# 6 ways to significantly speed up Pandas with a couple lines of code. Part 2

The second part of the article where I review and compare various Pandas optimization tools.

Magomed Aliev


In previous article, we looked at some simple ways to speed up Pandas through jit-compilation and

multiprocessing using tools like Numba and Pandarallel. This time we will talk about more powerful tools with which you can not only speed up pandas, but also c**luster it, thus allowing you to process big data**.

## Chapter 1:

- Numba
- Multiprocessing
- Pandarallel

## Chapter 2:

- Swifter
- Modin
- Dask

## Swifter

Swifter is another small but smart pandas wrapper. Depending on the situation, it chooses the most effective optimization method out of the possible ones — vectorization, parallelization or pandas implementations. Unlike pandarallel, it uses Dask instead of bare multiprocessing to organize parallel computing, we will talk about it later.

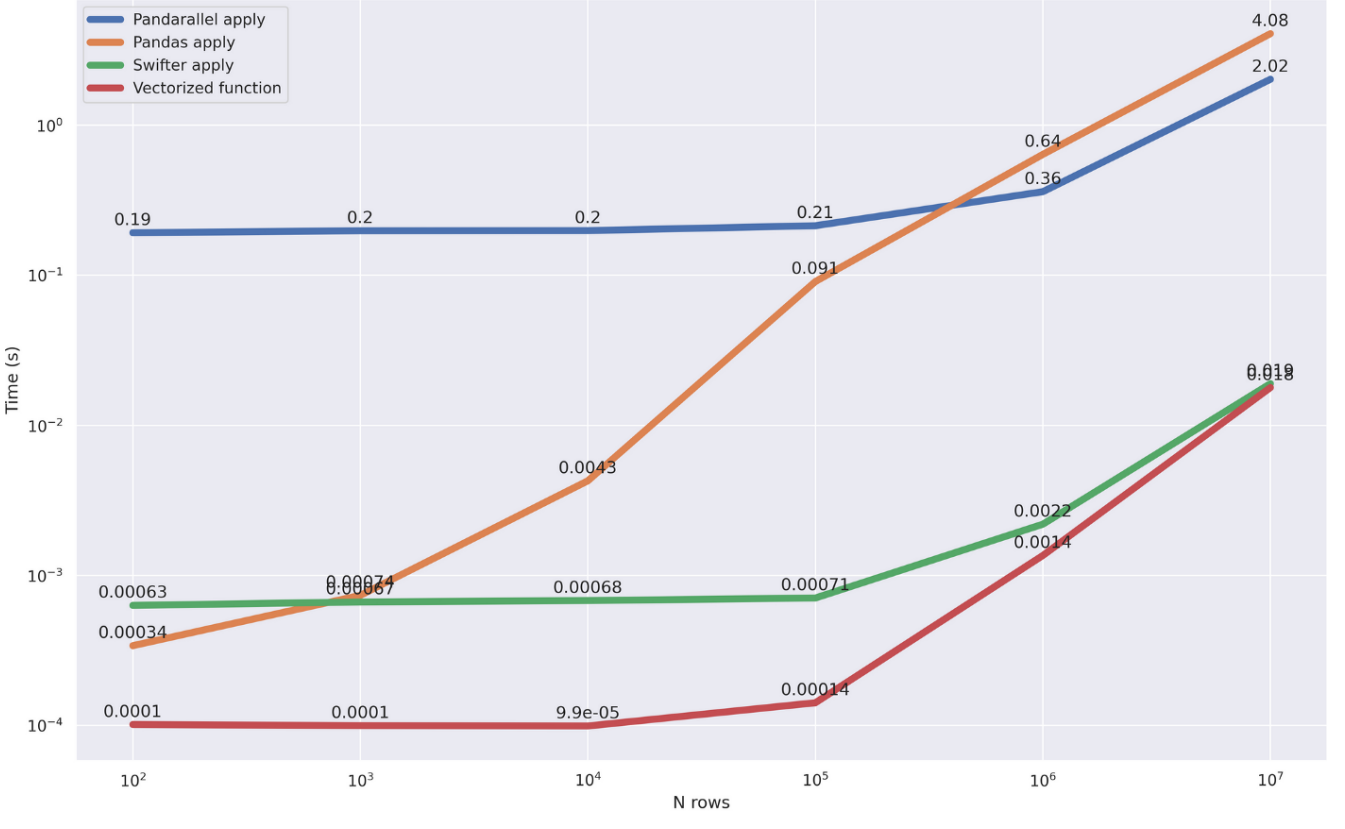
We will conduct two tests on the same news data as I used in previous chapter:

1. Test with function that can be vectorized (vectorized funcs are really fast)
2. Test with complicated function that cannot be vectorized.

