Week 4 Technical Lab

NoSQL Databases

Trevor Thomas

College of Computer and Information Sciences, Regis University

MSDS 610: Data Engineering

Paul Andrus

May 31, 2020

Overview and Objectives

MongoDB is a NoSQL database that stores data in document format. It uses JSON-like objects of key*value pairs to store information, and is considered one of the better general-use NoSQL databases available. For this lab, I will install MongoDB on my computer and then use my terminal to load and interact with data from the LA Parking Ticket dataset. Initially, I plan to query the database directly using the MongoDB shell. Then I will download and import pymongo, a Python-MongoDB interface, and use a Python shell to further explore the data. Finally, I will provide answers to the four numbered exercises in the assignment and discuss the results after that. The only tools used will be MongoDB, Python, and my terminal (Ubuntu 18.04 LTS).

Methods and Results

Install MongoDB and Import Dataset

I chose to install MongoDB locally on my own machine and completed the assignment in my terminal. First, I used `apt-get` to install MongoDB and then `mongo` to launch the MongoDB shell in my terminal.

```
(base) dpredbeard@dpred: sudo apt-get install mongodb
(base) dpredbeard@dpred: mongo
>
```

Then I left the shell with `exit` and used `wget`, `unzip`, and `mongoimport` to download the data zip file, extract it, and load it into the MongoDB database, respectively. I needed to add `sudo` to the `mongoimport` command for permission to run it.

```
(base) dpredbeard@dpred:wget https://www.dropbox.com/s/nh1h2r8ryimtm7h/los-
angeles-parking-citations.zip

(base) dpredbeard@dpred:unzip los-angeles-parking-citations.zip

(base) dpredbeard@dpred:sudo mongoimport -d parking -c tickets --type csv –
file parking-citations.csv –headerline
```

This completed the setup allowing me to query the newly built database.

Use MongoDB Shell to Query Database

I used `mongo` again to launch the MongoDB shell, and ran `show dbs`, `use parking`, and `show collections` to see information about the data we've loaded.

```
> show dbs
admin    0.000GB
config   0.000GB
local    0.000GB
parking  1.170GB

> use parking
switched to db parking

> show collections
tickets
>
```

This confirmed the data was successfully loaded into a database called "parking," moved the command prompt "into" the database, and that all the data was in a single collection called "tickets." Next, I started running a few queries to learn more about the data. MongoDB uses Javascript, so all the queries use dot notation to call methods and allow for the chaining of several methods together. They follow the general syntax of `db.<collection name>.method().other_method();`. Below are several queries I ran followed by the output.

```
> db.tickets.findOne();
{
      "_id" : ObjectId("5ed403e3bf227fe8817bddfa"),
      "Ticket number" : 1103341116,
      "Issue Date" : "2015-12-21T00:00:00",
      "Issue time" : 1251,
      "Meter Id" : "",
      "Marked Time" : "",
      "RP State Plate" : "CA",
      "Plate Expiry Date" : 200304,
      "VIN" : "",
      "Make" : "HOND",
      "Body Style" : "PA",
      "Color" : "GY",
      "Location" : "13147 WELBY WAY",
      "Route" : 1521,
      "Agency" : 1,
      "Violation code" : "4000A1",
      "Violation Description" : "NO EVIDENCE OF REG",
      "Fine amount" : 50,
      "Latitude" : 99999,
      "Longitude" : 99999
}

> db.tickets.find().limit(2).pretty();
{
      "_id" : ObjectId("5ed403e3bf227fe8817bddfa"),
      "Ticket number" : 1103341116,
      "Issue Date" : "2015-12-21T00:00:00",
      "Issue time" : 1251,
      "Meter Id" : "",
      "Marked Time" : "",
      "RP State Plate" : "CA",
      "Plate Expiry Date" : 200304,
      "VIN" : "",
      "Make" : "HOND",
      "Body Style" : "PA",
      "Color" : "GY",
      "Location" : "13147 WELBY WAY",
      "Route" : 1521,
      "Agency" : 1,
      "Violation code" : "4000A1",
      "Violation Description" : "NO EVIDENCE OF REG",
      "Fine amount" : 50,
      "Latitude" : 99999,
      "Longitude" : 99999
}
{
      "_id" : ObjectId("5ed403e3bf227fe8817bddfb"),
      "Ticket number" : 1103700150,
      "Issue Date" : "2015-12-21T00:00:00",
      "Issue time" : 1435,
      "Meter Id" : "",
      "Marked Time" : "",
      "RP State Plate" : "CA",
      "Plate Expiry Date" : 201512,
      "VIN" : "",
      "Make" : "GMC",
      "Body Style" : "VN",
      "Color" : "WH",
      "Location" : "525 S MAIN ST",
      "Route" : "1C51",
      "Agency" : 1,
      "Violation code" : "4000A1",
      "Violation Description" : "NO EVIDENCE OF REG",
      "Fine amount" : 50,
      "Latitude" : 99999,
      "Longitude" : 99999
}
```

