

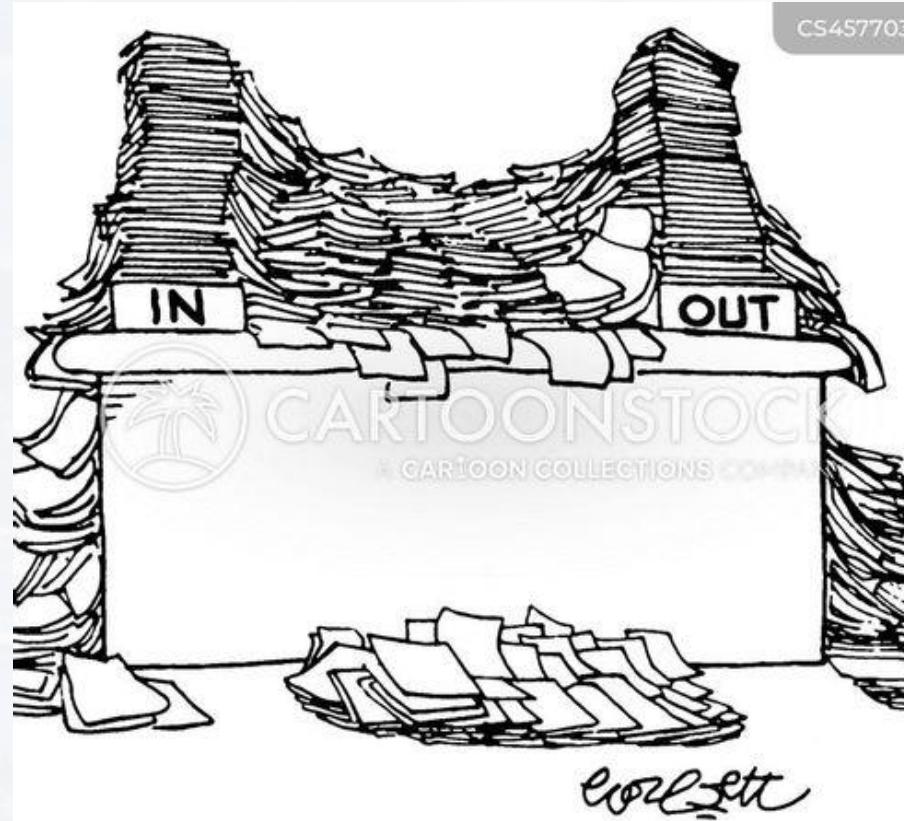
Lecture 02b:

Larger Than Memory Files



Markus Hohle
University California, Berkeley

Data Science for Scientific
Computing
MSSE 277A, 3 Units



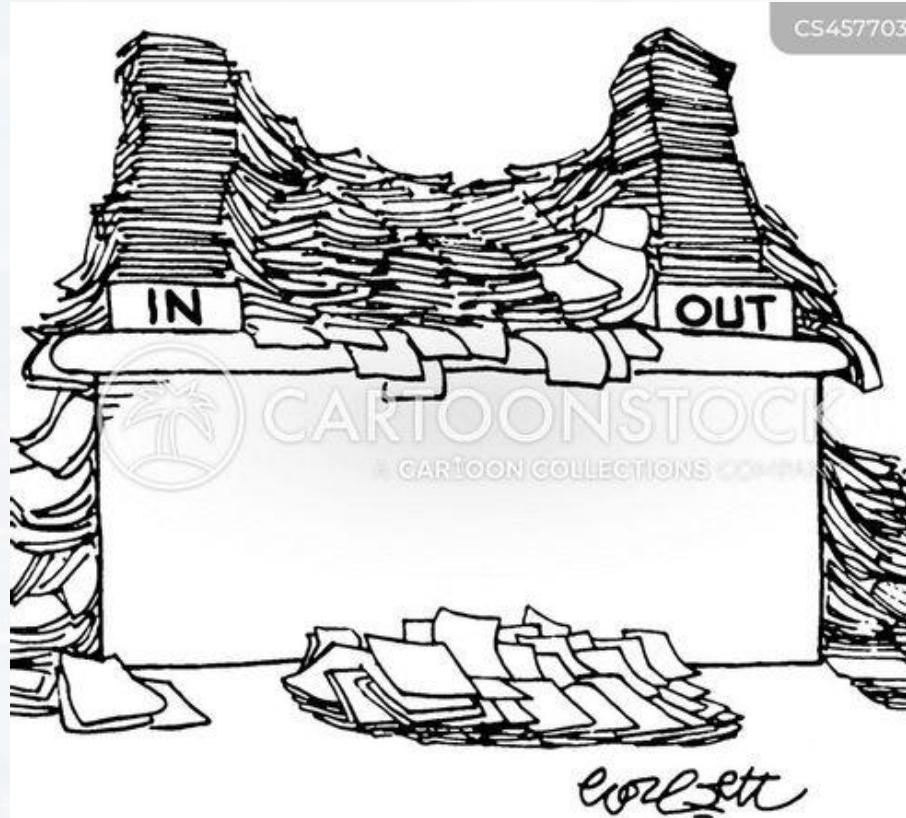
Outline

Standard Pandas

Line By Line

Chunks With Dask

PySpark



Outline

Standard Pandas

Line By Line

Chunks With Dask

PySpark



Picking the right tool matters!



.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



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**

```
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
```

decorator
to measure runtime



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

```
import psutil
import functools
import tracemalloc
```

