Does pandas Stop Where Big Data Begins?

J.J.H. Smit
10321977
University of Amsterdam
jorijn.smit@student.uva.nl

ABSTRACT

The popular data science tool pandas is limited by the fact that it needs to load all data into memory (plus overhead) in order to view or manipulate it. This paper researches methods that deal with situations in which there is more data than memory available. A thorough comparison with Koalas is made, a package which tries to mimic pandas' API but performs its processes using the parallel processing engine Apache Spark. However, due to its configuration, it is not as easy to deal with as pandas. In this experiment, letting pandas make use of swap memory is therefore a more efficient method of dealing with memory overload.

KEYWORDS

pandas, Koalas, swap memory, parallel processing

1 INTRODUCTION

Over the last year or so, the data analysis and manipulation tool pandas [4] has seen its popularity rise. Because it has been built on top of Python, it is super easy to incorporate into bigger programming projects. The API is also very *pythonic* in the sense that code written in pandas is very readable and easy to understand. Alternatives such as R or MATLAB lack both this portability and readability.

One thing pandas isn't so good at however is efficiency. Python, being an interpreted and a dynamically typed programming language, is not the fastest languages out there. The fact that pandas is built on top of that then doesn't help. Yes, pandas uses a lot of numpy[1] where possible and even Cython[2] in certain cases [5, p. 4]—which is great—but this only solves part of the issues.

A more pressing aspect of pandas' efficiency is in regards to the volume of the data: its memory use. In order to analyse or manipulate a dataset, pandas needs to fully load it into memory. A first problem we then encounter is that there is quite some memory overhead; for example a simple comma-separated values (CSV) file of 30.2MB already needs 47.8MB of memory as a `pandas.DataFrame` And this is before performing any alterations! Does this mean we have to reserve roughly twice the storage space in memory in order to just look at the data?

Let's say we take this memory overhead for granted. An obvious question then becomes: what happens when the dataset doesn't fit in memory anymore? Are we then restricted from using pandas completely? Is pandas only suitable for "laptop sized" or "small" data?

1.1 Single Node Approach

Simply loading in a file too big for memory will freeze Python. No warning, just an unusable session. The solution the pandas library offers to this is *chunking*. By chunking the data into smaller pieces, we can just iterate through the data step by step until all is processed. This might offer a solution in cases where the processing steps are known. However, running ad hoc analysis might provide more of a challenge. For instance, let's say I am interested in the maximum of a column. Let's also say that the total size of the raw file is of such a size that we are forced to load it into memory in chunks. Does `.max()` now return the maximum of the chunk or of the whole data? If the former is the case, analysis using chunks becomes practically impossible; we can only perform calculations on parts of the data. In the latter case, all chunks have to be read completely for every command. This can become tremendously slow. Is analysis of such big datasets still feasible?

1.2 Parallel Approach

A very promising solution to these limits of pandas is Koalas, introduced last year by Databricks[3]. It tries to bridge the gap between pandas and Apache's Spark[3], a big data processing engine. By making the pandas API available in a big data environment, Koalas users are able to analyse their data in the same fashion they are used to on a single node. On the background however, the computations are performed in parallel by Spark.

Dask provides a similar alternative to pandas' way of representing a dataset: `dask.dataframe`[4]. Again, it mimics the pandas API and parallelises it quietly in the background.

For both of these packages however, we should ask whether it offers all the features and functions pandas has. If it does, is it worth just always using these packages instead? Or; when does one need to make the switch from single to parallel

[1] https://numpy.org/
[2] https://cython.org/
[3] https://spark.apache.org/
[4] https://docs.dask.org/en/latest/dataframe.html

node? And in the case these parallel packages are not feature complete, how do we go about performing the missing functions?

In their announcement, Databricks talks about a "easy transition" from `pandas` to Apache Spark. Surely this would be a great solution; if both packages let themselves be used in exactly the same fashion, one could even contemplate dropping `pandas` altogether and always working with `Koalas`. It would be just a fast locally and at the same time be directly deployable on a parallel machine.

The main question that I want to answer in the rest of this paper is the following: **when a `pandas` dataframe becomes too big for memory, can Koalas provide an easy transition to parallel processing and how much better does it then perform?**

2 IMPLEMENTATION

In order to make the comparison described above, we need data. I decided on a dataset of historical trading data of cryptocurrencies. It is numerical, requires light computation but is big enough to fill a regular laptop's memory twice.

