**Tips and Tricks for Loading Large CSV Files into Pandas DataFrames — Part 2**

**Learn how to selectively load part of your CSV file into a DataFrame and at the same time reduce it’s memory footprint**



Photo by [Shubham Dhage](https://unsplash.com/@theshubhamdhage?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com?utm_source=medium&utm_medium=referral)

In my previous article —**Tips and Tricks for Loading Large CSV Files into Pandas DataFrames — Part 1**…

**Our Dataset**

For our examples in this article, I am using the **Flight Prices** dataset from <https://www.kaggle.com/datasets/dilwong/flightprices>. Each row in the dataset is a purchasable ticket found on Expedia between 2022–04–16 and 2022–10–05, to/from the following airports: ATL, DFW, DEN, ORD, LAX, CLT, MIA, JFK, EWR, SFO, DTW, BOS, PHL, LGA, IAD, OAK.

License type: Attribution 4.0 International (CC BY 4.0)

The main reason for choosing this dataset is that it is really large — 31GB in size and contains more than 82 million rows, and 27 columns. So this should be a good and realistic challenge to those of you whose machine has limited amount of memory to try to work with this dataset.

**Loading the entire CSV file into a DataFrame**

The first thing I want to try is to load the CSV file normally into a dataframe. I would like to time the loading process and see how long it takes to load such a file:

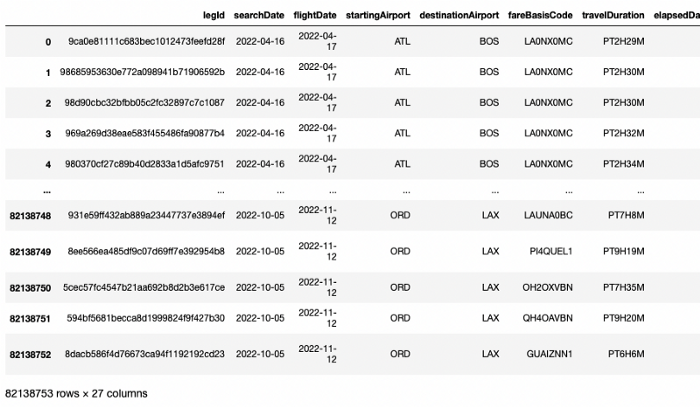
import time  
import pandas as pd  
  
start = time.time()  
  
df = pd.read\_csv('itineraries.csv')  
  
print(time.time() - start, 'seconds')

And so I waited and waited. In the end, it took 1164 seconds (close to 20 minutes) to load the entire CSV file into a Pandas DataFrame.

For reference, my machine is the Mac Studio with 32GB RAM. The above code snippet could not even run on my older M1 Mac Mini with 8GB RAM (the kernel died while trying to load the CSV file).

Let’s have a look at the dataframe:

display(df)



The dataframe has 82,138,753 (> 82 million) rows and 27 columns. To see how much memory is used by the dataframe, use the info() function and set the memory\_usage parameter to deep:

display(df.info(memory\_usage='deep'))

From the result below, you can see that the dataframe has a memory footprint of 110.5 GB!

<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 82138753 entries, 0 to 82138752  
Data columns (total 27 columns):  
 # Column Dtype   
--- ------ -----   
 0 legId object   
 1 searchDate object   
 2 flightDate object   
 3 startingAirport object   
 4 destinationAirport object   
 5 fareBasisCode object   
 6 travelDuration object   
 7 elapsedDays int64   
 8 isBasicEconomy bool   
 9 isRefundable bool   
 10 isNonStop bool   
 11 baseFare float64  
 12 totalFare float64  
 13 seatsRemaining int64   
 14 totalTravelDistance float64  
 15 segmentsDepartureTimeEpochSeconds object   
 16 segmentsDepartureTimeRaw object   
 17 segmentsArrivalTimeEpochSeconds object   
 18 segmentsArrivalTimeRaw object   
 19 segmentsArrivalAirportCode object   
 20 segmentsDepartureAirportCode object   
 21 segmentsAirlineName object   
 22 segmentsAirlineCode object   
 23 segmentsEquipmentDescription object   
 24 segmentsDurationInSeconds object   
 25 segmentsDistance object   
 26 segmentsCabinCode object   
dtypes: bool(3), float64(3), int64(2), object(19)  
**memory usage: 110.5 GB**

**Loading the DataFrame by Chunks**

Obviously, loading such a large CSV file is a time consuming and laborious affair (if it is even possible in the first place). So there must be better ways to optimize the process.

