Beyond Pandas: Spark, Dask, Vaex and other big data technologies battling head to head

API and performance comparison on a billion rows dataset. What should you use?

Jonathan Alexander

Photo by Pietro Mattia on Unsplash

Why

When confronting a new data science problem, one of the first questions to ask is which technology to use. There is hype; there are standard tools; there are bleeding-edge technologies, entire platforms and off-the-shelf solutions.

Over the last few years, I've been building proof of concepts and solutions with any technology I could get my hands on. If there is a new platform, I register for the trial; if any of the big cloud platforms releases a new feature, I will try it out; and when a new technology comes up, you can be damn sure I'll run through some tutorials and try it on my datasets.

From this perspective, I've decided to compare data wrangling technologies to choose the one most suitable for tabular data exploration, cleaning and wrangling for my next projects. I also take this as an opportunity to get back in touch with technologies I haven't been using for a few years, which I assume have gotten better with time.

TL;DR

Vaex is on the rise as the new big data technology. But, If you're already using a PySpark platform or have PySpark talent available, is it still a fine choice.

What

In the following, I assume basic proficiency with Python API and big data capabilities. The data I chose was the

Taxi **billion rows 100GB** dataset. The goal is to compare the technologies on their APIs, performance, and ease of use. I consider Pandas as the baseline having the most natural API (which is debatable I admit), as it is the most common solution by far, but can not handle big data.

You have two conventional approaches/ways to deal with big datasets: stronger/distributed computations where you match the memory to the size of the data, or an out-of-core solution, where data is only read in memory when it's necessary.

The difference in costs is immense, so I've decided only to consider solutions which can work out-of-core.

The competitors:

- Dask DataFrame — Flexible parallel computing library for analytics.
- PySpark — A unified analytics engine for large-scale data processing based on Spark.
- Koalas — Pandas API on Apache Spark.
- Vaex — A Python library for lazy Out-of-Core dataframes.
- Turicreate — A relatively clandestine machine learning package with its dataframe structure — SFrame, which qualifies.
- Datatable — The backbone of H2O's Driverless.ai. A dataframe package with specific emphasis on speed and big data support for a single node.

Some honourable mentions:

- H2O — The standard in-memory dataframe is well-rounded. Still, with the recommendations of a cluster four times the size of the dataset, you need deep pockets to use it for exploration and development.
- cuDF (RapidAI) — A GPU dataframe package is an exciting concept. For big data, you must use distributed GPUs with Dask to match your data size, perfect for bottomless pockets.
- Modin — A tool to scale Pandas without changes to the API which uses Dask or Ray in the backend. Sadly at this moment, it can only read a single parquet file while I already had a chunked parquet dataset. With the prospect of getting similar results as Dask DataFrame, it didn't seem to be worth pursuing by merging all parquet files to a single one at this point.
- Pandas — Pandas has a chunking feature, but for exploration and dynamic interactions, it is not in the same league as the others.
- Vaex does have a GPU and numba support for heavy calculations which I did not benchmark.

How

I used a $0.95 per hour ml.c5d.4xlarge instance on AWS Sagemaker so it would be easy to replicate the benchmarks. It has 16 vCPUs, 32 GB of RAM plus 500 SSD which has a subtle resemblance for a solid laptop.

Although all of the competitors can read a CSV file, a more optimized approach is to use the binary version that is most suitable for each technology. For PySpark, Koalas, and Dask DataFrame, I used Parquet, while for Vaex, I used HDF5. Turicreates' SFrame has a particular compressed binary version. Datatable is a special case optimized for the jay format. While I could not read the CSV files with Datatable, regardless of the size of my instance (even with more than double the data size in RAM), hacked a way to use HDF5 with it. It would be nice to do an update when Datatable can reliably read multiple CSV, Arrow, or Parquet files.

I'd like to note that the development of this benchmarks was a bit expensive, as technologies kept crashing many times after running for a few hours, though just running the code as is shouldn't break your wallet.

Results

Coding complexity

Just eyeballing the APIs can give you a sense of the amount of code and design patterns of each technology.

