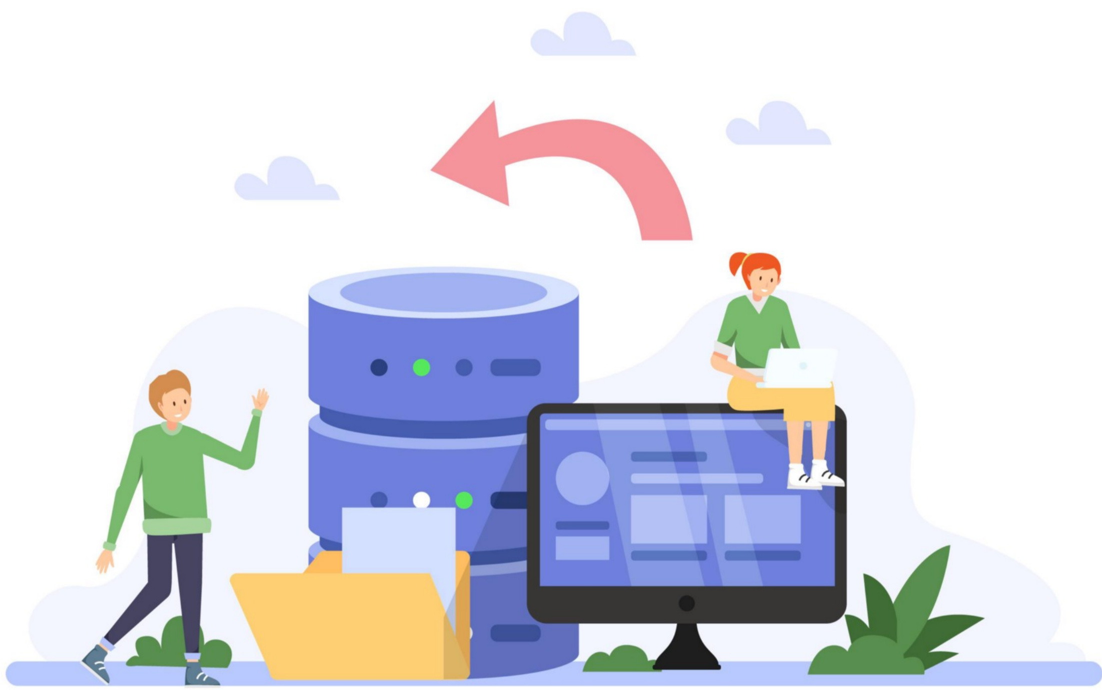
Loading large datasets in Pandas

Effectively using Chunking and SQL for reading large datasets in pandas

Parul Pandey



Business vector created by freepik

The pandas' library is a vital member of the Data Science ecosystem. However, the fact that it is unable to analyze datasets larger than memory makes it a little tricky for big data. Consider a situation when we want to analyze a large dataset by using only pandas. What kind of problems can we run into? For instance, let's take a file comprising 3GB of data summarising yellow taxi trip data for March in 2016. To perform any sort of analysis, we will have to import it into memory. We readily use the pandas' `read_csv()` function to perform the reading operation as follows:

```
import pandas as pd
df = pd.read_csv('yellow_tripdata_2016-03.csv')
```

When I ran the cell/file, my system threw the following **Memory Error.** (The memory error would depend upon the capacity of the system that you are using).

