Berkeley
MSSE
MOLECULAR SCIENCE & SOFTWARE ENGINEERING

Lecture 02b:

Larger Than Memory Files

Markus Hohle

University California, Berkeley

Data Science for Scientific Computing

MSSE 277A, 3 Units

Outline

Standard Pandas

Line By Line

Chunks With Dask

PySpark

<u>Outline</u>

Standard Pandas

Line By Line

Chunks With Dask

PySpark

Larger Than Memory Files:

Picking the right tool matters!

	pandas	dask	polars
.xlsx	**83 sec**	na	8.2 sec **10.2 sec**
.csv	1.2 sec	0.016 sec **1.6 sec**	0.27 sec **0.20 sec**
.txt	**1.5 sec**	0.016 sec 0.92 sec	0.25 sec **0.23 sec**

check out Benchmark_Pandas_Dask_Polars.py

Larger Than Memory Files:

sometimes, a file is larger than the machines memory

- exploring the file **line wise**
- reading only that **part** of the data which is **actually needed**
- reading/writing in **chunks**

Let us work with a dataset that is actually *smaller* than the memory

idea: comparing different methods (including pandas) and measure **run time** and **memory usage**

decorator
to measure runtime

```
import time

def my_timer(my_function):
       def get_args(*args,**kwargs):
              t1      = time.monotonic()
              results = my_function(*args,**kwargs)
              t2      = time.monotonic()
              dt      = t2 - t1
              print("Total runtime: " + str(dt) + " seconds")
              return results, dt
       return get_args
```

Larger Than Memory Files:

idea: comparing different methods (including pandas) and measure **run time** and **memory usage**

decorator

to measure memory usage

```
import psutil
import functools
import tracemalloc

def memory_usage_decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
                tracemalloc.start()
                result          = func(*args, **kwargs)
                current, peak = tracemalloc.get_traced_memory()
                tracemalloc.stop()
                print(f"Memory usage for {func.__name__}:")
                print(f"  Current: {current / (1024 * 1024):.2f} MB")
                print(f"  Peak:    {peak / (1024 * 1024):.2f} MB")
                return result
        return wrapper
```

memory currently used (varies over time)

peak memory (maximum), most relevant for reading/writing process

Larger Than Memory Files: Standard Pandas

idea: comparing different methods (including pandas) and measure **run time** and **memory usage**

```python
@my_timer
@memory_usage_decorator
def StandardPandas(file_path: str = '../Datasets/Data_Set.txt'):
        return pd.read_csv(file_path)
```

Data_Set.txt ≈ 182 MB

Data_Set.txt

File Edit View

```
,Unnamed: 0,time,sample 0,sample 1,sample 2,sample 3,sample 4,sample 
22,sample 23,sample 24,sample 25,sample 26,sample 27,sample 28,sample
45,sample 46,sample 47,sample 48,sample 49,sample 50,sample 51,sample
68,sample 69,sample 70,sample 71,sample 72,sample 73,sample 74,sample
91,sample 92,sample 93,sample 94,sample 95,sample 96,sample 97,sample
0,0,0.0,23.81060063040245,25.312445111977155,24.445055636868688,-0.29
20787716331884,5.536482312004993,-0.0188483495386808,36.598205345472
242,12.59225136601085 6,6.014981095548715,19.890693672369228,7.4669029
46781951631,31.741152636474556,11.44780909965696,23.95132391632089,-1
4,12.382364613628308,31.2694836555739,32.133294016977544,26.295337299
7820664,25.858505737438215,9.754672654965368,28.95382074285343,6.4842
```

Larger Than Memory Files: Standard Pandas

idea: comparing different methods (including pandas) and measure **run time** and **memory usage**

```
@my_timer
@memory_usage_decorator
def StandardPandas(file_path: str = '../Datasets/Data_Set.txt'):
        return pd.read_csv(file_path)
```

