Analysis of Parking Violations in New York City



Justin Nichols: [jnich56@lsu.edu](mailto:jnich56@lsu.edu)

Bryce Lee: [blee57@lsu.edu](mailto:blee57@lsu.edu)

CSC 4740 Final Project

30 April 2022

# Table of Contents

[Table of Contents](#_h8rj5frmyyk9)

[Introduction](#_u0kg9rg9fm2e)

[Motivation](#_crw27ijxscvb)

[Project Description](#_ekothn2ns9d9)

[Division of Roles](#_11mmw4oxj19f)

[System Design](#_pevlf646eybu)

[Big Data Frameworks](#_ouk89dtafs4r)

[Dataset](#_3oyflmactfo)

[Detailed Description of Components](#_mhscw0e6ayhi)

[Hadoop MapReduce Driver](#_6gfewi93gn9t)

[Hadoop MapReduce Partitioner](#_92svat9ejbrv)

[Hadoop MapReduce Mapper](#_xzcumojd3b9m)

[Hadoop MapReduce Reducer](#_v40uphjv6xrl)

[PySpark](#_q4762mcqh7i4)

[Software Manual](#_u8buz95ac724)

[General Instructions](#_xowl2nln3fcn)

[Hadoop MapReduce](#_6jm7e1gt6r5q)

[PySpark](#_uj2was7s66vr)

[Test Environment](#_67qggj9uk1jy)

[Test Results](#_i5olbcdf1ftb)

[registrationState](#_q6vsqsqoqg7b)

[plateType](#_phhzcujb2tny)

[vehicleBodyType](#_4t075o9sw9z5)

[vehicleMake](#_oj4yofw6bj4d)

[vehicleColor](#_gy4llk439ixc)

[vehicleYear](#_w68fqanig764)

[Comparing Hadoop MapReduce and PySpark Development Process](#_xk8la7ekey10)

[Comparing Hadoop MapReduce and PySpark Performance](#_yhmgy68i0fd9)

[Screenshots](#_wfzs9xxnk4c)

[Hadoop MapReduce](#_v35wbtjr24bo)

[PySpark](#_qztq44m5p4p6)

[Conclusion](#_8qh93v10fzjt)

[Appendix](#_bceldi1qhu76)

# Introduction

## Motivation

In many major cities in the United States, selecting a parking spot is a decision that can have costly financial implications. Cities are notorious for having obscured parking laws that are designed to profit off those who unknowingly break them. In 2015, New York City collected $565 million in traffic and parking penalties. Considering most parking tickets in NYC are given out in high quantities and cost over $100, this big data application is aimed at analyzing how people can reduce their chances of getting fined. The real-world value associated with our analysis is knowing which types of vehicles receive the costliest parking violations and are more likely to be ticketed when driving in and around a city. This could be useful for people traveling to a city as well as for local city governments wanting to know the types of vehicles that more often violate parking laws.

## Project Description

The Parking Violations big data application is aimed at identifying certain correlations between which types of vehicles (based on the registration state, plate type, vehicle body type, vehicle make, vehicle color, or vehicle year) received the costliest violations depending on the season (summer, fall, winter, spring). The project analyzes if, for example, commercial vehicles are more likely to be ticketed more frequently and higher during the winter holidays than at other times of the year due to the mass amounts of goods and packages being delivered. Another example includes, if cars more suitable to drive in warmer months, such as sports cars, convertibles, or motorcycles, receive costlier violations in the summer compared to those driven in cooler months, such as traditional sedans or SUVs. The application focuses on a 9GB dataset containing parking violations issued by the NYC DMV and was implemented in both Java utilizing Hadoop MapReduce and in Python utilizing PySpark. Other than analyzing the data, our project was also directed towards comparing and contrasting not only the development process between Hadoop MapReduce and PySpark but also the performance.

## Division of Roles

During the timeline of the project, we have worked simultaneously, equally committing the same amount of work and time. Throughout the research and development process, we have successfully worked virtually and have consistently been meeting once a week while also keeping constant communication. It has also allowed us to identify aspects of the project that can be improved and to shift our focus accordingly. We have utilized pair programming to help streamline the development process. From finding the data to writing the code and report, we have both contributed equally and have kept an open collaborative environment to allow ideas to bounce off one another.

# System Design

## Big Data Frameworks

The project utilizes Apache Hadoop MapReduce and Apache Spark (PySpark interface) using the Cloudera virtual machine in pseudo-distributed mode. For code development, we used Eclipse for Hadoop MapReduce (Java) and a basic text editor and Linux terminal for PySpark (Python). We used Apache Hue to visually view and analyze our results.

## Dataset

The dataset contains parking violation data around the NYC metro area. There are a total of 51 comma-separated values for each entry in the dataset. We used the information regarding the vehicle body type, vehicle make, vehicle year, vehicle color, registration state, plate type, issue date, and violation code. The dataset is around 9GB and contains approximately 42.3 million entries. The dataset can be found at the link, <https://www.kaggle.com/new-york-city/nyc-parking-tickets>.

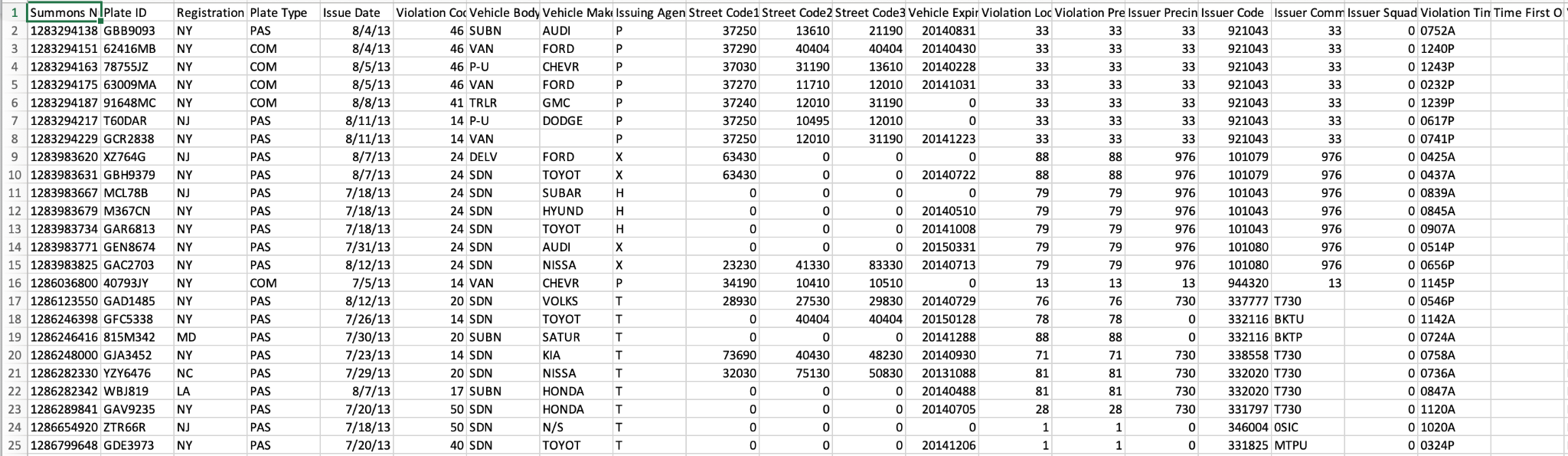
The dataset contains 4 separate files:

1. Parking\_Violations\_Issued\_-\_Fiscal\_Year\_2014\_\_August\_2013\_\_\_June\_2014\_.csv (1.87 GB)
2. Parking\_Violations\_Issued\_-\_Fiscal\_Year\_2015.csv (2.86 GB)
3. Parking\_Violations\_Issued\_-\_Fiscal\_Year\_2016.csv (2.15 GB)
4. Parking\_Violations\_Issued\_-\_Fiscal\_Year\_2017.csv (2.09 GB)

However, the dataset size was trimmed for submission purposes and contains the first 10,000 entries for each file, totaling 40,000 entries:

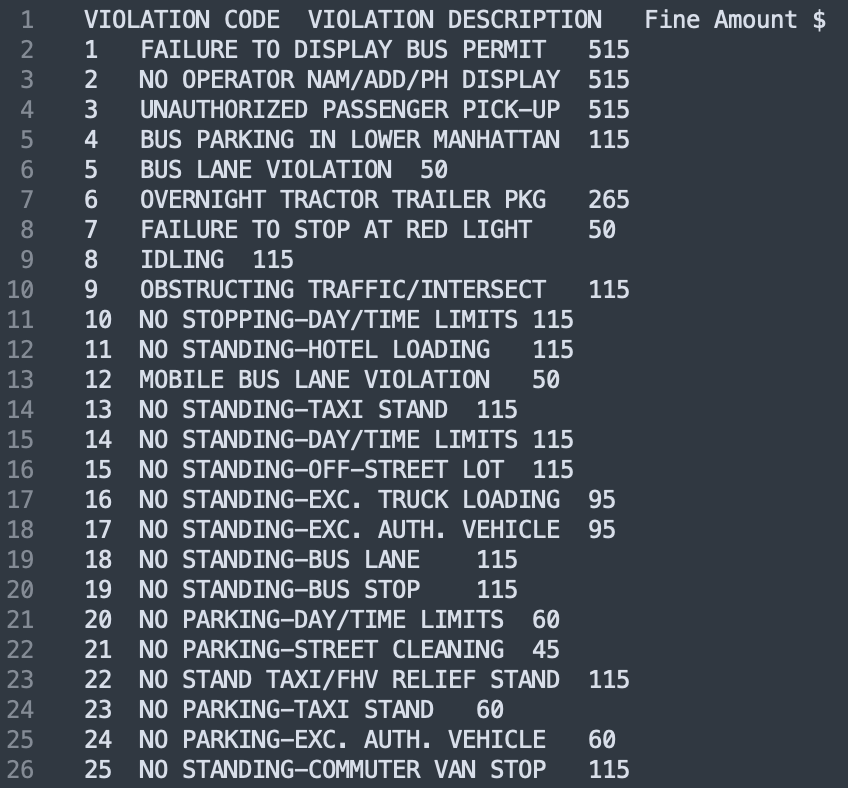
1. Parking\_Violations\_2013\_2014.csv (1.9 MB)
2. Parking\_Violations\_2015.csv (2.5 MB)
3. Parking\_Violations\_2016.csv (1.9 MB)
4. Parking\_Violations\_2017.csv (1.9 MB)

Below is a subset containing the first 25 entries from Parking\_Violations\_2013\_2014.csv.