For the first one, I use a simple math operation:

```
def multiply(x):
        return x * 5

# df['publish_date'].apply(multiply)
# df['publish_date'].swifter.apply(multiply)
# df['publish_date'].parallel_apply(multiply)
# multiply(df['publish_date'])
```

To make it clear which approach swifter chose, I included pandas processing, a vectorized approach, as well as pandarallel in the test:

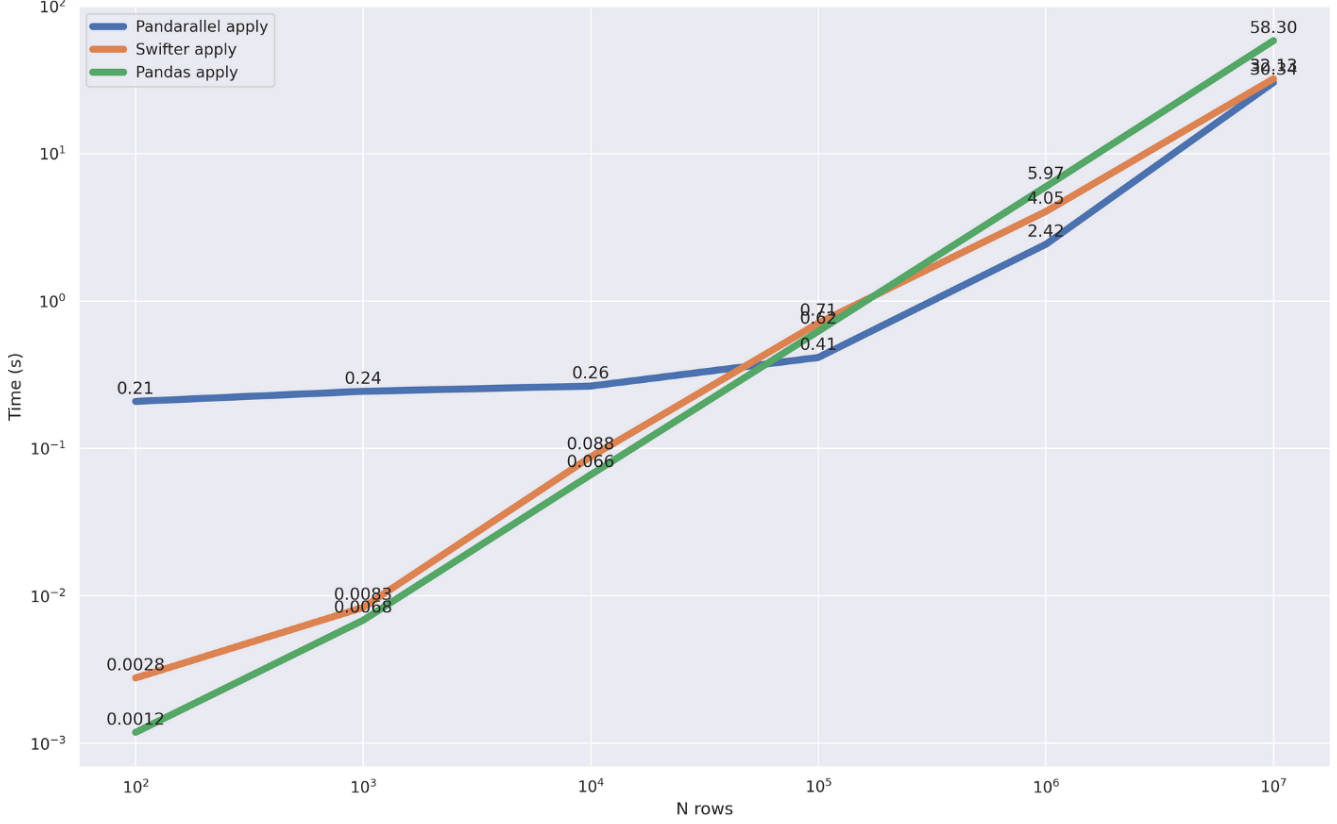


The graph clearly shows that **it goes on par with the vectorized version**, which is the most effective (with the exception of a small overhead that swifter spends on calculating the best method). This means that the optimization is carried out correctly.