Data_Set.txt ≈ 182 MB

StandardPandas()

```
Memory usage for StandardPandas:
 Current: 78.63 MB
 Peak:    157.26 MB
Total runtime: 1.2190000000118744 seconds
Out[12]:
(       Unnamed: 0.1  Unnamed: 0    time  ...  sample 97
 0                 0           0   0.000  ...  18.899622
 1                 1           1   0.001  ...  16.669575
 2                 2           2   0.002  ...  15.750601
 3                 3           3   0.003  ...  16.527525
 4                 4           4   0.004  ...  18.793216
 ...             ...         ...     ...  ...        ...
 99995         99995       99995  99.995  ...  19.910798
 99996         99996       99996  99.996  ...  22.044467
 99997         99997       99997  99.997  ...  25.177334
 99998         99998       99998  99.998  ...  28.300678
 99999         99999       99999  99.999  ...  30.434906

 [100000 rows x 103 columns],
 1.2190000000118744)
```

peak memory corresponds approximately to file size

well structured file

total runtime in seconds

Outline

Standard Pandas

Line By Line

Chunks With Dask

PySpark

Larger Than Memory Files: Line by Line

often, we don't know the structure of a data file.
if too large: **read first entries line by line** and print/store the output

```
@my_timer
@memory_usage_decorator
def TextFileLineByLine1(file_path: str = '../Datasets/Data_Set.txt',\
                                    k: int = 5):

        with open(file_path, 'r') as file:
                for i, line in enumerate(file):
                        if i>=k:
                                break
                        print(line)
```

reading the first five lines:
`TextFileLineByLine1()`

```
Memory usage for TextFileLineByLine1:
  Current: 0.15 MB
  Peak:    0.18 MB
Total runtime: 0.016000000000325963 seconds
```

minimal memory consumption

Larger Than Memory Files: Line by Line

reading the first five lines:
`TextFileLineByLine1()`

column names (keywords), **separated by comma**

data is structured

```
In [13]: TextFileLineByLine1()
,Unnamed: 0,time,sample 0,sample 1,sampl
12,sample 13,sample 14,sample 15,sample
26,sample 27,sample 28,sample 29,sample
40,sample 41,sample 42,sample 43,sample
54,sample 55,sample 56,sample 57,sample
68,sample 69,sample 70,sample 71,sample
82,sample 83,sample 84,sample 85,sample
96,sample 97,sample 98,sample 99

0,
0,0.0,23.81060063040245,25.312445111977
2740792067254,5.155855356386692,15.7023
8,36.59820534547204,16.936315893981185,1
437353255,18.05148342734458,6.855827508
42263420235605,-1.6129158279890152,21.9
1631,31.741152636474556,11.447809099656
446949008987,13.279841996028694,35.9203
,26.29533729976512,23.282536825428583,1
7308402,26.54782218211423,31.256188897
352444483523,26.95935326436113,14.04815
.509918246417644,12.424489553318868,16.
02,6.71431687590743,19.439262989661604,
6513,3.7759978478618814,9.6595054971236,
647936

1,
1,0.001,23.24191960143792,24.7557643097
6790093636127,4.718116935111938,15.3020
5.14989069912229,16.938697410251578,10.
59,18.63793433138781,6.903883334547536,
687214,-0.3577945473491559,21.905161840
3706376091183,13.340358676360577,23.076
```

Larger Than Memory Files:

often we need only parts of the data (say columns 1, 3, 5 etc) → keyword search to **find columns**!

```
@my_timer
@memory_usage_decorator
def TextFileLineByLine2(file_path: str = '../Datasets/Data_Set.txt', **keywords):

    locations = {v: None for v in keywords.values()}

    with open(file_path, 'r') as file:
        for line in file:
            ListSplit = line.split(',')

            for k in locations.keys():

                if locations[k] is None and k in ListSplit:
                    locations[k] = ListSplit.index(k)

            if all(locations.values()):
                break
```

storing which column belongs to the keyword

we saw earlier that column names are separated by comma (usually **not** hardcoded)

we are done, once all columns that we need are located

Larger Than Memory Files:

often we need only parts of the data (say columns 1, 3, 5 etc) → keyword search to **find columns**!

```
@my_timer
@memory_usage_decorator
def TextFileLineByLine2(file_path: str = '../Datasets/Data_Set.txt', **keywords):

        locations = {v: None for v in keywords.values()}
...

                if all(locations.values()):
                    break

        if not all(locations.values()):
            print('the following keywords were not found:\n')
            for k, v in zip(locations.keys(), locations.values()):
                if not v:
                    print(k + '\n')

    return locations
```

in case some keywords were not found:

Larger Than Memory Files: Line by Line

often we need only parts of the data (say columns 1, 3, 5 etc) → keyword search to **find columns**!

```
(Location, _) = TextFileLineByLine2(word1 = 'time', word2 = 'sample 0',\
                                     word3 = 'sample 4')
```

output form decorator not needed

```
Memory usage for TextFileLineByLine2:
  Current: 0.00 MB
  Peak:    0.03 MB
Total runtime: 0.0 seconds
```

minimal memory consumption

```
Location
{'time': 2, 'sample 0': 3, 'sample 4': 7}
```

data of time, sample 0 and sample 4 are located in columns 2, 3, and 7 respectively

Larger Than Memory Files:

often we need only parts of the data (say columns 1, 3, 5 etc) → keyword search to **find columns**!

```
(Location, _) = TextFileLineByLine2(word1 = 'time', word2 = 'sample 0',\
                                    word3 = 'XYZ', word4 = 'Sample 0')
```

upper case!

```
the following keywords were not found:

XYZ

Sample 0

Memory usage for TextFileLineByLine2:
  Current: 0.00 MB
  Peak:    0.04 MB
Total runtime: 7.047000000020489 seconds
```

Doesn't find keyword, if not consistent. **Requires fuzzy match** (see next lecture!)

minimal memory consumption, but needs to search the entire file, which takes more time

```
Location
{'time': 2, 'sample 0': 3, 'XYZ': None, 'Sample 0': None}
```

Larger Than Memory Files: Line by Line

often we need only parts of the data (say columns 1, 3, 5 etc) → keyword search to **find columns**!

```
(Location, _) = TextFileLineByLine3(word1 = 'time', word2 = 'sample 0',\
                                    word3 = 'XYZ', word4 = 'Sample 0')
```

Adding lines for reading specific columns once found in the data:

```
L = [[] for i in range(len(keywords))]

with open(file_path, 'r') as file:
        for l, line in enumerate(file):

                if l>1
                        ListSplit = line.split(',')

                        for i, (k, v) in enumerate(zip(locations.keys(),
                                                        locations.values())):
                            L[i].append(ListSplit[v])
```

preparing empty list

selecting the columns

saving content as list

Larger Than Memory Files:

With some small modifications:

storing the data once we located it in the file

turning extracted data into dataframe

```
A       = np.array(L, dtype = float).T
A_df    = pd.DataFrame(A, columns = locations.keys())
```

saving data as .csv

```
A_df.to_csv('ExtractedData.csv', index = False)
```

storing as .pkl file

```
with open('ExtractedData.pkl', 'wb') as f:
               pickle.dump(A_df, f)
```

.pkl files

- needs `import pickle`
- not human readable
- requires **half the space** of a .csv file
- reading is faster
- preferred format, if large date set has to be stored and **read during workflow**

```
with open(filename, 'rb') as file:
       loaded_data = pickle.load(file)
```

Larger Than Memory Files: Line by Line

With some small modifications:

storing the data once we located it in the file

turning extracted data into dataframe

```
A       = np.array(L, dtype = float).T
A_df    = pd.DataFrame(A, columns = locations.keys())
```

saving data as .csv

```
A_df.to_csv('ExtractedData.csv', index = False)
```

storing as .pkl file

```
with open('ExtractedData.pkl', 'wb') as f:
            pickle.dump(A_df, f)
```

Microsoft Excel Comma Separated Values File			
ExtractedData	19/10/2025 01:47	Microsoft Excel Com...	4.542 KB
PKL File			
ExtractedData.pkl	19/10/2025 01:47	PKL File	2.345 KB

