

Chapter 1-1

Bonus Recipe - Working with large data files

Working with large data files

One of the advantages of using pandas is that it provides data structures for in-memory analysis, which results in a performance advantage when working with data. However, this advantage can also become a constraint when working with large datasets, as the amount of data you can load is limited by the available memory. When datasets exceed the available memory, it can lead to performance degradation, especially when pandas creates intermediate copies of the data for certain operations.

In real-world scenarios, there are general best practices to mitigate these limitations, including the following:

- **Sampling or loading a small number of rows for your exploratory data analysis (EDA):** Before applying your data analysis strategy to the entire dataset, it is a good practice to sample or load a small number of rows. This allows you to get a better understanding of your data, gain some intuition, and identify unnecessary columns that can be eliminated, thus reducing the overall dataset size.
- **Reduce the number of columns:** Keeping only the columns necessary for your analysis can significantly reduce the memory footprint of the dataset.
- **Chunking:** Utilizing the `chunksize` parameter (available in many of the reader functions in pandas) allows you to process the data in smaller, manageable chunks. This technique helps in handling large datasets by processing them piece by piece.
- **Use other libraries for large datasets:** There are alternative libraries specifically designed for working with large datasets that offer a similar API to pandas, such as **Dask**, **Polars**, and **Modin**.

In this recipe, you will learn about techniques within pandas to handle large datasets, such as *chunking*. Afterward, you will explore three new libraries: **Dask**, **Polars**, and **Modin**. These libraries serve as alternatives to pandas and can be particularly useful when dealing with large datasets.

Getting ready

In this recipe, you will install the Dask and Polars libraries.

To install using pip, you can use the following command:

```
>>> pip install "dask[complete]"
>>> pip install "modin[dask]"
>>> pip install polars
```

To install using Conda, you can use the following:

```
>>> conda install -c conda-forge dask -y
>>> conda install -c conda-forge polars -y
>>> conda install -c conda-forge modin-dask -y
```

In this recipe, you will be working with the New York Taxi dataset from <https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page>, and we will be working with Yellow Taxi Trip Records for 2023 (covering *January to May*). In the GitHub repository of the book, I have provided the `run_once()` function, which you will need to execute once. It will combine all *five* months of datasets (five Parquet files) and produce one large CSV dataset (around 1.72 GB).

Here is the script as a reference:

```
# Script to create one large data file

import pandas as pd
import glob

def run_once():
    # Directory path where Parquet files are located
    directory = '.././datasets/Ch1/yellow_tripdata_2023-*.parquet'

    # Get a list of all Parquet files in the directory
    parquet_files = glob.glob(directory)

    # Read all Parquet files into a single DataFrame
    dfs = []
    for file in parquet_files:
        df = pd.read_parquet(file)
        dfs.append(df)

    # Concatenate all DataFrames into a single DataFrame
    combined_df = pd.concat(dfs)

    combined_df.to_csv('.././datasets/Ch1/yellow_tripdata_2023.csv',
index=False)

run_once()
```

How to do it...

In this recipe, you will explore three different methods for handling large datasets for ingestion purposes. These methods include the following:

- Using the `chunksize` parameter, which is available in many of the reader functions in pandas
- Using the Dask library
- Using the Polars library

The `memory_profiler` library will be utilized for illustration purposes to show memory consumption. You can install the library using `pip`:

```
pip install -U memory_profiler
pip install -U setuptools
```

To use `memory_profiler` in Jupyter Notebook, you will need to run the following once:

```
%load_ext memory_profiler
```

Once loaded, you can use it inside any code cell. You just need to start the cell with `%memit` or `%%memit` in a Jupyter code cell. A typical output will show peak memory size and increment size:

- **Peak memory** represents the maximum memory usage during the execution of a specific line of code
- **Increment** represents the difference in memory usage between the current line and the previous line

Using chunksize

Several reader functions in pandas support chunking through the `chunksize` parameter. This approach is convenient when you have a large dataset that you need to ingest, but it may not be suitable if you need to perform complex logic on each chunk, which requires coordination between the chunks.

Some of the reader functions in pandas that support the `chunksize` parameter are as follows: `pandas.read_csv()`, `pandas.read_table()`, `pandas.read_sql()`, `pandas.read_sql_query()`, `pandas.read_sql_table()`, `pandas.read_json()`, `pandas.read_fwf()`, `pandas.read_sas()`, `pandas.read_spss()`, and `pandas.read_stata()`.