It is clear from the above output that MongoDB is storing entries as key*value pairs. The objects it is returning are called cursors, and they are JSON-like objects that are stored as documents, which is MongoDB's storage format and enables the flexibility for which it is known. Next, I used the `.find()`,

`.sort()`, `.limit()`, and `.pretty()` methods to return the most recent ticket (highest ticket number) and the oldest ticket (lowest ticket number).

```
> db.tickets.find().sort({'Ticket number': -1}).limit(1).pretty();
{
      "_id" : ObjectId("5ed404ebbf227fe881d47402"),
      "Ticket number" : "1117031296D",
      "Issue Date" : "2017-08-12T00:00:00",
      "Issue time" : 1151,
      "Meter Id" : "",
      "Marked Time" : "",
      "RP State Plate" : "CA",
      "Plate Expiry Date" : 201801,
      "VIN" : "",
      "Make" : "BUIC",
      "Body Style" : "PA",
      "Color" : "SI",
      "Location" : "3000 CANYON LAKE DR",
      "Route" : "",
      "Agency" : 4,
      "Violation code" : 80560000,
      "Violation Description" : "RED ZONE",
      "Fine amount" : 93,
      "Latitude" : 99999,
      "Longitude" : 99999
}

> db.tickets.find().sort({'Ticket number': 1}).limit(1).pretty();
{
      "_id" : ObjectId("5ed4052ebf227fe881e71f51"),
      "Ticket number" : 1001096320,
      "Issue Date" : "2018-03-09T00:00:00",
      "Issue time" : 1651,
      "Meter Id" : "",
      "Marked Time" : "",
      "RP State Plate" : "CA",
      "Plate Expiry Date" : 200617,
      "VIN" : "",
      "Make" : "AUDI",
      "Body Style" : "PA",
      "Color" : "BK",
      "Location" : "REAR OF 642 28TH ST",
      "Route" : 328,
      "Agency" : 1,
      "Violation code" : "8061#",
      "Violation Description" : "STANDING IN ALLEY",
      "Fine amount" : 68,
      "Latitude" : 6473268.42892846,
      "Longitude" : 1720171.01612351
}
```

Each of these queries is taking a couple seconds because of the size of the data-set. To speed up the results, I sampled down the data to 1000 observations. I first declared a variable called "sample" made up of 1000 observations in the data-set using `.aggregate()` and `.toArray()`. Then I created a new collection in the "parking" database called "sampled_tickets" using `.insert()`. I also used `.count()` to make sure the new collection was built correctly.

```
> var sample = db.tickets.aggregate( [ { $sample: {size:
1000} } ] ).toArray();

> db.sampled_tickets.insert(sample);
> db.sampled_tickets.count()
1000
```

Finally, from this smaller sample, I ran a new query that was only to return the ticket number and violation description of the most recent ticket.

```
> db.sampled_tickets.find({}, {'Ticket number': 1, 'Violation Description':
1}).sort({'Ticket number': -1}).limit(1).pretty();

{
      "_id" : ObjectId("5ed40578bf227fe881035e3a"),
      "Ticket number" : NumberLong("4347993554"),
      "Violation Description" : "DISPLAY OF PLATES"
}
```

This was much faster given it only had 1000 observations to sort. The "_id" key always appears by default unless a parameter is added to explicitly omit it. That command looks like the following:

```
> db.sampled_tickets.find({}, {'Ticket number': 1, 'Violation Description':
1, '_id':0}).sort({'Ticket number': -1}).limit(1).pretty();

{
      "Ticket number" : NumberLong("4347993554"),
      "Violation Description" : "DISPLAY OF PLATES"
}
```

After this I closed the MongoDB shell using `exit` and moved on to connecting Python to MongoDB.