Larger Than Memory Files:

With some small modifications:

storing the data once we located it in the file

```
A       = np.array(L, dtype = float).T
A_df    = pd.DataFrame(A, columns = locations.keys())
```

turning extracted data into dataframe

```
A_df.to_csv('ExtractedData.csv', index = False)
```

saving data as .csv

```
with open('ExtractedData.pkl', 'wb') as f:
        pickle.dump(A_df, f)
```

storing as .pkl file

```
print(A_df.head())
```

	time	sample 0	sample 4
0	0.001	23.241920	7.778409
1	0.002	22.002649	8.569085
2	0.003	20.490190	9.503710
3	0.004	19.192838	10.498987
4	0.005	18.527045	11.466261

```
Memory usage for TextFileLineByLine3:
  Current: 51.78 MB
  Peak:    91.39 MB
Total runtime: 25.98499999998603 seconds
```

Outline

Standard Pandas

Line By Line

Chunks With Dask

PySpark

Larger Than Memory Files:

dask uses **parallel processing**

idea: reading and writing data in **chunks** in order to save memory!

	pandas	dask
.xlsx	**83 sec**	na
.csv	1.2 sec	0.016 sec **1.6 sec**
.txt	**1.5 sec**	0.016 sec 0.92 sec

Larger Than Memory Files:

```python
@memory_usage_decorator
def Chunks(df):
    return [chunk.compute() for chunk in df.to_delayed()]

@my_timer
def CSVFileDask1(file_path: str = '../Datasets/Data_Set.csv',\
                 blocksize: int = 2):

    df = dd.read_csv(file_path, blocksize = int(blocksize * 1024 * 1024))

    with ProgressBar():
        chunks = Chunks(df)

    return pd.concat(chunks, ignore_index = True)
```

reading the file in chunks of defined size (=blocksize in MB)

compute turns dask objects into actual dataframes

Larger Than Memory Files:


```
@memory_usage_decorator
def Chunks(df):
    return [chunk.compute() for chunk in df.to_delayed()]

@my_timer
def CSVFileDask1(file_path: str = '../Datasets/Data_Set.csv',\
                 blocksize: int = 2):

    df = dd.read_csv(file_path, blocksize = int(blocksize * 1024 * 1024))

    with ProgressBar():
        chunks = Chunks(df)

    return pd.concat(chunks, ignore_index = True)
```

```
out = CSVFileDask1(blocksize = 200)
```

```
[########################################] | 100% Completed | 1.47 ss
Memory usage for Chunks:
  Current: 77.89 MB
  Peak:    512.74 MB
Total runtime: 1.5 seconds
```

fast, but memory intense!

Larger Than Memory Files:


```
@memory_usage_decorator
def Chunks(df):
    return [chunk.compute() for chunk in df.to_delayed()]

@my_timer
def CSVFileDask1(file_path: str = '../Datasets/Data_Set.csv',\
                 blocksize: int = 2):

    df = dd.read_csv(file_path, blocksize = int(blocksize * 1024 * 1024))

    with ProgressBar():
        chunks = Chunks(df)

    return pd.concat(chunks, ignore_index = True)
```

```
out = CSVFileDask1(blocksize = 2)
```

```
Memory usage for Chunks:
  Current: 80.02 MB
  Peak:    85.15 MB
Total runtime: 9.984000000025844 seconds
```

slower, but requires less memory

Larger Than Memory Files:


```
Memory usage for Chunks:
  Current: 80.02 MB
  Peak:    85.15 MB
Total runtime: 9.984000000025844 seconds
```

needs way more memory than blocksize!
reason: file is still stored in memory!

solution: **saving data in chunks as we read it**!