It is important to note that Hadoop MapReduce and PySpark were tested using both the original dataset and the trimmed dataset. Besides the primary dataset, a file named ParkingViolationCodes.txt was used to translate the 98 NYC violation codes used in the dataset into a violation dollar amount. This file was used directly in the programs.

Below is a subset containing the first 25 entries from ParkingViolationCodes.txt.



Finally, 2 helper PDF files, regristration\_class\_body\_types\_colors.pdf and plate\_types\_state\_codes.pdf, provided by the NYC DMV were used to translate many of the abbreviations and parking technical codes. These files were not directly used in the programs but were used to manually decode the output files.

# Detailed Description of Components

## Hadoop MapReduce Driver

The primary functionality of the Driver is to set configurations and coordinate the execution of the MapReduce task. ToolRunner was added to the Driver to capture command-line arguments to modify the configuration of the MapReduce program on the fly. The command-line argument that the user can enter is called “taskType” and it specifies the parameter the user wants to use as the key during MapReduce. Along with this, LocalJobRunner was utilized in order to enable rapid development iterations, also known as Agile programming, within Eclipse. Besides linking the Mapper and Reducer classes to the Driver, a Partitioner class was also linked to enable its useful capabilities. The number of reduce tasks was set to 4, one for each season. Subsequently, we set the Mapper and Reducer output key and value type as Text. The total time the job takes to execute is also tracked.

## Hadoop MapReduce Partitioner

The primary functionality of the Partitioner is to partition data based on a specific parameter. In this case, the Partitioner parses out the season from each value passed from the Mapper and distributes it to one of the 4 Reducers using a “seasons” hashmap.

## Hadoop MapReduce Mapper

The primary functionality of the Mapper is to parse through each line in the dataset and output a key-value pair. The implementation of the Mapper in our project is separated into two parts, the setup method, and the map method.

In the setup method, the user-provided <taskType> value is validated and throws an IOException error if the input is invalid. If the user specifies a <taskType> value, it is checked that it equals a value held in the “options” array, that being either “registrationState”, “plateType”, “vehicleBodyType”, “vehicleMake”, “vehicleColor”, or “vehicleYear”. Otherwise, the “vehicleMake” is used as the default. Subsequently, the two hashmaps, “columns” and “month\_season”, are populated. The “columns” hashmap is used as an index to fetch the column data in each line entry and the “month\_season” hashmap is used for translating the date, specifically the month, into an equivalent season.

In the map method, the line is split using a regular expression that separates the input by commas since the data is in a CSV format. The commas correspond to individual columns within the dataset. Lines that have less than 35 columns (commas technically) are filtered out. Those that are in a valid format are indexed by column using the “columns” hashmap to store the relevant information, that being the vehicle body type, vehicle make, vehicle year, vehicle color, registration state, plate type, issue date, and violation code. The first line in each input file, which contains the legend, is filtered out. Then, depending on the <taskType> value, the Mapper filters out entries missing the column data for the specific <taskType>, however, allows other unnecessary columns to be missing. For example, if the <taskType> is “vehicleMake”, then the “vehicleMake” column must not be missing in the entry or it will be filtered out, however, if for instance the “vehicleColor” is missing, that is acceptable. Ultimately, entries missing the <taskType>, “issueDate”, and “violationCode” are filtered out considering they are vital and are used for emitting the key-value pair. The “issueDate” is parsed using the “Calendar” and “SimpleDateFormat” packages. The issue date month is extracted and converted into an equivalent season using the “month\_season” hashmap, otherwise, entries with misformatted issue dates are filtered out. Finally, the Mapper emits a key-value pair in the form (<taskType>, [season, violationCode]).

## Hadoop MapReduce Reducer

The primary function of the Reducer is to perform operations on the shuffled and sorted key-value pairs. The implementation of the Reducer in our project is separated into three parts, a helper method called violationInfo, used for extracting the violation information, the setup method, and the map method.

In the header method violationInfo, the ParkingViolationCodes.txt file, which contains the 98 parking violations and their equivalent dollar amounts, is parsed and stored in a hashmap called “violationInfo”. In the setup method, a legend is outputted at the top of each of the 4 output files.

In the reduce method, the violation code emitted from the Mapper is translated to a dollar amount that was gathered from the helper method. The number of values per key and the total dollar amount is also tracked. Keys with less than 5 values are filtered out. Then the average is computed by dividing the total dollar amount by the number of values per key. Finally, the Reducer emits a key-value pair in the form (<taskType>, [count, totalDollars, average]).

## PySpark

The primary functionality of PySpark is to filter, map, reduce, and partition the dataset. At its core, it performs the same operations as the Hadoop MapReduce implementation described above, however, it does it through the use of transformations using resilient distributed datasets (RDDs) and various actions. PySpark also utilizes lineage tracking, pipelined expressions, and lazy evaluation.

Similar to Hadoop MapReduce, the “seasons”, “columns”, and “month\_season” dictionaries are declared, along with another dictionary called “options” comparable to the “options” array in MapReduce. The <task\_type> command-line argument is checked for validity and defaults to “vehicleMake” if the argument is not passed or throws a ValueError if the value passed is invalid. The Parking\_ViolationCodes.txt files, which contain the 98 parking violations and their equivalent dollar amounts, are parsed and stored in a dictionary called “violationInfo”.

Using the “columns” dictionary, the lines in the dataset files are then stored in an RDD, split by commas, and filtered. This process involves chained transformations. The filtered data includes removing the legend at the top of each input file and columns with missing or misformatted data. Similar to Hadoop MapReduce, lines with less than 35 columns (commas technically) or lines missing the task type, issue date, or violation code columns are filtered out. Review the [Hadoop MapReduce Mapper](#_xzcumojd3b9m) filtering process description for more details. Lines with a misformatted issue date or invalid violation code are also filtered out. This process uses the filter function which is a one-to-one transformation.

The filtered data is then transformed into a pair RDD, where the key is in the form ([season, task\_type], violation\_code). The season is obtained with the help of indexing the issue date column using the “seasons”, “month\_season”, and “columns” dictionaries. The “task\_type” data is obtained with the help of indexing the specific column the “task\_type” equals (default is “vehicleMake”) and uses the “options” dictionaries to help. The “violation\_code” is obtained with the help of indexing the violation code column using the “columns” dictionary. This process uses the map function which is a one-to-one transformation.

The pair RDD is then broadcasted to a Python list using the “countsByKey” function which counts how many values each key contains. Similar to the [Hadoop MapReduce Reducer](#_v40uphjv6xrl), all keys with less than 5 values are discarded. The pair RDD is adjusted to be in the form ([season, task\_type], violation\_amount). The RDD is then reduced by the key and the average is found by adding up the total dollar amount and dividing it by the number of values. A new pair RDD is created using the form ([season, task\_type], [count, total\_dollars, average]). The pair RDD is adjusted again to prepare for partitioning using the form (season, [task\_type, violation\_amount]). The pair RDD is sorted by key and partitioned by season, emitting 1 output file per season. The result is written to the output files. The pair RDD that is outputted is in the form (<task\_type>, [count, total\_dollars, average]). The total time the job takes to execute is also tracked.

## Software Manual

### General Instructions

1. Open up the Cloudera-Training-CAPSpark-Student-VM-cdh5.4.3a-vmware virtual machine in VMWare. Download VMWare using this [link](https://www.vmware.com/products/workstation-player/workstation-player-evaluation.html) and the VM using this [link](https://lsumail2-my.sharepoint.com/:u:/g/personal/klee76_lsu_edu/EaKfXeEzunNAt-OXW7Lhy4MByuJ_gP1Hxxw41N5pXZ15bg?e=R1xfI7).
2. Copy and paste the ParkingViolations.zip deliverables file in the /home/training directory on the VM.
3. Extract the folder from the zip file.
4. Open a terminal window and use the command:

hdfs dfs -put ParkingViolations/Data

This will put the CSV files containing the data in the Hadoop Distributed File System.

Note, the CSV files included in the deliverables are smaller versions of the original files, since the VM has, by default, limited disk space.

### Hadoop MapReduce

1. To execute the MapReduce task, use the command:

hadoop jar ParkingViolations/Code/ParkingViolations.jar parking\_violations.ParkingViolationsDriver -D taskType=<taskType> Data output

Where <taskType> is one of the options: “registrationState”, “plateType”, “vehicleBodyType”, “vehicleMake”, “vehicleColor”, “vehicleYear”

1. When the MapReduce task completes, open up Firefox on the VM and click the Hue bookmark at the top.
2. Click the File System tab, then view the output directory to see the 4 output files, part-r-00000, part-r-00001, part-r-00002, part-r-00003, and part-r-00004.

### PySpark

1. To execute the PySpark task, use the command:

spark-submit ParkingViolations/Code/parking\_violations.py Data <task\_type>

Where <task\_type> is one of the options: registrationState, plateType, vehicleBodyType, vehicleMake, vehicleColor, vehicleYear

1. When the PySpark task completes, open up Firefox on the VM and click the Hue bookmark at the top.
2. Click the File System tab, then view the output2 directory to see the 4 output files, part-r-00000, part-r-00001, part-r-00002, part-r-00003, and part-r-00004.

## Test Environment

The cluster configuration we used was the pseudo-distributed environment in the Cloudera-Training-CAPSpark-Student-VM-cdh5.4.3a-vmware VM.

## Test Results

For evaluation, documents provided by the NYC government helped decode the various abbreviations and codes throughout the data into tangible meanings. The provided documents included regristration\_class\_body\_types\_colors.pdf and plate\_types\_state\_codes.pdf and helped translate the registration state, body type, colors, plate types, and state codes. Although the output files generated were sorted by task type, we also wanted to sort by the count, total dollars, and the average to help make evaluation easier. To do this, a simple python script, analyze.py, was used to sort the output files accordingly.

Overall, most vehicles, regardless of specific attributes, were ticketed higher in winter and spring than compared to the summer and fall. This was a trend throughout all the years the data covered, from 2013 to 2017.

### registrationState

Based on the total dollars collected and the total number of violations issued, the states with the greatest total dollar amounts and number of violations issued were those in close proximity to New York City, that being New York (NY), New Jersey (NJ), Pennsylvania (PA), and Connecticut (CT). Subsequently, states located in the Northeastern region of the United States also had greater total dollar amounts and number of violations issued, that being Massachusetts (MA), Virginia (VA), and Maryland (MD). Consistently, the states in the top 15 included Florida (FL), Indiana (IN), North Carolina (NC), Illinois (IL), Georgia (GA), Arizona (AZ), and Texas (TX). Beyond that, the states were roughly ordered based on their population combined with their proximity to New York. These trends were apparent for all seasons.