1. First, let's read this large file using the traditional approach with `read_csv` without chunking:

```
import pandas as pd
from pathlib import Path
file_path = Path('../..../datasets/Ch1/yellow_tripdata_2023.csv')

%%time
%%memit
df_pd = pd.read_csv(file_path, low_memory=False)
```

Given that we have two magic commands, `%%time` and `%%memit`, the output will display memory usage and CPU time, as shown here:

```
peak memory: 10085.03 MiB, increment: 9922.66 MiB
CPU times: user 21.9 s, sys: 2.64 s, total: 24.5 s
Wall time: 25 s
```

2. Using the same `read_csv` function, you utilize the `chunksize` parameter, which represents the number of lines to read per chunk. In this example, you will use `chunksize=10000`, which will create a chunk every 10,000 rows:

```
%%time
%%memit

df_pd = pd.read_csv(file_path, low_memory=False, chunksize=10000)
```

This will produce the following output:

```
peak memory: 3101.41 MiB, increment: 0.77 MiB
CPU times: user 25.5 ms, sys: 13.4 ms, total: 38.9 ms
Wall time: 516 ms
```

The reason the execution occurred so fast is that what has been returned is an **iterator** object of the `TextFileReader` type, as shown:

```
type(df_pd)
pandas.io.parsers.readers.TextFileReader
```

3. To retrieve the data in each chunk, you can use the `get_chunk()` method to retrieve one chunk at a time, or use a loop to retrieve all chunks, or simply use the `pandas.concat()` function:
 - **Option 1:** Using the `get_chunk()` method or Python `next()` function. This will retrieve one chunk at a time, at 10,000 records per chunk. Every time you run `get_chunk()` or `next()`, you will get the next chunk:

```
%%time
%%memit

df_pd = pd.read_csv(
    file_path, low_memory=False, chunksize=10000)
df_pd.get_chunk()
>>
peak memory: 6823.64 MiB, increment: 9.14 MiB
CPU times: user 72.3 ms, sys: 40.8 ms, total: 113 ms
Wall time: 581 ms
# this is equivalent to
df_pd = pd.read_csv(
    file_path, low_memory=False, chunksize=10000)
next(df_pd)
```

- **Option 2:** Looping through the chunks. This is useful if you want to perform simple operations on each chunk before combining each chunk:

```
%%time
%%memit

df_pd = pd.read_csv(
    file_path, low_memory=False, chunksize=10000)
final_result = pd.DataFrame()
for chunk in df_pd:
    final_result = pd.concat([final_result, chunk])
```

- **Option 3:** Using `pd.concat()` to retrieve all the chunks at once in one operation. This may not be as useful in terms of overall performance:

```
%%time
%%memit

df_pd = pd.read_csv(
    file_path, low_memory=False, chunksize=10000)
final_result = pd.concat(df_pd)
>>

peak memory: 9145.42 MiB, increment: 697.86 MiB
CPU times: user 14.9 s, sys: 2.81 s, total: 17.7 s
Wall time: 18.8 s
```

The memory and CPU time measurements are added for illustration purposes. As you can observe, looping through the chunks and appending each chunk can be a time-consuming process.

Next, you will learn how to use Polars, which provides a very similar API to that of pandas, making the transition to learn Polars a simpler task.

Using Polars

Similar to pandas, the Polars library is designed to be used on a single machine but offers higher performance than pandas when working with large datasets. Unlike pandas, which is single-threaded and cannot leverage multiple cores on a single machine, Polars can utilize all available cores on a single machine for efficient parallel processing. In terms of memory usage, pandas provides in-memory data structures, hence its popularity for in-memory analytics. This also means that when you load your CSV file, the entire dataset is loaded into memory; hence, working with datasets beyond your memory's capacity can be problematic. Polars, on the other hand, requires less memory than pandas for similar operations.

Polars is written in Rust, a programming language that offers similar performance to C and C++ and is becoming a very popular programming language in the land of machine learning operations (MLOps) due to its performance advantage compared to Python.