The first question to ask yourself is: do you really need all the rows and columns in the CSV file? For example, you might only need all rows whose starting airport is ATL, and ignore the rest. However, loading the entire CSV file into a dataframe only to drop all the rows that you don’t need is not an option as your computer might not even have enough memory to hold the entire dataframe. This is where *chunking* is useful.

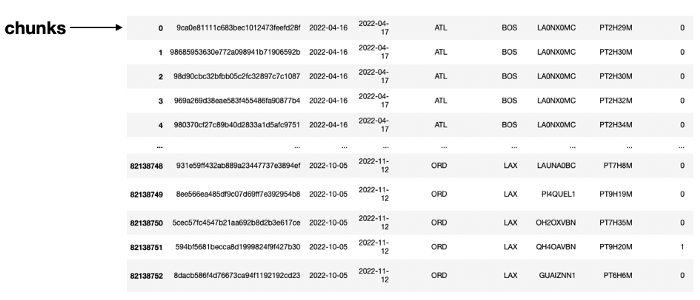
Chunking allows you to load the dataframe in smaller chunks (hence its name), rather than all in one go. Each chunk is a subset of the entire dataframe that you are loading, and the size of each chunk can be customized to be as small or as large as you want. When each chunk is loaded, you have the opportunity to filter the rows or columns in the dataframe and retain only those rows or columns that you need for further analytics.

To perform chunking on your dataframe, use the chunksize parameter in the read\_csv() function:

import time  
import pandas as pd  
  
start = time.time()  
chunks = pd.read\_csv('itineraries.csv', chunksize=100000)  
print(time.time() - start, ' seconds')  
  
# result is a TextFileReader object  
chunks

In the above code, the control returns almost immediately when you run it, and the output shows that the chunks variable is of type pandas.io.parsers.readers.TextFileReader.

Think of chunks as a file pointer that points to the first row in the CSV file, and it is ready to start reading the first 100,000 rows (as specified in the chunksize parameter).



In fact, chunks is an *iterable*, where you can iterate through it to load all the rows in the CSV file into a dataframe, 100,000 rows at a time.

To see how you can iterate through the chunks variable, let’s define a function named process\_chunk(), where I will only fetch all rows where the startingAirport column has the value of “**ATL**”:

def process\_chunk(df):  
 df = df.query('startingAirport == "ATL"')   
 print(df.shape)  
 return df

You can also filter columns if you wish in this function.

The following code snippet shows how to load the entire CSV file using the chunks iterable:

chunk\_list = [] # used for storing dataframes  
for chunk in chunks: # each chunk is a dataframe   
 # perform data filtering   
 filtered\_chunk = process\_chunk(chunk)  
   
 # Once the data filtering is done, append the filtered chunk to list  
 chunk\_list.append(filtered\_chunk)

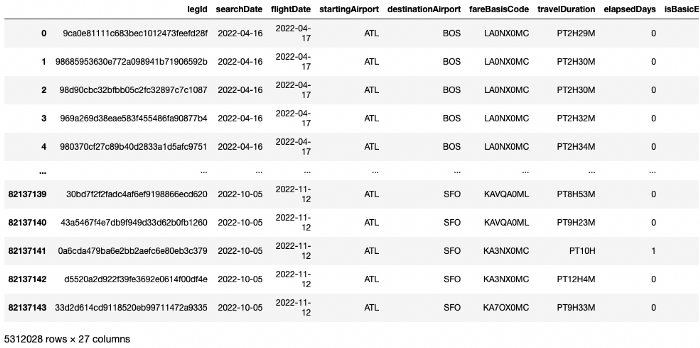
For each iteration of the chunks variable, the chunk iterator will contain a dataframe containing 100,000 rows. The dataframe is then passed into process\_chunk(), where the filtering is performed. When the filtered dataframe is returned, it is appended to a list named chunk\_list.

To combine all the dataframes in the list, you use the concat() function:

# concat all the dfs in the list into a single dataframe   
df\_concat = pd.concat(chunk\_list)

Let’s print out the final dataframe that is loaded:

display(df\_concat)



You will see that the final dataframe contains 5,312,028 rows.

By using chunking, you can now selectively load part of the CSV into the dataframe, instead of trying to load everything from the entire CSV file.