Use Python to Connect to MongoDB

I began by downloading the Python-MongoDB interface called pymongo using `pip install pymongo` and then launched a Python shell in my terminal using the `python` command.

```
(base) dpredbeard@dpred:~$ pip install pymongo
(base) dpredbeard@dpred:~$ python
Python 3.7.6 | packaged by conda-forge | (default, Jan  7 2020, 20:28:53)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
```

Next, I imported the MongoClient library from pymongo and initialized a few variables. Ultimately, I created a variable called “tickets” that contains the ticket data-set as an object upon we I could call methods. The first method I called was `.find_one()` to retrieve one of the records. The output was noticeably less user-friendly without the `.pretty()` method in the MongoDB native query language.

```
>>> from pymongo import MongoClient
>>> client = MongoClient()
>>> db = client['parking']
>>> tickets = db['tickets']
>>> tickets.find_one()

{'_id': ObjectId('5ed403e3bf227fe8817bddfa'), 'Ticket number': 1103341116,
'Issue Date': '2015-12-21T00:00:00', 'Issue time': 1251, 'Meter Id': '',
'Marked Time': '', 'RP State Plate': 'CA', 'Plate Expiry Date': 200304,
'VIN': '', 'Make': 'HOND', 'Body Style': 'PA', 'Color': 'GY', 'Location':
'13147 WELBY WAY', 'Route': 1521, 'Agency': 1, 'Violation code': '4000A1',
'Violation Description': 'NO EVIDENCE OF REG', 'Fine amount': 50, 'Latitude':
99999, 'Longitude': 99999}
>>>
```

Then I created a new variable called “registration_tickets” by filtering the original “tickets” variable. To do this, I executed the `.find()` method and fed it a parameter to only return the first 10 results where the “Violation Description” key held the value “NO EVIDENCE OF REG.” Once this variable was created, I indexed it to print the first record to the console.

```
>>> registration_tickets = tickets.find({'Violation Description': {'$eq': 'NO
EVIDENCE OF REG'}}).limit(10)

>>> registration_tickets[0]

{'_id': ObjectId('5ed403e3bf227fe8817bddfa'), 'Ticket number': 1103341116,
'Issue Date': '2015-12-21T00:00:00', 'Issue time': 1251, 'Meter Id': '',
'Marked Time': '', 'RP State Plate': 'CA', 'Plate Expiry Date': 200304,
'VIN': '', 'Make': 'HOND', 'Body Style': 'PA', 'Color': 'GY', 'Location':
'13147 WELBY WAY', 'Route': 1521, 'Agency': 1, 'Violation code': '4000A1',
'Violation Description': 'NO EVIDENCE OF REG', 'Fine amount': 50, 'Latitude':
99999, 'Longitude': 99999}
>>>
```