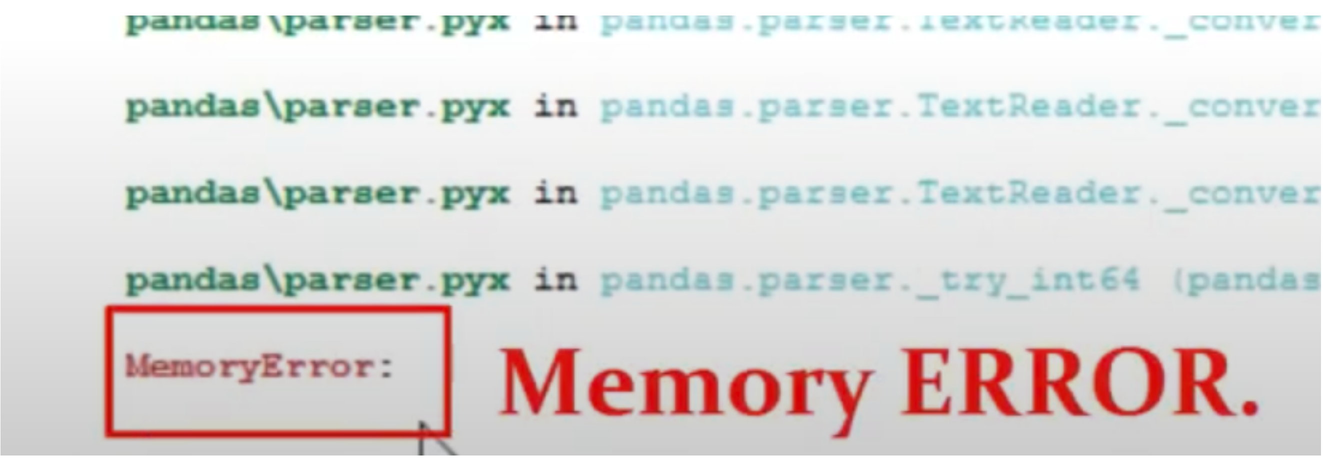


Image by Author

Any Alternatives?

Before criticizing pandas, it is important to understand that pandas may not always be the right tool for every task. Pandas lack multiprocessing support, and other libraries are better at handling big data. One such alternative is Dask, which gives a pandas-like API foto work with larger than memory datasets. Even the pandas' documentation explicitly mentions that for big data:

it's worth considering not using pandas. Pandas isn't the right tool for all situations.

In this article, however, we shall look at a method called chunking, by which you can load out of memory datasets in pandas. This method can sometimes offer a healthy way out to manage the out-of-memory problem in pandas but may not work all the time, which we shall see later in the chapter. Essentially we will look at two ways to import large datasets in python:

- Using `pd.read_csv()` with chunksize
- Using SQL and pandas

💡 Chunking: subdividing datasets into smaller parts

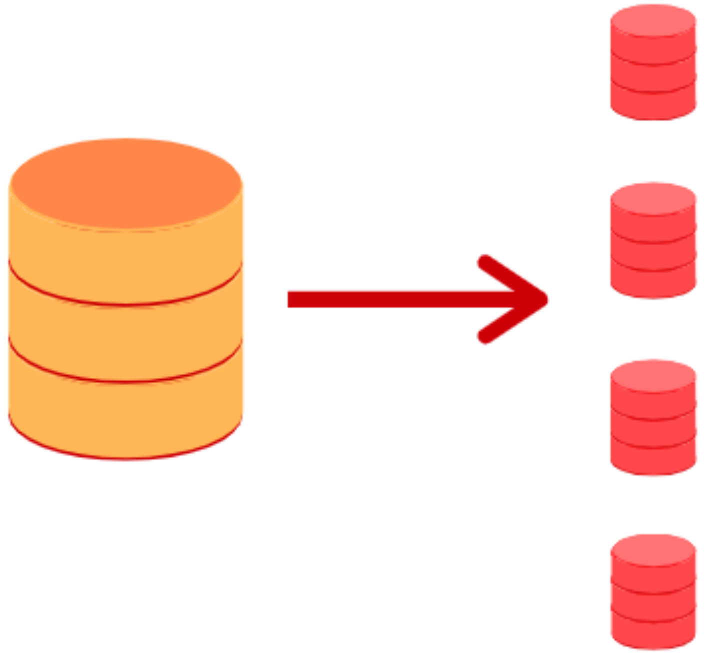


Image by Author

Before working with an example, let's try and understand what we mean by the work chunking. According to [Wikipedia](),

*[**Chunking**]() refers to strategies for improving performance by using special knowledge of a situation to aggregate related memory-allocation requests.*

In order words, instead of reading all the data at once in the memory, we can divide into smaller parts or chunks. In the case of CSV files, this would mean only loading a few lines into the memory at a given point in time.

Pandas' `read_csv()` function comes with a **chunk size [parameter]()** that controls the size of the chunk. Let's see it in action. We'll be working with the exact dataset that we used earlier in the article, but instead of loading it all in a single go, we'll divide it into parts and load it.

✴️ Using pd.read_csv() with chunksize

To enable chunking, we will declare the size of the chunk in the beginning. Then using `read_csv()` with the chunksize parameter, returns an object we can iterate over.

```
chunk_size=50000
batch_no=1for chunk in pd.read_csv('yellow_tripda
    chunk.to_csv('chunk'+str(batch_no)+'.csv',inc
    batch_no+=1
```

We choose a chunk size of 50,000, which means at a time, only 50,000 rows of data will be imported. Here is a video of how the main CSV file splits into multiple files.

