Pandas 2, Polars or Dask?

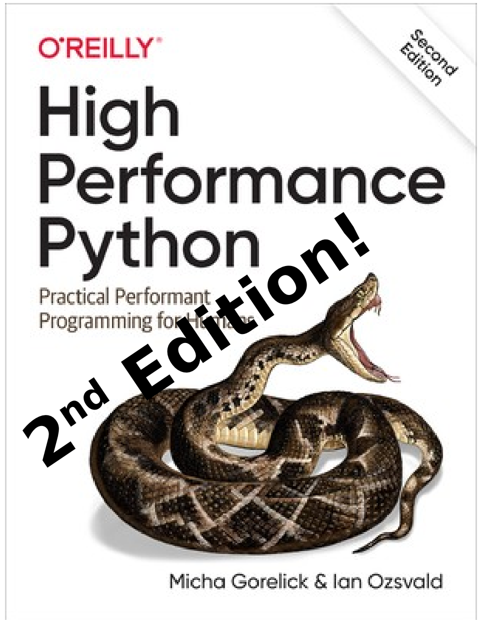
PyData London 2023 Talk

@IanOzsvald – ianozsvald.com

@GilesWeaver – ???

We are Ian Ozsvald & Giles Weaver

- Interim Chief Data Scientist
- Data Scientist



Part of **PyData – 216 groups**

PyData London Meetup

London, United Kingdom

11,880 members · Public group

3 interesting DataFrame libraries

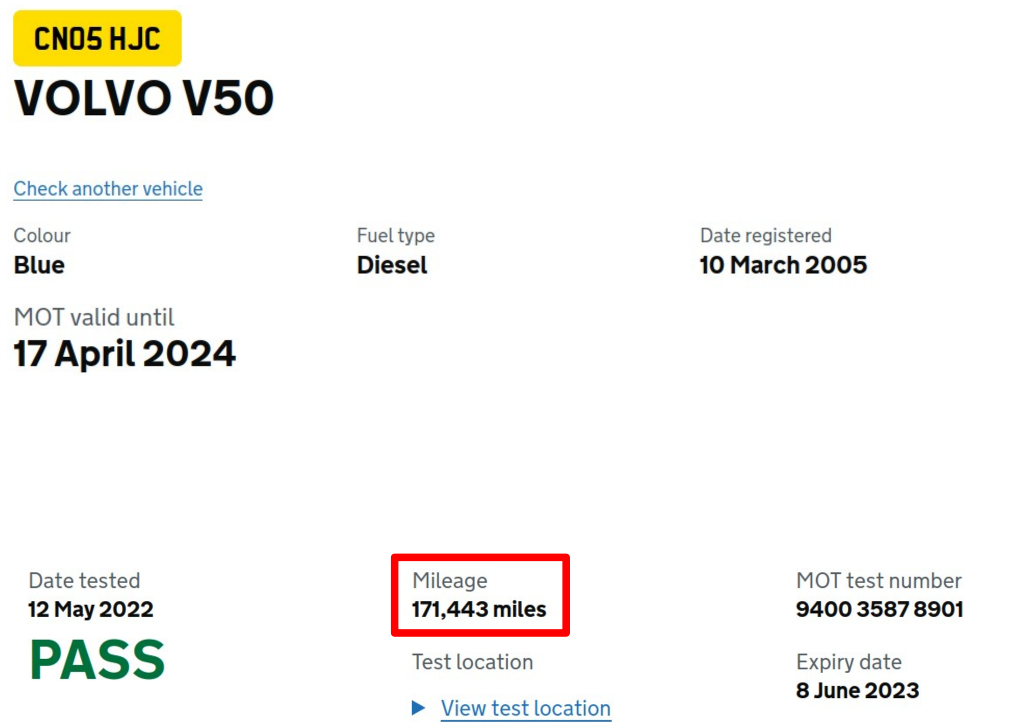
- Lots of change in the ecosystem in recent years
- Which library should you use?
- We learned Polars in 2 weeks
- We benchmark. All benchmarks are lies

pandas dask polars

https://www.check-mot.service.gov.uk/results?registration=cn05hjc

Car Test Data (UK DVLA)

- 20 years of roadtest pass or fails
- 30M vehicles/year, [C|T]SV text files
- Text→Parquet made easy with Dask
- 600M rows in total



