Week 6: File ingestion and schema validation

Name: Niyusha Baghayi
Batch Code: LICAN01
Submission Data: 4/14/2021
Submitted to: Data Glacier

Table of Contents

Introduction

Data pipelining is an important part of the data engineering, we should know how to read data in an optimal way to use resources efficiently.

Here we have a sample to see what's the difference between different ways of reading data and find the optimal way to do so and after that try to validate input data by YAML file and if the file is valid, then write it in a .gz file.

Dataset

The dataset that I used is “Parking_Violations_Issued_-_Fiscal_Year_2015” Dataset. The below features are the features of this dataset. The size of the dataset is 2.67 GB.

- Summons Number: Number
- Plate ID:Plain Text
- Registration State: Plain Text
- Plate Type: Plain Text
- Issue Date: Date & Time
- Violation Code: Number
- Vehicle Body Type: Plain Text
- Vehicle Make: Plain Text
- Issuing Agency: Plain Text
- Street Code1: Number
- Street Code2: Number
- Street Code3: Number
- Vehicle Expiration Date: Number
- Violation Location: Plain Text
- Violation Precinct: Number
- Issuer Precinct: Number
- Issuer Code: Number
- Issuer Command: Plain Text
- Issuer Squad: Plain Text
- Violation Time: Plain Text
- Time First Observed: Plain Text
- Violation County: Plain Text
- Violation In Front Of Or Opposite: Plain Text
- House Number: Plain Text
- Street Name: Plain Text
- Intersecting Street: Plain Text
- Date First Observed: Number
- Law Section: Number
- Sub Division: Plain Text
- Violation Legal Code: Plain Text

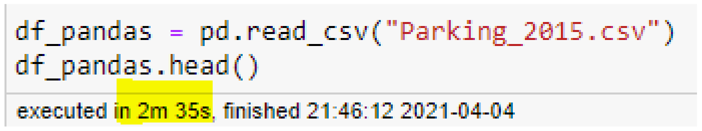
- Days Parking In Effect: Plain Text
- From Hours In Effect: Plain Text
- To Hours In Effect: Plain Text
- Vehicle Color: Plain Text
- Unregistered Vehicle?: Plain Text
- Vehicle Year: Number
- Meter Number: Plain Text
- Feet From Curb: Number
- Violation Post Code: Plain Text
- Violation Description: Plain Text
- No Standing or Stopping Violation: Plain Text
- Hydrant Violation: Plain Text
- Double Parking Violation: Plain Text
- Latitude: Number
- Longitude: Number
- Community Board: Number
- Community Council: Number
- Census Tract: Number
- BIN: Number
- BBL: Number
- NTA: Plain Text[1]

[1] https://www.kaggle.com/new-york-city/nyc-parking-tickets

Solutions to Read Data

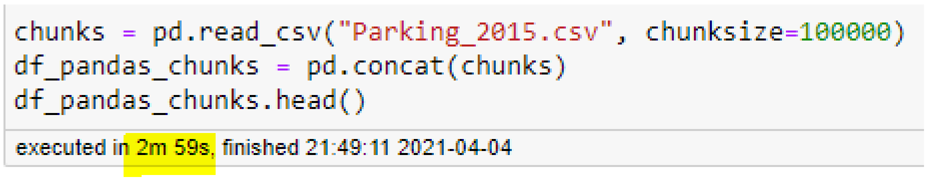
Here I have tried to test different libraries to read and in the following parts we can see them with their results:

Pandas:



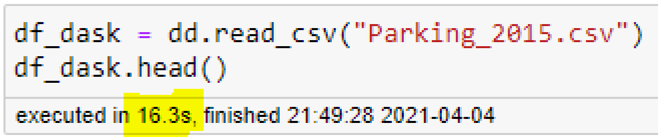
```
df_pandas = pd.read_csv("Parking_2015.csv")
df_pandas.head()
executed in 2m 35s, finished 21:46:12 2021-04-04
```

Pandas with Chunks:



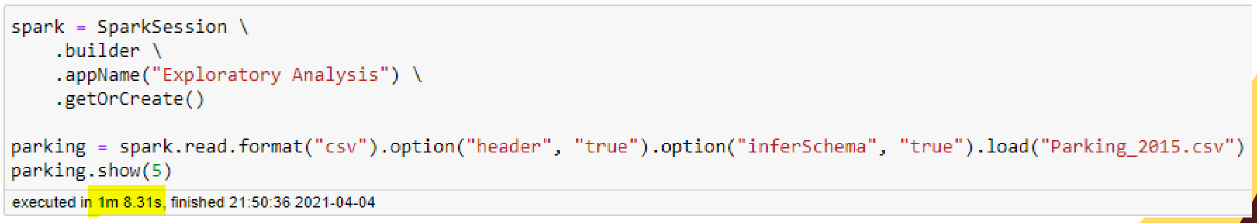
```
chunks = pd.read_csv("Parking_2015.csv", chunksize=100000)
df_pandas_chunks = pd.concat(chunks)
df_pandas_chunks.head()
executed in 2m 59s, finished 21:49:11 2021-04-04
```

Dask:



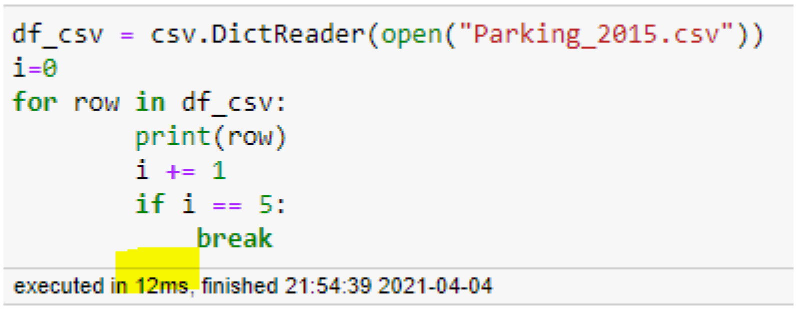
```
df_dask = dd.read_csv("Parking_2015.csv")
df_dask.head()
executed in 16.3s, finished 21:49:28 2021-04-04
```

Pyspark:



```
spark = SparkSession \
    .builder \
    .appName("Exploratory Analysis") \
    .getOrCreate()

parking = spark.read.format("csv").option("header", "true").option("inferSchema", "true").load("Parking_2015.csv")
parking.show(5)
executed in 1m 8.31s, finished 21:50:36 2021-04-04
```

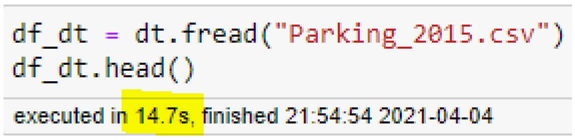
CSV Dict Reader:



```
df_csv = csv.DictReader(open("Parking_2015.csv"))
i=0
for row in df_csv:
        print(row)
        i += 1
        if i == 5:
            break
```

```
executed in 12ms, finished 21:54:39 2021-04-04
```