Now let's see how he will cope with a more complex, non-vectorized case. Let's take our word processing function from prev chapter by adding work with swifter there:

```
# calculate the average word length in the title
def mean_word_len(line):
    # this cycle just complicates the task
    for i in range(6):
        words = [len(i) for i in line.split()]
        res = sum(words) / len(words)
    return res

# to work with strings, we initiate a special fu
df['headline_text'].swifter.allow_dask_on_string
```

Compare speed:




This case is even more interesting. While the amount of data is small (up to 100,000 lines), swifter uses the methods of pandas itself, which is clearly visible. Further, when pandas is no longer so efficient, parallel processing is enabled, and swifter starts working on several cores, aligning in speed with pandarallel.

# Summary

- Swifter **allows not only to parallelize, but also to vectorize functions**
- **Automatically** determines the best strategy for optimizing calculations, allowing you not to think about where to use it and where not to
- Unfortunately, it is not yet able to use `apply` on grouped data (`groupby`)

# Modin

Modin is a parallelization tool, that uses Dask or Ray, and is not much different from previous projects. However, this is a pretty powerful tool, and I can't call it just a wrapper. Modin implements its own `dataframe` class (although pandas is still used under the hood), in which at the moment there is already ~ 80% of the original functionality, and the remaining 20% refer to pandas implementations, thus **repeating its API completely**.

So, let's get started with the configuration, for which we only need to set the `env` variable to the desired engine and import the dataframe class:

```
# The Dask engine is currently considered experi
%env MODIN_ENGINE=ray
import modin.pandas as mpd
```

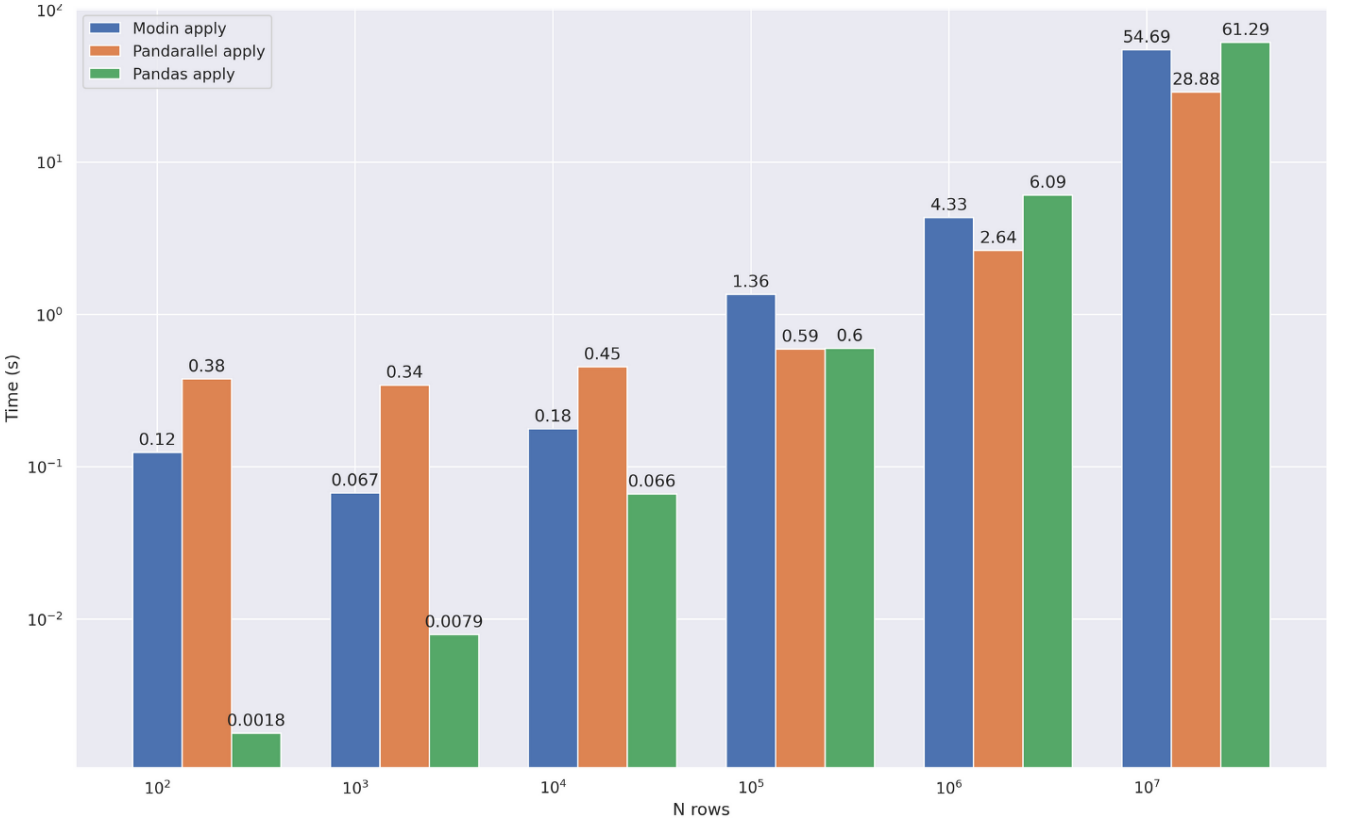
An interesting modin feature is **optimized file reading**. For the test, create a csv file with a size of 1.2 GB:

```
df = mpd.read_csv('abcnews-date-text.csv', heade
df = mpd.concat([df] * 15)
df.to_csv('big_csv.csv')
```

