Analysis of Parking Violations in New York City

Justin Nichols: jnich56@lsu.edu

Bryce Lee: blee57@lsu.edu

CSC 4740 Final Project

30 April 2022

Table of Contents

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:

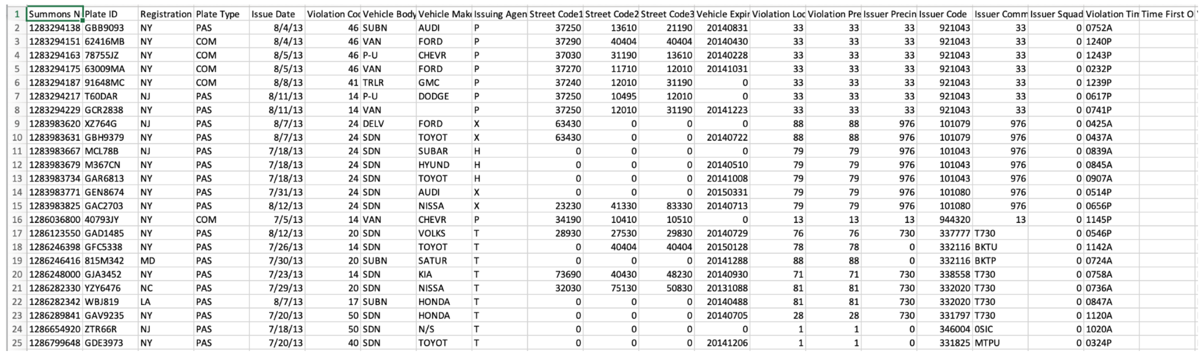
1. `Parking_Violations_Issued_-_Fiscal_Year_2014__August_2013___June_2014_.csv` (1.87 GB)

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)

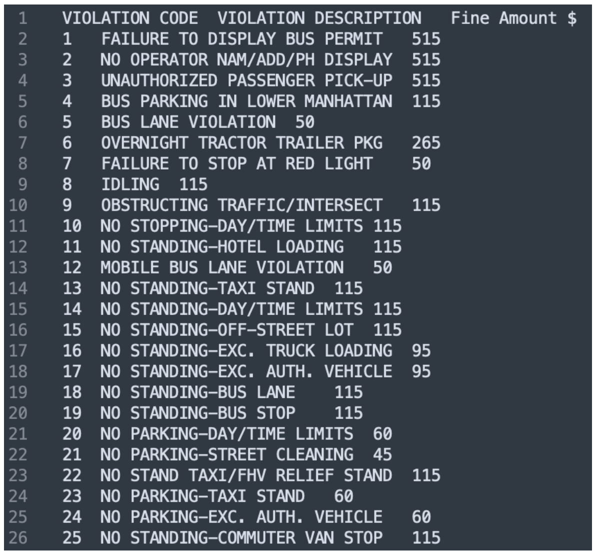
Below is a subset containing the first 25 entries from `Parking_Violations_2013_2014.csv`.



1	Summons N	Plate ID	Registration	Plate Type	Issue Date	Violation Coc	Vehicle Body	Vehicle Make	Issuing Agen	Street Code1	Street Code2	Street Code3	Vehicle Expir	Violation Loc	Violation Pre	Issuer Precin	Issuer Code	Issuer Comm	Issuer Squad	Violation Tin	Time First O
2	1283294138	GBB9093	NY	PAS	8/4/13	46	SUBN	AUDI	P	37250	13610	21190	20140831	33	33	33	921043	33	0	0752A	
3	1283294151	62416MB	NY	COM	8/4/13	46	VAN	FORD	P	37290	40404	40404	20140430	33	33	33	921043	33	0	1240P	
4	1283294163	78755JZ	NY	COM	8/5/13	46	P-U	CHEVR	P	37030	31190	13610	20140228	33	33	33	921043	33	0	1243P	
5	1283294175	63009MA	NY	COM	8/5/13	46	VAN	FORD	P	37270	11710	12010	20141031	33	33	33	921043	33	0	0232P	
6	1283294187	91648MC	NY	COM	8/8/13	41	TRLR	GMC	P	37240	12010	31190	0	33	33	33	921043	33	0	1239P	
7	1283294217	T60DAR	NJ	PAS	8/11/13	14	P-U	DODGE	P	37250	10495	12010	0	33	33	33	921043	33	0	0617P	
8	1283294229	GCR2838	NY	PAS	8/11/13	14	VAN		P	37250	12010	31190	20141223	33	33	33	921043	33	0	0741P	
9	1283983620	XZ764G	NJ	PAS	8/7/13	24	DELV	FORD	X	63430	0	0	0	88	88	976	101079	976	0	0425A	
10	1283983631	GBH9379	NY	PAS	8/7/13	24	SDN	TOYOT	X	63430	0	0	20140722	88	88	976	101079	976	0	0437A	
11	1283983667	MCL78B	NJ	PAS	7/18/13	24	SDN	SUBAR	H	0	0	0	0	79	79	976	101043	976	0	0839A	
12	1283983679	M367CN	NY	PAS	7/18/13	24	SDN	HYUND	H	0	0	0	20140510	79	79	976	101043	976	0	0845A	
13	1283983734	GAR6813	NY	PAS	7/18/13	24	SDN	TOYOT	H	0	0	0	20141008	79	79	976	101043	976	0	0907A	
14	1283983771	GEN8674	NY	PAS	7/31/13	24	SDN	AUDI	X	0	0	0	20150331	79	79	976	101080	976	0	0514P	
15	1283983825	GAC2703	NY	PAS	8/12/13	24	SDN	NISSA	X	23230	41330	83330	20140713	79	79	976	101080	976	0	0656P	
16	1286036800	40793JY	NY	COM	7/5/13	14	VAN	CHEVR	P	34190	10410	10510	0	13	13	13	944320	13	0	1145P	
17	1286123550	GAD1485	NY	PAS	8/12/13	20	SDN	VOLKS	T	28930	27530	29830	20140729	76	76	730	337777	T730	0	0546P	
18	1286246398	GFC5338	NY	PAS	7/26/13	14	SDN	TOYOT	T	0	40404	40404	20150128	78	78	0	332116	BKTU	0	1142A	
19	1286246416	815M342	MD	PAS	7/30/13	20	SUBN	SATUR	T	0	0	0	20141288	88	88	0	332116	BKTP	0	0724A	
20	1286248000	GJA3452	NY	PAS	7/23/13	14	SDN	KIA	T	73690	40430	48230	20140930	71	71	730	338558	T730	0	0758A	
21	1286282330	YZY6476	NC	PAS	7/29/13	20	SDN	NISSA	T	32030	75130	50830	20131088	81	81	730	332020	T730	0	0736A	
22	1286282342	WBJ819	LA	PAS	8/7/13	17	SUBN	HONDA	T	0	0	0	20140488	81	81	730	332020	T730	0	0847A	
23	1286289841	GAV9235	NY	PAS	7/20/13	50	SDN	HONDA	T	0	0	0	20140705	28	28	730	331797	T730	0	1120A	
24	1286654920	ZTR66R	NJ	PAS	7/18/13	50	SDN	N/S	T	0	0	0	0	1	1	0	346004	0SIC	0	1020A	
25	1286799648	GDE3973	NY	PAS	7/20/13	40	SDN	TOYOT	T	0	0	0	20141206	1	1	0	331825	MTPU	0	0324P	