Function	Pandas/Modin/Koala	Vaex	Dask DataFrame	Turicreate	H2O	PySpark	Datatable
Read file	pd.read_csv() pd.read_parquet()	vaex.read_csv() vaex.open()	dd.read_csv() dd.read_parquet()	tc.read_csv() tc.SFrame()	h20. upload_file() h2o.import_file()	sqlContext.read.csv() sqlContext.read.parquet()	dt.fread() dt.open()
Count	len(df)	len(df)	len(df)	len(df)	len(df)	df.count()	df.shape[0]
Mean	df.x.mean()	df.x.mean()	df.x.mean() .compute()	df['x'].mean() tc.Sketch(df['x']).mean()	df['x'].mean()	df.select(f.mean('x')) .collect()	df[:, dt.mean(dt.f.x)]
Standard deviation	df.x.std()	df.x.std()	df.x.std() .compute()	df['x'].std() tc.Sketch(df['x']).std()	df['x'].sd()	df.select(f.stddev('x')) .collect()	df[:, dt.sd(dt.f.x)]
Sum columns	df['x']+df['y'] df.x + df.y	df['x']+df['y'] df.x + df.y	df['x']+df['y'] df.x + df.y	df['x']+df['y']	df['x']+df['y']	df['x']+df['y']	df[:, f.x + f.y]
Sum columns mean	(df.x + df.y).mean()	(df.x + df.y).mean()	(df.x + df.y).mean() .compute()	(df['x'] + df['y']) .mean()	(df['x'] + df['y']) .mean()	df.select(f.mean(df['x'] + df['y'])) .collect()	df[:, dt.mean (f.x + f.y)]
Value counts	df.x.value_counts()	df.x.value_counts()	df.x.value_counts() .compute()	df['x'].value_counts()	df['x'].table()	df.select('x').distinct() .collect()	df[:,dt.count(f.x),'x']
Group-by	df.groupby(by='z') .agg({ 'x': ['mean', 'std'], 'y': ['mean', 'std']})	df.groupby(by='z') .agg({ 'x': ['mean', 'std'], 'y': ['mean', 'std']})	df.groupby(by='z') .agg({ 'x': ['mean', 'std'], 'y': ['mean', 'std']}) .compute()	df.groupby('z', operations={ 'c1':tc.aggregate.MEAN('x'), 'c2':tc.aggregate.STD('x'), 'c3':tc.aggregate.MEAN('y'), 'c4':tc.aggregate.STD('y')})	df.group_by('z') .mean(col = ['x', 'y']) .sd(col = ['x', 'y']) .get_frame()	df.groupby('z') .agg(f.mean('x'), f.stddev('x'), f.mean('y'), f.stddev('y'))	aggs = { 'c1': dt.mean(f.x), 'c2': dt.sd(f.x), 'c3': dt.mean(f.y), 'c4': dt.sd(f.y),} df[:, aggs, dt.by(f.z)]
Join	df.join(other, on = 'key') pd.merge(df, other)	df.join(other, on = 'key')	dd.merge(df, other)	df.join(other, on = 'key')	df.merge(other)	df.join(other, on = 'key')	other.key = 'key' df[:,:,dt.join(other)]

Commonly used methods (Image by author)

Winners — Vaex, Dask DataFrame, Turicreate, and Koalas have a very Pandas-like code (for Koalas it's identical), it's easy to do whatever you want.

Losers — PySpark and Datatable as they have their own API design, which you have to learn and adjust. Not a difficult task, but if you are used to working with Pandas, it's a disadvantage.

Features

Winners — PySpark/Koalas, and Dask DataFrame provide a wide variety of features and functions. Note that in some complex cases when using PySpark, you might need " map-reduce" knowledge to write algorithms to fit your needs. With Dask DataFrame, you might need to know when you can or cannot use a sklearn feature that can scale without a large memory footprint.

While Vaex and Turicreate have some missing features, they do cover most core functionalities.

Losers — Datatable seems a bit immature and far behind.

Pipelines

Often, when building solutions for machine learning and backend API (unlike with visualizations), you need to program a pipeline of your process. For example, when normalizing a column, you need to remember the mean and standard deviation to normalize new observations. Here simplicity, flexibility, and writing less code and are essential. For many data-science applications, this can be 80% of the work.