**Examining the Memory Footprint of the DataFrame**

Instead of the original 82 million rows, our dataframe now has a more manageable size of 5 millions rows. Let’s see how much memory it uses:

df\_concat.info()

Take note of the last line of output:

<class 'pandas.core.frame.DataFrame'>  
Int64Index: 5312028 entries, 0 to 82137143  
Data columns (total 27 columns):  
 # Column Dtype   
--- ------ -----   
 0 legId object   
 1 searchDate object   
 2 flightDate object   
 3 startingAirport object   
 4 destinationAirport object   
 5 fareBasisCode object   
 6 travelDuration object   
 7 elapsedDays int64   
 8 isBasicEconomy bool   
 9 isRefundable bool   
 10 isNonStop bool   
 11 baseFare float64  
 12 totalFare float64  
 13 seatsRemaining int64   
 14 totalTravelDistance float64  
 15 segmentsDepartureTimeEpochSeconds object   
 16 segmentsDepartureTimeRaw object   
 17 segmentsArrivalTimeEpochSeconds object   
 18 segmentsArrivalTimeRaw object   
 19 segmentsArrivalAirportCode object   
 20 segmentsDepartureAirportCode object   
 21 segmentsAirlineName object   
 22 segmentsAirlineCode object   
 23 segmentsEquipmentDescription object   
 24 segmentsDurationInSeconds object   
 25 segmentsDistance object   
 26 segmentsCabinCode object   
dtypes: bool(3), float64(3), int64(2), object(19)  
**memory usage: 1.0+ GB**

The memory usage of 1+ GB is just an estimate. To see the full memory usage, you need to set the memory\_usage parameter to deep:

df\_concat.info(memory\_usage='deep')

The result now shows that the actual memory footprint is 7.1 GB:

<class 'pandas.core.frame.DataFrame'>  
Int64Index: 5312028 entries, 0 to 82137143  
Data columns (total 27 columns):  
 # Column Dtype   
--- ------ -----   
 0 legId object   
 1 searchDate object   
 2 flightDate object   
 3 startingAirport object   
 4 destinationAirport object   
 5 fareBasisCode object   
 6 travelDuration object   
 7 elapsedDays int64   
 8 isBasicEconomy bool   
 9 isRefundable bool   
 10 isNonStop bool   
 11 baseFare float64  
 12 totalFare float64  
 13 seatsRemaining int64   
 14 totalTravelDistance float64  
 15 segmentsDepartureTimeEpochSeconds object   
 16 segmentsDepartureTimeRaw object   
 17 segmentsArrivalTimeEpochSeconds object   
 18 segmentsArrivalTimeRaw object   
 19 segmentsArrivalAirportCode object   
 20 segmentsDepartureAirportCode object   
 21 segmentsAirlineName object   
 22 segmentsAirlineCode object   
 23 segmentsEquipmentDescription object   
 24 segmentsDurationInSeconds object   
 25 segmentsDistance object   
 26 segmentsCabinCode object   
dtypes: bool(3), float64(3), int64(2), object(19)  
**memory usage: 7.1 GB**

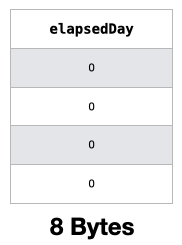
Why the huge difference? To understand why, let’s use the memory\_usage() function to see how much memory each column uses:

df\_concat.memory\_usage()

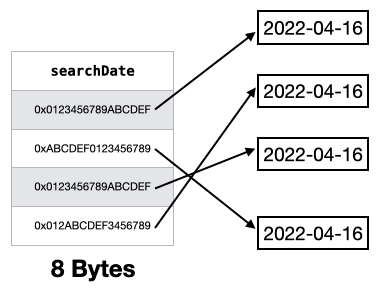
Interesting, it seems like all columns (except the three bool columns —**isBasicEconomy**, **isRefundable**, and **isNonStop**) uses the same amount of memory:

Index 42496224  
legId 42496224  
searchDate 42496224  
flightDate 42496224  
startingAirport 42496224  
destinationAirport 42496224  
fareBasisCode 42496224  
travelDuration 42496224  
elapsedDays 42496224  
isBasicEconomy 5312028  
isRefundable 5312028  
isNonStop 5312028  
baseFare 42496224  
totalFare 42496224  
seatsRemaining 42496224  
totalTravelDistance 42496224  
segmentsDepartureTimeEpochSeconds 42496224  
segmentsDepartureTimeRaw 42496224  
segmentsArrivalTimeEpochSeconds 42496224  
segmentsArrivalTimeRaw 42496224  
segmentsArrivalAirportCode 42496224  
segmentsDepartureAirportCode 42496224  
segmentsAirlineName 42496224  
segmentsAirlineCode 42496224  
segmentsEquipmentDescription 42496224  
segmentsDurationInSeconds 42496224  
segmentsDistance 42496224  
segmentsCabinCode 42496224  
dtype: int64

