**4 Techniques for Scaling Pandas to Large Datasets**

**Tips for increasing the efficiency**



Photo by [S. Tsuchiya](https://unsplash.com/@s_tsuchiya?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText) on [Unsplash](https://unsplash.com/s/photos/large?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText)

Pandas is one of the most frequently used tools in the data science ecosystem. It makes it quite easy to manipulate and analyze tabular data by providing numerous functions with an easy-to-understand syntax.

While Pandas is on top of the competition in data analysis and manipulation, its…

**What makes Pandas slower**

The main reason is that Pandas does in-memory analytics so if the dataset is larger than memory, it becomes very difficult, or impossible, to use Pandas.

Moreover, even if there is enough memory for the dataset, it can be a challenge to use Pandas as some operations make intermediate copies. In order to have a smooth experience with Pandas, the dataset should be relatively smaller than the memory.

Since we are talking about performance, it is inevitable to mention that Pandas uses a single CPU core to execute the operations. On very large datasets, this makes Pandas slower than the tools that offer distributed computing.

In this article, we will go over 4 techniques that help make Pandas more efficient when working with very large datasets.

It is important to note that there are other tools and libraries that are a better choice for very large datasets such as Dask, Vaex, Modin, and Datatable.

We will not cover these tools in this article. Instead, our focus is how to make Pandas more applicable and perform better.

**1. Do you really need all of the dataset?**

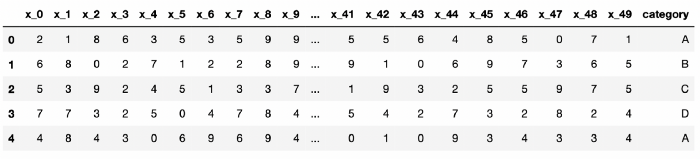
Data is the most valuable asset in data science so we tend to collect as much data as possible. However, we do not need every piece of data for all tasks.

The dataset might contain redundant columns or we might simply need only a few columns for a particular task.

Instead of reading the entire dataset and then filtering the required columns, a better approach is to only read the columns we need.

Let’s first generate a dataset. The following code snippet creates a Pandas DataFrame with 51 columns and 10 million rows. 50 columns are filled with random integers between 0 and 10 and the other column contains string values of A, B, C, and D. It took my computer about 1 minute to generate this dataset.

import pandas as pd  
import numpy as np  
  
df = pd.DataFrame(np.random.randint(0, 100, size=(10000000, 50)))  
df = df.rename(columns={i:f"x\_{i}" for i in range(50)})  
df["category"] = ["A", "B", "C", "D"] \* 2500000



The first 5 rows of df (image by author)

The memory usage of this DataFrame is approximately 4 GB.

np.round(df.memory\_usage().sum() / 10\*\*9, 2)  
  
# output  
4.08

We might have much larger datasets than this one in real-life but it is enough to demonstrate our case.

I would like to do a simple filtering operation and measure how long it takes.

%time df[df["category"]=="A"]  
  
# output  
CPU times: user 519 ms, sys: 911 ms, total: 1.43 s  
Wall time: 2.43 s

It took about half a second. We can also measure how long it takes to sort the rows by the values in a column or columns.

%time df.sort\_values(by=["x\_0", "x\_1"])  
  
# output  
CPU times: user 2.84 s, sys: 1.19 s, total: 4.03 s  
Wall time: 4.52 s

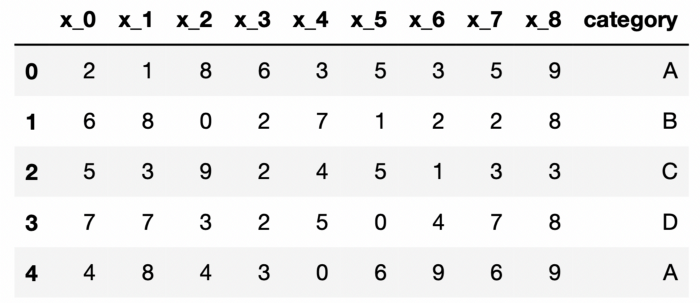
It took 2.84 seconds to sort the rows by the x\_0 and x\_1 columns.

Let’s save this DataFrame as a CSV file.

df.to\_csv("very\_large\_dataset.csv", index=False)

Consider a case where we only need the first 10 columns in this dataset. We can select a list of columns to read using the usecols parameter of the read\_csv function.

cols = ["category", "x\_0", "x\_1", "x\_2", "x\_3", "x\_4", "x\_5", "x\_6", "x\_7", "x\_8"]  
  
df = pd.read\_csv("very\_large\_dataset.csv", usecols=cols)  
  
df.head()



The first 5 rows of df (image by author)

The memory usage went down to 0.8 GB from 4 GB.

np.round(df.memory\_usage().sum() / 10\*\*9, 2)  
  
# output  
0.8

Let’s do the same filtering operation as we did with the entire dataset.

%time df[df["category"]=="A"]  
  
# output  
CPU times: user 389 ms, sys: 147 ms, total: 535 ms  
Wall time: 629 ms

It took 389 ms, which means 25% faster compared to working with the entire dataset. The speed gain is even more on the sorting operation.