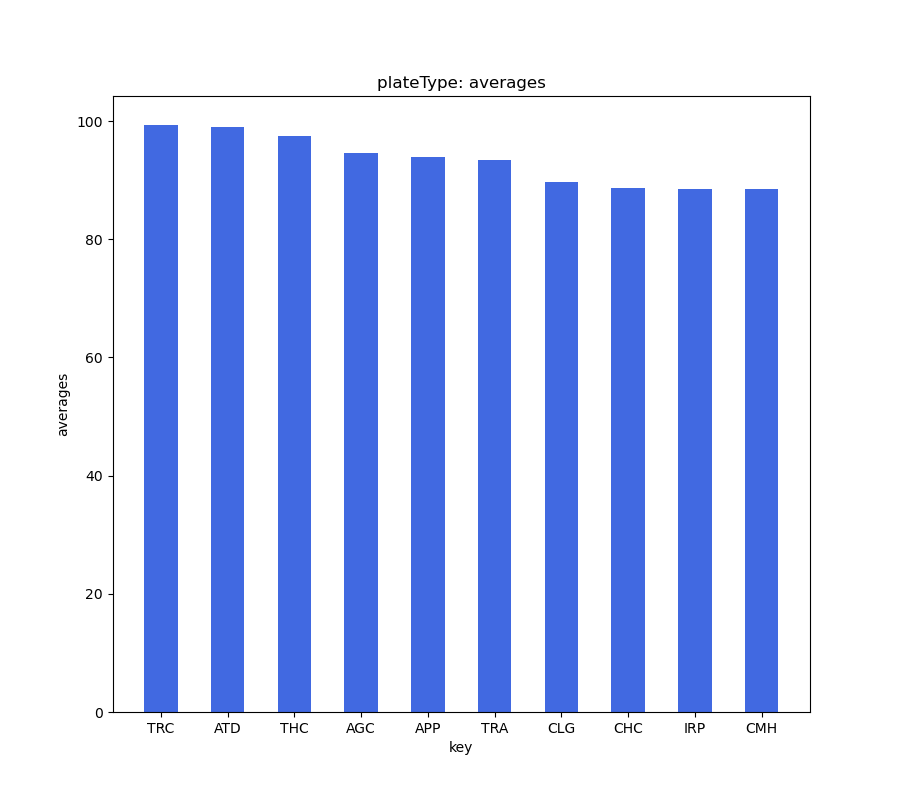
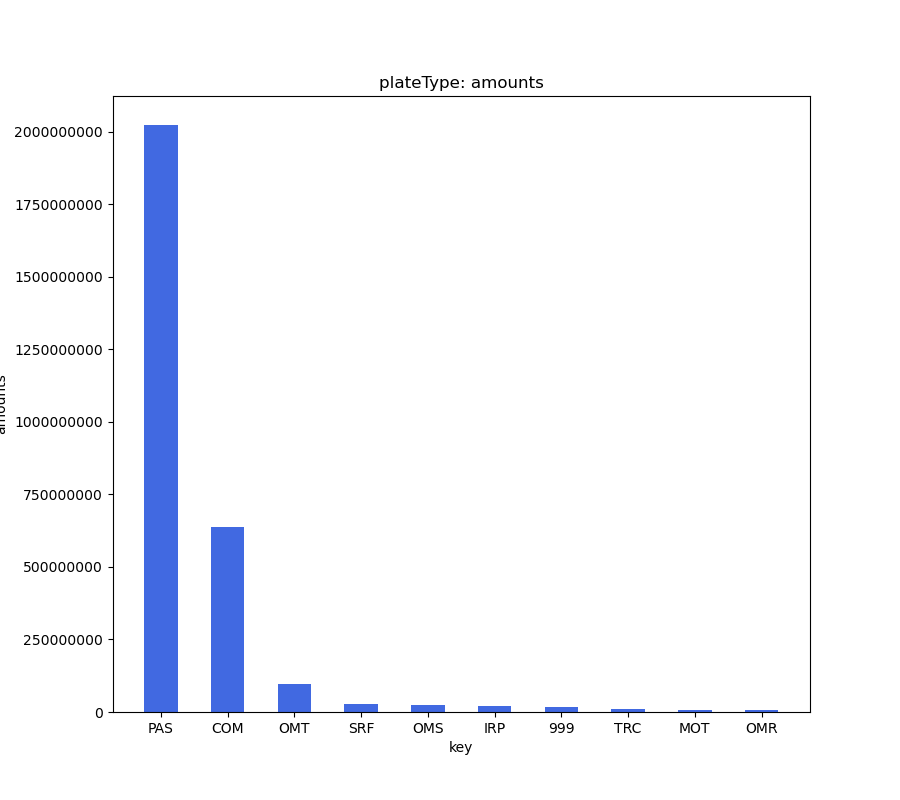
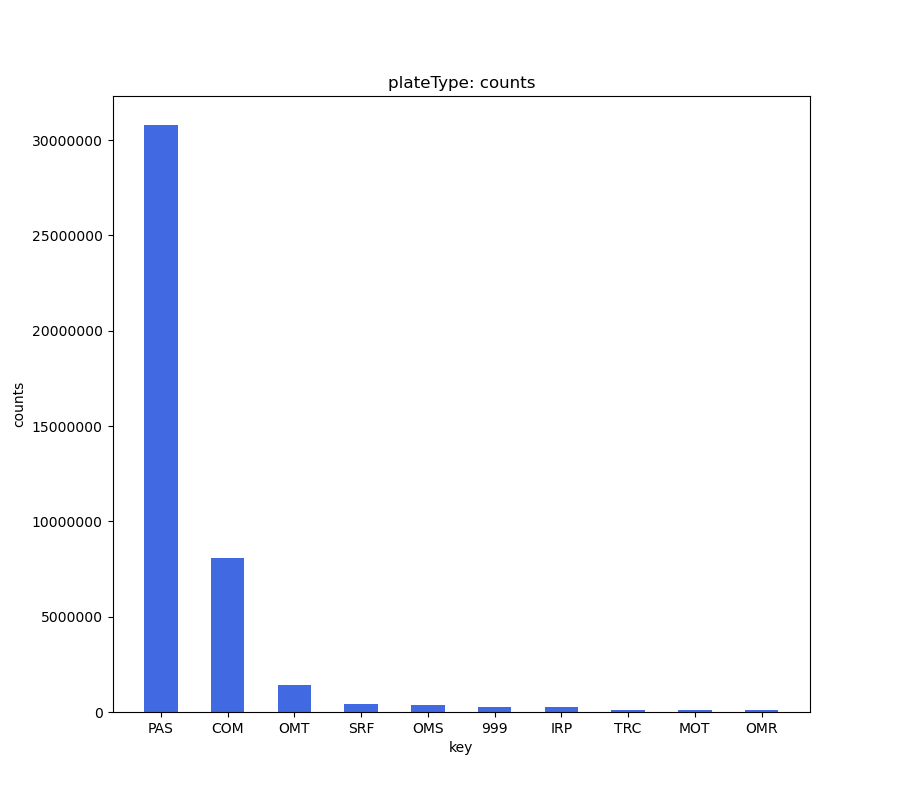
However, on average, the priciest parking violations were given to U.S. Government (GV) and U.S. State Department (DP) plates across all seasons. Subsequently, states that received the costliest violations were Minnesota (MN), Indiana (IN), Oklahoma (OK), New Jersey (NJ), and Idaho (ID). Foreign plates (FO), including Canadian provinces and Mexican states, also received high fees. Canadian provinces and territories that received costly violations included New Brunswick (NB), British Columbia (BC), Quebec (QC), Ontario (ON), Saskatchewan (SK), Manitoba (MB), and Yukon (YK). One Mexican state, Mexico (MX) was also ticketed higher.

### 

### plateType

Based on the total dollars collected and the total number of violations issued, passenger (PAS), commercial (COM), taxi (OMT), specialized passenger (SRF), international (IRP), rental (OMS), motorcycle (MOT), tractor (TRC), and bus (OMR) vehicles had greater total dollar amounts and number of violations issued. These trends were apparent for all seasons.