Now read it with modin and pandas:

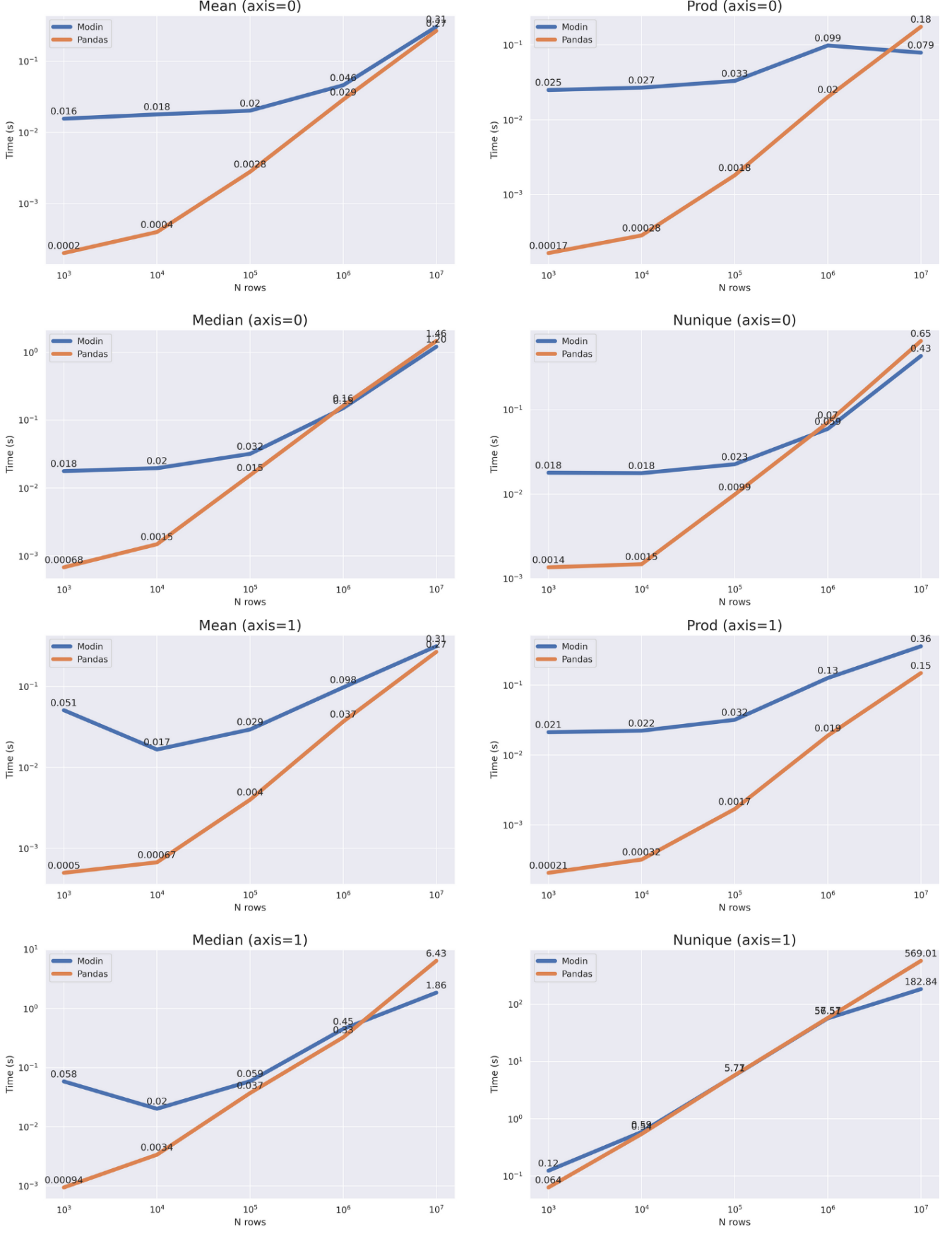
```
In [1]: %timeit mpd.read_csv('big_csv.csv', heade
8.61 s ± 176 ms per loop (mean ± std. dev. of 5 r

In [2]: %timeit pd.read_csv('big_csv.csv', header
22.9 s ± 1.95 s per loop (mean ± std. dev. of 5 r
```