decorator
to measure memory usage

```
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



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 $\approx 182 \text{ MB}$

The screenshot shows a window titled "Data_Set.txt" with a CSV file open. The window has a standard OS X style with a close button (X) and a plus sign (+) for new tabs. The menu bar includes "File", "Edit", and "View". The main area displays the contents of the CSV file, which consists of a header row and approximately 182 MB of data rows. The data rows are comma-separated values.

Header	Value
1	,
2	Unnamed: 0
3	time
4	sample 0
5	sample 1
6	sample 2
7	sample 3
8	sample 4
9	sample 5
10	sample 6
11	sample 7
12	sample 8
13	sample 9
14	sample 10
15	sample 11
16	sample 12
17	sample 13
18	sample 14
19	sample 15
20	sample 16
21	sample 17
22	sample 18
23	sample 19
24	sample 20
25	sample 21
26	sample 22
27	sample 23
28	sample 24
29	sample 25
30	sample 26
31	sample 27
32	sample 28
33	sample 29
34	sample 30
35	sample 31
36	sample 32
37	sample 33
38	sample 34
39	sample 35
40	sample 36
41	sample 37
42	sample 38
43	sample 39
44	sample 40
45	sample 41
46	sample 42
47	sample 43
48	sample 44
49	sample 45
50	sample 46
51	sample 47
52	sample 48
53	sample 49
54	sample 50
55	sample 51
56	sample 52
57	sample 53
58	sample 54
59	sample 55
60	sample 56
61	sample 57
62	sample 58
63	sample 59
64	sample 60
65	sample 61
66	sample 62
67	sample 63
68	sample 64
69	sample 65
70	sample 66
71	sample 67
72	sample 68
73	sample 69
74	sample 70
75	sample 71
76	sample 72
77	sample 73
78	sample 74
79	sample 75
80	sample 76
81	sample 77
82	sample 78
83	sample 79
84	sample 80
85	sample 81
86	sample 82
87	sample 83
88	sample 84
89	sample 85
90	sample 86
91	sample 87
92	sample 88
93	sample 89
94	sample 90
95	sample 91
96	sample 92
97	sample 93
98	sample 94
99	sample 95
100	sample 96
101	sample 97
102	sample 98
103	sample 99
104	sample 100
105	sample 101
106	sample 102
107	sample 103
108	sample 104
109	sample 105
110	sample 106
111	sample 107
112	sample 108
113	sample 109
114	sample 110
115	sample 111
116	sample 112
117	sample 113
118	sample 114
119	sample 115
120	sample 116
121	sample 117
122	sample 118
123	sample 119
124	sample 120
125	sample 121
126	sample 122
127	sample 123
128	sample 124
129	sample 125
130	sample 126
131	sample 127
132	sample 128
133	sample 129
134	sample 130
135	sample 131
136	sample 132
137	sample 133
138	sample 134
139	sample 135
140	sample 136
141	sample 137
142	sample 138
143	sample 139
144	sample 140
145	sample 141
146	sample 142
147	sample 143
148	sample 144
149	sample 145
150	sample 146
151	sample 147
152	sample 148
153	sample 149
154	sample 150
155	sample 151
156	sample 152
157	sample 153
158	sample 154
159	sample 155
160	sample 156
161	sample 157
162	sample 158
163	sample 159
164	sample 160
165	sample 161
166	sample 162
167	sample 163
168	sample 164
169	sample 165
170	sample 166
171	sample 167
172	sample 168
173	sample 169
174	sample 170
175	sample 171
176	sample 172
177	sample 173
178	sample 174
179	sample 175
180	sample 176
181	sample 177
182	sample 178
183	sample 179
184	sample 180
185	sample 181
186	sample 182
187	sample 183
188	sample 184
189	sample 185
190	sample 186
191	sample 187
192	sample 188
193	sample 189
194	sample 190
195	sample 191
196	sample 192
197	sample 193
198	sample 194
199	sample 195
200	sample 196
201	sample 197
202	sample 198
203	sample 199
204	sample 200
205	sample 201
206	sample 202
207	sample 203
208	sample 204
209	sample 205
210	sample 206
211	sample 207
212	sample 208
213	sample 209
214	sample 210
215	sample 211
216	sample 212
217	sample 213
218	sample 214
219	sample 215
220	sample 216
221	sample 217
222	sample 218
223	sample 219
224	sample 220
225	sample 221
226	sample 222
227	sample 223
228	sample 224
229	sample 225
230	sample 226
231	sample 227
232	sample 228
233	sample 229
234	sample 230
235	sample 231
236	sample 232
237	sample 233
238	sample 234
239	sample 235
240	sample 236
241	sample 237
242	sample 238
243	sample 239
244	sample 240
245	sample 241
246	sample 242
247	sample 243
248	sample 244
249	sample 245
250	sample 246
251	sample 247
252	sample 248
253	sample 249
254	sample 250
255	sample 251
256	sample 252
257	sample 253
258	sample 254
259	sample 255
260	sample 256
261	sample 257
262	sample 258
263	sample 259
264	sample 260
265	sample 261
266	sample 262
267	sample 263
268	sample 264
269	sample 265
270	sample 266
271	sample 267
272	sample 268
273	sample 269
274	sample 270
275	sample 271
276	sample 272
277	sample 273
278	sample 274
279	sample 275
280	sample 276
281	sample 277
282	sample 278
283	sample 279
284	sample 280
285	sample 281
286	sample 282
287	sample 283
288	sample 284
289	sample 285
290	sample 286
291	sample 287
292	sample 288
293	sample 289
294	sample 290
295	sample 291
296	sample 292
297	sample 293
298	sample 294
299	sample 295
300	sample 296
301	sample 297
302	sample 298
303	sample 299
304	sample 300
305	sample 301
306	sample 302
307	sample 303
308	sample 304
309	sample 305
310	sample 306
311	sample 307
312	sample 308
313	sample 309
314	sample 310
315	sample 311
316	sample 312
317	sample 313
318	sample 314
319	sample 315
320	sample 316
321	sample 317
322	sample 318
323	sample 319
324	sample 320
325	sample 321
326	sample 322
327	sample 323
328	sample 324
329	sample 325
330	sample 326
331	sample 327
332	sample 328
333	sample 329
334	sample 330
335	sample 331
336	sample 332
337	sample 333
338	sample 334
339	sample 335
340	sample 336
341	sample 337
342	sample 338
343	sample 339
344	sample 340
345	sample 341
346	sample 342
347	sample 343
348	sample 344
349	sample 345
350	sample 346
351	sample 347
352	sample 348
353	sample 349
354	sample 350
355	sample 351
356	sample 352
357	sample 353
358	sample 354
359	sample 355
360	sample 356
361	sample 357
362	sample 358
363	sample 359
364	sample 360
365	sample 361
366	sample 362
367	sample 363
368	sample 364
369	sample 365
370	sample 366
371	sample 367
372	sample 368
373	sample 369
374	sample 370
375	sample 371
376	sample 372
377	sample 373
378	sample 374
379	sample 375
380	sample 376
381	sample 377
382	sample 378
383	sample 379
384	sample 380
385	sample 381
386	sample 382
387	sample 383
388	sample 384
389	sample 385
390	sample 386
391	sample 387
392	sample 388
393	sample 389
394	sample 390
395	sample 391
396	sample 392
397	sample 393
398	sample 394
399	sample 395
400	sample 396
401	sample 397
402	sample 398
403	sample 399
404	sample 400
405	sample 401
406	sample 402
407	sample 403
408	sample 404
409	sample 405
410	sample 406
411	sample 407
412	sample 408
413	sample 409
414	sample 410
415	sample 411
416	sample 412
417	sample 413
418	sample 414
419	sample 415
420	sample 416
421	sample 417
422	sample 418
423	sample 419
424	sample 420
425	sample 421
426	sample 422
427	sample 423
428	sample 424
429	sample 425
430	sample 426
431	sample 427
432	sample 428
433	sample 429
434	sample 430
435	sample 431
436	sample 432
437	sample 433
438	sample 434
439	sample 435
440	sample 436
441	sample 437
442	sample 438
443	sample 439
444	sample 440
445	sample 441
446	sample 442
447	sample 443
448	sample 444
449	sample 445
450	sample 446
451	sample 447
452	sample 448
453	sample 449
454	sample 450
455	sample 451
456	sample 452
457	sample 453
458	sample 454
459	sample 455
460	sample 456
461	sample 457
462	sample 458
463	sample 459
464	sample 460
465	sample 461
466	sample 462
467	sample 463
468	sample 464
469	sample 465
470	sample 466
471	sample 467
472	sample 468
473	sample 469
474	sample 470
475	sample 471
476	sample 472
477	sample 473
478	sample 474
479	sample 475
480	sample 476
481	sample 477
482	sample 478
483	sample 479
484	sample 480
485	sample 481
486	sample 482
487	sample 483
488	sample 484
489	sample 485
490	sample



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



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: 0.0160000000325963 seconds
```