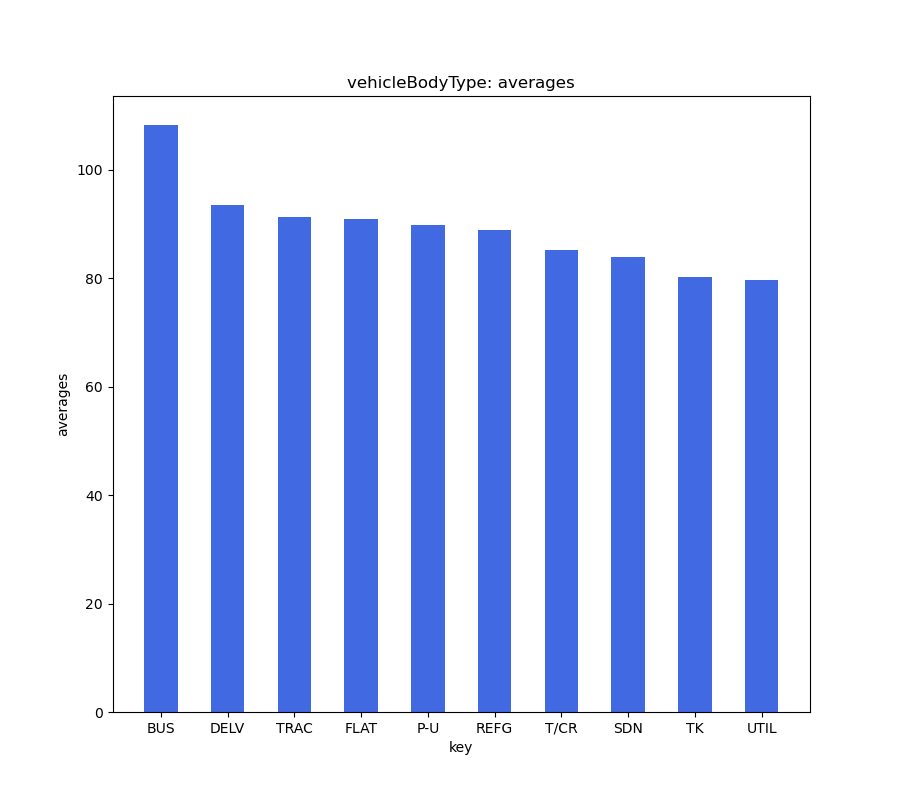
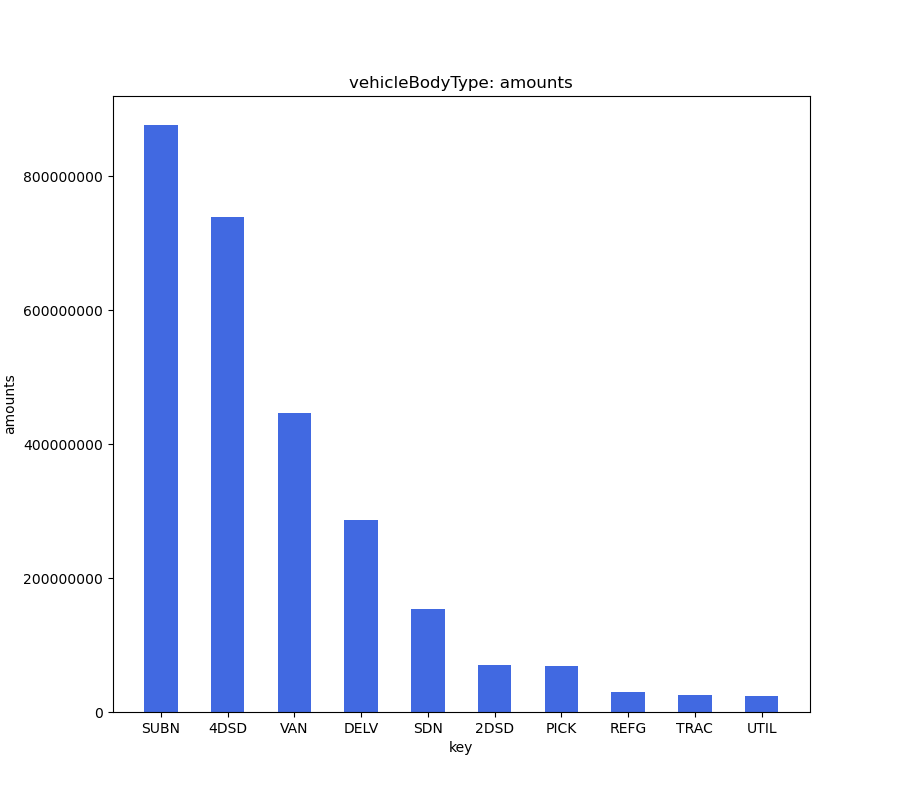
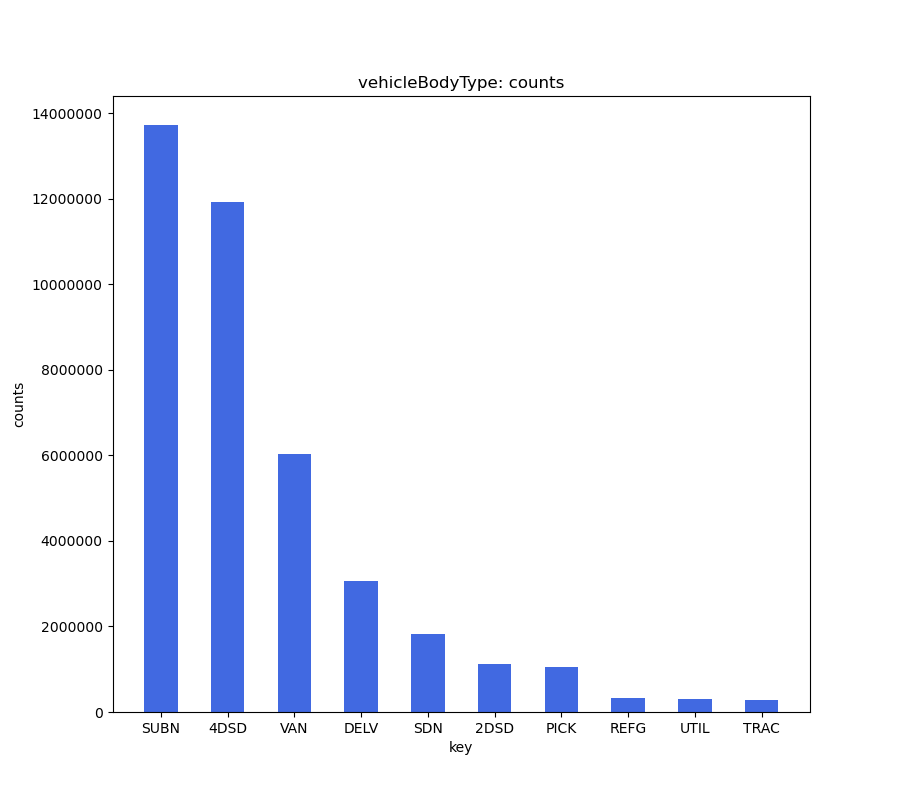
However, on average, the more common plate types, such as passenger and commercial, received among the cheapest violations. The highest violations on average were given to tractors (TRC, THC), all terrain deales (ATD), amateur radio vehicles (HAM), and farm vehicles (FAR). Government plate types, including New York City Council (NYC), Congress Medal of Honor (CMH), county legislators (CLG), governor’s additional car (GAC), and U.S. Congress (USC) were also on average ticketed higher. In the spring and summer, omnibuses (OMO) were ticketed higher than in the cooler months. In the fall, historical motorcycle (HSM) plates were abnormally high.



### vehicleBodyType

Based on the total dollars collected and the total number of violations issued, the vehicle body types with the costliest and highest number of tickets included suburbans (SUBN) and 4 door sedans (4DSD). Subsequently, vans (VAN), delivery vehicles (DELV), sedans (SDN), pick-up trucks (PICK), 2 door sedans (2DSD), refrigerated trailers (REFG), tractors (TRAC), utility vehicles (UTIL), taxis (TAXI), buses (BUS), and convertibles (CONV) also had greater total dollar amounts and number of violations issued. These trends were apparent for all seasons.

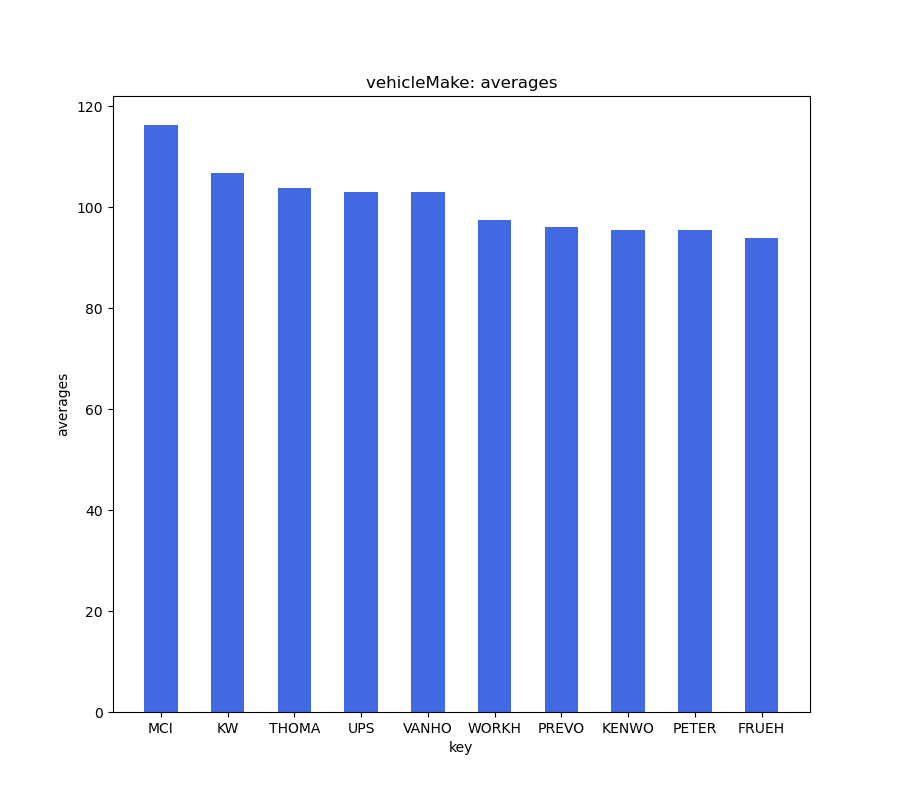
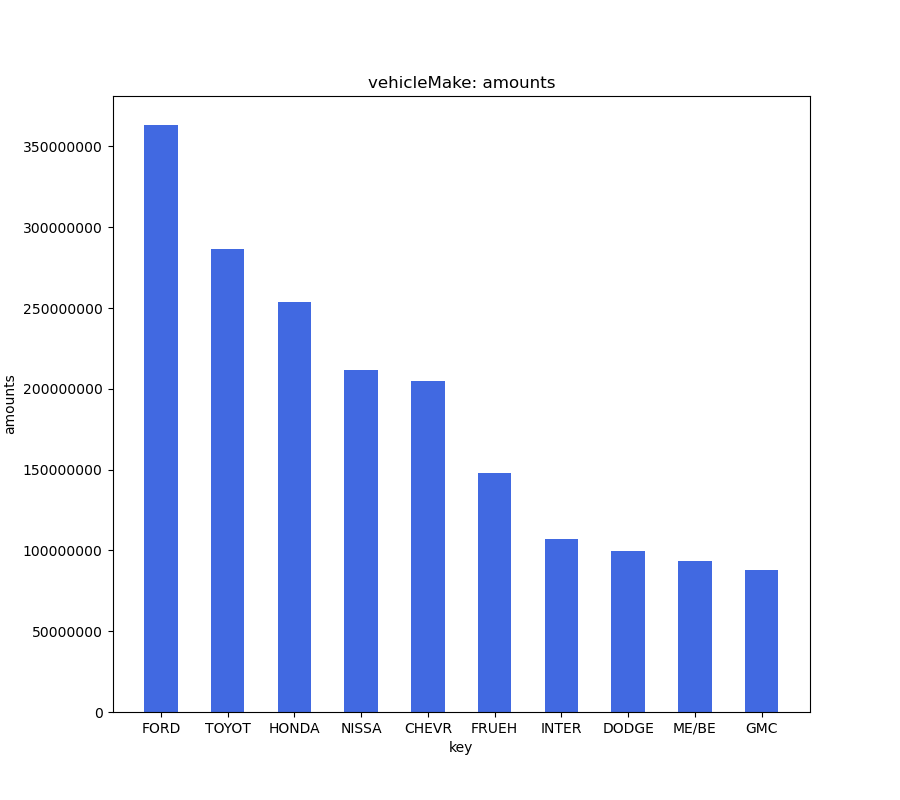
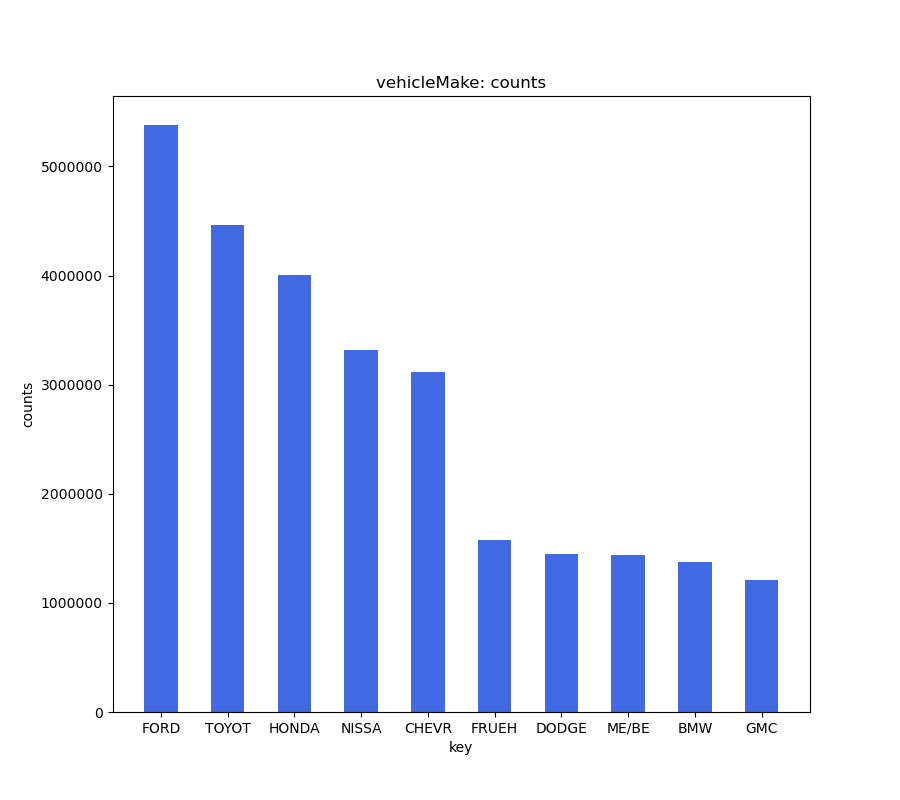
However, on average, more heavy duty vehicles, such as tractors (TRC, TRAC), buses (BUS), delivery vehicles (DELV), trucks (TRUC), flatbed trucks (FLAT), and limousines (LIM), were ticketed higher in the winter than any other season. For example, buses (BUS) were ticketed the highest in the winter and spring compared to the summer and fall. Subsequently, flatbed hitch trailers (TL) on average were ticketed higher in the spring and summer than in the fall and winter.