In this recipe, you will explore the basics of Polars, primarily reading a large CSV file using the `read_csv()` reader function. The Polars `DataFrame` API is designed to be familiar to pandas users while offering significant performance improvements through parallel execution and memory efficiency:

1. Start by importing the Polars library:

```
import polars as pl
from pathlib import Path

file_path = Path('.././datasets/Ch1/yellow_tripdata_2023.csv')
```

2. You can now read the CSV file using the `read_csv` function, similar to how you have done it using pandas. Notice the use of the `%%time` and `%%memit` Jupyter magic commands for illustration purposes:

```
%%time
%%memit

df_pl = pl.read_csv(file_path)
>>
peak memory: 8633.58 MiB, increment: 2505.14 MiB
CPU times: user 4.85 s, sys: 3.28 s, total: 8.13 s
Wall time: 2.81 s
```

3. You can use the `.head()` method to print out the first five records of the Polars `DataFrame`, similar to pandas:

```
df_pl.head()
```

4. To get the total number of rows and columns of the Polars `DataFrame`, you can use the `.shape` property:

```
df_pl.shape
>>
(16186386, 20)
```

5. If you decided to use Polars for processing large datasets but later decided to output the results back as a pandas `DataFrame`, you can do so using the `.to_pandas()` method:

```
df_pd = df_pl.to_pandas()
```

Just from these simple code runs, it becomes clear how fast Polars is. This can be even more obvious from the memory and CPU metrics when comparing `read_csv` in the Polars library to `read_csv` in pandas.

Using Dask

Another popular library for working with large datasets is Dask. It has a similar API to pandas but differs in its distributed computing capabilities, allowing it to scale beyond a single machine. Dask integrates well with other popular libraries such as pandas, scikit-learn, NumPy, and XGBoost.

Additionally, you can install Dask-ML, an add-on library that provides scalable machine learning alongside popular Python ML libraries such as scikit-learn, XGBoost, PyTorch, and TensorFlow/Keras.

In this recipe, you will explore the basics of Dask, primarily reading a large CSV file using the `read_csv()` reader function and understanding Dask's **lazy evaluation** approach:

1. Start by importing the `dataframe` module from the `dask` library:

```
import dask.dataframe as dd
from pathlib import Path
file_path = Path('../..../datasets/Ch1/yellow_tripdata_2023.csv')
```

2. You can now read the CSV file using the `read_csv` function, similar to how you have done it using pandas. Notice the use of the `%%time` and `%%memit` Jupyter magic commands for illustration purposes:

```
%%time
%%memit

df_dk = dd.read_csv(file_path)

>>
peak memory: 153.22 MiB, increment: 3.38 MiB
CPU times: user 44.9 ms, sys: 12.1 ms, total: 57 ms
Wall time: 389 ms
```

This is interesting output in terms of memory and CPU utilization. One would assume nothing was read. Let's run a few tests to understand what is happening.

3. You will explore the `df_dk` DataFrame using familiar techniques you would normally use in pandas, such as checking the size of the DataFrame:

```
df_dk.shape
>>
(Delayed('int-0ab72188-de09-4d02-a76e-4a2c400e918b'), 20)

df_dk.info()
<class 'dask.dataframe.core.DataFrame'>
Columns: 20 entries, VendorID to airport_fee
dtypes: float64(13), int64(4), string(3)
```


When using `read_csv`, Dask does not load the entire dataset yet. It only reads the data when you perform specific operations or functions. For example, using the `head()` method will retrieve only the first five records, and that's it. Thus saving memory and improving performance.

5. Print the first five records of the Dask DataFrame using the `head()` method:

```
df_dk.head()
```

You will notice that the first five records are printed out similar to how you would expect when using pandas.

6. To get the total number of records in the dataset, you can use the `compute()` method, which will force evaluation:

```
%%time
%%memit

print(df_dk.shape[0].compute())
>>

16186386
peak memory: 6346.53 MiB, increment: 1818.44 MiB
CPU times: user 19.3 s, sys: 5.29 s, total: 24.6 s
Wall time: 10.5 s
```

7. Lastly, if you were able to reduce the size of the DataFrame to be easier to load in pandas, you can convert from a Dask DataFrame to a pandas DataFrame using the `compute()` method:

```
df_pd = df_dk.compute()
type(df_pd)
>>
pandas.core.frame.DataFrame
```

The Dask library offers different APIs. You only explored the DataFrame API, which is similar to the pandas library. Dask offers many optimization capabilities and has a learning curve for those working with very large datasets and needing to scale their current workflows, whether it be from NumPy, scikit-learn, or pandas.

How it works...

Due to pandas' popularity, many libraries (such as Polars and Dask) were inspired by pandas' simplicity and API. This is because these libraries are designed to target pandas users to provide a solution to one of pandas' most significant limitations: the lack of ability to scale and work with very large datasets that cannot fit into memory.

There's more...