Using some simple calculations, you can see that each row in all these columns (except the three bool columns) uses 8 bytes (42496224 bytes / 5312028 rows). Why is this so? For columns with type int64 and float64, this is reasonable as 64 bits is equal to 8 bytes. For example, each value for the **elapsedDay** column will occupy 8 bytes (int64):



How about those object columns? Internally, object columns store their values separately in other memory locations. So the actual value stored for an object column is really the memory address of the location that stores the object value. Here is an example for the **searchDate** column, which is of object type:



To find out exactly how much storage is used by an object columns, you need to set the deep parameter in the memory\_usage() function to True:

df\_concat.memory\_usage(deep=True)

The output now shows a more accurate picture of how much memory is used by each column:

Index 42496224  
legId 472770492  
searchDate 355905876  
flightDate 355905876  
startingAirport 318721680  
destinationAirport 318721680  
fareBasisCode 343071603  
travelDuration 339818490  
elapsedDays 42496224  
isBasicEconomy 5312028  
isRefundable 5312028  
isNonStop 5312028  
baseFare 42496224  
totalFare 42496224  
seatsRemaining 42496224  
totalTravelDistance 42496224  
segmentsDepartureTimeEpochSeconds 403417584  
segmentsDepartureTimeRaw 579572987  
segmentsArrivalTimeEpochSeconds 403417584  
segmentsArrivalTimeRaw 579572987  
segmentsArrivalAirportCode 338518225  
segmentsDepartureAirportCode 338518225  
segmentsAirlineName 410952127  
segmentsAirlineCode 329246888  
segmentsEquipmentDescription 438243766  
segmentsDurationInSeconds 350013948  
segmentsDistance 341426543  
segmentsCabinCode 357085873  
dtype: int64

For example, each of value in the **searchDate** column uses **67** bytes (355905876 bytes / 5312028 rows), and each of the value in the **startingAirport** column uses **60** bytes (318721680 bytes / 5312028 rows).

**Converting Types**

As you can observe from the previous section, the memory used by the two columns — **searchDate** and **flightDate**, each value (which is a date) takes up 67 bytes when stored as an object type. As these column contains dates, it is inefficient to store them as object type as manipulating them later would be difficult. Instead, it is better to convert them to the datetime64 type. Doing so would also save memory as they now will only take up 8 bytes each.

So let’s now convert these columns into thedatetime64 type:

df\_concat["searchDate"] = df\_concat["searchDate"].astype('datetime64')  
df\_concat["flightDate"] = df\_concat["flightDate"].astype('datetime64')

You can now check the memory usage of each column:

df\_concat.memory\_usage(deep=True)

You can now see that the memory usage of both columns have been reduced to 42496224 bytes. Each value takes 8 bytes (42496224 bytes / 5312028 rows) as expected:

Index 42496224  
legId 472770492  
**searchDate 42496224  
flightDate 42496224**startingAirport 318721680  
destinationAirport 318721680  
fareBasisCode 343071603  
travelDuration 339818490  
elapsedDays 42496224  
isBasicEconomy 5312028  
isRefundable 5312028  
isNonStop 5312028  
baseFare 42496224  
totalFare 42496224  
seatsRemaining 42496224  
totalTravelDistance 42496224  
segmentsDepartureTimeEpochSeconds 403417584  
segmentsDepartureTimeRaw 579572987  
segmentsArrivalTimeEpochSeconds 403417584  
segmentsArrivalTimeRaw 579572987  
segmentsArrivalAirportCode 338518225  
segmentsDepartureAirportCode 338518225  
segmentsAirlineName 410952127  
segmentsAirlineCode 329246888  
segmentsEquipmentDescription 438243766  
segmentsDurationInSeconds 350013948  
segmentsDistance 341426543  
segmentsCabinCode 357085873  
dtype: int64

And we want to know the memory footprint of our dataframe now:

df\_concat.info(memory\_usage='deep')

It has now been reduced from 7.1 GB to 6.5 GB:

<class 'pandas.core.frame.DataFrame'>  
Int64Index: 5312028 entries, 0 to 82137143  
Data columns (total 27 columns):  
 # Column Dtype   
--- ------ -----   
 0 legId object   
 **1 searchDate datetime64[ns]  
 2 flightDate datetime64[ns]**  
 3 startingAirport object   
 4 destinationAirport object   
 5 fareBasisCode object   
 6 travelDuration object   
 7 elapsedDays int64   
 8 isBasicEconomy bool   
 9 isRefundable bool   
 10 isNonStop bool   
 11 baseFare float64   
 12 totalFare float64   
 13 seatsRemaining int64   
 14 totalTravelDistance float64   
 15 segmentsDepartureTimeEpochSeconds object   
 16 segmentsDepartureTimeRaw object   
 17 segmentsArrivalTimeEpochSeconds object   
 18 segmentsArrivalTimeRaw object   
 19 segmentsArrivalAirportCode object   
 20 segmentsDepartureAirportCode object   
 21 segmentsAirlineName object   
 22 segmentsAirlineCode object   
 23 segmentsEquipmentDescription object   
 24 segmentsDurationInSeconds object   
 25 segmentsDistance object   
 26 segmentsCabinCode object   
dtypes: bool(3), datetime64[ns](2), float64(3), int64(2), object(17)  
**memory usage: 6.5 GB**