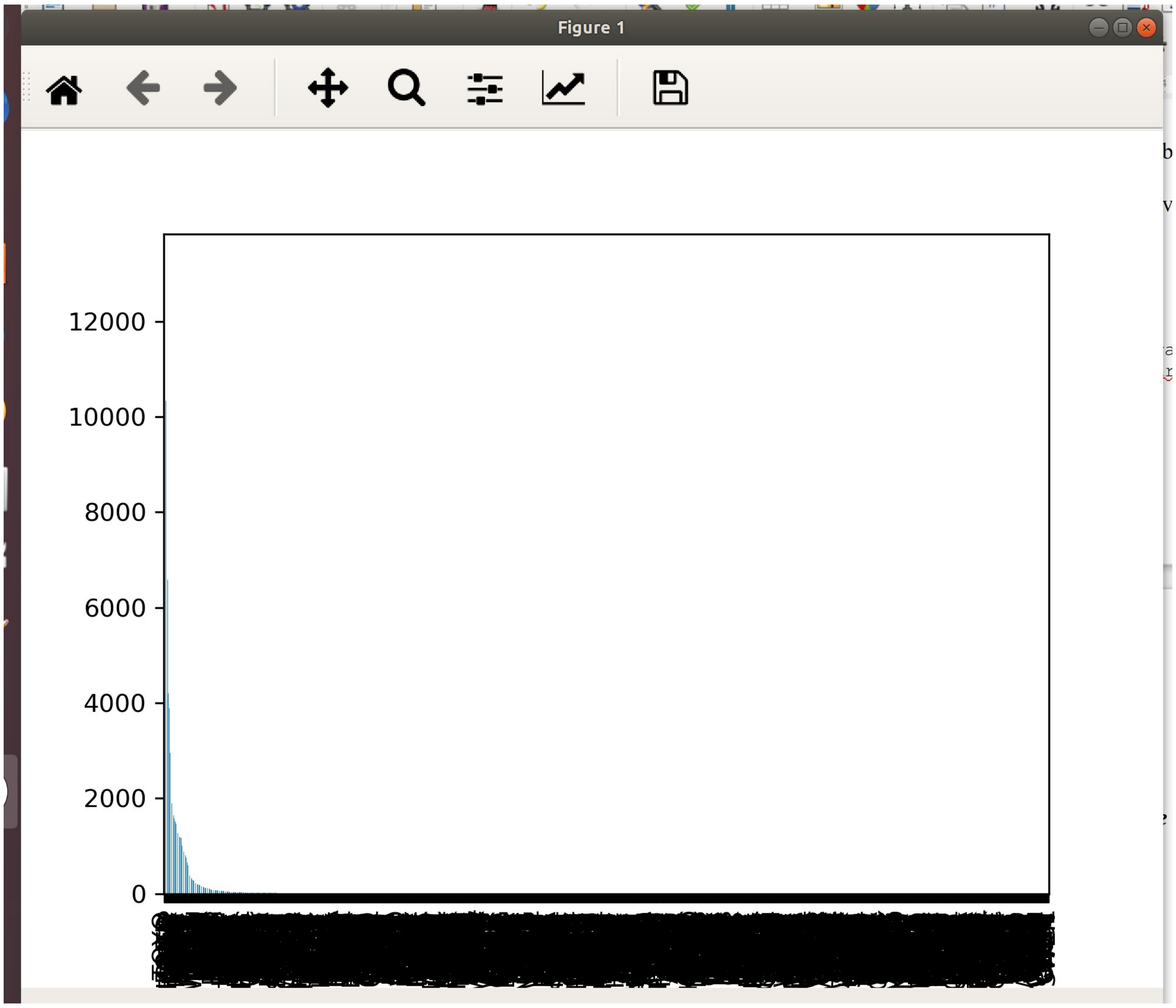
I declared another variable called “registration_tickets_makes” that took the contents of “registration_tickets” and stripped it of everything except the make.

```
>>> registration_tickets_makes = tickets.find({'Violation Description':
{'$eq': 'NO EVIDENCE OF REG'}}, {'Make': 1, '_id': 0})
```

```
>>> registration_tickets_makes[0]
{'Make': 'HOND'}
>>>
```