We got **x3 acceleration**. Of course, reading a file is not the most common operation, but still nice. Let's see how modin behaves on our main text processing case:




We see that `apply` is not the strongest side of modin, most likely it will start to benefit on larger data, but I did not have enough RAM to check it. However, modin arsenal does not end there, so we'll check other operations:

```
# This time I used numerical data
df = pd.DataFrame(np.random.randint(0, 100, size=
```




What do we see? Very big overhead. In case of `median` and`nunique`, we only got acceleration when the size of the dataframe grew to `10 ** 7`, while in the case of`mean` and `prod (axis = 1)`, this did not happen, but from the graph it can be seen that pandas computation time is growing

faster and with a size of `10 ** 8` modin will already be more efficient in all cases.

## Summary

- **Modin dataframe API is identical to pandas** and to adapt the code for big data, just change one line
- **Very large overhead**, should not be used on small data. According to my calculations, it is relevant to use it on data larger than 1GB
- **Support for a large number of methods** — at the moment more than 80% methods have an optimized version
- Modin is **capable not only of parallel computing, but also clustering** — you can configure the Ray / Dask cluster and modin will connect to it
- There is a very useful feature that allows you to **use the disk if the RAM is full**
- Both Ray and Dask bring up a pretty useful dashboard in the browser. I used Ray:

# Dask

Dask is the last and most powerful tool on my list. It has a huge number of features and deserves a separate article or even several of them. In addition to working with numpy and pandas, he also has machine learning models — dask has integrations with **sklearn** and **xgboost** , as well as many of its own models and tools. And all this can work **both on the cluster and on your local machine** with all cores connected. However, in this article I will focus on pandas.

All you need to do to configure dask is to set up a cluster of workers.

```
from distributed import Client
# create local cluster
client = Client(n_workers=8)
```

Dask, like modin, uses its own `dataframe` class, which covers all the main functionality:

```
import dask.dataframe as dd
```

Now we can start testing. Lets compare speed of reading file:

```
In [1]: %timeit dd.read_csv('big_csv.csv', heade
```