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`.



```
VIOLATION CODE  VIOLATION DESCRIPTION   Fine Amount $
1   FAILURE TO DISPLAY BUS PERMIT   515
2   NO OPERATOR NAM/ADD/PH DISPLAY  515
3   UNAUTHORIZED PASSENGER PICK-UP  515
4   BUS PARKING IN LOWER MANHATTAN  115
5   BUS LANE VIOLATION  50
6   OVERNIGHT TRACTOR TRAILER PKG   265
7   FAILURE TO STOP AT RED LIGHT    50
8   IDLING  115
9   OBSTRUCTING TRAFFIC/INTERSECT   115
10  NO STOPPING-DAY/TIME LIMITS 115
11  NO STANDING-HOTEL LOADING   115
12  MOBILE BUS LANE VIOLATION   50
13  NO STANDING-TAXI STAND  115
14  NO STANDING-DAY/TIME LIMITS 115
15  NO STANDING-OFF-STREET LOT  115
16  NO STANDING-EXC. TRUCK LOADING  95
17  NO STANDING-EXC. AUTH. VEHICLE  95
18  NO STANDING-BUS LANE    115
19  NO STANDING-BUS STOP    115
20  NO PARKING-DAY/TIME LIMITS  60
21  NO PARKING-STREET CLEANING  45
22  NO STAND TAXI/FHV RELIEF STAND  115
23  NO PARKING-TAXI STAND   60
24  NO PARKING-EXC. AUTH. VEHICLE   60
25  NO STANDING-COMMUTER VAN STOP   115
```

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 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, 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 and the VM using this link.
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”`
2. When the MapReduce task completes, open up Firefox on the VM and click the Hue bookmark at the top.
3. 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`
2. When the PySpark task completes, open up Firefox on the VM and click the Hue bookmark at the top.
3. 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

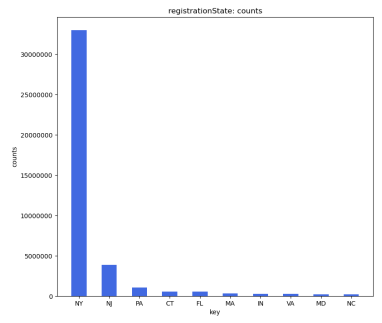
For evaluation, documents provided by the NYC government helped decode the various abbreviations and codes throughout the data into tangible meanings. The provided documents included `regristration_class_body_types_colors.pdf` and `plate_types_state_codes.pdf` and helped translate the registration state, 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.

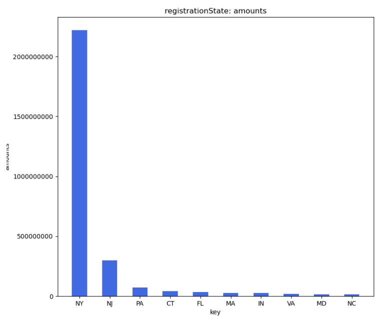
Overall, most vehicles, regardless of specific attributes, were ticketed higher in winter and spring than compared to the summer and fall. This was a trend throughout all the years the data covered, from 2013 to 2017.

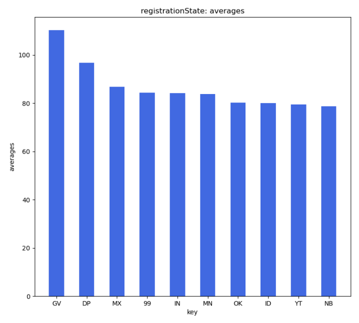
registrationState

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.





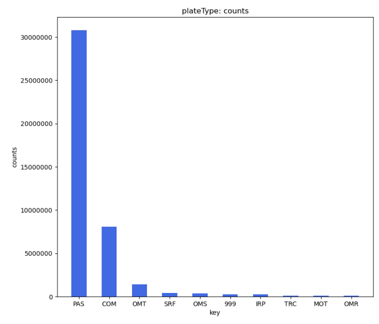


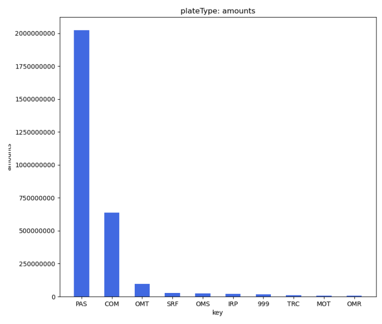
plateType

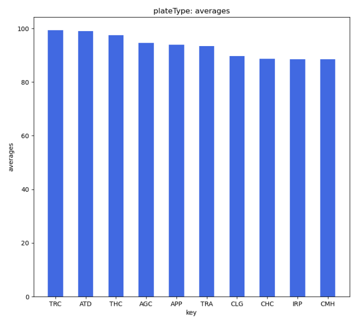
Based on the total dollars collected and the total number of violations issued, passenger (PAS), commercial (COM), taxi (OMT), specialized passenger (SRF), international (IRP), rental

(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.