In total the dataset is 17GB and consists of 198 CSV files, each file containing the full history of a single pair of two currencies. Every row summarises a minute of trading data: candlestick data (opening, high, low and closing price), volume, amount of trades and more. The dataset goes back to 2017, so for well-traded pairs this means a CSV file with well over 1 million rows.[5]

2.1 Goal

The eventual goal is to combine all the 198 pairs into one dataframe. For simplicity's sake, all the price data of a single minute will be brought back to one single decimal and *all* the other columns will be dropped. This will leave us with 198 clean time series (one for each pair) which can then be joined using their timestamp. Minutes in which no trades took place (and thus no data is available) will be forward filled in with the last known price. This will result in a clean dataframe which is well-suited for further analysis.

2.2 Single Node Approach

Instead of trying to load in all files at once, the transformation was performed on a sample of the data first. This made it easy to see what exactly was happening while it was being performed. Once that proof of concept worked it was scaled up to include all files.

Using a chunking approach, the script (see the appendix for source code) loops over all the filenames found in the data folder and performs the following steps:

(1) load file into memory as a `pandas.DataFrame`
(2) compute a proper `pandas.DatetimeIndex` based on the timestamp column
(3) compute a new column with a single price for each minute by averaging the open and the close price
(4) join this new column to the dataframe `result`
(5) resample the `result` dataframe to 1 minute intervals and forward fill missing entries.

Resampling aligns timestamps that are not exactly the same but do fall within the same time span (in this case that of 1 minute).

2.3 Parallel Approach

Porting the code to a parallel approach is rather straightforward; only slight adjustments were needed (again, see appendix for the source). In theory, Databricks states that the only change needed is to replace the `pandas` package by `databricks.koalas`. However, a first difficulty arose when creating an empty dataframe for each subsequent dataframe to be joined on. Apparently `koalas` is not able to create such an object yet.[6] This was easily solved by implementing an additional if-statement.

The code then ran in multiple cluster environments; Google Colab[2], Databricks and Binder[1]—each single node having a configuration as close as possible to the MacBook used in the single node approach.

3 EVALUATION & DISCUSSION

In the single node approach it took a 2015 MacBook (1.1Ghz dual-core, 8GB RAM) 35 minutes and 28 seconds to build the resulting dataframe:

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1385760 entries,
2017-07-14 04:00:00 to 2020-03-02 11:59:00
Freq: T
Columns: 198 entries, RVN to FUN
dtypes: float64(198)
memory usage: 2.1 GB
```

As expected the memory gets maxed out during this process. But that is surprisingly well handled by the operating system through activating its swap memory. And even though the swap file got as big as 3.86GB and is an obvious constraint to the processing speed, it is good to know that the process is not entirely impossible on a single node.

Not so much luck on the distributed alternative. The biggest problem were the error messages thrown by the three cluster

[5] A newer version of this dataset can now be found at https://www.kaggle.com/jorijnsmit/binance-full-history. Note however it now is made up of Parquet instead of CSV files, compressing the size of the dataset by a factor of 5.

[6] https://github.com/databricks/koalas/issues/1084

environments this experiment was performed on. All three of them returned the same very unhelpful errors (with hundred of lines) which could only loosely be tied to mismatching Java versions. Multiple Java packages and versions were tried out (Oracle JDK, OpenJDK) but to no avail:

```
Py4JJavaError:
An error occurred while calling
o89554.collectToPython.
: java.lang.StackOverflowError
```

For what is is worth; it did take Colab only 16 minutes and 3 seconds to fully load the files and generate this error message. The code was executed successfully in all three cases but the resulting dataframe or any of its attributes remained uncallable.

3.1 Reproducibility

Much of the fact that `Koalas` failed to work 'out-of-the-box' has to do with the underlying configurations of the servers. A considerable part of this research was spent on these configurations. And although it is not the focus of the experiment, it does play an important role. Therefore a GitHub repository[7] has been made public, including configuration files for Binder. Binder is a service that starts a virtual machine and renders an environment from scratch, strictly based on a repository's specific branch, tag or commit and the configuration files it contains. In this case this is a `conda` environment based on `environment.yml`. This ensures reproducible results with consistent package versions. The source for this configuration file is included in the appendix. Visit the repository itself to see it action.

3.2 Conclusion

The main question that this paper is trying to answer is: when a `pandas` dataframe becomes too big for memory, can Koalas provide an easy transition to parallel processing and how much better does it then perform?

In the single node approach, we saw the rather surprising result that swap memory can grow to quite large sizes compares to the RAM. And albeit slower, it does create a very low threshold for continuing to work on these datasets without having to change your existing system or its configuration. The only requirement being that the use of swap memory is enabled.

Unfortunately the results in regards to parallel processing are not so clear. We do see a faster processing time, almost two-fold. This can also be due to the processor speed varying in both experiments. Also we don't know if the dataframe was actually built correctly; the object is never accessible. And because of this, `Koalas` is not a viable solution to the memory problem of `pandas` in this experiment. It is probable that the errors thrown on all three (!) cluster environments can be solved. For this particular task however, it is obvious that it is not worth the additional setup time. **`Koalas` is not able to provide the easy transition it promises. It appears it would be faster, but this can not be said with any certainty.** Swap memory is a wonderful saviour in the single node approach and is in this experiment the best method to solve situations in which the dataset does not fit in memory anymore.

`Koalas` makes for a persuasive alternative. As can be seen from the source code, the changes that need to be made are minimal. Active development of the package will further improve this. It is however not the API which is causing the problems but the back-end configuration. Running into Java error messages is unexpected and not what a Python developer or data scientist is hoping for when trying to find an easy way to scale a `pandas` project. Unless presented with a fully working setup of multiple nodes able to deal with `Koalas` without fault, closer-to-home solutions such as chunking, optimising memory use and swap memory are a lot more realistic in solving "big data" memory issues with `pandas`.

Lastly something can be said about the setup of the experiment itself. Parallel processing benefits the most from heavy processes that can run simultaneously. And although it is true that the process of loading a CSV file from disk can be parallelised, it also creates a new problem. Clusters are more likely to use network attached storage (NAS), opposed to having its own dedicated hard-disks. In retrospect it appears that the speed at which the data can be read from disk into memory can be a substantial factor here. To make a good comparison between single and parallel processing, an experiment which relies more on computational power instead of memory loading would be more suited.

[7] https://github.com/jorijnsmit/uva-pandas_vs_koalas

REFERENCES

[1] M Bussonnier, J Forde, J Freeman, B Granger, T Head, C Holdgraf, K Kelley, G Nalvarte, A Osheroff, et al. 2018. Binder 2.0-Reproducible, Interactive, Sharable Environments for Science at Scale. In *Proceedings of the 17th Python in Science Conference*, Vol. 113. 120.

[2] Tiago Carneiro, Raul Victor Medeiros Da Nóbrega, Thiago Nepomuceno, Gui-Bin Bian, Victor Hugo C De Albuquerque, and Pedro Pedrosa Reboucas Filho. 2018. Performance Analysis of Google Colaboratory as a Tool for Accelerating Deep Learning Applications. *IEEE Access* 6 (2018), 61677–61685.

[3] T. Liu and T. Hunter. 2019. *Koalas: Easy Transition from pandas to Apache Spark*. Databricks. https://databricks.com/blog/2019/04/24/koalas-easy-transition-from-pandas-to-apache-spark.html Accessed 2020-02-03.

[4] Wes McKinney. 2011. pandas: a Foundational Python Library for Data Analysis and Statistics. *Python for High Performance and Scientific Computing* 14 (2011).

[5] Wes McKinney. 2012. *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython.* " O'Reilly Media, Inc.".

A SOURCE CODE

A.1 Single Node Approach

```
import os
import pandas as pd

result = pd.DataFrame([])
for filename in os.listdir(base_path):
    full_path = f'{base_path}/{filename}'
    name = filename.split('-')[0]
    # (1) load file in memory
    new = pd.read_csv(full_path)
    # (2) compute index based on timestamp
    new['open_time'] = pd.to_datetime(
        new['open_time'],
        unit='ms'
    )
    new = new.set_index('open_time', drop=True)
    # (3) compute single price per minute
    new[name] = (new['open'] + new['close']) / 2
   # (4) join price column to the result dataframe
    result = result.join(new[name], how='outer')
# (5) resample to 1 minute intervals and forward fill
result = result.resample('1min').asfreq().ffill()
```

A.2 Parallel Approach

```
import os
import databricks.koalas as ks

first_run = True
for filename in os.listdir(base_path):
    full_path = f'{base_path}/{filename}'
    name = filename.split('-')[0]
    # (1) load file in memory
    new = ks.read_csv(full_path)
    # (2) compute index based on timestamp
    new['open_time'] = ks.to_datetime(
        new['open_time'],
        unit='ms'
    )
    new = new.set_index('open_time', drop=True)
    # (3) compute single price per minute
    new[name] = (new['open'] + new['close']) / 2
   # (4) join price column to the result dataframe
    if first_run:
        result = ks.DataFrame(new[name])
        first_run = False
    else:
     result = result.join(new[name], how='outer')
# (5) resample to 1 minute intervals and forward fill
result = result.resample('1min').asfreq().ffill()
```

A.3 Anaconda Configuration

```
name: pandas_vs_koalas
channels:
  - conda-forge
dependencies:
  - arrow-cpp
  - cython
  - htop
  - ipyparallel
  - koalas
  - openjdk
  - pandas
  - pip
  - pyarrow=0.8.0
  - python=3.7
  - pyspark
  - pip:
    - gsutil
```