Converting date-related columns to the datetime64 type not only saves memory, but it also makes it easier to perform your data analytics.

**Down-casting Columns**

The next improvement we can make to the dateframe is down-casting all the numeric fields. Consider the **seatsRemaining** column. If you check the range of values (smallest and largest) in this column, you will see that the minimum is 0 and maximum is 10:

df\_concat['seatsRemaining'].min() # 0   
df\_concat['seatsRemaining'].max() # 10

And yet, an int64 type was used to store this column. An int64 column can store values from -(2⁶³ — 1) to 2⁶³. Apparently this datatype is an overkill for this column, where it stores only non-negative values and whose maximum value is no more than 10. The same might apply to other columns, such as **baseFare**, **totalFare**, as well as **totalTravelDistance**. To reduce such wastage of memory, we need to downcast the types used by such columns to types that are just sufficient to hold their values. For example, the **seatsRemaining** column could work just fine with an uint8 data type, which can hold values from 0 to (2⁸ — 1), or 0 to 255.

Let’s now try to downcast all the numeric columns (integers and floating-points).

First, define a function named memory\_usage() to take in either a Pandas DataFrame or Series and return its total memory usage:

def mem\_usage(obj):  
 if isinstance(obj, pd.DataFrame):  
 usage\_b = obj.memory\_usage(deep=True).sum()  
 else: # we assume if not a df then it's a series  
 usage\_b = obj.memory\_usage(deep=True)  
  
 usage\_mb = usage\_b / 1024 \*\* 2 # bytes to megabytes   
 return "{:03.2f} MB".format(usage\_mb)

Next, define a function named downcast\_type() to take in a dataframe, and convert columns of one type and downcast them to a new type:

def downcast\_type(df, old\_type, new\_type):  
 # get all the columns with the specified type – old\_type  
 df\_oldtype = df.select\_dtypes(include=[old\_type])  
  
 # convert all the columns from old\_type to new\_type  
 df\_newtype = df\_oldtype.apply(pd.to\_numeric, downcast=new\_type)  
   
 # print out the memory usage of old\_type and new\_type  
 print(f'{old\_type} memory usage: {mem\_usage(df\_oldtype)}')  
 print(f'{new\_type} memory usage: {mem\_usage(df\_newtype)}')  
  
 return df\_newtype

The downcast parameter in the apply() function downcasts the resulting data to the smallest numerical data type possible according to the following rules:

* ‘integer’ or ‘signed’: smallest signed int dtype (min.: np.int8)
* ‘unsigned’: smallest unsigned int dtype (min.: np.uint8)
* ‘float’: smallest float dtype (min.: np.float32)

**Source**: <https://pandas.pydata.org/docs/reference/api/pandas.to_numeric.html>

Let’s try to downcast all the integer columns to *unsigned*:

df\_converted\_int = downcast\_type(df\_concat, "int", "unsigned")

For Windows, in the above statement replace “int” with “int64”

You will see the following output:

int memory usage: 121.58 MB  
unsigned memory usage: 50.66 MB

This means that the memory of all the integer columns in the dataframe have been reduced from 121.58 MB to 50.66 MB.

Let’s also downcast all the floating-point columns:

df\_converted\_float = downcast\_type(df\_concat, "float", "float")

The output shows the reduction in memory usage for these floating point columns:

float memory usage: 162.11 MB  
float memory usage: 101.32 MB

I will now create a copy of the dataframe in df\_concat and then replace all the columns that have been downcasted:

# make a copy of the dataframe and replace the columns with the downcasted types  
optimized\_df = df\_concat.copy()  
optimized\_df[df\_converted\_int.columns] = df\_converted\_int  
optimized\_df[df\_converted\_float.columns] = df\_converted\_float

You can examine the various columns that have their types downcasted:

optimized\_df.info(memory\_usage='deep')

The following output in bold shows the columns that have been downcasted:

<class 'pandas.core.frame.DataFrame'>  
Int64Index: 5312028 entries, 0 to 82137143  
Data columns (total 27 columns):  
 # Column Dtype   
--- ------ -----   
 0 legId object   
 1 searchDate datetime64[ns]  
 2 flightDate datetime64[ns]  
 3 startingAirport object   
 4 destinationAirport object   
 5 fareBasisCode object   
 6 travelDuration object   
 **7 elapsedDays uint8**    
 8 isBasicEconomy bool   
 9 isRefundable bool   
 10 isNonStop bool   
 **11 baseFare float32   
 12 totalFare float32   
 13 seatsRemaining uint8**    
 **14 totalTravelDistance float32**   
 15 segmentsDepartureTimeEpochSeconds object   
 16 segmentsDepartureTimeRaw object   
 17 segmentsArrivalTimeEpochSeconds object   
 18 segmentsArrivalTimeRaw object   
 19 segmentsArrivalAirportCode object   
 20 segmentsDepartureAirportCode object   
 21 segmentsAirlineName object   
 22 segmentsAirlineCode object   
 23 segmentsEquipmentDescription object   
 24 segmentsDurationInSeconds object   
 25 segmentsDistance object   
 26 segmentsCabinCode object   
dtypes: bool(3), datetime64[ns](2), float32(3), object(17), uint8(2)  
**memory usage: 6.4 GB**

The memory footprint has been further reduced to 6.4 GB.

Down-casting helps you to save memory by automatically checking the range of values in each column and converting the column to the data type that is sufficient to hold the existing values.

**Converting Object Type to Category**

The memory footprint for the dataframe is still relatively large. The real culprits are those columns of type object that we have seen earlier.

Let’s find all those object columns and examine them in more details:

# get all the columns with the Object type  
df\_obj = optimized\_df.select\_dtypes(include=['object']).copy()  
df\_unique\_objs = df\_obj.describe()  
  
print(df\_unique\_objs)

Here’s the output:

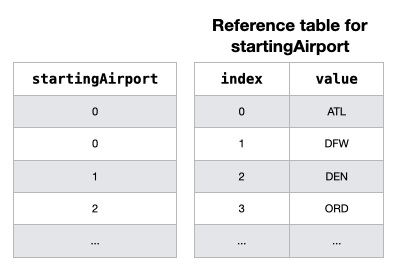
legId startingAirport destinationAirport \  
count 5312028 **5312028** **5312028**   
unique 362460 **1** **15**   
top 3c18dd8a485f8f15b31e68982f53db00 ATL LAX   
freq 60 5312028 709809   
  
 fareBasisCode travelDuration segmentsDepartureTimeEpochSeconds \  
count 5312028 5312028 5312028   
unique 6035 1480 320710   
top KAUOA0MQ PT2H15M 1663672500   
freq 107336 79204 377   
  
 segmentsDepartureTimeRaw segmentsArrivalTimeEpochSeconds \  
count 5312028 5312028   
unique 321658 344387   
top 2022-09-20T07:15:00.000-04:00 1661900340   
freq 377 237   
  
 segmentsArrivalTimeRaw segmentsArrivalAirportCode \  
count 5312028 5312028   
unique 348155 1439   
top 2022-08-30T18:59:00.000-04:00 LGA   
freq 237 192573   
  
 segmentsDepartureAirportCode segmentsAirlineName segmentsAirlineCode \  
count 5312028 5312028 5312028   
unique 637 59 59   
top ATL Delta DL   
freq 1754883 1031841 1031841   
  
 segmentsEquipmentDescription segmentsDurationInSeconds \  
count 5155578 5312028   
unique 2514 24456   
top Airbus A321 8100   
freq 427699 79204   
  
 segmentsDistance segmentsCabinCode   
count 5312028 5312028   
unique 1352 34   
top None||None coach||coach   
freq 556848 3168404

The idea is to see how the number of unique values in each of the object columns. For example, consider the **startingAirport** column which has only 1 unique value (which is ATL).

You might even argue that we could altogether drop this **startingAirport** column! But for clarity I will keep this column.

In this case, storing the string “**ATL”** 5312028 times in the dataframe is not a very wise choice. Likewise, for the **destinationAirport** column, there are only 15 unique values, and hence there are a lot of duplicate values stored in the column.