%time df.sort\_values(by=["x\_0", "x\_1"])  
  
# output  
CPU times: user 919 ms, sys: 298 ms, total: 1.22 s  
Wall time: 1.33 s

It took 919 ms, which means 67% faster compared to 2.84 seconds with the entire dataset.

Saving only 1 second may not seem like a significant improvement but it adds up when you do lots of operations for a typical data cleaning or data analysis task. The more important point here is the memory usage went down to 0.8 GB from 4.8 GB.

**2. More efficient data type for categorical data**

Each column in DataFrame has a data type. Choosing the data types efficiently might reduce memory consumption and thus helps scaling Pandas to larger datasets.

If we have a categorical feature with low-cardinality, using the category data type instead of object or string saves a substantial amount of memory.

Low-cardinality means having very few distinct values compared to the total number of values. For instance, the category column in our DataFrame has only 4 distinct values compared to a total of 10 million.

df["category"].unique()  
# output  
array(['A', 'B', 'C', 'D'], dtype=object)  
  
len(df["category"])  
# output  
10000000

It is currently stored with object data type. Let’s check its memory usage.

df["category"].dtypes  
# output  
dtype('O')  
  
np.round(df["category"].memory\_usage() / 10\*\*6, 2)  
# output  
80.0

The memory usage of the category columns is 80 MB. Let’s change its data type to category and check the memory usage again.

df["category"] = df["category"].astype("category")  
  
np.round(df["category"].memory\_usage() / 10\*\*6, 2)  
# output  
10.0

It went down to 10 MB from 80 MB, which means 87.5% reduction in memory usage.

**3. Downcast numeric columns**

The data types for numerical columns might be causing unnecessary usage of memory. For instance, the default integer data type in Pandas is “int64”, which can store numbers between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807. In most cases, we don’t need such a gigantic range for integer values.

We can downcast integer columns to int16 or int8 to reduce memory usage. A more practical approach is to use the to\_numeric function, which can do the proper downcast for us.

Let’s first check the memory consumption of an integer column.

df["x\_0"].dtypes  
# output  
dtype('int64')  
  
np.round(df["x\_0"].memory\_usage() / 10\*\*6, 2)  
# output  
80.0

It’s 80 MB. Let’s downcast the data type of this column and check the memory usage again.

df["x\_0"] = pd.to\_numeric(df["x\_0"], downcast="unsigned")  
  
df["x\_0"].dtypes  
# output  
dtype('uint8')  
  
np.round(df["x\_0"].memory\_usage() / 10\*\*6, 2)  
# output  
10.0

The new data type is unsigned integer 8, which results in a memory usage of 10 MB. This also means 87.5% reduction in memory usage.

We can do this on any numerical columns either integer or float. In the case of working with floats, we can set the value of the downcast parameter as “float”.

**4. Use special data structures for sparse data**

We can use sparse objects for efficiently storing sparse data. Consider we have numerical columns that contain mostly zeroes. The memory consumption can be greatly reduced by converting these columns to sparse data type.

It does not have to be “mostly zeroes”. It can be NaN or any other value. Sparse objects can be viewed as being “compressed” where any data matching a specific value (0, NaN, or any other value) is omitted. The compressed values are not actually stored in the array.

Let’s go over an example to demonstrate this case.

df\_new = df[["x\_6", "x\_7", "x\_8"]].replace(  
 {2: 0, 3: 0, 4: 0, 5: 0, 6: 0, 7: 0, 8: 0, 9: 0}  
)  
  
df\_new["x\_6"].value\_counts()  
# output  
0 8999426  
1 1000574  
Name: x\_6, dtype: int64

The DataFrame above (df\_new) contains 3 columns, which consist of mostly zeroes (approx. 90%). The other values are 1. Let’s check the data types and memory consumption of each column.

df\_new.dtypes  
# output  
x\_6 int64  
x\_7 int64  
x\_8 int64  
dtype: object  
  
np.round(df\_new.memory\_usage() / 10\*\*6, 2)  
# output  
Index 0.0  
x\_6 80.0  
x\_7 80.0  
x\_8 80.0

The data type is int64 and each column consumes 80 MB of memory. If we convert the data type to unsigned int8, each column will take up a memory of 10 MB.

df\_new = df\_new.astype("uint8")  
  
np.round(df\_new.memory\_usage() / 10\*\*6, 2)  
# output  
Index 0.0  
x\_6 10.0  
x\_7 10.0  
x\_8 10.0

Let’s use a sparse data type for further improvement on memory usage.

sdf = df\_new.astype(pd.SparseDtype("uint8", 0))  
  
np.round(sdf.memory\_usage() / 10\*\*6, 2)  
# output  
Index 0.00  
x\_6 5.00  
x\_7 4.99  
x\_8 5.00

It went down to 5 MB, which is a significant improvement as well.

**Conclusion**

We have covered 4 different techniques that make Pandas more eligible to work with large datasets. It is important to note that there are other alternatives but if you want to keep working with Pandas, you can use these techniques for increasing the efficiency.

*You can become a* [*Medium member*](https://sonery.medium.com/membership) *to unlock full access to my writing, plus the rest of Medium. If you already are, don’t forget to* [*subscribe*](https://sonery.medium.com/subscribe) *if you’d like to get an email whenever I publish a new article.*