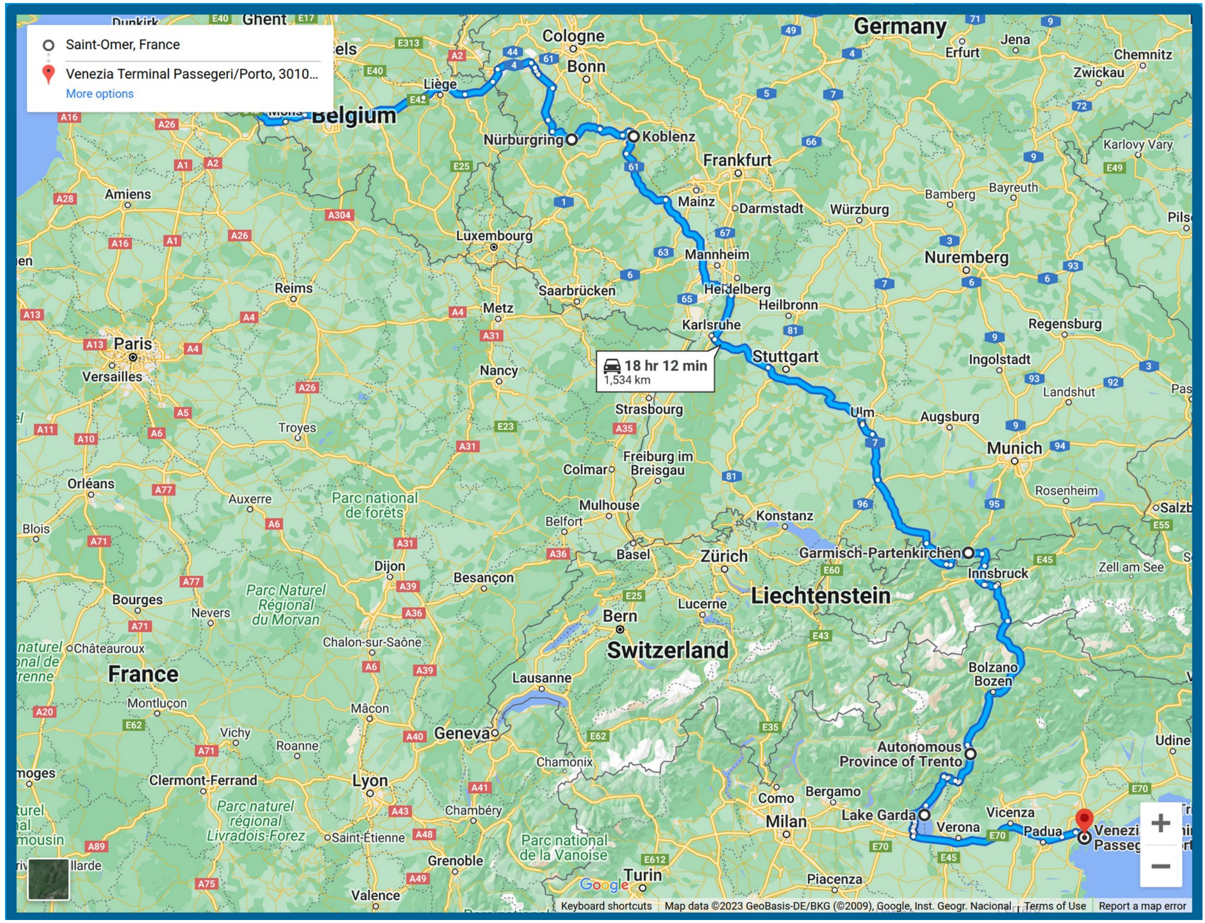
test_date	test_class_id	test_type	test_result	test_mileage	make	model	colour	fuel_type	cylinder_capacity	first_use_date
2022-05-12 00:00:00	4	NT	F	171443	VOLVO	V50	BLUE	DI	1997	2005-03-10 00:00:00
2022-05-12 00:00:00	4	RT	P	171443	VOLVO	V50	BLUE	DI	1997	2005-03-10 00:00:00

Motoscape Charity Rally

Rob, Edd and Ian's fundraiser for Parkinson's UK

MOTOSCAPE BANGER RALLY 2023, 2 September 2023

- Ian - “Let’s do something silly”
- 2,000 mile round trip <£1k car
- Ideally it shouldn’t break or explode



Ian Ozsvald

Pandas 2 – what's new?

- PyArrow first class alongside numpy
- Internal clean-ups so less RAM used
- Copy on Write (off by default)

10:15 AM 40min
New Developments in Pandas and Dask Dataframes
Matthew Rocklin

Beaumont

11:00 AM 40min
Executives at PyData
Ian Ozsvald

PyArrow vs NumPy – which to use?