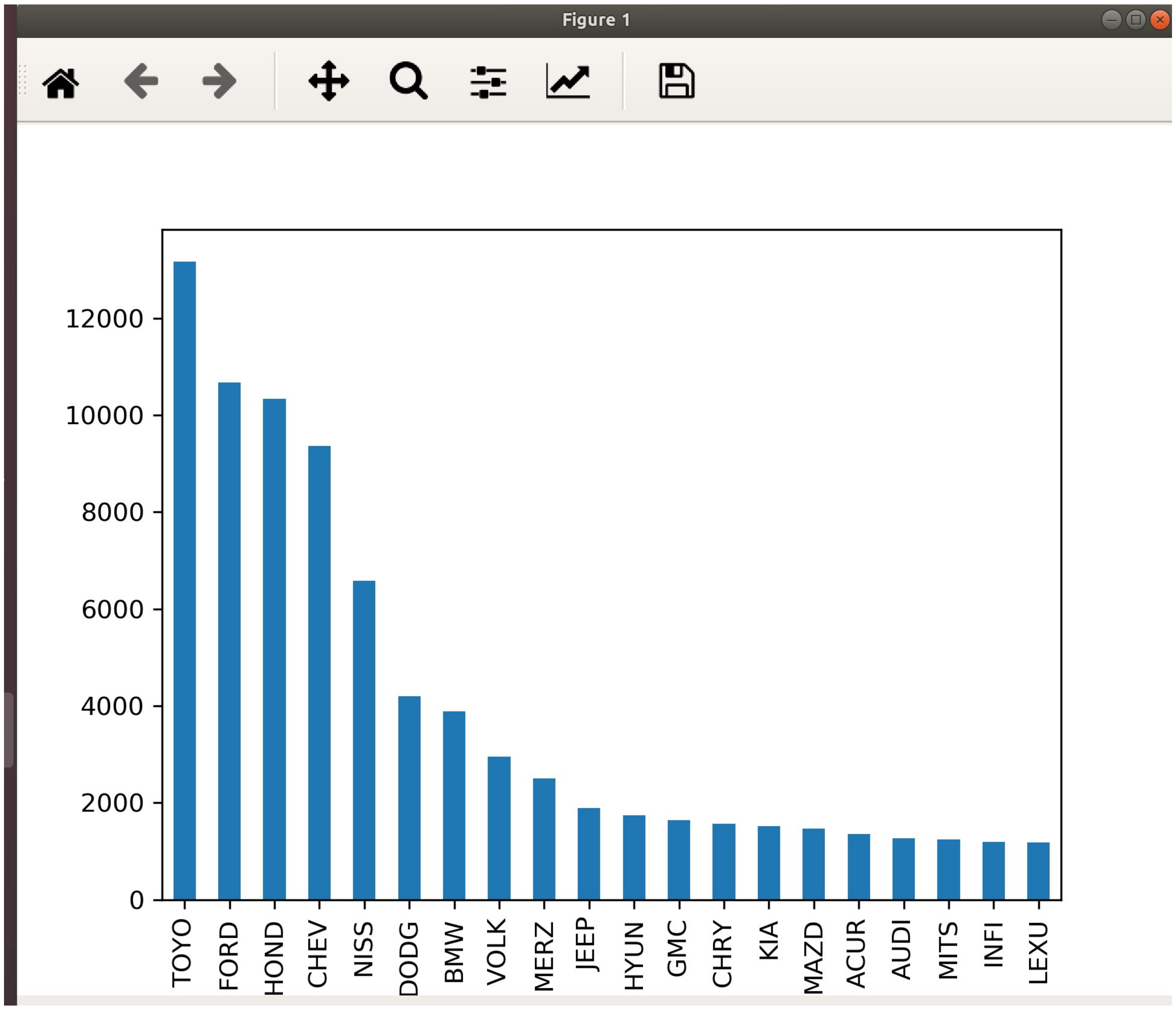
Finally, I imported pandas and matplotlib and used them to build a sorted bar chart of the makes from "registration_tickets_makes." I assigned the "df" and "ax" variables and used `plt.show()` to pull up the chart.

```
>>> import pandas as pd
>>> import matplotlib.pyplot as plt
>>> df = pd.io.json.json_normalize(registration_tickets_makes)
>>> ax = df['Make'].value_counts().plot.bar()
>>> plt.show()
```



This clearly has some problems with it. This was based on the entire data-set, so the majority of vehicle makes that only show up in the dataset a few times don't even have visible bars in relation to the most common makes. Additionally, There are so many observations that the horizontal axis appears as a near-solid black bar. To address this, I changed the "ax" variable to only show the top 20 results.

```
>>> ax = df['Make'].value_counts()[:20].plot.bar()
>>> plt.show()
```



This is obviously much better. I saved the output and moved on the the numbered exercises.

```
>>> fig = ax.get_figure()
>>> fig.savefig('make_barplot.png')
```

Exercises

1) What are the amounts and violation descriptions for the top 10 most expensive tickets?

While still in the Python shell, I started by trying to simply sort the data in descending order according to ‘Fine amount’, assigning it to a new variable called “expensive_tickets” where I eliminated all key*value pairs except for ‘Fine amount’ and ‘Violation Description.’ Unfortunately, this placed any tickets without a fine associated with them at the top of the list, so the first several items returned tickets for which there was no fine entry.

```
>>> expensive_tickets = tickets.find({}, {'Fine amount' : 1, 'Violation
Description' : 1, '_id' : 0}).sort('Fine amount', -1)

>>> expensive_tickets[0]
{'Violation Description': '17104h', 'Fine amount': ''}

>>> expensive_tickets[1]
{'Violation Description': '17104h', 'Fine amount': ''}
```