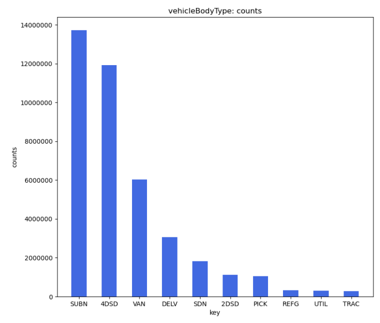


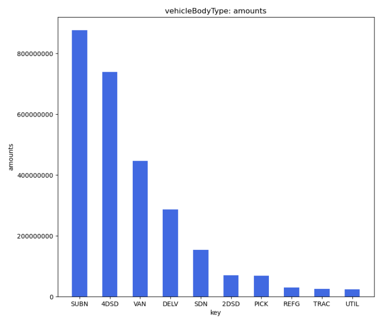


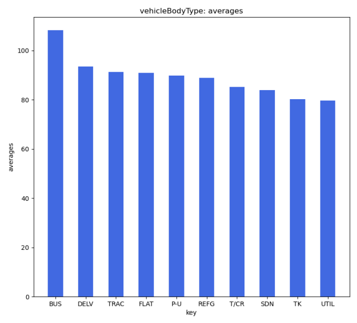
vehicleBodyType

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.



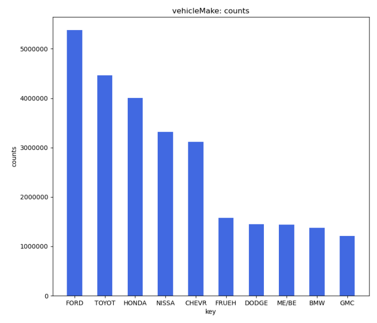


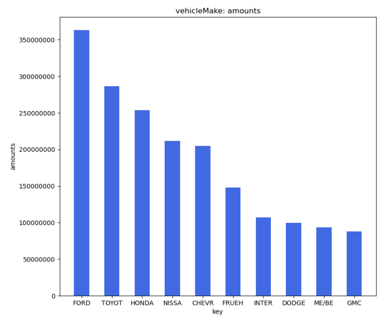


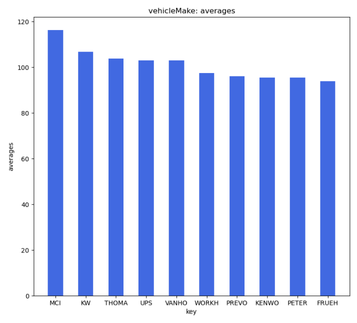
vehicleMake

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.



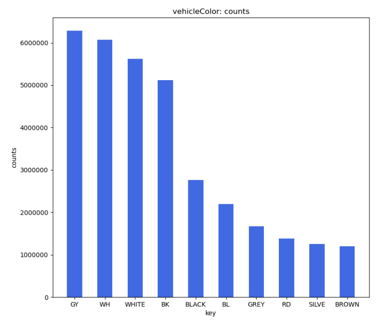


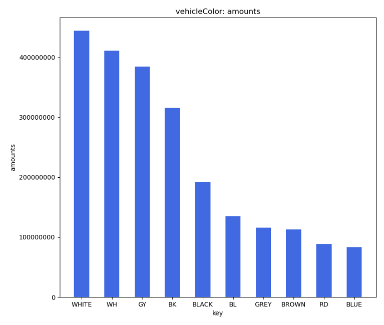


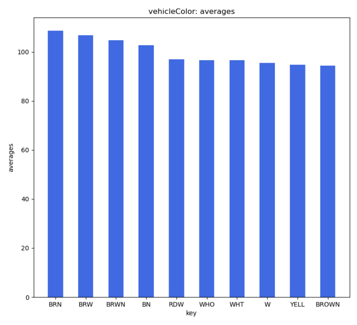
vehicleColor

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.



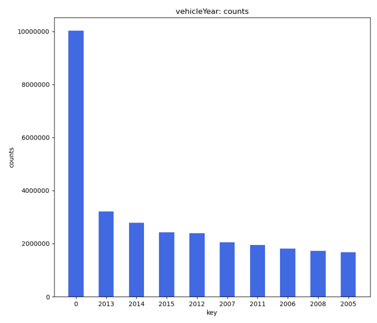


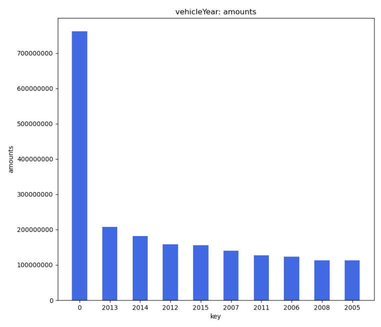


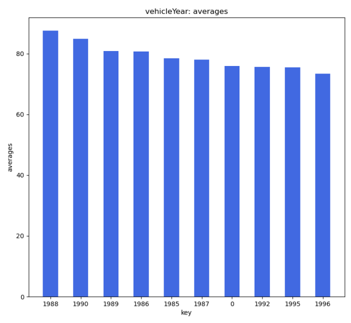
vehicleYear

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.