Datatable fread:



```
df_dt = dt.fread("Parking_2015.csv")
df_dt.head()
```

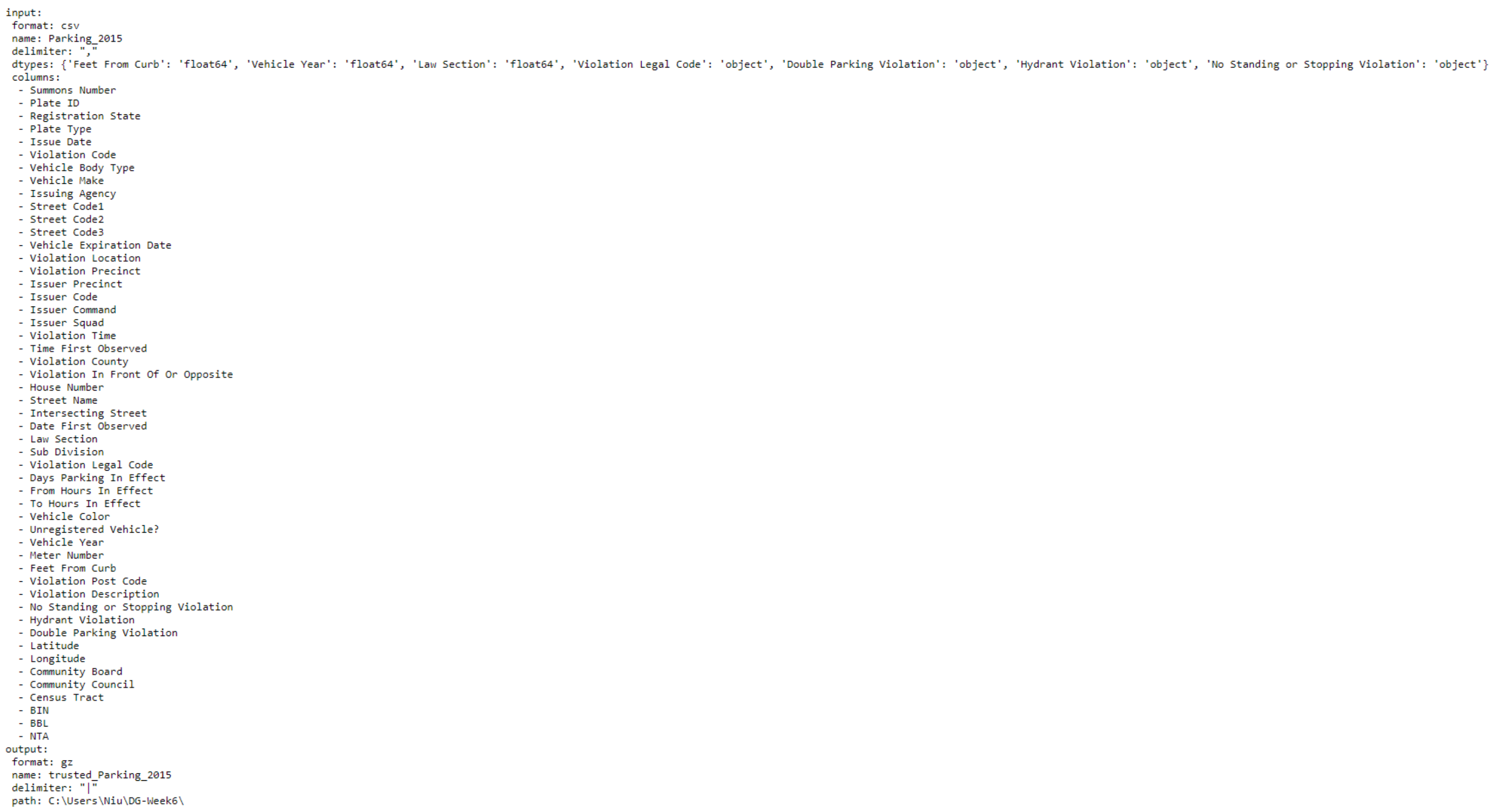
```
executed in 14.7s, finished 21:54:54 2021-04-04
```

Best Solution

For sure as the executing times illustrate totally “CSV Dict Reader” is the best one, but "Dask" is the good choice in this scenario to read data in Data frame mode. So I have used “Dask” for the rest parts.