So far, you have been introduced to better options when working with large files than using pandas, especially if you have memory constraints and cannot fit all the data into the memory available. The pandas library is a single-core framework and does not offer parallel computing capabilities. Instead, there are specialized libraries and frameworks for parallel processing designed to work with big data. Such frameworks do not rely on loading everything into memory and instead can utilize multiple CPU cores, disk usage, or expand into multiple worker nodes (think multiple machines). Earlier, you explored **Dask**, which chunks your data, creates a computation graph, and parallelizes the smaller tasks (chunks) behind the scenes, thus speeding the overall processing time and reducing memory overhead.

Using Modin

These frameworks are great, but will require you to spend time learning the framework and may necessitate that you rewrite the original code to leverage these capabilities. So, there might be a learning curve initially. Luckily, this is where the **Modin** project comes into play. Modin can use acts as a wrapper or, more specifically, an abstraction on top of **Dask** or **Ray** (another open source distributed computing framework) that uses an API similar to that of pandas. Modin makes optimizing your pandas code much more straightforward without learning another framework; all it takes is a single line of code.

Start by importing the necessary libraries:

```
from pathlib import Path
from modin.config import Engine
Engine.put("dask") # Modin will use Dask
import modin.pandas as pd
file_path = Path('../datasets/Ch1/yellow_tripdata_2023.csv')
```

Notice a few things here. First, we specified the engine to be used. In this case, we opted to use **Dask**. **Modin** supports other engines, including Ray and MPI. Second, notice the `import modin.pandas as pd` statement; this single line is all that is needed to scale your existing pandas code. Keep in mind that the Modin project is in active development, which means that as pandas matures and adds additional features and functionalities, Modin may still be catching up.

Let's read our CSV file and compare the metrics in terms of CPU and memory utilization:

```
%%time
%%memit
df_md = pd.read_csv(file_path)
>>
peak memory: 348.02 MiB, increment: 168.59 MiB
CPU times: user 1.23 s, sys: 335 ms, total: 1.57 s
Wall time: 8.26 s
```

Your data is loaded fast, and you can run other pandas functions to further inspect your data, such as `df_md.head()`, `df_md.info()`, and `df_md.head()`, and you will notice how fast the results appear:

```
df_md.info()
>>
<class 'modin.pandas.dataframe.DataFrame'>
RangeIndex: 16186386 entries, 0 to 16186385
Data columns (total 20 columns):
#   Column                                Dtype
---  -
0   VendorID                             int64
1   tpep_pickup_datetime                  object
2   tpep_dropoff_datetime                 object
3   passenger_count                       float64
4   trip_distance                         float64
5   RatecodeID                           float64
6   store_and_fwd_flag                    object
7   PULocationID                         int64
8   DOLocationID                         int64
9   payment_type                          int64
10  fare_amount                           float64
11  extra                                 float64
12  mta_tax                               float64
13  tip_amount                            float64
14  tolls_amount                          float64
15  improvement_surcharge                  float64
16  total_amount                           float64
17  congestion_surcharge                   float64
18  Airport_fee                            float64
19  airport_fee                            float64
dtypes: float64(13), int64(4), object(3)
memory usage: 2.4+ GB
```

Using **Modin** allows you to utilize your existing pandas code, skillset, and experience with the library without having to learn a new framework. This includes access to the pandas I/O functions (reader and writer functions) and all the parameters you would expect from the pandas library.

Choosing the right framework for large data

When working with large datasets that exceed pandas' capacity, consider these frameworks based on your specific needs:

Framework	Best For	Key Advantages	Limitations
pandas with Chunking	Memory-constrained sequential operations	Familiar API No additional dependencies Simple implementation	Sequential processing only No parallelism Memory inefficient for complex operations
Polars	High-performance, single-machine queries	Extremely fast on a single machine Memory efficient Familiar DataFrame concepts	Limited to a single machine Not all pandas operations are supported Steeper learning curve than Modin
Dask	Distributed time series pipelines	Scales to clusters Full ecosystem (arrays, ML) Great for complex workflows	Complex setup for clusters Performance overhead for small datasets Lazy evaluation requires different thinking
Modin	Scaling pandas with minimal code changes	Drop-in replacement for pandas Scales existing pandas code Minimal learning curve	Not all pandas functions are optimized Still in active development Less performant than specialized solutions

Table 1.1: Comparison of large data processing frameworks in Python

See also

Other Python projects are dedicated to making working with large datasets more scalable and performant, and, in some cases, better options than pandas;

- Dask: <https://dask.org/>
- Modin: <https://modin.readthedocs.io/en/latest/>
- Polars: <https://pola.rs>
- Ray: <https://ray.io/>
- Vaex: <https://vaex.io/>