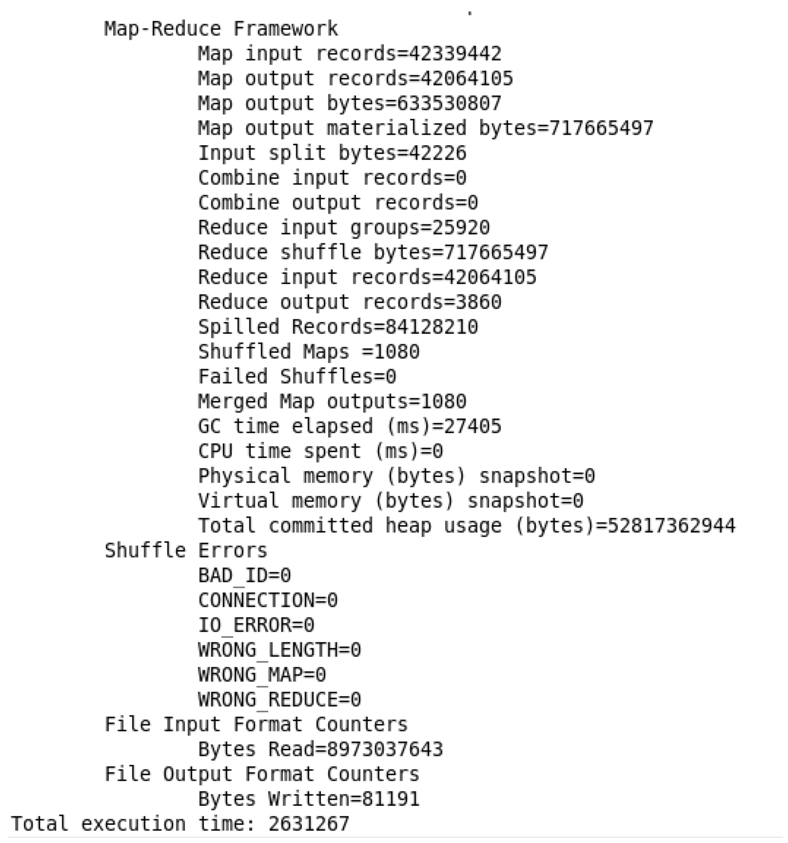
Comparing Hadoop MapReduce and PySpark Performance

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

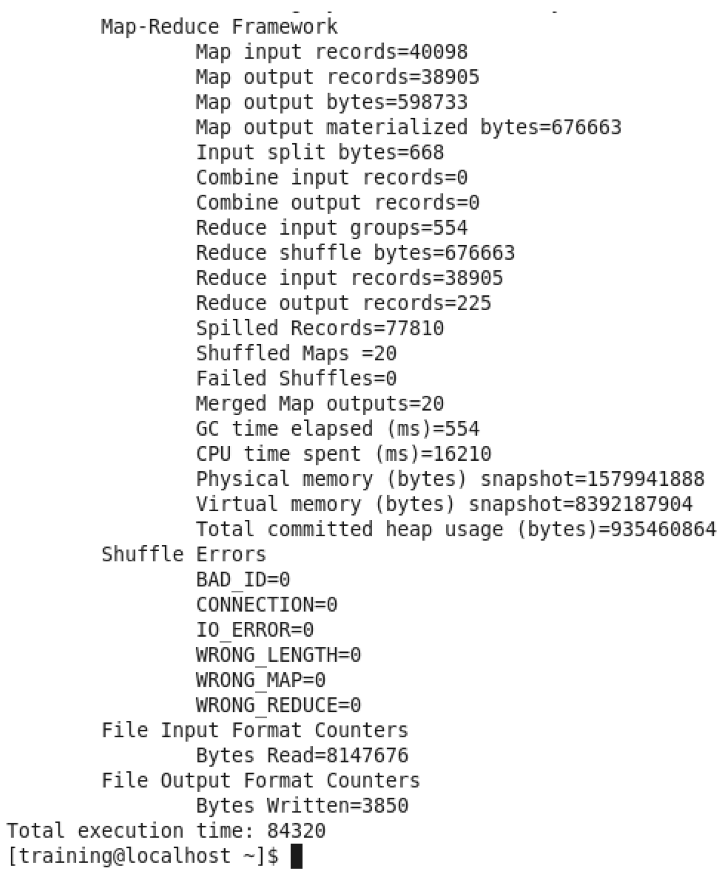
Below is an image of the terminal output after running the entire dataset in Hadoop MapReduce:



```
        Map-Reduce Framework
                Map input records=42339442
                Map output records=42064105
                Map output bytes=633530807
                Map output materialized bytes=717665497
                Input split bytes=42226
                Combine input records=0
                Combine output records=0
                Reduce input groups=25920
                Reduce shuffle bytes=717665497
                Reduce input records=42064105
                Reduce output records=3860
                Spilled Records=84128210
                Shuffled Maps =1080
                Failed Shuffles=0
                Merged Map outputs=1080
                GC time elapsed (ms)=27405
                CPU time spent (ms)=0
                Physical memory (bytes) snapshot=0
                Virtual memory (bytes) snapshot=0
                Total committed heap usage (bytes)=52817362944
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=8973037643
        File Output Format Counters
                Bytes Written=81191
Total execution time: 2631267
```