Instead of storing all these duplicate values, it would be better to convert columns like **startingAirport** and **destinationAirport** to the category data type. In the category data type, the unique values in the column are stored in a separate reference table, and the indices of the values are then stored within that column. The following figure illustrates this concept:



By converting object columns into category type, you can *save a lot* of memory. However, there is one caveat. If a column contains a lot of unique values, then converting it into a category field is not going to save you any memory — in fact it is going to use more memory. Hence, I want to calculate the percentage of unique values for each object column:

df\_unique\_objs.iloc[1,:] / df\_unique\_objs.iloc[0,:] \* 100

If more than 50% of the values in a column are unique, it is more feasible to continue using the object type rather than converting them to category type. The following output shows that none of the object columns has more than 50% uniqueness:

legId 6.823383  
startingAirport 0.000019  
destinationAirport 0.000282  
fareBasisCode 0.11361  
travelDuration 0.027861  
segmentsDepartureTimeEpochSeconds 6.037431  
segmentsDepartureTimeRaw 6.055277  
segmentsArrivalTimeEpochSeconds 6.483155  
segmentsArrivalTimeRaw 6.554088  
segmentsArrivalAirportCode 0.027089  
segmentsDepartureAirportCode 0.011992  
segmentsAirlineName 0.001111  
segmentsAirlineCode 0.001111  
segmentsEquipmentDescription 0.048763  
segmentsDurationInSeconds 0.460389  
segmentsDistance 0.025452  
segmentsCabinCode 0.00064  
dtype: object

We can now go ahead and convert the object columns to the category data type:

df\_obj\_cat = df\_obj.astype('category')

Once the columns are converted, assign it back to the optimized\_df dataframe:

optimized\_df[df\_obj\_cat.columns] = df\_obj\_cat

We can now examine the memory usage for each column:

optimized\_df.memory\_usage(deep=True)

Straight-away, we can see that the various previously-object columns now uses much lesser memory:

Index 42496224  
legId 61961236  
searchDate 42496224  
flightDate 42496224  
**startingAirport 5312196  
destinationAirport 5313484  
fareBasisCode 11147023  
travelDuration 10752574**  
elapsedDays 5312028  
isBasicEconomy 5312028  
isRefundable 5312028  
isNonStop 5312028  
baseFare 21248112  
totalFare 21248112  
seatsRemaining 5312028  
totalTravelDistance 21248112  
**segmentsDepartureTimeEpochSeconds 55265918  
segmentsDepartureTimeRaw 67923918  
segmentsArrivalTimeEpochSeconds 57071505  
segmentsArrivalTimeRaw 70785925  
segmentsArrivalAirportCode 10755425  
segmentsDepartureAirportCode 10684758  
segmentsAirlineName 5319331  
segmentsAirlineCode 5318025  
segmentsEquipmentDescription 10949169  
segmentsDurationInSeconds 12869511  
segmentsDistance 10750114  
segmentsCabinCode 5315719**  
dtype: int64

For example, each value of the **startingAirport** column now uses about 1 byte on average (5312196 bytes / 5312028 rows).

And let’s check the final footprint of our optimized dataframe:

optimized\_df.info(memory\_usage='deep')

The final memory footprint of our dataframe is 605.9 MB, as compared to its initial size of 7.1 GB!

<class 'pandas.core.frame.DataFrame'>  
Int64Index: 5312028 entries, 0 to 82137143  
Data columns (total 27 columns):  
 # Column Dtype   
--- ------ -----   
 0 legId category   
 1 searchDate datetime64[ns]  
 2 flightDate datetime64[ns]  
 3 startingAirport category   
 4 destinationAirport category   
 5 fareBasisCode category   
 6 travelDuration category   
 7 elapsedDays uint8   
 8 isBasicEconomy bool   
 9 isRefundable bool   
 10 isNonStop bool   
 11 baseFare float32   
 12 totalFare float32   
 13 seatsRemaining uint8   
 14 totalTravelDistance float32   
 15 segmentsDepartureTimeEpochSeconds category   
 16 segmentsDepartureTimeRaw category   
 17 segmentsArrivalTimeEpochSeconds category   
 18 segmentsArrivalTimeRaw category   
 19 segmentsArrivalAirportCode category   
 20 segmentsDepartureAirportCode category   
 21 segmentsAirlineName category   
 22 segmentsAirlineCode category   
 23 segmentsEquipmentDescription category   
 24 segmentsDurationInSeconds category   
 25 segmentsDistance category   
 26 segmentsCabinCode category   
dtypes: bool(3), category(17), datetime64[ns](2), float32(3), uint8(2)  
**memory usage: 605.9 MB**

With your dataframe’s memory footprint greatly reduced, it should now be easier and faster to perform data analytics on your dataframe.

Converting object columns to category type often leads to huge savings in memory.