Winner — Vaex. With its expression system, any transformation to the dataset is saved in the background such that you can easily apply it to new data. That makes pipelining not just painless, but practically a non-task.

Runner up is Dask DataFrame, which has a variety of pre-processing tools. However, you might need to implement your transformers and consider which sklearn transformers can be processed efficiently.

Next is PySpark. Even though building pipelines is one of PySpark strongest suits, you need to write much code to make it happen. Another thing that holds PySpark back is that installation and deployment of models and pipelines are far from trivial. You almost surely need (or would

prefer) using an expensive platform like Databricks and Domino, or rely heavily on the infrastructures of Sagemaker, Dataproc or Azure.

Losers — Koalas, Turicreate, Datatable

Turicreate and Datatable have no pipelining features to speak of.

Although Koalas has a better API than PySpark, it rather unfriendly for creating pipelines. One can convert a Koalas to a PySpark dataframe and back easy enough, but for the purpose of pipelining it is tedious, and leads to various challenges.

Lazy evaluation

Lazy evaluation is a feature where calculations only run when needed. For example, if I have two columns A and B, creating a new column A*B takes practically 0 seconds and no memory. If I want to peek at the first few values of that column, only those are calculated, and not the entire column.

Lazy evaluation makes the process of feature engineering and exploration MUCH faster, more comfortable, and prevents you from having other massive columns in memory. It's particularly valuable when working with big datasets as engineering new columns, joining tables, or filtering data too big to fit in memory is likely to crash your machine.

Performance-wise, as you can see in the following section, I created a new column and then calculated it's mean. Dask DataFrame took between 10x- 200x longer than other technologies, so I guess this feature is not well optimized.

Winners — Vaex, PySpark, Koalas, Datatable, Turicreate.

Losers — Dask DataFrame.

Performance

> "All benchmarks are wrong, but some are useful" — misquoting George Box

Since performance can vary, I was inspired by this blog to run every benchmark twice. I'd like to consider the first run to be more relevant for a batch job (and more indicative of disk read speed), and the second one to be more representative of the experience when you are working interactively (the actual speed of the method).

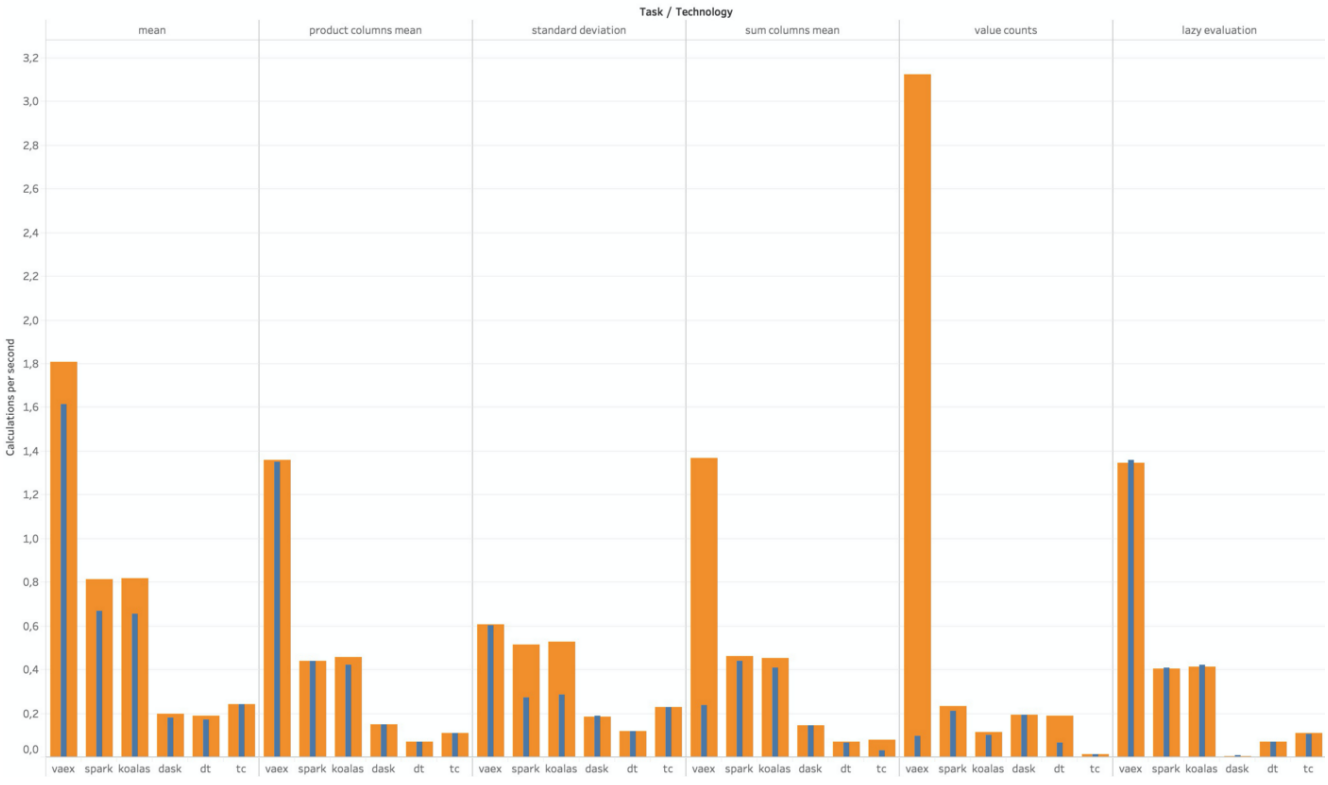
In all of the following graphs:

- First runs are represented by the blue bars and the second in orange.
- Turicreate has the Sketch feature which calculates a bunch of statistics and estimations at the same time; it's better for statistical analysis.
- For a more condensed name visualization, I used aliases: "dt" for Datatable, "tc" for Turicreate, "spark"

for PySpark and “dask” for Dask DataFrame.

Basic Statistics

Here I tested the basics; mean, standard deviation, value counts, mean of a product of two columns, and creating a lazy column and calculating it’s average.



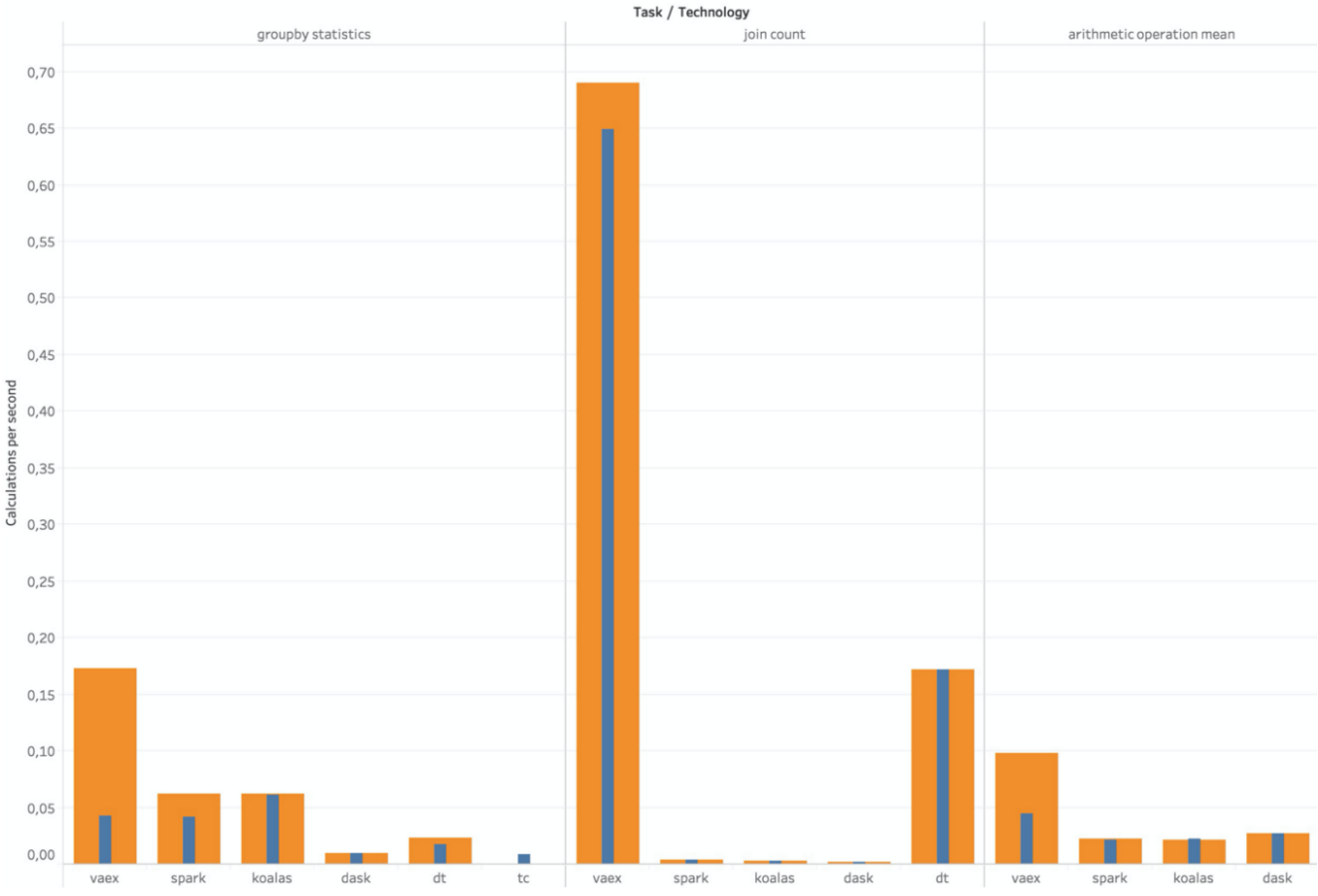
Bigger is better (Image by author)

- Vaex is in the lead.
- Koalas gets similar results to PySpark, which make sense as it uses PySpark in the background.
- Dask DataFrame, Turicreate and Datatable are falling behind the pack.

Heavy calculations

I run group-by with mean and standard deviation on two columns, then joined it to the original dataset and counted

the rows so I won't have to handle the entire joined dataset in memory. I also run a super-duper complicated mathematical expression to explore the impact of a lengthy feature engineering process.



Bigger is better (Image by author)

- Vaex does so well on join it distorts the graph.
- Datatable did remarkably well on join. This result might be related to the way I wrapped Datatable with HDF5. I didn't include all the string columns, which led to a much smaller footprint for the dataframe. But a sweet second place to our dark horse.
- Dask DataFrame and Turicreate again fall far behind.
- Please note that when I run a join on a newly created column, all technologies but Vaex crashed. Take it into account when planning your feature engineering.