To fix this, I needed to filter out results where ‘Fine amount’ is empty. I used the `‘$ne’` parameter for the `.find()` method, which is ‘not equal’ in MongoDB’s query language. I then used a `for` loop to print the first 10 entries to the console.

```
>>> expensive_tickets = tickets.find({'Fine amount' : {'$ne' : ''}}, {'Fine
amount' : 1, 'Violation Description' : 1, '_id' : 0}).sort('Fine amount', -1)

>>> for i in range(10):
...     expensive_tickets[i]
...

{'Violation Description': '8755**', 'Fine amount': 505}
{'Violation Description': '8755**', 'Fine amount': 505}
{'Violation Description': '8755**', 'Fine amount': 505}
{'Violation Description': '8755**', 'Fine amount': 505}
{'Violation Description': '8755**', 'Fine amount': 505}
{'Violation Description': '8755**', 'Fine amount': 505}
{'Violation Description': 'HANDICAP/NO DP ID', 'Fine amount': 363}
{'Violation Description': 22522, 'Fine amount': 363}
{'Violation Description': 'HANDICAP/CROSS HATCH', 'Fine amount': 363}
{'Violation Description': 'HANDICAP/NO DP ID', 'Fine amount': 363}
>>>
```

The 6 most expensive tickets were for a violation with code 8755** and were for $505 each. The four after that were all for $363, and three of them appear to be related to parking in a handicap space. The one of those four that was not explicitly related to a handicap violation was the 8th entry, with a violation description of 22522.

2) Use Python to connect to MongoDB

