Processing of NYC Yellow Taxi Data (5GB+)**

## Objective

This task demonstrates how to efficiently process a 5GB+ NYC Yellow Taxi CSV dataset locally using PySpark, without causing memory crashes. The goal was to perform exploratory data analysis (EDA) and transformations in a low-memory virtual machine environment using Spark’s distributed processing capabilities.

## Environment Setup on Ubuntu 22.04 LTS

* Install Java 11:  
   sudo apt install openjdk-11-jdk -y



* Set JAVA\_HOME:  
   echo 'export JAVA\_HOME=/usr/lib/jvm/java-11-openjdk-amd64' >> ~/.bashrc  
   echo 'export PATH=$JAVA\_HOME/bin:$PATH' >> ~/.bashrc  
   source ~/.bashrc
* Install Python and PySpark:  
   sudo apt install python3 python3-pip -y  
   pip3 install pyspark
* Verify Installation:  
   java -version  
   python3 -c "import pyspark; print('PySpark is ready!')"

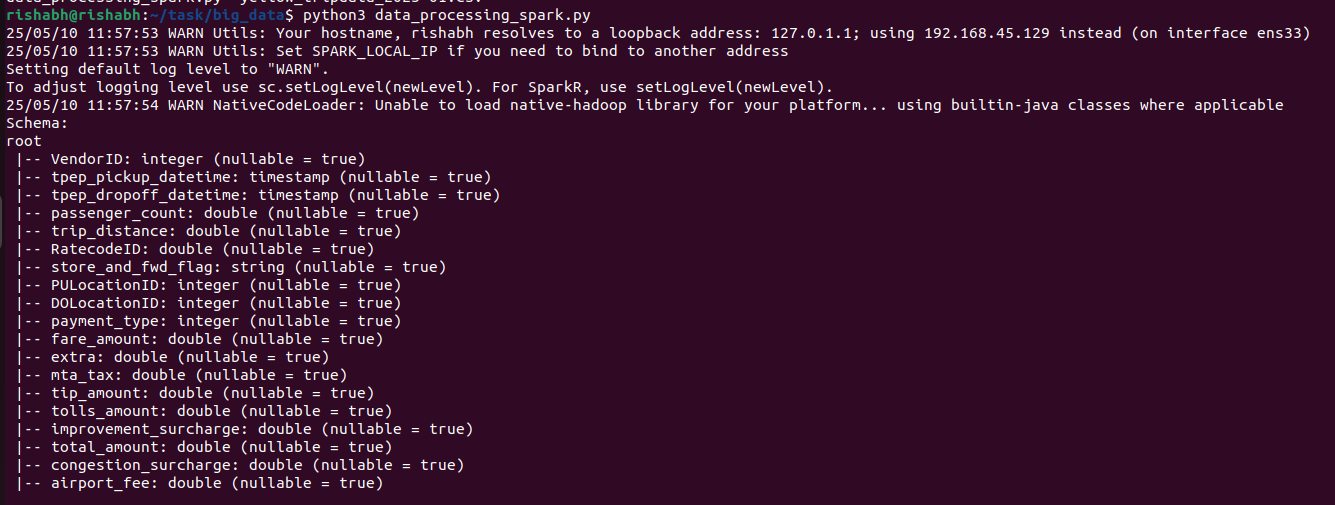
## Dataset Details

• Source: NYC TLC Trip Record Data  
• File: yellow\_tripdata\_2023-01.csv  
• Size: 5.1 GB (CSV format)  
• Rows: ~7 million taxi trip records  
• Converted from Parquet using pandas:

import pandas as pd  
df = pd.read\_parquet('yellow\_tripdata\_2023-01.parquet')  
df.to\_csv('yellow\_tripdata\_2023-01.csv', index=False)

## **Data Loading with PySpark**

PySpark does not immediately load the full data into memory, making it ideal for large datasets. I created a SparkSession and read the dataset using spark.read.csv:

from pyspark.sql import SparkSession  
  
spark = SparkSession.builder \  
 .appName("YellowTaxiTripDataProcessing") \  
 .config("spark.driver.memory", "4g") \  
 .getOrCreate()  
  
df = spark.read.csv("yellow\_tripdata\_2023-01.csv", header=True, inferSchema=True)

## **Exploratory Data Analysis**

### **Schema Inspection**

df.printSchema()  
df.show(5)

### **Total Trip Count**

df.count()

### **Average Trip Distance**

df.selectExpr("avg(trip\_distance)").first()[0]

### **Filter Trips Over 10 Miles**

long\_trips = df.filter(df.trip\_distance > 10)  
long\_trips.count()

### **Trip Count by Payment Type**

df.groupBy("payment\_type").count().show()

## 

## **Output Export**

Trips over 10 miles were exported into CSV files using:  
long\_trips.write.csv("output/long\_trips\_10plus", header=True, mode="overwrite")

## **Key Learnings and Advantages**

* Lazy Evaluation: Avoids loading entire dataset into memory
* Distributed Computation: Efficient on low-RAM VMs
* inferSchema=True: Automatically detects column data types
* No .collect() Used: Prevents memory crashes
* .write.csv(): Splits output into parallelized chunks

## Summary

Using PySpark, I successfully processed a 5GB+ NYC taxi dataset on a modest virtual machine, performed essential EDA, and exported filtered results — all without any memory errors. This project demonstrates how PySpark enables large-scale processing on low-resource systems by leveraging lazy execution and distributed task scheduling.