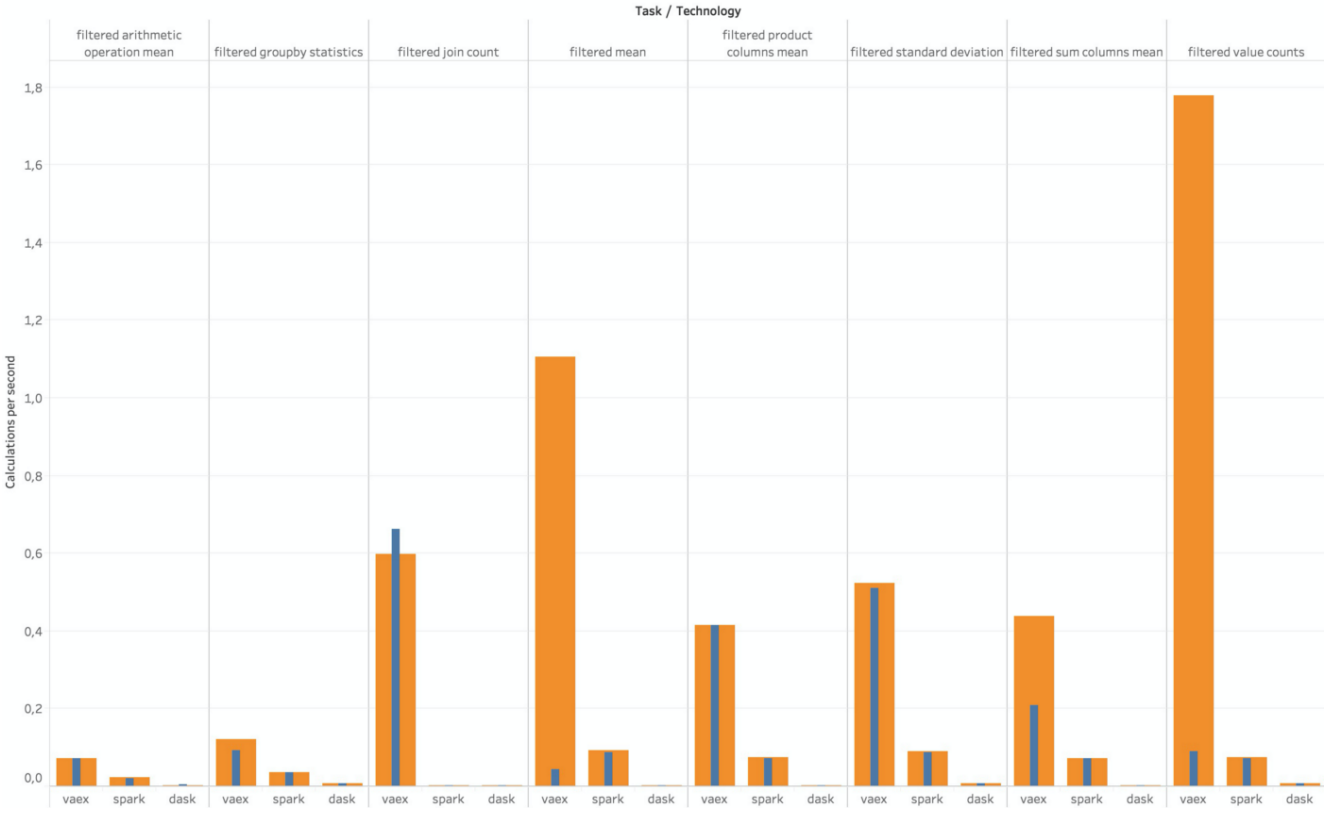
A definite win for Vaex while Dask DataFrame and Turicreate are getting further behind.

Filtering

Here I first filtered the data and repeated the operations above.

Unfortunately, Koalas, Datatable and Turicreate crashed at this point.

These results are a good indication of data cleaning performance.



Bigger is better (Image by author)

- It seems like Vaex is prevailing in most cases, especially in the second run.
- Dask DataFrame gets an honourable mention as it didn't crash, but the results were about 5x–100x

slower than Vaex and PySpark.

The numbers

Below you can see the tabulated results.

- The results are in seconds.
- Where a result is missing, the technology crashed.
- Values smaller than one second indicate lazy evaluation.
- Interactive table.