This was completed above by installing pymongo and importing the MongoClient from that library. I have been using the Python shell in my terminal to execute the commands.

3) Within Python, extract the entries from the DB where the license plate is not CA.

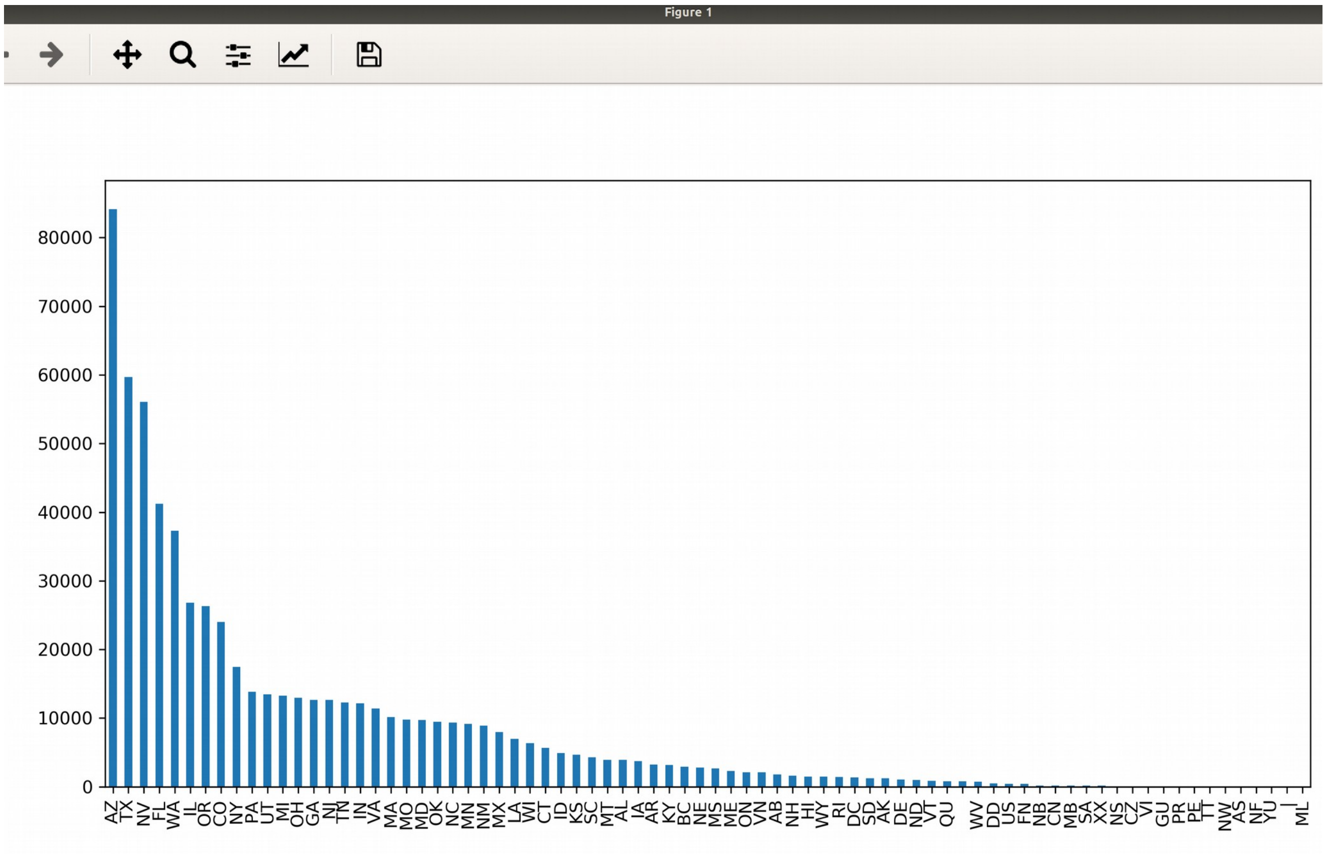
The commands used for this were very similar to the commands used for question #1. I needed to create a new variable made up of the 'tickets' collection with the California license plates filtered out. I used the same methods as before, and I stripped the results of everything except the 'RP State Plate' and 'Ticket number' fields. Using the .count() method, I showed there were 646,975 ticket records for non-CA plates. This was obviously too much to meaningfully list, so I used another for loop to list the top 10 entries.

```
>>> non_CA_license = tickets.find({'RP State Plate' : {'$ne' : 'CA'}},
{'Ticket number' : 1, 'RP State Plate' : 1, '_id' : 0})

>>> non_CA_license.count()
646975

>>> for i in range(10):
...     non_CA_license[i]
...

{'Ticket number': 1107780822, 'RP State Plate': 'FL'}
{'Ticket number': 1109455266, 'RP State Plate': 'NY'}
{'Ticket number': 1109704934, 'RP State Plate': 'OR'}
{'Ticket number': 1111884093, 'RP State Plate': 'WA'}
{'Ticket number': 1111884130, 'RP State Plate': 'NJ'}
{'Ticket number': 1111927526, 'RP State Plate': 'CO'}
{'Ticket number': 1112058872, 'RP State Plate': 'AR'}
{'Ticket number': 1112058931, 'RP State Plate': 'MA'}
{'Ticket number': 1112071726, 'RP State Plate': 'TX'}
{'Ticket number': 1112078704, 'RP State Plate': 'TX'}
```