minimal memory consumption



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,sample  
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,  
0,0.0,23.81060063040245,25.3124451119771  
2740792067254,5.155855356386692,15.70238  
8,36.59820534547204,16.936315893981185,1  
437353255,18.05148342734458,6.8558275083  
42263420235605,-1.6129158279890152,21.96  
1631,31.741152636474556,11.4478090996569  
446949008987,13.279841996028694,35.92034  
,26.29533729976512,23.282536825428583,17  
7308402,26.54782218211423,31.25618889782  
352444483523,26.95935326436113,14.048154  
.509918246417644,12.424489553318868,16.5  
02,6.71431687590743,19.439262989661604,3  
6513,3.7759978478618814,9.6595054971236,  
647936
```

```
1,  
1,0.001,23.24191960143792,24.75576430971  
6790093636127,4.718116935111938,15.30209  
5.14989069912229,16.938697410251578,10.2  
59,18.63793433138781,6.903883334547536,1  
687214,-0.3577945473491559,21.9051618406  
3706376091183,13.340358676360577,23.0767
```



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()} ←
        storing which column
        belongs to the keyword
    with open(file_path, 'r') as file:
        for line in file:
            ListSplit = line.split(',') ←
                we saw earlier that column
                names are separated by
                comma (usually not
                hardcoded)
            for k in locations.keys():

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

            if all(locations.values()): ←
                we are done, once all
                columns that we need are
                located
                break
```



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:



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',\n                                     word3 = 'sample 4')
```

output form decorator
not needed

```
Memory usage for TextFileLineByLine2:\n  Current: 0.00 MB\n  Peak:    0.03 MB\nTotal runtime: 0.0 seconds
```

minimal memory consumption

```
Location\n{'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



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',\n                                     word3 = 'XYZ', word4 = 'Sample 0')
```

the following keywords were not found:

XYZ

Sample 0

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

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

upper case!

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



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))]
```

preparing empty list

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

selecting the
columns

```
if l>1
```

```
    ListSplit = line.split(',')  
for i, (k, v) in enumerate(zip(locations.keys(),
```

saving content as list

```
locations.values())):  
    L[i].append(ListSplit[v])
```



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

.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)
```



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

▼ 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



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())
```