Backend	String dtype	Nullable integer dtype

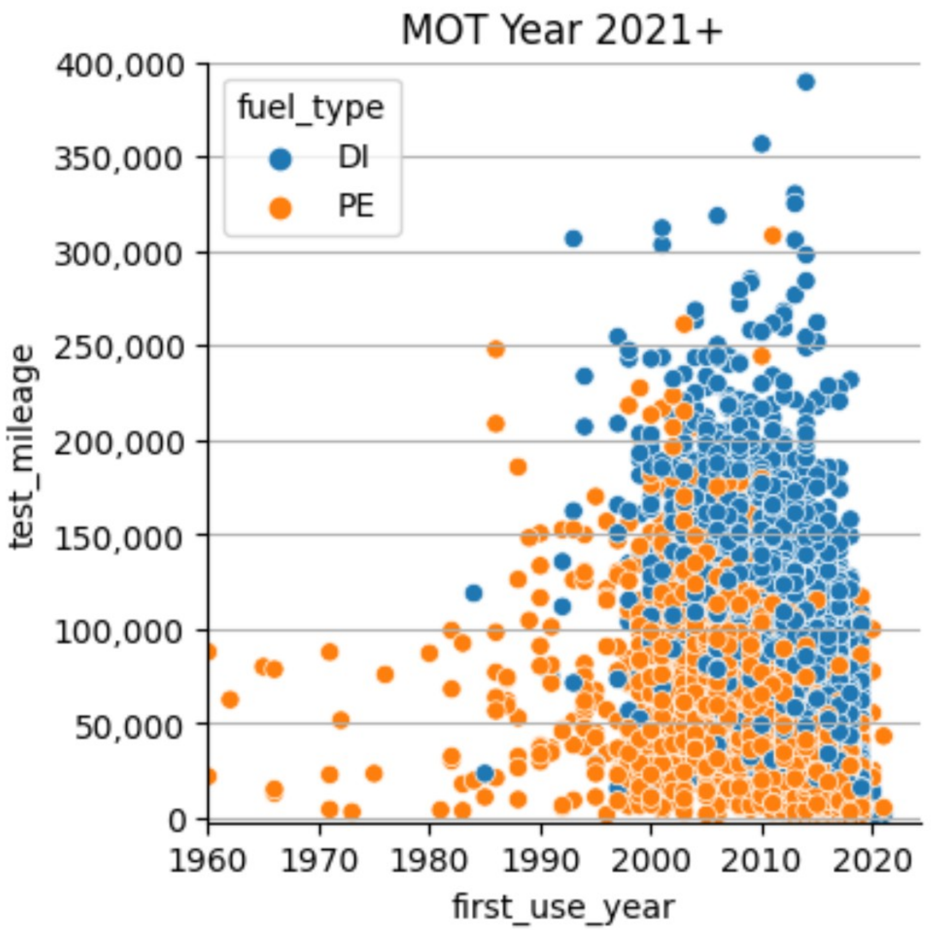
ARROW	`%timeit dfpda['make'].str.len()`
	`# e.g. TOYOTA, VOLKSWAGEN`
	`1.09 s ± 130 ms per loop (mean`
		`%timeit dfpda['test_mileage'].mean()`
		`60 ms ± 1.76 ms per loop (mean ± std`

NumPy	`%timeit dfpdn['make'].str.len()` `7.65 s ± 246 ms per loop (mean`	`%timeit dfpdn['test_mileage'].mean()` `140 ms ± 1.18 ms per loop (mean ± st`
	NumPy strings expensive in RAM, much cheaper in Arrow	NumExpr & bottleneck both installed Checks for identical results in notebook

Pandas+Arrow, query, Seaborn

```
%%time
df_fuel = (
    dfpda.query('test_result=="P"')
    .sample(10_000)
    .assign(first_use_year=lambda dfx: dfx["first_use_date"].dt.year)[
        ["test_mileage", "fuel_type", "first_use_year"]
    ]
    .dropna()
    .query("fuel_type in ['PE', 'DI']")
)
```

```
CPU times: user 8.94 s, sys: 5.11 s, total: 14 s
Wall time: 13.9 s
```



You can optimise **by hand** – mask, then choose columns to go faster

Polars – what's in it?