	Run / Technology											
	1						2					
Task	vaex	spark	koalas	dask	dt	tc	vaex	spark	koalas	dask	dt	tc
arithmetic operation	0,00		0,14	0,02			0,00		0,13	0,02		
arithmetic operation mean	22,25	46,49	45,09	37,03			10,20	45,68	45,79	36,52		
count	0,00	0,38	0,36	7,84	0,00	0,00	0,00	0,30	0,37	7,94	0,00	0,00
filter data	1,09	0,29		0,00		0,00	1,11	0,09		0,00		0,00
filtered arithmetic operation mean	14,10	46,23		247,54			14,02	45,62		286,51		
filtered groupby statistics	10,81	27,22		126,43			8,29	26,77		127,69		
filtered join count	1,51	266,50		763,51			1,68	262,22		762,84		
filtered mean	23,17	11,25		640,88			0,90	10,71		694,67		
filtered product columns mean	2,41	13,93		503,49			2,42	13,48		274,04		
filtered standard deviation	1,96	11,38		119,48			1,91	11,21		121,12		
filtered sum columns mean	4,77	13,80		521,56			2,28	13,62		625,51		
filtered value counts	10,94	13,71		122,57			0,56	13,60		121,52		
groupby statistics	23,55	23,87	16,32	102,54	56,00	120,65	5,77	16,09	16,14	104,01	42,72	
join	0,65		0,17		5,68		1,00		0,16		5,82	
join count	1,54	254,71	294,84	636,31	5,83		1,45	258,15	300,70	646,42	5,82	
lazy evaluation	0,74	2,45	2,37	101,83	14,48	9,51	0,74	2,47	2,42	153,02	14,48	9,23
mean	0,62	1,50	1,53	5,55	5,88	4,17	0,55	1,23	1,22	5,06	5,28	4,14
product columns mean	0,74	2,28	2,36	6,74	14,48	9,25	0,73	2,27	2,18	6,63	14,52	9,21
read_file	0,01	0,11	0,24	0,77	0,01	0,00	0,01	0,10	0,16	0,81	0,01	0,00
standard deviation	1,66	3,68	3,51	5,23	8,29	4,34	1,65	1,94	1,89	5,38	8,28	4,37
sum columns mean	4,23	2,28	2,45	6,90	15,54	32,70	0,73	2,17	2,21	6,82	14,61	12,82
value counts	10,56	4,72	9,70	5,17	15,00	81,59	0,32	4,32	8,70	5,16	5,27	75,38

Smaller is better (Image by author)

Winners — A clear win for Vaex.

Second place goes to PySpark and Koalas.

Losers — Dask DataFrame, Turicreate, Datatable.

Final points

- This code lets you compare the APIs and run the

benchmarks yourself. All the datasets are available for download in the right binary format.

- I had all kind of issues with Dask DataFrame, and to get the best performance possible, I went to great lengths restarting the kernel and re-read the data before some calculations. Although this was unpleasant, I did my best to get the best performance I could. If you just run the notebook as is, you might wait for hours or it might crash.
- As expected, Koalas and PySpark have very similar results as both use Spark behind the scenes.
- As mentioned before, I could not apply the tests on a newly created column not persisted in the file as it would crash Dask DataFrame and PySpark. To combat it, I further calculated the mean or the count of the results instead, to force a single value.
- I did not use any string columns because of our hack to Datatable, which did not work on string columns.
- Disclaimer — I am a part of vaex.io, I know personally one of the creators of Turicreate, a contributor to Spark, but I attempted to be as unbiased as I could, doing it as a personal, professional project.

Conclusions

I was using Turicreate as my go-to package for a few years since it was open-sourced, and PySpark before that, but I am switching to **Vaex**. Although still in it's early stages and a bit raw, the expressions and transfer of

states allow me to write much less code sacrificing features I don't commonly use, and it's just super fast.

I was surprised to see how well PySpark and Koalas did. Still, setting it up, deploying solutions without using a ready platform, the issues with pipelines, the unexplainable errors during development (and funky PySpark APIs) are just too much for me.

Dask DataFrame was an unfortunate challenge. It crashed numerous times, and I went through hoops to have it competitive in performance (check out the notebook).

All in all, PySpark and Dask DataFrame were the most expensive in time and money during the benchmark development.

Aftermath

I am interested to see how Datatable grows in the future.

Don't expect to see much progress for Turicreate as they are currently concentrating on their deep-learning development.

If you have any questions or ideas, you are welcome to comment/issue in the Github repo for benchmarks.

I plan to re-run and update the benchmarks if I find a reasonably fair way to optimize any of the technologies, and including string column benchmarks while using

different instances.

You can follow me on Twitter, Linkedin.

Updates

- 01/06/20 — After a great comment by , I added PySpark lazy evaluation to the benchmarks and updated all plots. The new plots are indistinguishable from the original version besides the addition of the PySpark lazy evaluation results in the first plot and a small change in numbers in the final table. I added PySpark to the lazy evaluation winners as well.
- 16/06/20 — H2O does not have a GPU implementation for its Dataframes. I redacted that comment.