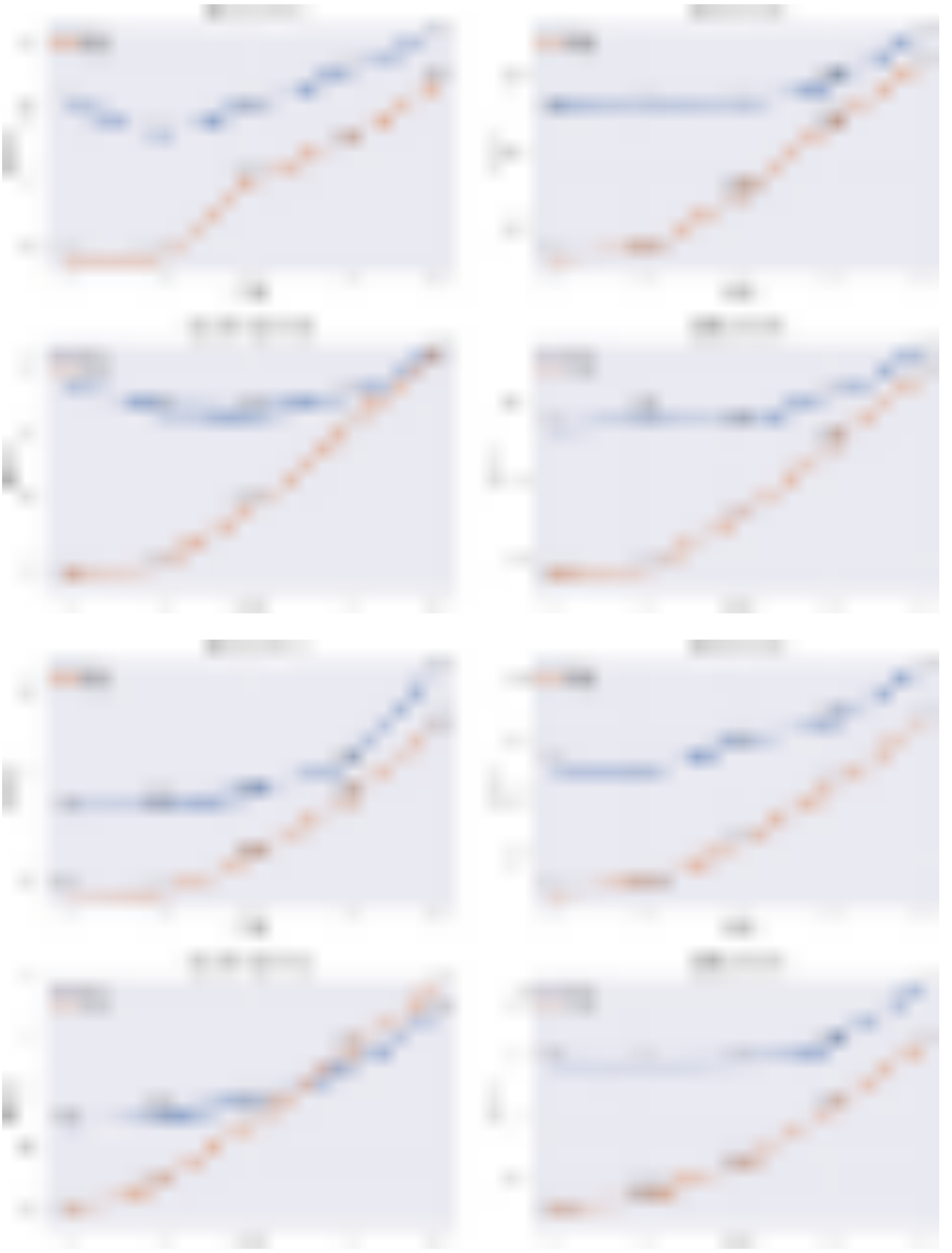
```
6.79 s ± 798 ms per loop (mean ± std. dev. of 7 r
19.8 s ± 2.75 s per loop (mean ± std. dev. of 7 r
```

Dask worked somewhere **3 times faster**. Let's see how it will cope with our main task — the `apply` acceleration. For comparison, I added [pandarallel and swifter](#) here, which in my opinion also did a good job:

```
# compute () is needed because all calculations :
# dd.from_pandas – method for converting pandas d
dd.from_pandas(df, npartitions=8).apply(mean_word
```

We can say that dask showed the best results, working out faster than anyone starting from `10 ** 4` lines. Now lets check out some other useful features:

```
# same numerical data as for modin
df = pd.DataFrame(np.random.randint(0, 100, size=
```