### vehicleMake

Based on the total dollars collected and the total number of violations issued, the vehicle makes with the costliest and highest number of tickets included Ford (FORD), Toyota (TOYOT), Honda (HONDA), Chevrolet (CHEVR), and Nissan (NISSA). Subsequently, Fruehauf (FRUEH), Dodge (DODGE), Mercedes Benz (ME/BE), BMW (BMW), GMC (GMC), Jeep (JEEP), and Hyundai (HYUND) vehicles also had greater total dollar amounts and number of violations issued. These trends were apparent for all seasons.

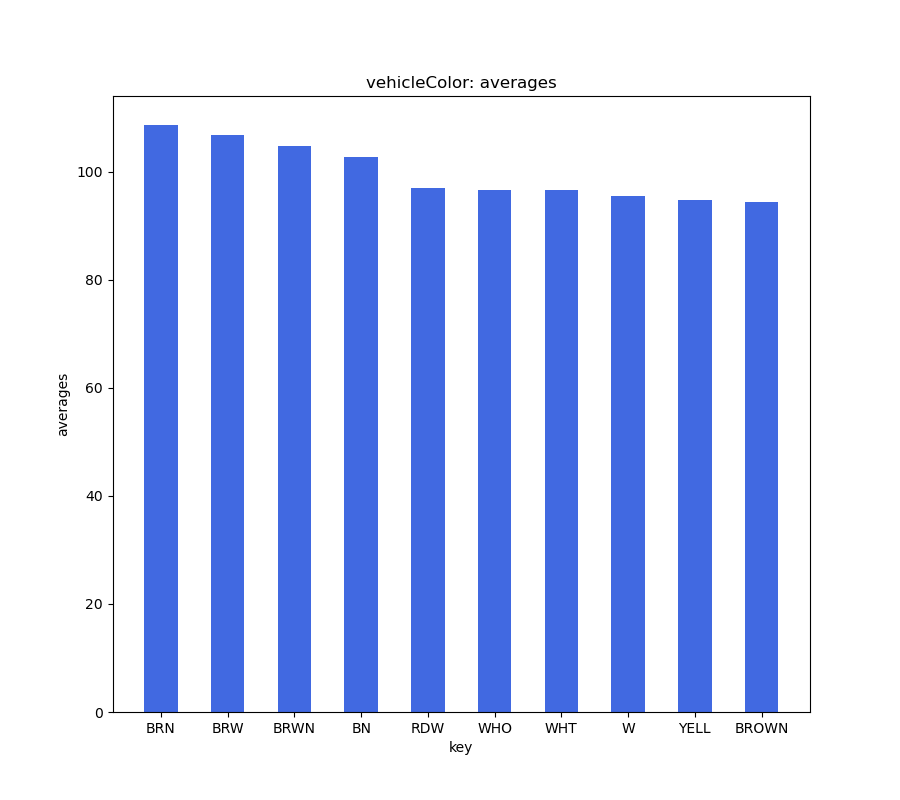
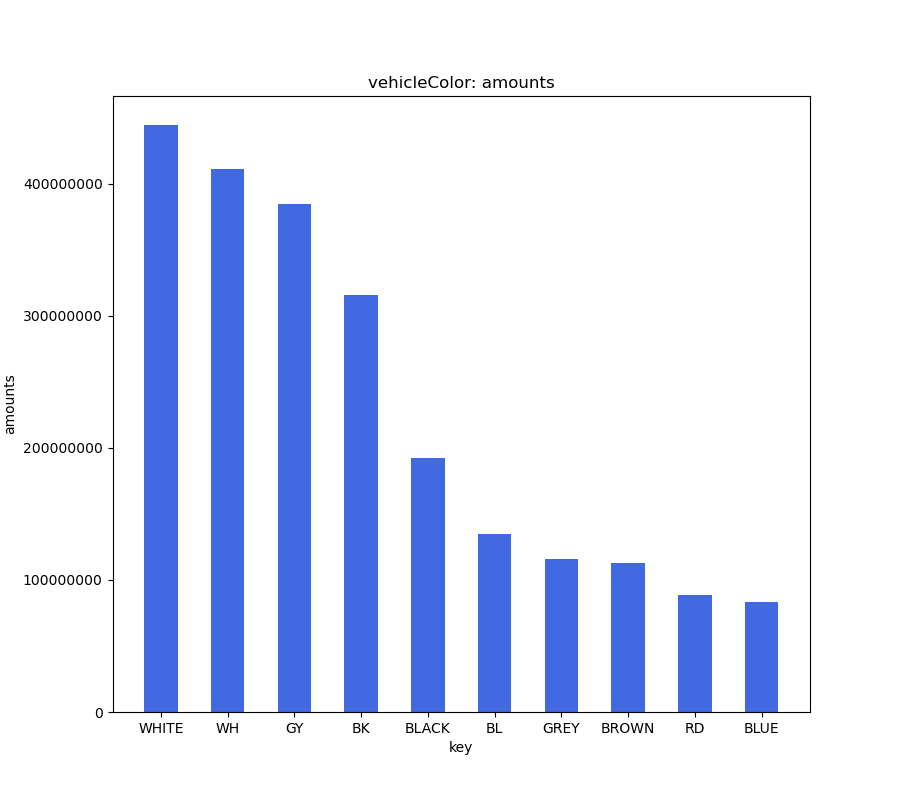
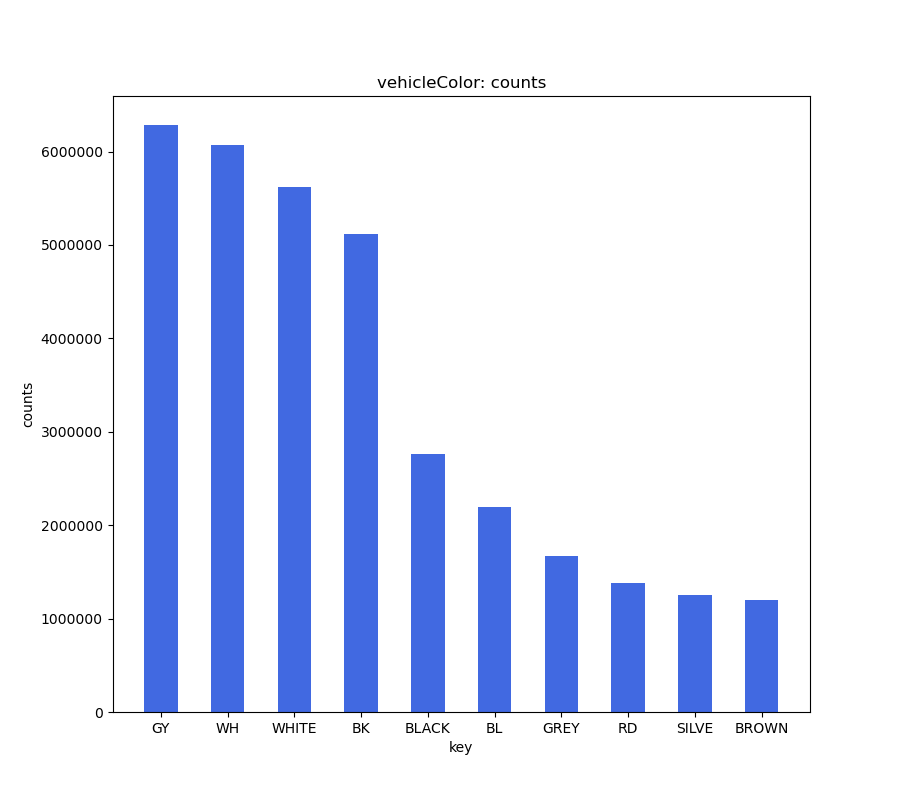
However, on average, vehicle makes that are more geared for the commercial market were ticketed higher. Commercial vehicles such as Motor Coach Industries (MCI), Van Hool (VANHOL), Prevost (PREVO), Workhorse (WORKH), Peterbilt (PETER), Kentwood (KW, KENWO), Fruehauf (FRUEH), Mack (MACK), Hino (HIN, HINO), UD (UK), and Isuzu (ISUZU) had high violation costs. In comparison most passenger vehicles featured average violation costs. American based brands such as GMC (GMC), Mercury (MERCU), Lincoln (MERCU), Dodge (DODGE), Ford (FORD), Cadillac (CADIL), Chevrolet (CHEVR), and Pontiac (PONTI) were on average ticketed higher than foreign made brands. However, on average, Mercedes Benz (ME/BE) were the highest ticketed German car brand compared to BMW (BMW), Volkswagen (VOLKS), and Audi (AUDI). Japanese and South Korean car brands were ticketed on average lower than both American and German car brands. These trends were apparent for all seasons.