Using YAML File

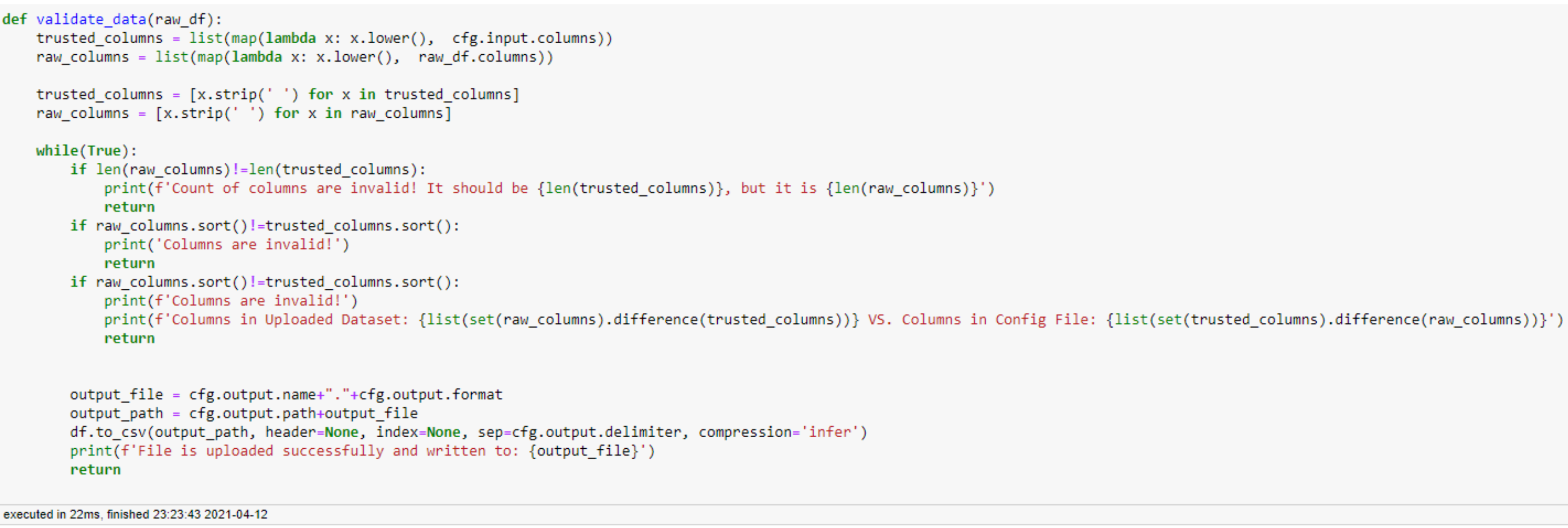
First of all we should create a YAML file, I have done that in this way:



```yaml
input:
 format: csv
 name: Parking_2015
 delimiter: ","
 dtypes: {'Feet From Curb': 'float64', 'Vehicle Year': 'float64', 'Law Section': 'float64', 'Violation Legal Code': 'object', 'Double Parking Violation': 'object', 'Hydrant Violation': 'object', 'No Standing or Stopping Violation': 'object'}
 columns:
  - Summons Number
  - Plate ID
  - Registration State
  - Plate Type
  - Issue Date
  - Violation Code
  - Vehicle Body Type
  - Vehicle Make
  - Issuing Agency
  - Street Code1
  - Street Code2
  - Street Code3
  - Vehicle Expiration Date
  - Violation Location
  - Violation Precinct
  - Issuer Precinct
  - Issuer Code
  - Issuer Command
  - Issuer Squad
  - Violation Time
  - Time First Observed
  - Violation County
  - Violation In Front Of Or Opposite
  - House Number
  - Street Name
  - Intersecting Street
  - Date First Observed
  - Law Section
  - Sub Division
  - Violation Legal Code
  - Days Parking In Effect
  - From Hours In Effect
  - To Hours In Effect
  - Vehicle Color
  - Unregistered Vehicle?
  - Vehicle Year
  - Meter Number
  - Feet From Curb
  - Violation Post Code
  - Violation Description
  - No Standing or Stopping Violation
  - Hydrant Violation
  - Double Parking Violation
  - Latitude
  - Longitude
  - Community Board
  - Community Council
  - Census Tract
  - BIN
  - BBL
  - NTA
output:
 format: gz
 name: trusted_Parking_2015
 delimiter: "|"
 path: C:\Users\Niu\DG-Week6\
```

As you can see here, I declared different parameters in nested way, for input and output data.

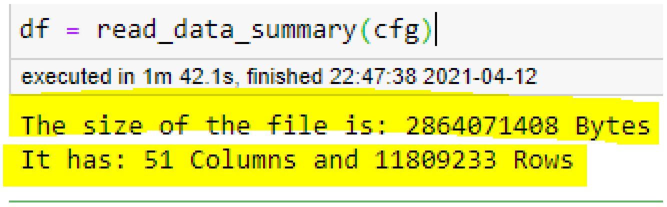
In the below function we get data and check all validation like count of columns, columns names and if it is OK, then we can write the file in .gz format as an output:



```
def validate_data(raw_df):
    trusted_columns = list(map(lambda x: x.lower(),  cfg.input.columns))
    raw_columns = list(map(lambda x: x.lower(),  raw_df.columns))

    trusted_columns = [x.strip(' ') for x in trusted_columns]
    raw_columns = [x.strip(' ') for x in raw_columns]

    while(True):
        if len(raw_columns)!=len(trusted_columns):
            print(f'Count of columns are invalid! It should be {len(trusted_columns)}, but it is {len(raw_columns)}')
            return
        if raw_columns.sort()!=trusted_columns.sort():
            print('Columns are invalid!')
            return
        if raw_columns.sort()!=trusted_columns.sort():
            print(f'Columns are invalid!')
            print(f'Columns in Uploaded Dataset: {list(set(raw_columns).difference(trusted_columns))} VS. Columns in Config File: {list(set(trusted_columns).difference(raw_columns))}')
            return


        output_file = cfg.output.name+"."+cfg.output.format
        output_path = cfg.output.path+output_file
        df.to_csv(output_path, header=None, index=None, sep=cfg.output.delimiter, compression='infer')
        print(f'File is uploaded successfully and written to: {output_file}')
        return
```

```
executed in 22ms, finished 23:23:43 2021-04-12
```

And also we can find the size of data in this way:



```
df = read_data_summary(cfg)
```

```
executed in 1m 42.1s, finished 22:47:38 2021-04-12
```

```
The size of the file is: 2864071408 Bytes
It has: 51 Columns and 11809233 Rows
```