4) Create a sorted bar chart of the states from #3.

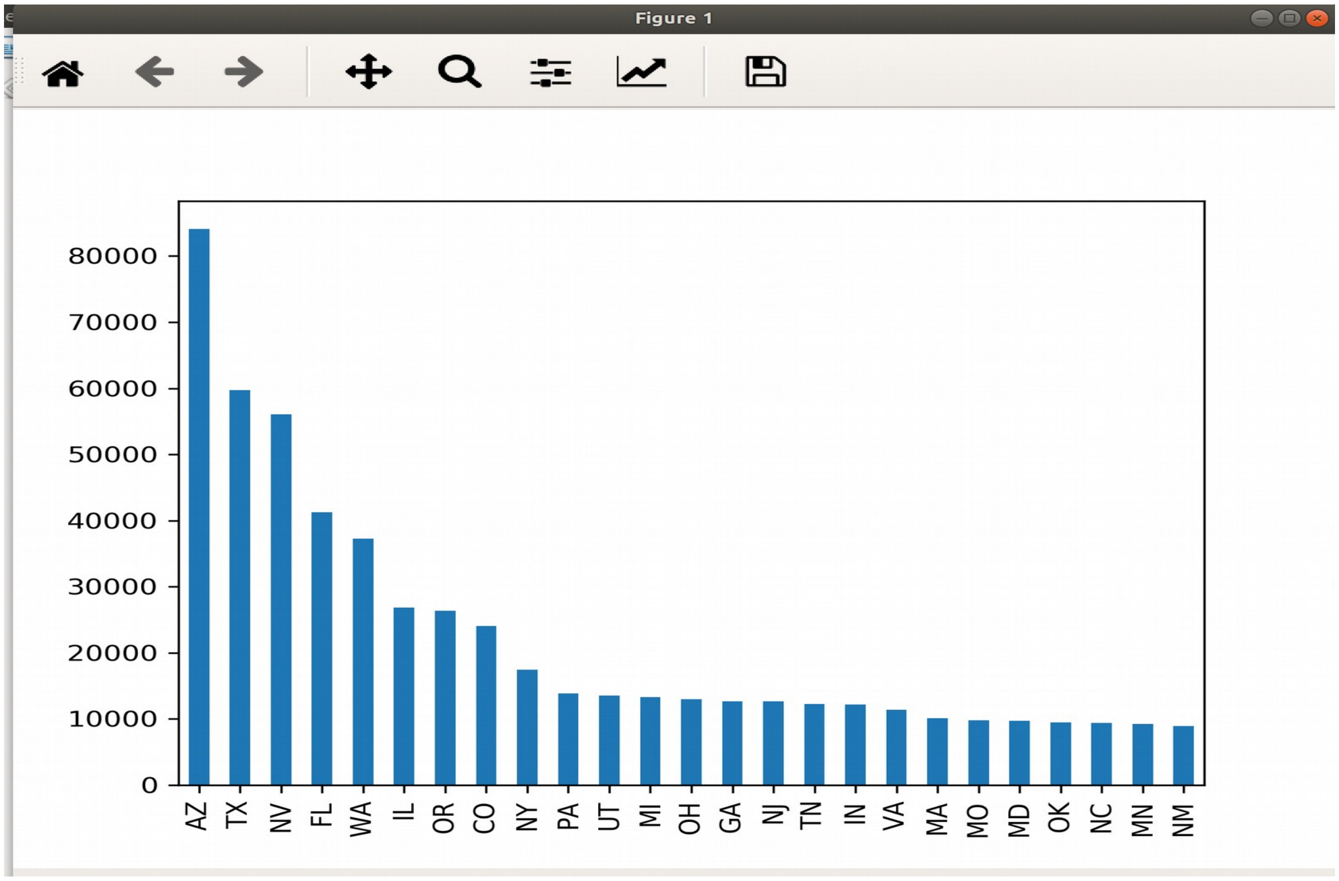
For the final step, I took the results from question #3 and plotted them in a bar chart. I was still in the same Python shell I ran the practice commands in, so I did not need to import pandas or matplotlib again. I reassigned the “df” and “ax” variables and then used plt.show() to display the chart.

```
>>> df = pd.io.json.json_normalize(non_CA_license)
>>> ax = df['RP State Plate'].value_counts().plot.bar()
>>> plt.show()
```



While this one is certainly more readable than the first graph of the vehicle makes, it’s still a little small and difficult to read. I repeated this step, limiting it to the 25 most frequent states.

```
>>> ax = df['RP State Plate'].value_counts()[:25].plot.bar()
>>> plt.show()
```



Discussion

It is clear to me now why MongoDB is so widely used and touted. After an initial learning curve, the queries are straightforward and easy to work with. When interfacing with Python, the ease of use is combined with Python's extensive set of libraries for data analysis and visualization, creating a synergistic relationship where the results are extremely powerful and versatile. As was the case with the SQL lab, this was my first time directly interacting with a NoSQL database, and I have to say I prefer the latter. While I was surprised at how much I enjoyed working with RDBMS and see many benefits to using them for specific problems, I found the key*value pairs and flexibility of NoSQL to be easier to work with. The two types of databases address different needs, and neither seem to be going anywhere soon.

To start this exercise, I installed MongoDB onto my laptop and used the Mongo shell to query the database. I expected the proprietary query language to be hard to work with and inferior to SQL, but this belief did not hold. As indicated before, there was certainly a learning curve – but I did not find a noticeable difference between the two once I understood the general syntax. Displaying specific ticket records to the console helped familiarize myself with the data and how it was stored. Eventually I moved on to link Python with MongoDB using Pymongo, and this was where I was truly sold on MongoDB and NoSQL as a whole. The freedom to work in a language with which I am very comfortable using data structures (cursors) that behave similarly to other objects I have worked with made me feel very much in control of the data. I am certain there are Python interfaces for SQL RDBMS out there, but it was the JSON-like objects that really sold me. After exploring using Python, I completed the numbered exercises and created the charts printed above.

Altogether, this lab was very informative and helped me "turn a corner" in terms of understanding databases. I've had significant knowledge gaps when it came to understanding databases in the past because they are frequently explained in terms of one another – RDBMS is contrasted to NoSQL and vice versa. The material is often geared toward those with a pre-existing understanding of databases, so I often left resources with more questions than I came with. Now that I have direct experience with both types, those gaps are starting to close. I am nowhere near being an expert, but I finally feel competent navigating and discussing databases and their qualities.