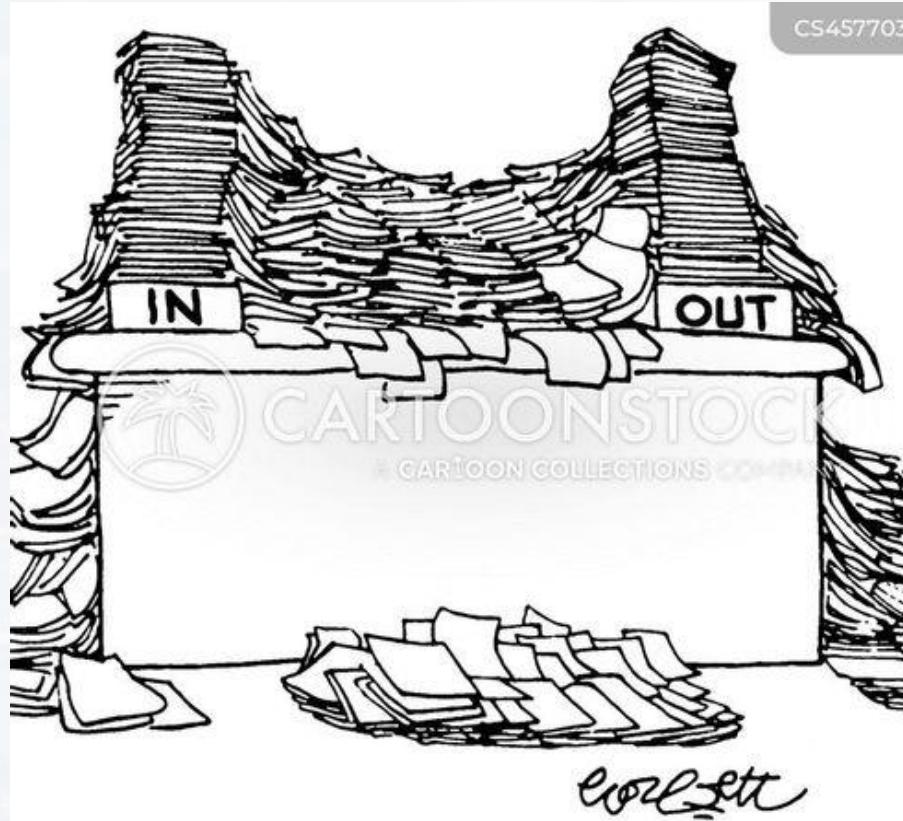
	time	sample 0	sample 4
0	0.001	23.241920	7.778409
1	0.002	22.002649	8.569085
2	0.003	20.490190	9.503710
3	0.004	19.192838	10.498987
4	0.005	18.527045	11.466261

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



dask uses **parallel processing**



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

.xlsx

83 sec

na

.csv

1.2 sec

0.016 sec
1.6 sec

.txt

1.5 sec

0.016 sec
0.92 sec



```
@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



```
@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!



```
@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)
```

slower, but requires less memory

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



Memory usage for Chunks:

Current: 80.02 MB

Peak: 85.15 MB

Total runtime: 9.98400000025844 seconds

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

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



solution: saving data in chunks as we read it!

```
@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)
```

compute turns dask
objects into actual
dataframes

actual saving part



```
@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
```



```
@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!')
```