### vehicleColor

Based on the total dollars collected and the total number of violations issued, the vehicle colors with the costliest and highest number of tickets included white (WHITE, WHT), gray (GY, GRY, GREY, GRAY), black (BK, BLACK, BLK), blue (BL, BLUE), and brown (BROWN, BR). Subsequently, red (RD, RED), green (GR, GREEN), silver (SILVE), yellow (YW, YELLO), tan (TN, TAN), and gold (GL, GOLD) vehicle colors also had greater total dollar amounts and number of violations issued. These trends were apparent for all seasons.

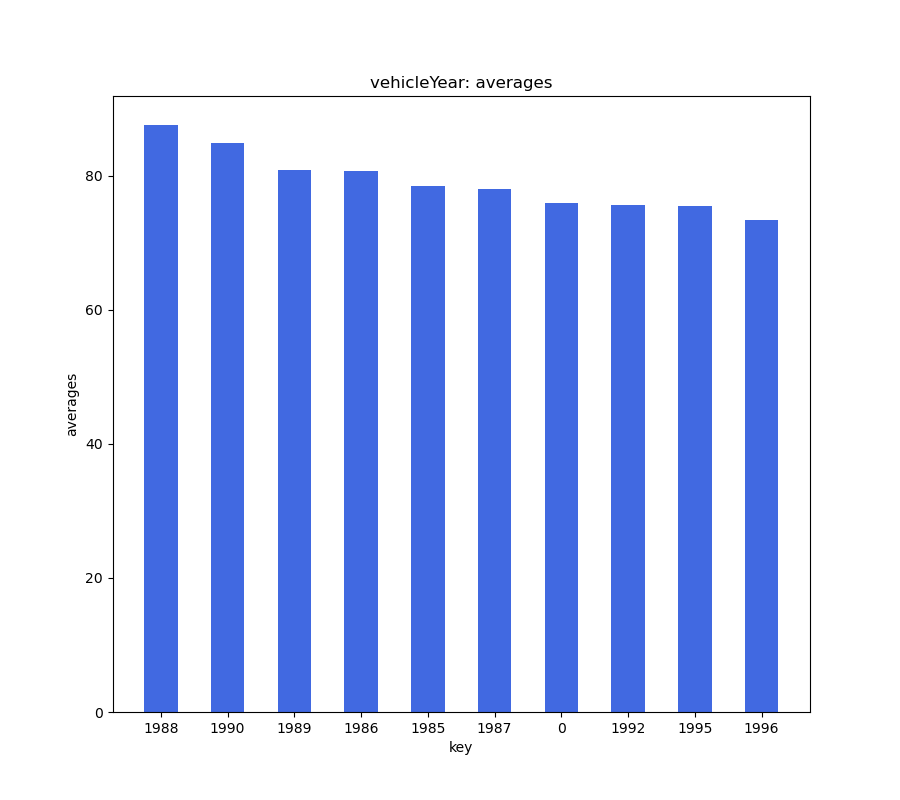
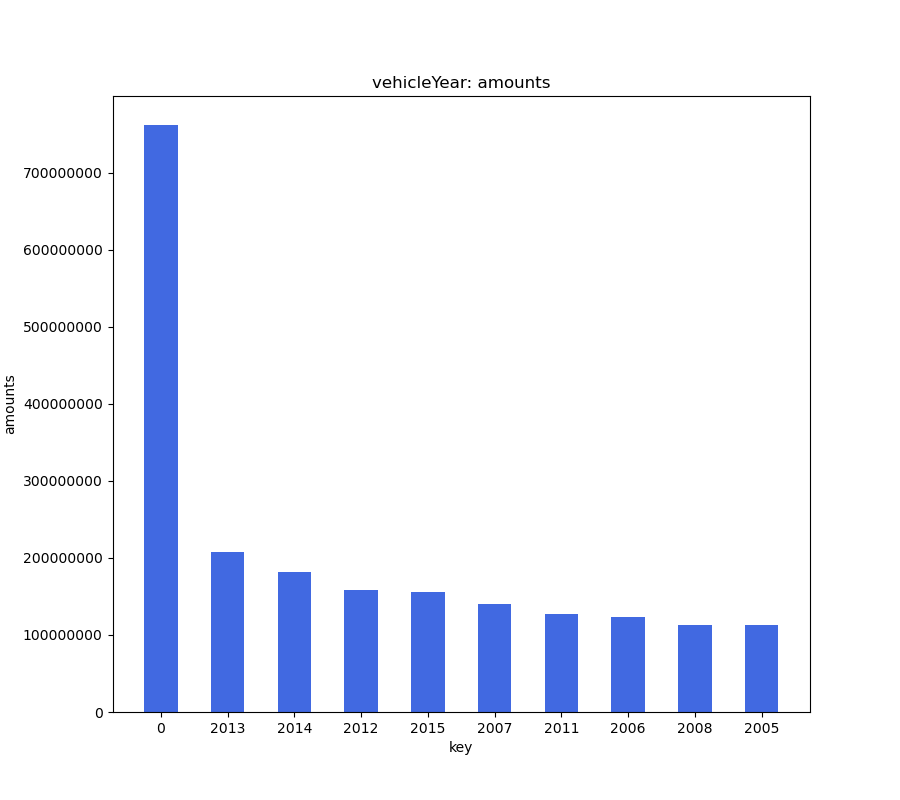
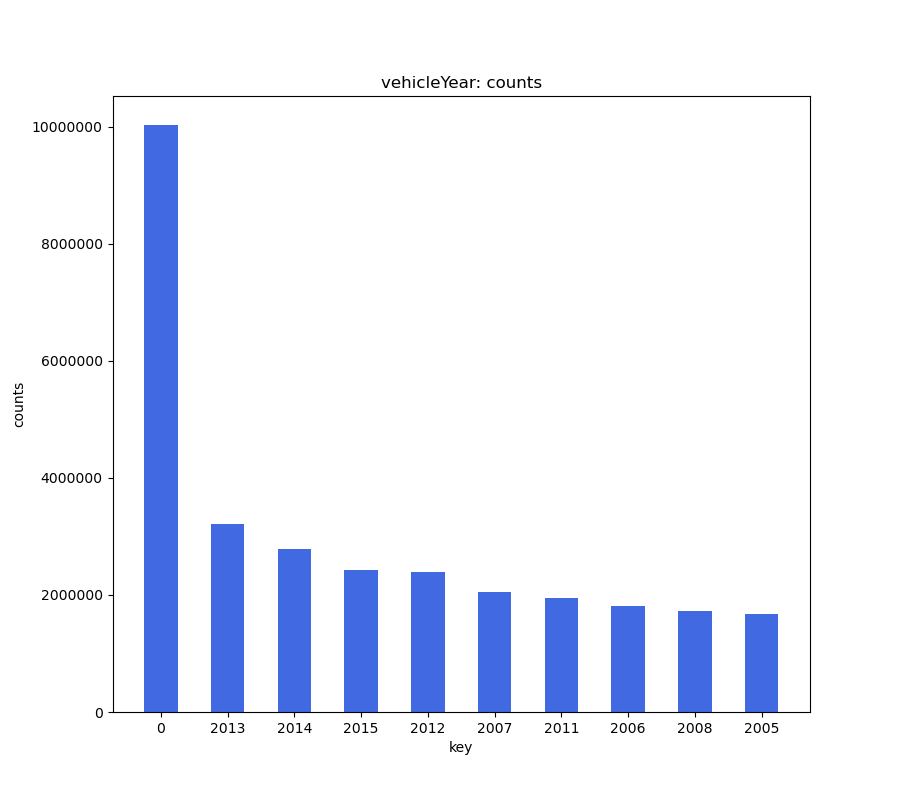
However, on average, abnormally colored vehicles, including brown (BRN, BN, BROWN), orange (ORANG), yellow (YELLO), green (GN, GRN, GREEN), and multi-colored (OTHER) were ticketed higher overall. Among the three most popular colors, on average, white (WHT, WHITE) vehicles were ticketed the highest followed by and black (BLK) vehicles then gray (GRAY, GRY) and silver (SILV, SILVR) vehicles. The shade of the color, or in other terms whether the color was light or dark, had no effect on the average ticket price. These trends were apparent for all seasons.



### vehicleYear

Based on the total dollars collected and the total number of violations issued, the vehicle years with the costliest and highest number of tickets included those from the 2000s and 2010s, with 2013 being the highest ticketed, followed by 2014, 2015, 2012, and a variety of other twenty-first century made vehicles. Vehicles from the 1990s, followed by the 1980 then 1970s all had fewer total dollars and total number of violations. These correlations reasonably make sense due to the fact that there are less vehicles on the road from previous decades. These trends were apparent for all seasons.

However, on average, newer vehicles (1998-2017) and vintage vehicles (1960-1984) received lower ticket prices than vehicles perceived as dated (1985-1997). Vehicles produced between the years 1985-1997, on average, correlated to $10-$20 higher ticket prices than vehicles made before or after those years. A possible explanation would be that older vehicles, not yet perceived as rare or vintage, generally result in a lower vehicle price. Perhaps these vehicles are targeted more by ticketing officials.



### Comparing Hadoop MapReduce and PySpark Development Process

Both Hadoop MapReduce and PySpark were developed using an Agile programming methodology, where code was rapidly developed and tested consistently. Hadoop MapReduce, however, was easier to develop due to having the ability to locally run it within Eclipse which enabled making quick changes for testing. For PySpark, this was not utilized and switching between a text editor and a terminal was common and tedious. Although PySpark had a more cumbersome development environment, the code itself was written fairly quickly due to the program only containing one file and the added ability to chain operations together. For Hadoop MapReduce, the code development lasted over a longer period of time due to there being 4 files that needed to be written. However, for error analysis, PySpark presented to be more challenging than Hadoop MapReduce. Debugging was a more streamlined process in Hadoop MapReduce because more useful error messages were provided compared to PySpark.