- Rust based, Python front-end
- Arrow (not NumPy) based
- Inherently multi-core and parallelised
- Beta out of core (medium-data) support
- Eager, also Lazy APIs + Query Planner

polars

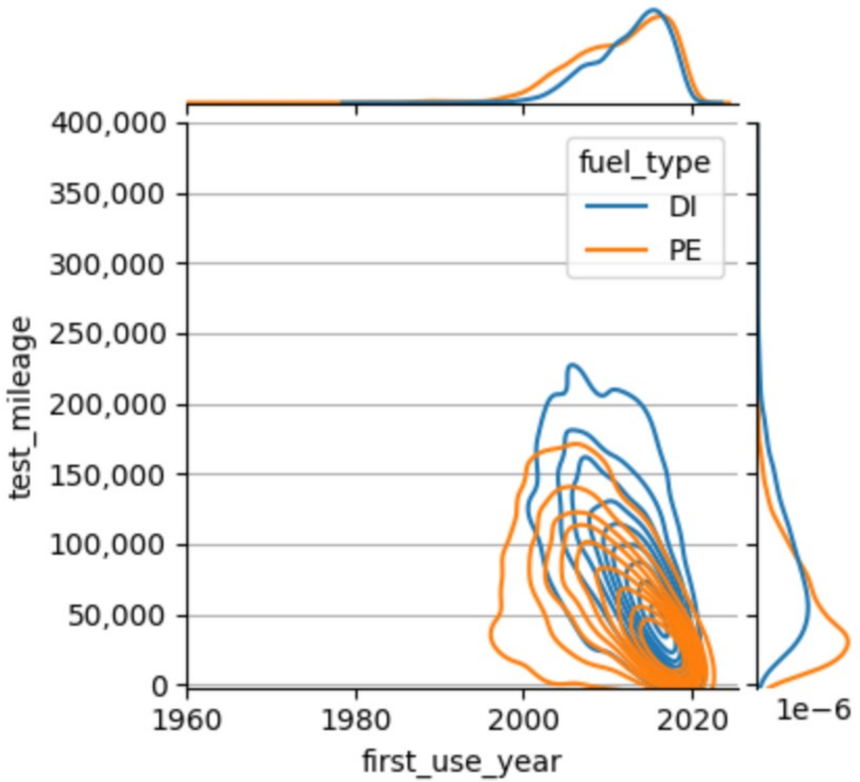
APACHE ARROW

Polars – same query & Seaborn

```
%%time
df_fuel = (
    dfple.filter(pl.col("test_result") == "P")
    .sample(10_000)
    .with_columns(pl.col("first_use_date").dt.year().alias("first_use_year"))
    .filter((pl.col("fuel_type").is_in({"PE", "DI"})))[
        ["test_mileage", "first_use_year", "fuel_type"]
    ]
)
```

```
CPU times: user 21.4 s, sys: 20.2 s, total: 41.6 s
Wall time: 5.59 s
```

(Lazy even faster)



A more advanced query

```
%%time
result = (
    dfpda.dropna(subset=["cylinder_capacity"])
    .groupby("make")["cylinder_capacity"]
    .agg(["median", "count"])
    .query("count > 10")
    .sort_values("median")
)
result
```

```
CPU times: user 12.4 s, sys: 6.33 s, total: 18
Wall time: 18.6 s
```

Pandas+NumPy takes 25s (i.e. slower)

Possibly we can further optimise this by hand (?)

```
%%time
result = (
    dfple.lazy()
    .filter(pl.col("cylinder_capacity").is_not_null())
    .groupby(by="make")
    .agg(
        [
            pl.col("cylinder_capacity").median().alias("median"),
            pl.col("cylinder_capacity").count().alias("count"),
        ]
    )
    .filter(pl.col("count") > 10)
    .sort(by="median")
    .collect()
)
```

```
CPU times: user 18 s, sys: 4.62 s, total: 22.6 s
Wall time 2.96 s
```

Polars eager (no “lazy() / collect()” call) takes 6s

First conclusions

- Pandas+Arrow *probably* faster than Pandas+NumPy
- Polars seems to be faster than Pandas+Arrow
- Maybe you can make Pandas “as fast”, but you have to experiment – Polars is “just fast”
- *All benchmarks are lies*