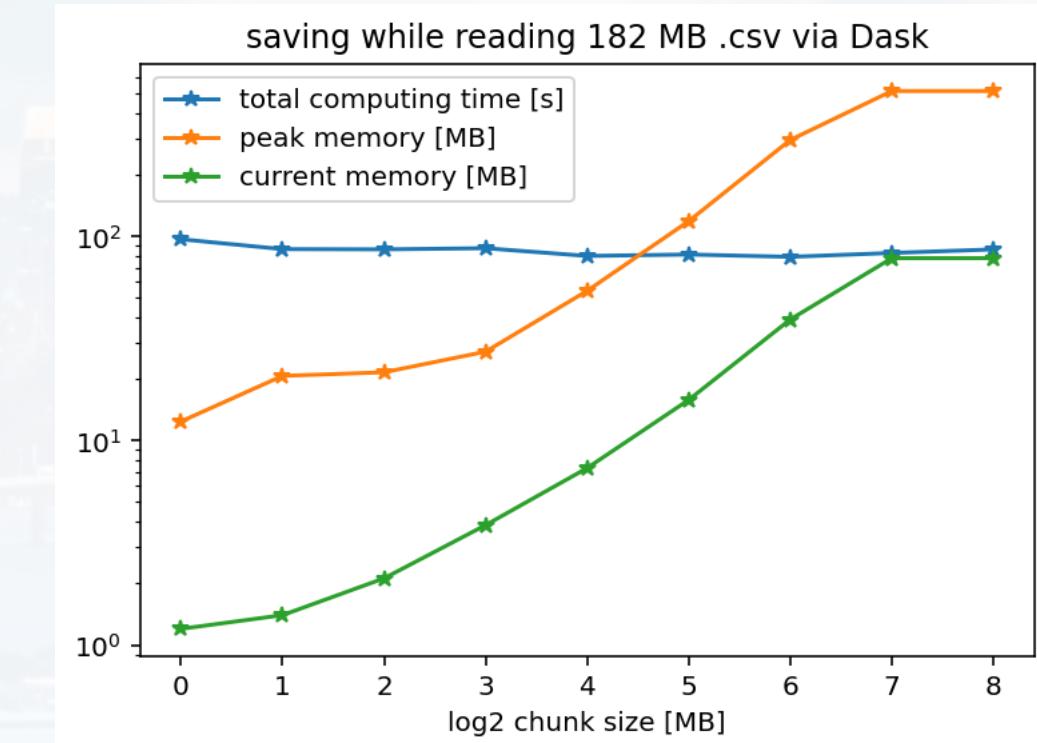
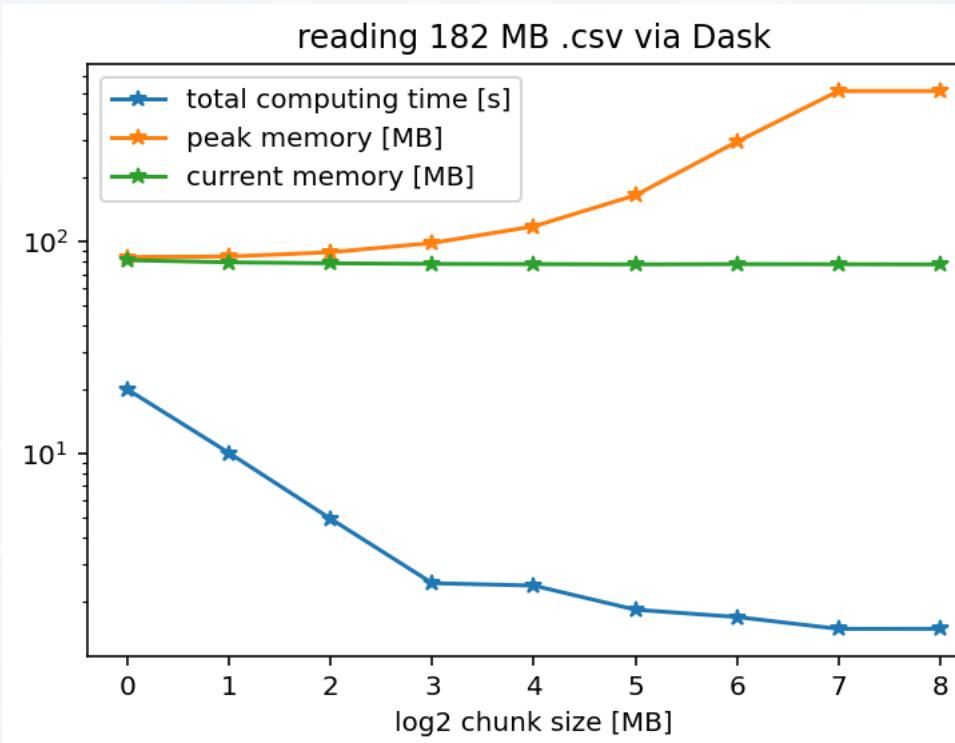
slower, but
requires less
memory

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

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



General tradeoff between memory use and speed!





Outline

Standard Pandas

Line By Line

Chunks With Dask

PySpark



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='sample 6',
     _c19='sample 17', _c20='sample 18', _c25='sample 25',
     _c28='sample 26', _c29='sample 29', _c36='sample 34',
     _c37='sample 35', _c38='sample 38')
```



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 **Exploratory Data Analysis**



Thank you very much for your attention!