Video by Author

Importing a single chunk file into pandas dataframe:

We now have multiple chunks, and each chunk can easily be loaded as a pandas dataframe.

```
df1 = pd.read_csv('chunk1.csv')
df1.head()
```

It works like a charm!. No more memory error. Let’s quickly look at the memory usage by this chunk:

```
df1.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50000 entries, 0 to 49999
Data columns (total 19 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   VendorID               50000 non-null  int64
 1   tpep_pickup_datetime   50000 non-null  object
 2   tpep_dropoff_datetime  50000 non-null  object
 3   passenger_count        50000 non-null  int64
 4   trip_distance          50000 non-null  float64
 5   pickup_longitude       50000 non-null  float64
 6   pickup_latitude        50000 non-null  float64
 7   RatecodeID             50000 non-null  int64
 8   store_and_fwd_flag     50000 non-null  object
 9   dropoff_longitude      50000 non-null  float64
 10  dropoff_latitude       50000 non-null  float64
 11  payment_type           50000 non-null  int64
 12  fare_amount            50000 non-null  float64
 13  extra                  50000 non-null  float64
 14  mta_tax                50000 non-null  float64
 15  tip_amount             50000 non-null  float64
 16  tolls_amount           50000 non-null  float64
 17  improvement_surcharge  50000 non-null  float64
 18  total_amount           50000 non-null  float64
dtypes: float64(12), int64(4), object(3)
memory usage: 7.2+ MB
```

Image by Author

🔴 A word of caution

Chunking creates various subsets of the data. As a result, it works well when the operation you're performing requires zero or minimal coordination between chunks. This is an important consideration. Another drawback of using chunking is that some operations like `groupby` are much harder to do chunks. In such cases, it is better to use alternative libraries.

✴️Using SQL and pandas to read

large data files[1]

(see References)

Image by Author

Another way around is to build an SQLite database from the chunks and then extract the desired data using SQL queries. SQLite is a relational database management system based on the SQL language but optimized for small environments. It can be integrated with Python using a Python module called sqlite3. If you want to know more about using Sqlite with python, you can refer to an article that I wrote on this very subject:

Programming with Databases in Python using SQLite

If you are aspiring to be a data scientist, you will be working with a lot of Data. Much of the data resides in…

SQLAlchemy is the Python SQL toolkit and Object Relational Mapper that gives application developers the full power and flexibility of SQL. It is used to build an engine for creating a database from the original data, which is a large CSV file, in our case.

For this article, we shall follow the following steps:

Import the necessary libraries

```
import sqlite3
from sqlalchemy import create_engine
```

Create a connector to a database

We shall name the database to be created as `csv_database`.

```
csv_database = create_engine('sqlite:///csv_datal
```

Creating a database from the CSV file with Chunking

This process is similar to what we have seen earlier in this article. The loop reads the datasets in bunches specified by the chunksize.

```
chunk_size=50000
batch_no=1for chunk in pd.read_csv('yellow_tripda
    chunk.to_sql('chunk_sql',csv_database, if_ex:
    batch_no+=1
```

```
    print('index: {}'.format(batch_no))
```

Note that we use the function. `chunk.to_sql instead of chunk.to_csv` since we are writing the data to the database i.e `csv_database`. Also, `chunk_sql` is an arbitrary name given to the chunk.

Constructing a pandas dataframe by querying SQL database

The database has been created. We can now easily query it to extract only those columns that we require; for instance, we can extract only those rows where the passenger count is less than `5` and the trip distance is greater than `10`. `pandas.read_sql_query`reads SQL query into a DataFrame.

We now have a dataframe that fits well into our memory and can be used for further analysis.

Conclusion

Pandas is a handy and versatile library when it comes to data analysis. However, it suffers from several bottlenecks when it comes to working with big data. In this article, we saw how chunking, coupled with SQL, could offer some solace for analyzing datasets larger than the system's memory. However, this alternative is not a 'one size fits all' solution, and getting to work with libraries created for handling big data would be a better option.

References

1. How to Read Very Big Files With SQL and Pandas in Python by Dr. Vytautas Bielinskas

2. Scaling to large datasets