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

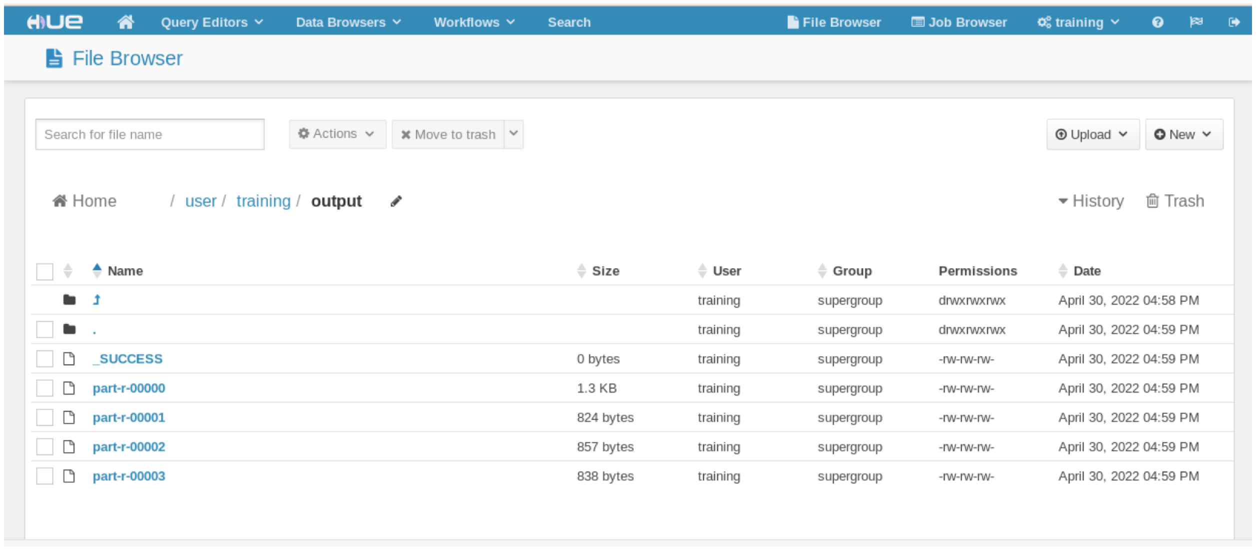
Below is an image of the terminal output after running the trimmed dataset in Hadoop MapReduce:



```
        Map-Reduce Framework
                Map input records=40098
                Map output records=38905
                Map output bytes=598733
                Map output materialized bytes=676663
                Input split bytes=668
                Combine input records=0
                Combine output records=0
                Reduce input groups=554
                Reduce shuffle bytes=676663
                Reduce input records=38905
                Reduce output records=225
                Spilled Records=77810
                Shuffled Maps =20
                Failed Shuffles=0
                Merged Map outputs=20
                GC time elapsed (ms)=554
                CPU time spent (ms)=16210
                Physical memory (bytes) snapshot=1579941888
                Virtual memory (bytes) snapshot=8392187904
                Total committed heap usage (bytes)=935460864
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=8147676
        File Output Format Counters
                Bytes Written=3850
Total execution time: 84320
[training@localhost ~]$
```

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:

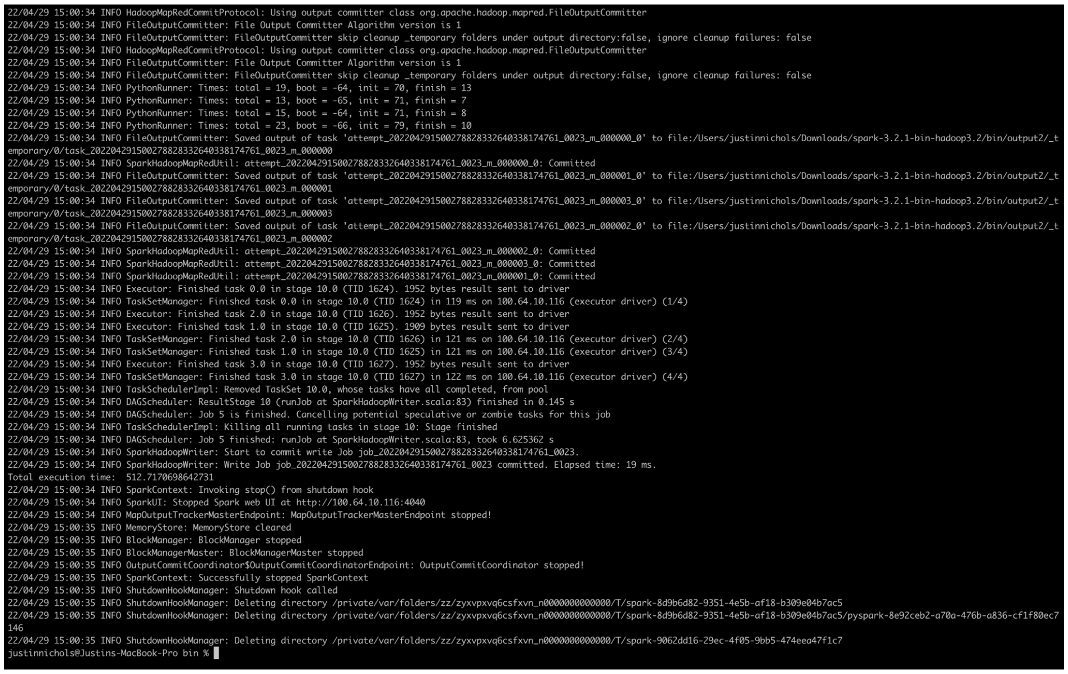


/ user / training / output

Name	Size	User	Group	Permissions	Date
↥		training	supergroup	drwxrwxrwx	April 30, 2022 04:58 PM
.		training	supergroup	drwxrwxrwx	April 30, 2022 04:59 PM
_SUCCESS	0 bytes	training	supergroup	-rw-rw-rw-	April 30, 2022 04:59 PM
part-r-00000	1.3 KB	training	supergroup	-rw-rw-rw-	April 30, 2022 04:59 PM
part-r-00001	824 bytes	training	supergroup	-rw-rw-rw-	April 30, 2022 04:59 PM
part-r-00002	857 bytes	training	supergroup	-rw-rw-rw-	April 30, 2022 04:59 PM
part-r-00003	838 bytes	training	supergroup	-rw-rw-rw-	April 30, 2022 04:59 PM