Larger Than Memory Files:

solution: **saving data in chunks as we read it**!

compute turns dask objects into actual dataframes

```
@memory_usage_decorator
def ChunksSave(df, outfilename: str):

    for i, chunk in enumerate(df.to_delayed()):
        computed_part = chunk.compute()

        if i == 0:
            computed_part.to_csv(outfilename, mode = 'w', index = False,\
                                 header = True)
        else:
            computed_part.to_csv(outfilename, mode = 'a', index = False,\
                                 header = False)
```

actual saving part

Larger Than Memory Files: Dask

```
@my_timer
def CSVFileDask2(file_path: str = '../Datasets/Data_Set.csv',\
                 blocksize: int = 2 ,outfilename: str = 'Out.csv')

    df = dd.read_csv(file_path, blocksize = int(blocksize * 1024 * 1024))

    with ProgressBar():
         ChunksSave(df, outfilename)

    print('Done!')
```

memory intense!

```
out = CSVFileDask2(blocksize = 200)
```

```
[########################################] | 100% Completed | 1.67 ss
Memory usage for ChunksSave:
  Current: 140.01 MB
  Peak:    629.59 MB
Done!
Total runtime: 80.73499999998603 seconds
```

Larger Than Memory Files:


```
@my_timer
def CSVFileDask2(file_path: str = '../Datasets/Data_Set.csv',\
                 blocksize: int = 2 ,outfilename: str = 'Out.csv')

    df = dd.read_csv(file_path, blocksize = int(blocksize * 1024 * 1024))

    with ProgressBar():
            ChunksSave(df, outfilename)

     print('Done!')
```

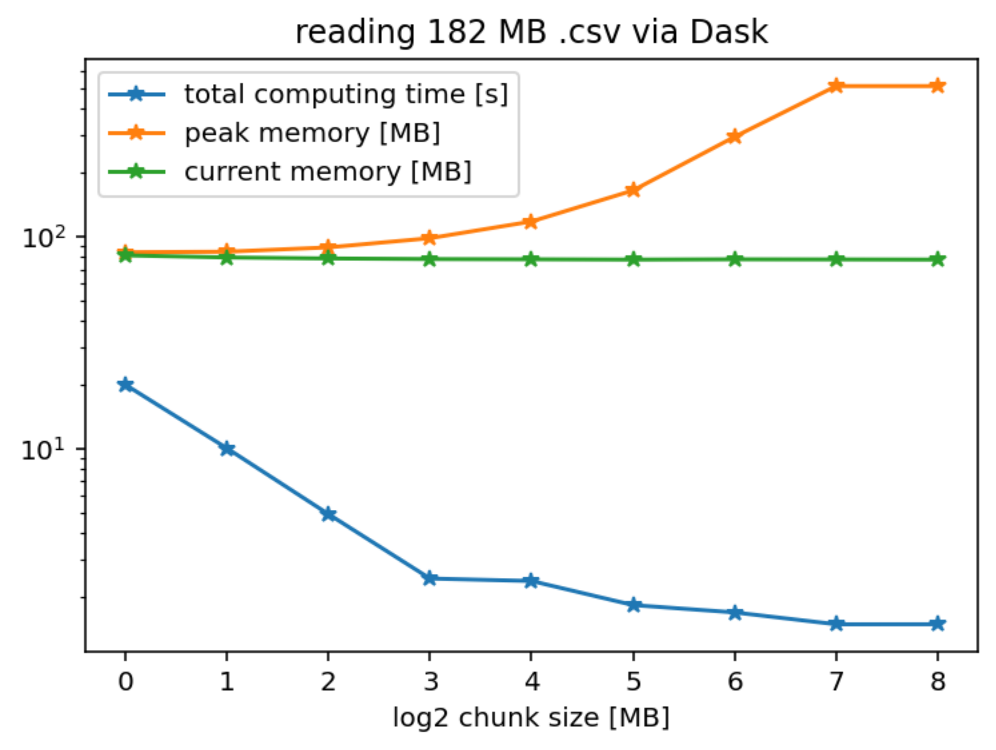
```
out = CSVFileDask2(blocksize = 2)
```