### Comparing Hadoop MapReduce and PySpark Performance

Both Hadoop MapReduce and PySpark performed fairly well. Both methods also generated the same outputs for the dataset. However, PySpark was generally faster at processing the dataset than Hadoop MapReduce. For comparison, when processing the entire dataset using the parameter “vehicleMake”, PySpark took 512.717 seconds while Hadoop MapReduce took 2631.267 seconds. When running the trimmed dataset using the parameter “vehicleMake”, PySpark took 17.093 seconds while Hadoop MapReduce took 84.32 seconds. This can be attributed to PySpark’s use of fast in-memory performance with reduced disk reading and writing operations compared to Hadoop MapReduce’s slower performance using disks for storage which negatively impacted the read and write speed. PySpark also keeps track of the different transformations and actions executed through the lineage graph. Lazy evaluation is also implemented for running jobs on PySpark, which reduces the amount of disk read and writes since data evaluation only occurs when an action is called. Hadoop MapReduce does not implement lazy evaluation and has to perform a significant amount of read and write operations to do the same task.

## Screenshots

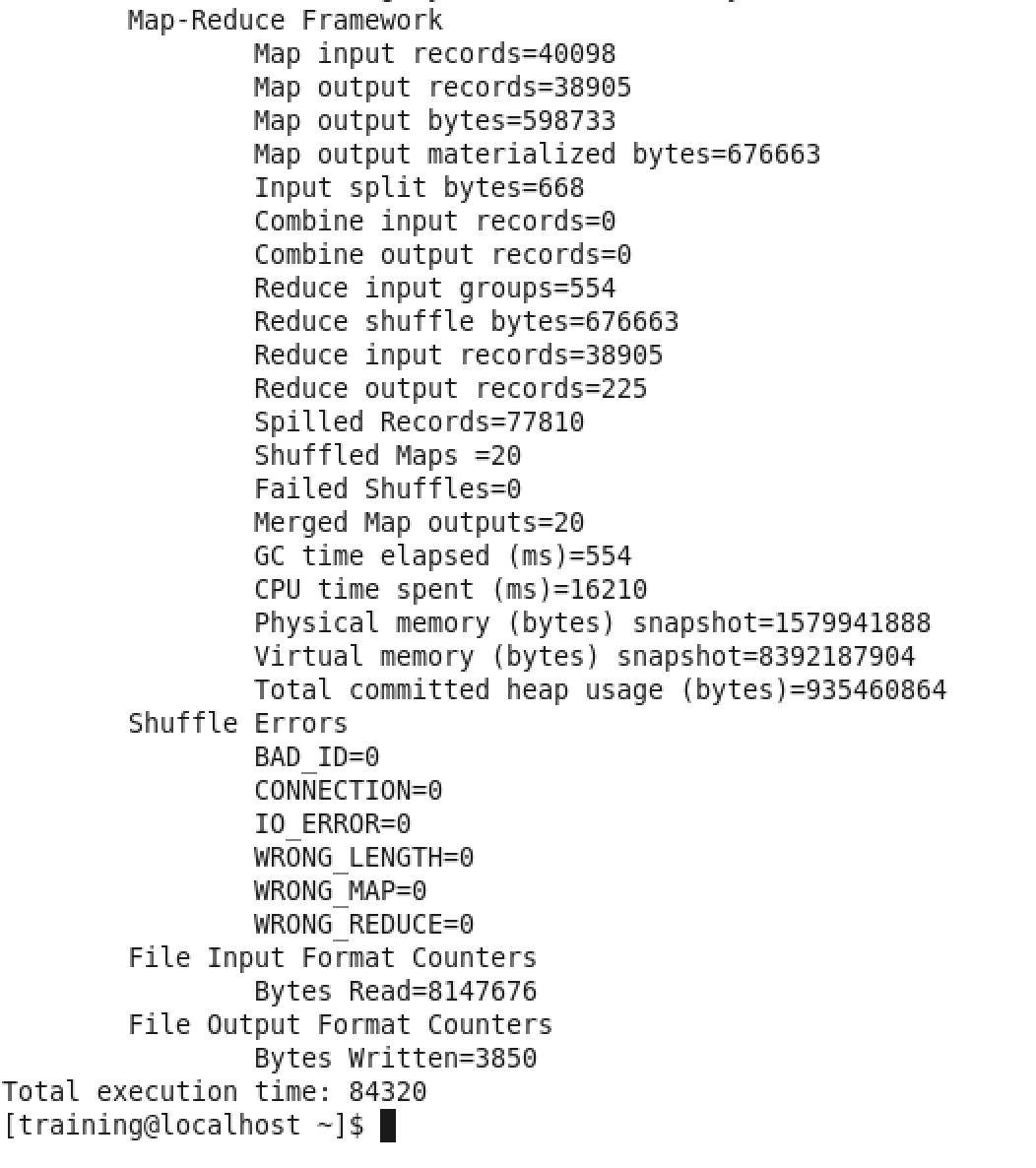
### Hadoop MapReduce

Below is an image of the terminal output after running the entire dataset in Hadoop MapReduce:



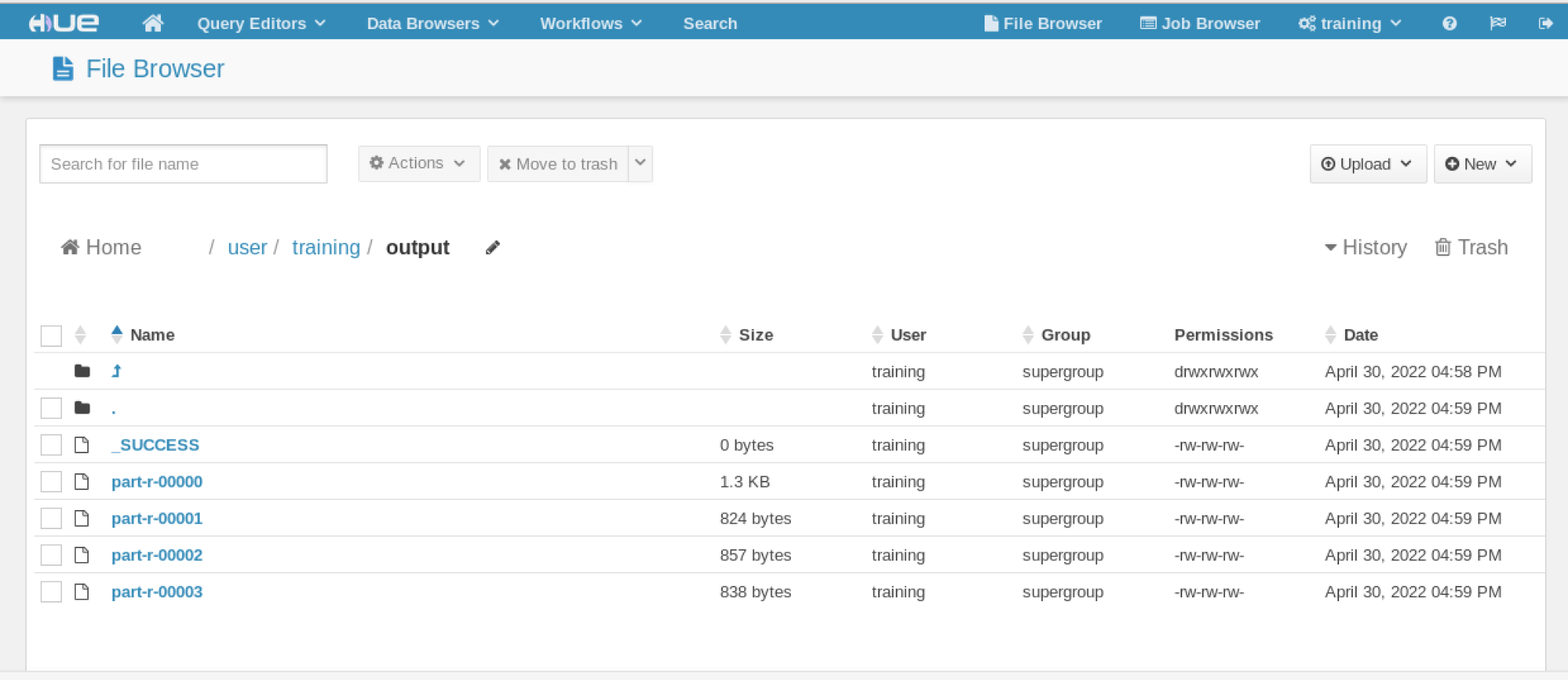
Total execution time is 2631.267 seconds or 2,631,267 milliseconds.

Below is an image of the terminal output after running the trimmed dataset in Hadoop MapReduce:



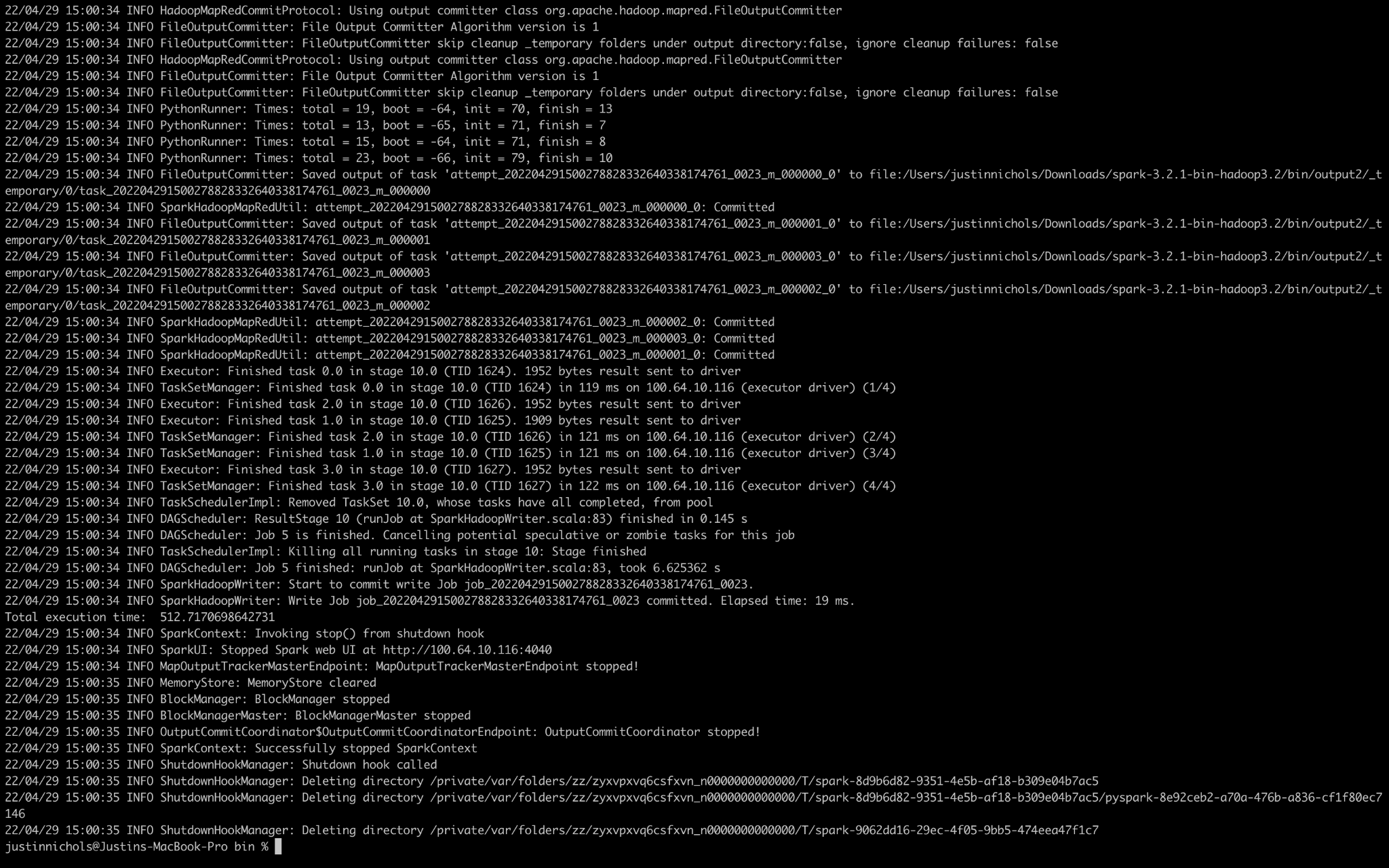
Total execution time is 84.32 seconds or 84,320 milliseconds.