**Persisting the Optimized DataFrame**

If you have taken all the trouble to filter your dataframe and optimize its memory footprint, the last thing you want to do is to save the dataframe back onto storage using the CSV format! If you do so, the next time you load back the CSV onto a dataframe, all the optimizations that you have painstakingly performed will go down the drain. A smarter approach is to save it using Pickle, through the to\_pickle() function:

optimized\_df.to\_pickle('optimized\_df.pkl')

Pickle serializes your dataframe to storage using binary mode, unlike saving as CSV, which saves the data as plain text.

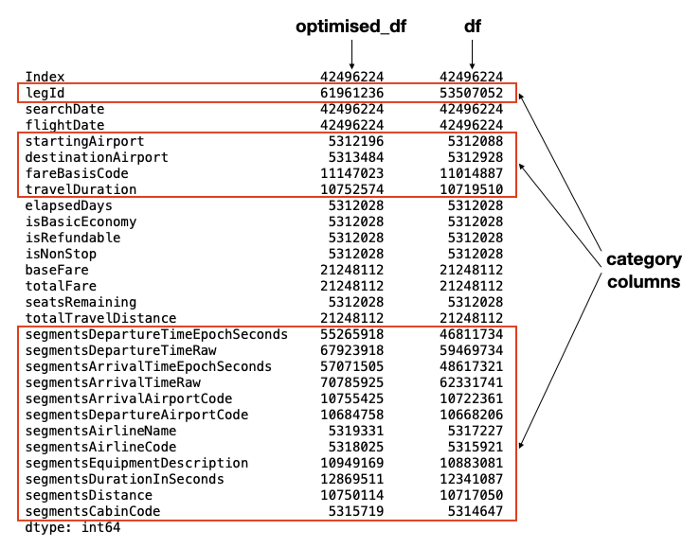
To verify that all the optimizations were not wasted, load back the pickle file and verify the data type of the newly loaded dataframe:

df = pd.read\_pickle('optimized\_df.pkl')  
df.info(memory\_usage='deep')

You will realize that the resultant memory footprint is now even smaller than earlier (which was 605.9 MB):

<class 'pandas.core.frame.DataFrame'>  
Int64Index: 5312028 entries, 0 to 82137143  
Data columns (total 27 columns):  
 # Column Dtype   
--- ------ -----   
 0 legId category   
 1 searchDate datetime64[ns]  
 2 flightDate datetime64[ns]  
 3 startingAirport category   
 4 destinationAirport category   
 5 fareBasisCode category   
 6 travelDuration category   
 7 elapsedDays uint8   
 8 isBasicEconomy bool   
 9 isRefundable bool   
 10 isNonStop bool   
 11 baseFare float32   
 12 totalFare float32   
 13 seatsRemaining uint8   
 14 totalTravelDistance float32   
 15 segmentsDepartureTimeEpochSeconds category   
 16 segmentsDepartureTimeRaw category   
 17 segmentsArrivalTimeEpochSeconds category   
 18 segmentsArrivalTimeRaw category   
 19 segmentsArrivalAirportCode category   
 20 segmentsDepartureAirportCode category   
 21 segmentsAirlineName category   
 22 segmentsAirlineCode category   
 23 segmentsEquipmentDescription category   
 24 segmentsDurationInSeconds category   
 25 segmentsDistance category   
 26 segmentsCabinCode category   
dtypes: bool(3), category(17), datetime64[ns](2), float32(3), uint8(2)  
**memory usage: 564.7 MB**

If you compare the memory usage of each column (see below), notice that all the category columns now uses much lesser memory (after reloading from the pickle file) than before. This is most likely due to the way the category columns are loaded and organized in memory when it is loaded directly from the pickle file.



**[Join Medium with my referral link - Wei-Meng Lee](https://weimenglee.medium.com/membership" \t "_blank)**

**[Read every story from Wei-Meng Lee (and thousands of other writers on Medium). Your membership fee directly supports…](https://weimenglee.medium.com/membership" \t "_blank)**

[weimenglee.medium.com](https://weimenglee.medium.com/membership" \t "_blank)

**Summary**

In this article, I went through a few techniques that you can deploy when dealing with extremely large CSV data files. Specifically, you learned that instead of loading the entire CSV file into a dataframe, you can use chunking to load the dataframe by parts. This gives you the flexibility to selectively load rows and columns that you need for your analytics purposes. In addition, there are several techniques that you can use to reduce the memory footprint of your dataframe. In general, use the following steps to reduce the memory usage:

* Convert date-related columns to the datetime64 type
* Downcast all numeric types to types that are wide enough to hold the values in these columns
* Convert all object columns to category type if feasible

Using these three simple techniques will yield significant savings in the memory usage of your dataframe. Try it out on your own dataset and you will be amazed!