PySpark

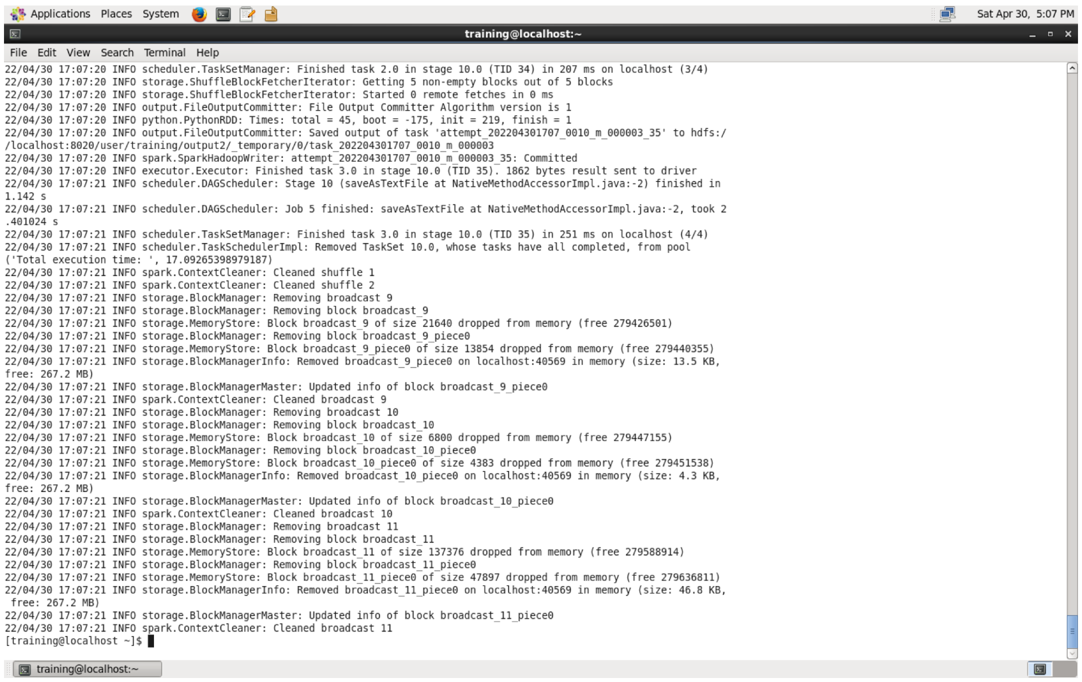
Below is an image of the terminal output after running the entire dataset in PySpark:



```
22/04/29 15:00:34 INFO HadoopMapRedCommitProtocol: Using output committer class org.apache.hadoop.mapred.FileOutputCommitter
22/04/29 15:00:34 INFO FileOutputCommitter: File Output Committer Algorithm version is 1
22/04/29 15:00:34 INFO FileOutputCommitter: FileOutputCommitter skip cleanup _temporary folders under output directory:false, ignore cleanup failures: false
22/04/29 15:00:34 INFO HadoopMapRedCommitProtocol: Using output committer class org.apache.hadoop.mapred.FileOutputCommitter
22/04/29 15:00:34 INFO FileOutputCommitter: File Output Committer Algorithm version is 1
22/04/29 15:00:34 INFO FileOutputCommitter: FileOutputCommitter skip cleanup _temporary folders under output directory:false, ignore cleanup failures: false
22/04/29 15:00:34 INFO PythonRunner: Times: total = 19, boot = -64, init = 70, finish = 13
22/04/29 15:00:34 INFO PythonRunner: Times: total = 13, boot = -65, init = 71, finish = 7
22/04/29 15:00:34 INFO PythonRunner: Times: total = 15, boot = -64, init = 71, finish = 8
22/04/29 15:00:34 INFO PythonRunner: Times: total = 23, boot = -66, init = 79, finish = 10
22/04/29 15:00:34 INFO FileOutputCommitter: Saved output of task 'attempt_202204291500278828332640338174761_0023_m_000000_0' to file:/Users/justinnichols/Downloads/spark-3.2.1-bin-hadoop3.2/bin/output2/_t
emporary/0/task_202204291500278828332640338174761_0023_m_000000
22/04/29 15:00:34 INFO SparkHadoopMapRedUtil: attempt_202204291500278828332640338174761_0023_m_000000_0: Committed
22/04/29 15:00:34 INFO FileOutputCommitter: Saved output of task 'attempt_202204291500278828332640338174761_0023_m_000001_0' to file:/Users/justinnichols/Downloads/spark-3.2.1-bin-hadoop3.2/bin/output2/_t
emporary/0/task_202204291500278828332640338174761_0023_m_000001
22/04/29 15:00:34 INFO FileOutputCommitter: Saved output of task 'attempt_202204291500278828332640338174761_0023_m_000003_0' to file:/Users/justinnichols/Downloads/spark-3.2.1-bin-hadoop3.2/bin/output2/_t
emporary/0/task_202204291500278828332640338174761_0023_m_000003
22/04/29 15:00:34 INFO FileOutputCommitter: Saved output of task 'attempt_202204291500278828332640338174761_0023_m_000002_0' to file:/Users/justinnichols/Downloads/spark-3.2.1-bin-hadoop3.2/bin/output2/_t
emporary/0/task_202204291500278828332640338174761_0023_m_000002
22/04/29 15:00:34 INFO SparkHadoopMapRedUtil: attempt_202204291500278828332640338174761_0023_m_000002_0: Committed
22/04/29 15:00:34 INFO SparkHadoopMapRedUtil: attempt_202204291500278828332640338174761_0023_m_000003_0: Committed
22/04/29 15:00:34 INFO SparkHadoopMapRedUtil: attempt_202204291500278828332640338174761_0023_m_000001_0: Committed
22/04/29 15:00:34 INFO Executor: Finished task 0.0 in stage 10.0 (TID 1624). 1952 bytes result sent to driver
22/04/29 15:00:34 INFO TaskSetManager: Finished task 0.0 in stage 10.0 (TID 1624) in 119 ms on 100.64.10.116 (executor driver) (1/4)
22/04/29 15:00:34 INFO Executor: Finished task 2.0 in stage 10.0 (TID 1626). 1952 bytes result sent to driver
22/04/29 15:00:34 INFO Executor: Finished task 1.0 in stage 10.0 (TID 1625). 1909 bytes result sent to driver
22/04/29 15:00:34 INFO TaskSetManager: Finished task 2.0 in stage 10.0 (TID 1626) in 121 ms on 100.64.10.116 (executor driver) (2/4)
22/04/29 15:00:34 INFO TaskSetManager: Finished task 1.0 in stage 10.0 (TID 1625) in 121 ms on 100.64.10.116 (executor driver) (3/4)
22/04/29 15:00:34 INFO Executor: Finished task 3.0 in stage 10.0 (TID 1627). 1952 bytes result sent to driver
22/04/29 15:00:34 INFO TaskSetManager: Finished task 3.0 in stage 10.0 (TID 1627) in 122 ms on 100.64.10.116 (executor driver) (4/4)
22/04/29 15:00:34 INFO TaskSchedulerImpl: Removed TaskSet 10.0, whose tasks have all completed, from pool
22/04/29 15:00:34 INFO DAGScheduler: ResultStage 10 (runJob at SparkHadoopWriter.scala:83) finished in 0.145 s
22/04/29 15:00:34 INFO DAGScheduler: Job 5 is finished. Cancelling potential speculative or zombie tasks for this job
22/04/29 15:00:34 INFO TaskSchedulerImpl: Killing all running tasks in stage 10: Stage finished
22/04/29 15:00:34 INFO DAGScheduler: Job 5 finished: runJob at SparkHadoopWriter.scala:83, took 6.625362 s
22/04/29 15:00:34 INFO SparkHadoopWriter: Start to commit write Job job_202204291500278828332640338174761_0023.
22/04/29 15:00:34 INFO SparkHadoopWriter: Write Job job_202204291500278828332640338174761_0023 committed. Elapsed time: 19 ms.
Total execution time:  512.7170698642731
22/04/29 15:00:34 INFO SparkContext: Invoking stop() from shutdown hook
22/04/29 15:00:34 INFO SparkUI: Stopped Spark web UI at http://100.64.10.116:4040
22/04/29 15:00:34 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint stopped!
22/04/29 15:00:35 INFO MemoryStore: MemoryStore cleared
22/04/29 15:00:35 INFO BlockManager: BlockManager stopped
22/04/29 15:00:35 INFO BlockManagerMaster: BlockManagerMaster stopped
22/04/29 15:00:35 INFO OutputCommitCoordinator$OutputCommitCoordinatorEndpoint: OutputCommitCoordinator stopped!
22/04/29 15:00:35 INFO SparkContext: Successfully stopped SparkContext
22/04/29 15:00:35 INFO ShutdownHookManager: Shutdown hook called
22/04/29 15:00:35 INFO ShutdownHookManager: Deleting directory /private/var/folders/zz/zyxvpxvq6csfxvn_n0000000000000/T/spark-8d9b6d82-9351-4e5b-af18-b309e04b7ac5
22/04/29 15:00:35 INFO ShutdownHookManager: Deleting directory /private/var/folders/zz/zyxvpxvq6csfxvn_n0000000000000/T/spark-8d9b6d82-9351-4e5b-af18-b309e04b7ac5/pyspark-8e92ceb2-a70a-476b-a836-cf1f80ec7
146
22/04/29 15:00:35 INFO ShutdownHookManager: Deleting directory /private/var/folders/zz/zyxvpxvq6csfxvn_n0000000000000/T/spark-9062dd16-29ec-4f05-9bb5-474eea47f1c7
justinnichols@Justins-MacBook-Pro bin %
```