Below is an image of the Hadoop MapReduce output files as shown in Hue:



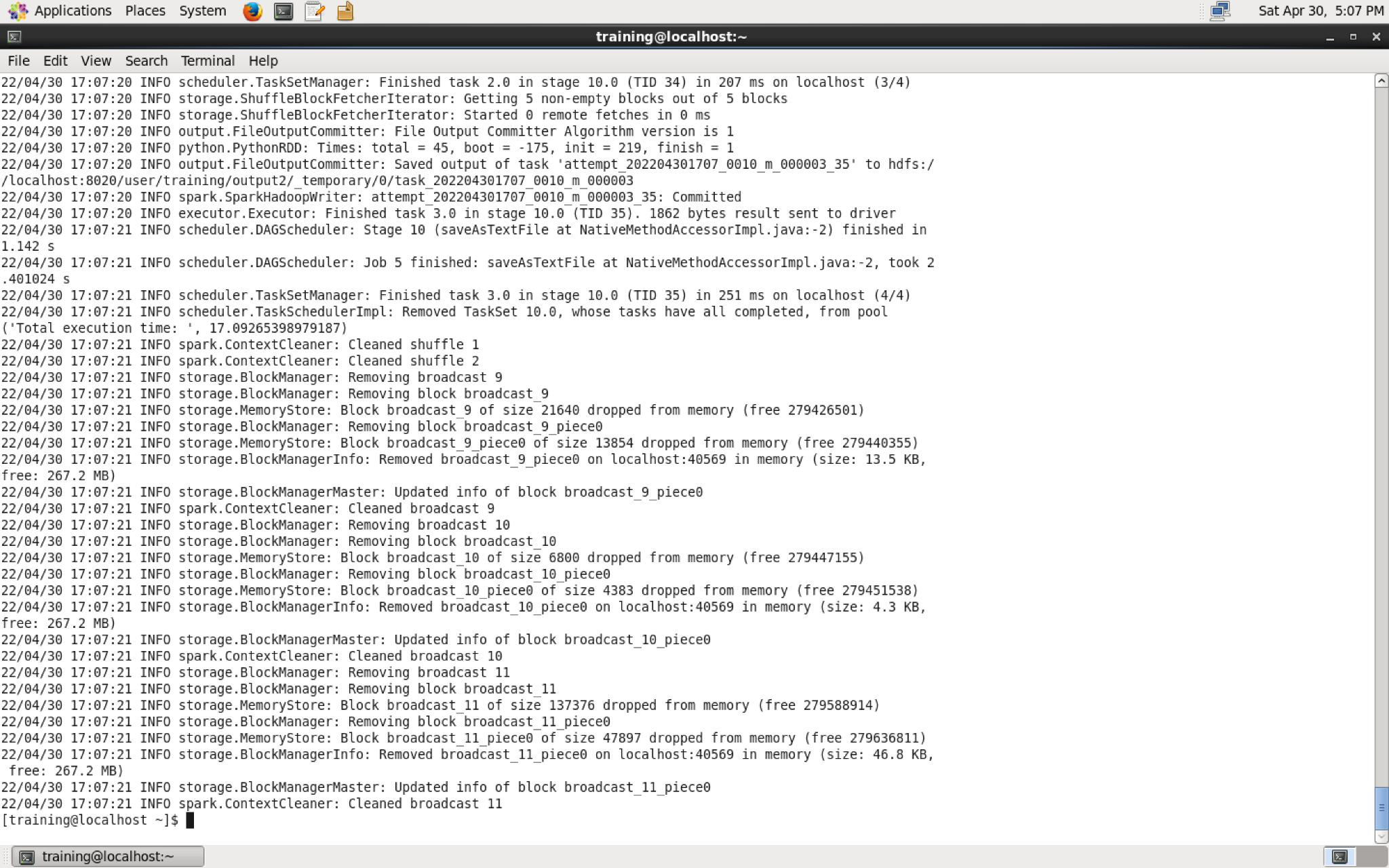
### PySpark

Below is an image of the terminal output after running the entire dataset in PySpark:

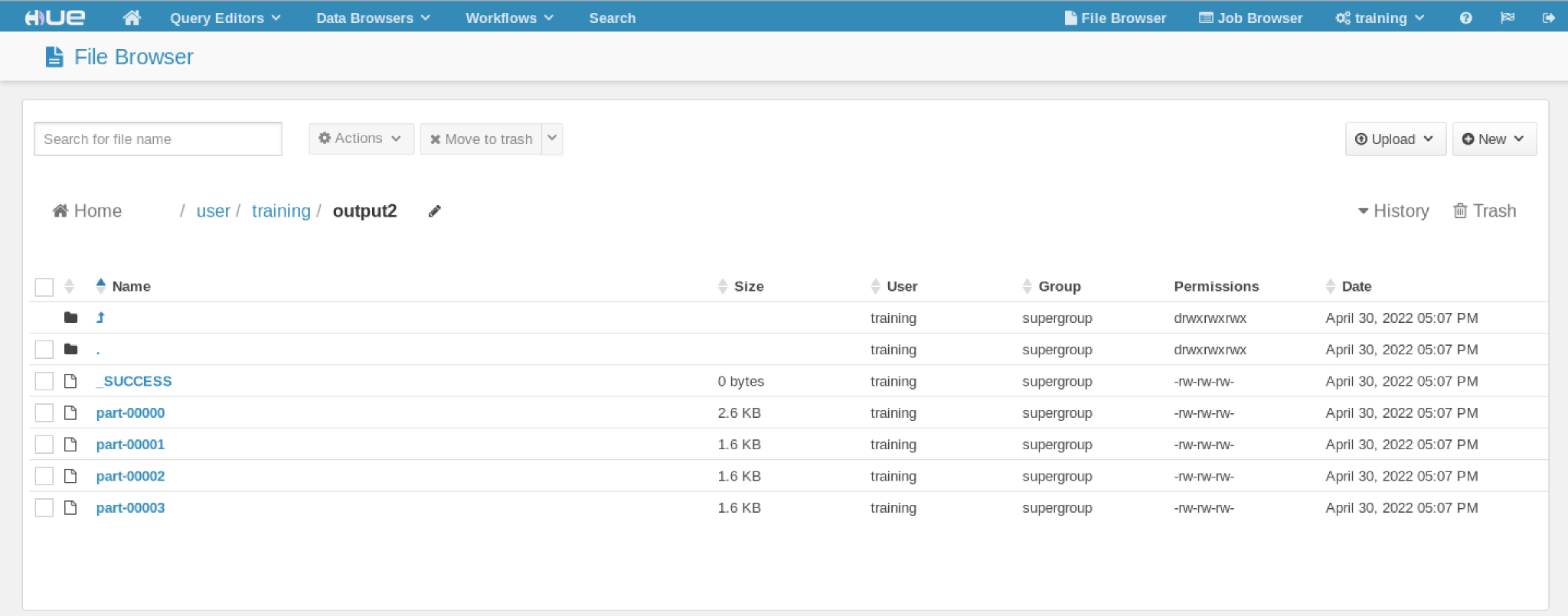


Total execution time is 512.717 seconds.

Below is an image of the terminal output after running the trimmed dataset in PySpark:



Total execution time is 17.093 seconds.

Below is an image of the PySpark output files as shown in Hue:

# Conclusion

In conclusion, across the entire dataset, vehicle ticket prices in the winter and spring were generally higher and given out at greater frequencies. PySpark had better performance than Hadoop MapReduce, however, in general, the development process was easier and more streamlined. The only downside of PySpark was the inability to use an IDE to locally run the jobs as well as the complicated error messages.

To improve for future analysis and evaluation, filtering out misformatted and irrelevant data would need to be adjusted. The main issues faced with filtering was not only coming across numerous unknown abbreviations and codes but also many alterations of the same key. For example, various colors had multiple abbreviations. Subsequently, based on the task type being used, adding functionality to change the minimum count limit needed to be accepted and written to the output files would be beneficial. The reasoning behind this is due to the fact that different task types required different numbers of keys to be filtered out other than just those less than 5.

With this, different task types were more prone to erroneous data than others and had to be filtered out. For example, the “registrationState” task type had minimal erroneous data due to there being only a limited number of state abbreviations compared to the “vehicleBodyType” and “vehicleColor” task types having a plethora of erroneous and unknown data. Another issue faced was managing and tackling the limited disk and memory space on the virtual machine. In the future, an environment with increased virtual hardware would need to be used to efficiently run the entire dataset.

# Appendix

| **File Name** | **Number of Lines** |
| --- | --- |
| ParkingViolationsDriver.java | 44 |
| ParkingViolationsPartitioner.java | 26 |
| ParkingViolationsMapper.java | 105 |
| ParkingViolationsReducer.java | 68 |
| parking\_violations.py | 40 |