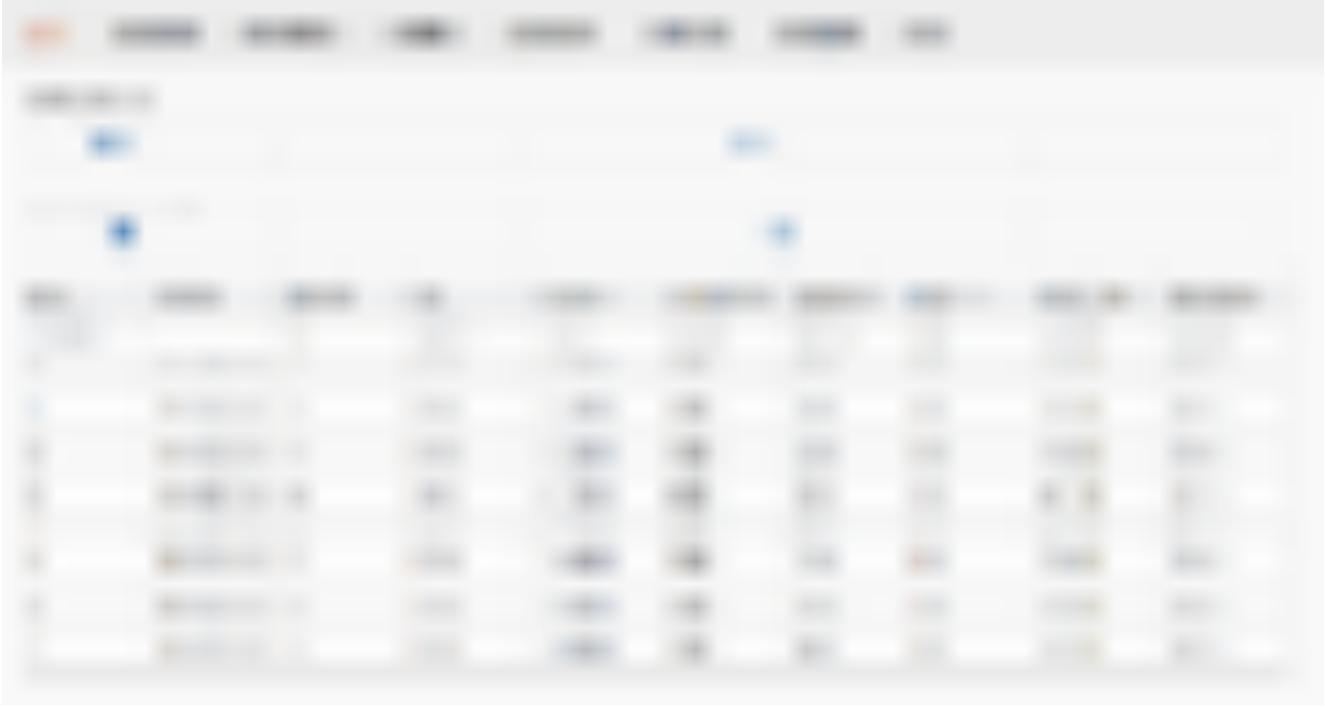
As with modin, we have a rather large overhead, but the results are rather mixed. For operations with the parameter `axis = 0` we did not get acceleration, but according to the growth rate it is clear that with `data size`

`> 10 ** 8` dask will perform better. For operations with `axis = 1`, you can even say that pandas will work faster (with the exception of the`quantile (axis = 1)`method).

Despite the fact that pandas proved to be better in many operations, do not forget that dask is primarily a cluster solution that can work where pandas can not cope (for example, with large data that does not fit into RAM).

## Summary

- Good at `apply` acceleration
- Very big overhead. **Designed to manipulate large datasets that do not fit in memory.**
- It **can work both in a cluster and on a single machine**
- **Dask API copies pandas**, but not completely, so adapting code under Dask by replacing only the dataframe class may fail
- **Support for** a large number of methods
- Useful dashboard



## Conclusion

It should be understood that **parallelization is not a solution to all problems, and you always need to start with optimizing your code**. Before parallelizing a function or applying cluster solutions like Dask, ask yourself: Is it possible to apply vectorization? Is data stored efficiently? Are indexes configured correctly? And if after answering these questions your opinion has not changed, then you really need the described tools, or you are too lazy to do optimization.

Thank you for your attention, I hope these tools come in handy!

> *P.s Trust, but verify — all code used in the article (benchmarks and charts drawing), I posted on github*

> *Originally posted on alievmagomed.com on may 27 2020*