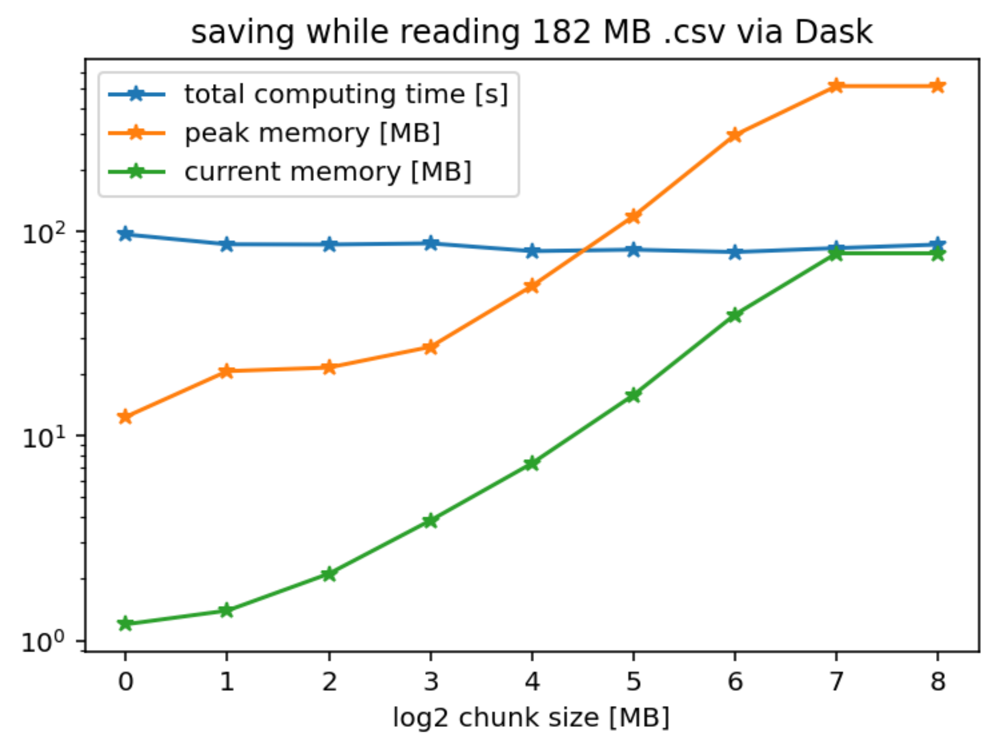
```
Memory usage for ChunksSave:
  Current: 1.56 MB
  Peak:    20.91 MB
Done!
Total runtime: 102.75 seconds
```

slower, but requires less memory

Larger Than Memory Files: Dask

General tradeoff between memory use and speed!





Outline

Standard Pandas

Line By Line

Chunks With Dask

PySpark

Larger Than Memory Files:

alternative to line by line: PySpark API **for large datasets across computers**

```
from pyspark.sql import SparkSession

@my_timer
@memory_usage_decorator
def CSVFilePySpark(file_path: str = '../Datasets/Data_Set.csv'):

    spark = SparkSession.builder.appName('LargeFileProcessing').getOrCreate()
    df    = spark.read.csv(file_path)

    print(df.head())
    print(df.schema)

    return df
```

```
out = CSVFilePySpark()
```

```
Memory usage for CSVFilePySpark:
  Current: 3.24 MB
  Peak:    3.40 MB
Total runtime: 9.469000000011874 seconds
```

```
Row(_c0=None, _c1='time', _c2='sample 0', _
c10='sample 8', _c11='sample 9', _c12='samp
6', _c19='sample 17', _c20='sample 18', _c
ample 25', _c28='sample 26', _c29='sample 
_c36='sample 34', _c37='sample 35', _c38='s
```

Larger Than Memory Files:

summary: see `LargerThanMemoryExample.ipynb`

reading file line by line using open
- **fast**, if only the first few lines
- **no external library** needed
- **low memory** usage
- for **quick overview** of data structure

alternative: PySpark

once we know the file structure:
- **only** reading **data needed** for further analysis (consistent keywords!)
- read and save data in **chunks**
- save extracted data as **.pkl** (further reading is more efficient)

These steps are usually called preprocessing and are the first step of **E**xploratory **D**ata **A**nalysis

Thank you very much for your attention!