Total execution time is 512.717 seconds.

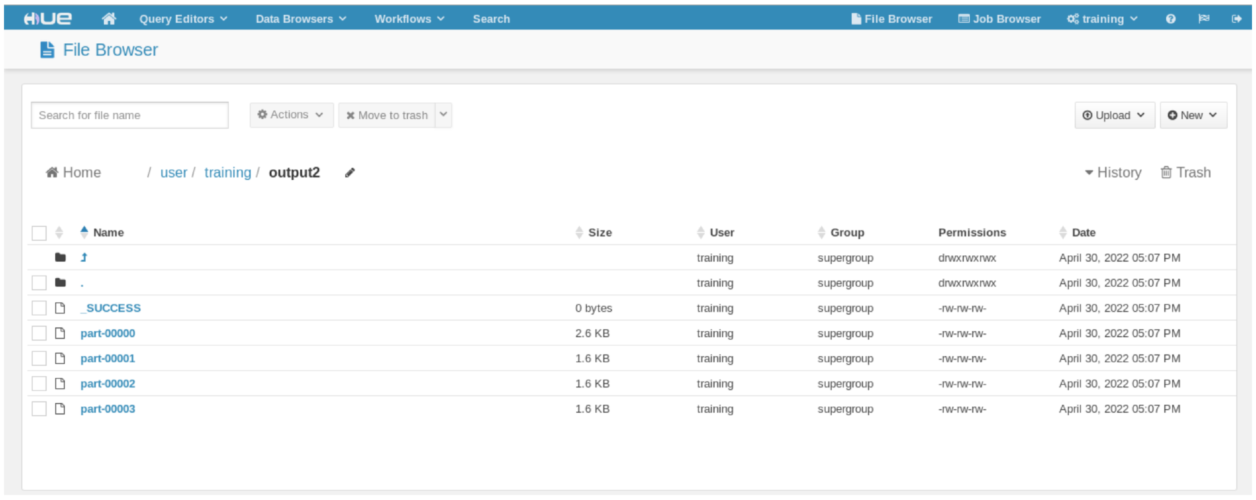
Below is an image of the terminal output after running the trimmed dataset in PySpark:


Applications Places System
Sat Apr 30, 5:07 PM
training@localhost:~
File Edit View Search Terminal Help
training@localhost:~

```
22/04/30 17:07:20 INFO scheduler.TaskSetManager: Finished task 2.0 in stage 10.0 (TID 34) in 207 ms on localhost (3/4)
22/04/30 17:07:20 INFO storage.ShuffleBlockFetcherIterator: Getting 5 non-empty blocks out of 5 blocks
22/04/30 17:07:20 INFO storage.ShuffleBlockFetcherIterator: Started 0 remote fetches in 0 ms
22/04/30 17:07:20 INFO output.FileOutputCommitter: File Output Committer Algorithm version is 1
22/04/30 17:07:20 INFO python.PythonRDD: Times: total = 45, boot = -175, init = 219, finish = 1
22/04/30 17:07:20 INFO output.FileOutputCommitter: Saved output of task 'attempt_202204301707_0010_m_000003_35' to hdfs:/
/localhost:8020/user/training/output2/_temporary/0/task_202204301707_0010_m_000003
22/04/30 17:07:20 INFO spark.SparkHadoopWriter: attempt_202204301707_0010_m_000003_35: Committed
22/04/30 17:07:20 INFO executor.Executor: Finished task 3.0 in stage 10.0 (TID 35). 1862 bytes result sent to driver
22/04/30 17:07:21 INFO scheduler.DAGScheduler: Stage 10 (saveAsTextFile at NativeMethodAccessorImpl.java:-2) finished in
1.142 s
22/04/30 17:07:21 INFO scheduler.DAGScheduler: Job 5 finished: saveAsTextFile at NativeMethodAccessorImpl.java:-2, took 2
.401024 s
22/04/30 17:07:21 INFO scheduler.TaskSetManager: Finished task 3.0 in stage 10.0 (TID 35) in 251 ms on localhost (4/4)
22/04/30 17:07:21 INFO scheduler.TaskSchedulerImpl: Removed TaskSet 10.0, whose tasks have all completed, from pool
('Total execution time: ', 17.09265398979187)
22/04/30 17:07:21 INFO spark.ContextCleaner: Cleaned shuffle 1
22/04/30 17:07:21 INFO spark.ContextCleaner: Cleaned shuffle 2
22/04/30 17:07:21 INFO storage.BlockManager: Removing broadcast 9
22/04/30 17:07:21 INFO storage.BlockManager: Removing block broadcast_9
22/04/30 17:07:21 INFO storage.MemoryStore: Block broadcast_9 of size 21640 dropped from memory (free 279426501)
22/04/30 17:07:21 INFO storage.BlockManager: Removing block broadcast_9_piece0
22/04/30 17:07:21 INFO storage.MemoryStore: Block broadcast_9_piece0 of size 13854 dropped from memory (free 279440355)
22/04/30 17:07:21 INFO storage.BlockManagerInfo: Removed broadcast_9_piece0 on localhost:40569 in memory (size: 13.5 KB,
free: 267.2 MB)
22/04/30 17:07:21 INFO storage.BlockManagerMaster: Updated info of block broadcast_9_piece0
22/04/30 17:07:21 INFO spark.ContextCleaner: Cleaned broadcast 9
22/04/30 17:07:21 INFO storage.BlockManager: Removing broadcast 10
22/04/30 17:07:21 INFO storage.BlockManager: Removing block broadcast_10
22/04/30 17:07:21 INFO storage.MemoryStore: Block broadcast_10 of size 6800 dropped from memory (free 279447155)
22/04/30 17:07:21 INFO storage.BlockManager: Removing block broadcast_10_piece0
22/04/30 17:07:21 INFO storage.MemoryStore: Block broadcast_10_piece0 of size 4383 dropped from memory (free 279451538)
22/04/30 17:07:21 INFO storage.BlockManagerInfo: Removed broadcast_10_piece0 on localhost:40569 in memory (size: 4.3 KB,
free: 267.2 MB)
22/04/30 17:07:21 INFO storage.BlockManagerMaster: Updated info of block broadcast_10_piece0
22/04/30 17:07:21 INFO spark.ContextCleaner: Cleaned broadcast 10
22/04/30 17:07:21 INFO storage.BlockManager: Removing broadcast 11
22/04/30 17:07:21 INFO storage.BlockManager: Removing block broadcast_11
22/04/30 17:07:21 INFO storage.MemoryStore: Block broadcast_11 of size 137376 dropped from memory (free 279588914)
22/04/30 17:07:21 INFO storage.BlockManager: Removing block broadcast_11_piece0
22/04/30 17:07:21 INFO storage.MemoryStore: Block broadcast_11_piece0 of size 47897 dropped from memory (free 279636811)
22/04/30 17:07:21 INFO storage.BlockManagerInfo: Removed broadcast_11_piece0 on localhost:40569 in memory (size: 46.8 KB,
 free: 267.2 MB)
22/04/30 17:07:21 INFO storage.BlockManagerMaster: Updated info of block broadcast_11_piece0
22/04/30 17:07:21 INFO spark.ContextCleaner: Cleaned broadcast 11
[training@localhost ~]$
```

Total execution time is 17.093 seconds.

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



Name	Size	User	Group	Permissions	Date
↥		training	supergroup	drwxrwxrwx	April 30, 2022 05:07 PM
.		training	supergroup	drwxrwxrwx	April 30, 2022 05:07 PM
_SUCCESS	0 bytes	training	supergroup	-rw-rw-rw-	April 30, 2022 05:07 PM
part-00000	2.6 KB	training	supergroup	-rw-rw-rw-	April 30, 2022 05:07 PM
part-00001	1.6 KB	training	supergroup	-rw-rw-rw-	April 30, 2022 05:07 PM
part-00002	1.6 KB	training	supergroup	-rw-rw-rw-	April 30, 2022 05:07 PM
part-00003	1.6 KB	training	supergroup	-rw-rw-rw-	April 30, 2022 05:07 PM

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