Buying a Volvo V50 – typical mileage?

```
dfpl_volvo = dfple.filter(
    (pl.col("make") == "VOLVO")
    & (pl.col("model") == "V50")
    & (pl.col("fuel_type") == "DI")
    & (pl.col("first_use_date").dt.year() == 2005)
    & (pl.col("test_date").dt.year() == 2022)
    & (pl.col("test_result") == "P")
)
```

```
# 80 is 80th percentile - we can use scipy on Arrow columns
percentile = scipy.stats.percentileofscore(
    dfpl_volvo["test_mileage"], 181_000, nan_policy="omit"
)
f"{percentile:0.0f}th percentile for mileage"
```

```
'80th percentile for mileage'
```

```
dfpl_volvo["test_mileage"].describe()
```

shape: (9, 2)

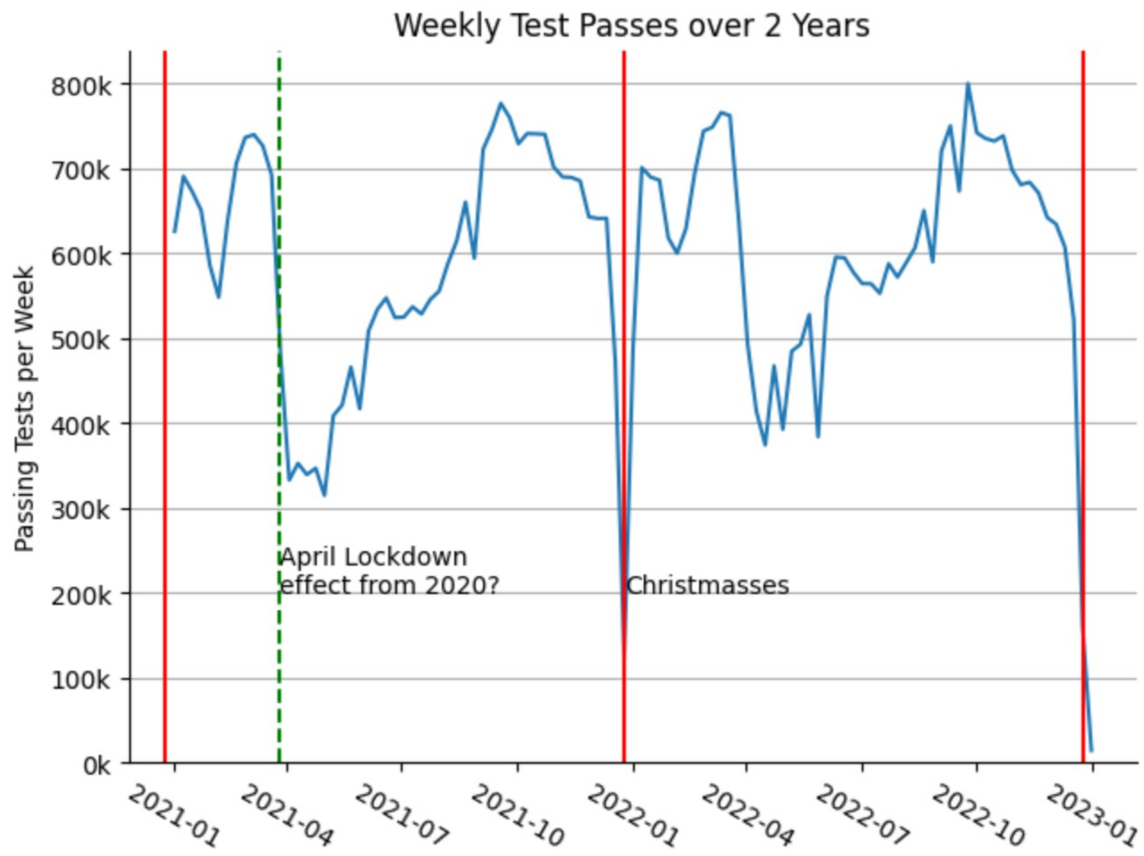
statistic	value
str	f64
"count"	1593.0
"null_count"	1.0
"mean"	150133.792714
"std"	39022.631791
"min"	27861.0
"max"	345116.0
"median"	150670.5
"25%"	125499.0
"75%"	174614.0

Volvo v50 lasts <24 hours

Resampling a timeseries

```
dfple = dfple.with_columns((pl.col("test_result") == pl.lit("P")). \
                                          alias("passed"))
result = (
     dfple.sort(pl.col("test_date"))
     .groupby_dynamic("test_date", every="1w")
     .agg(pl.col("passed").sum())
)
```

There's a **limit to how much we can instantiate into memory**, even if we're careful with sub-selection and dtypes



Scanning 640M rows of larger dataset

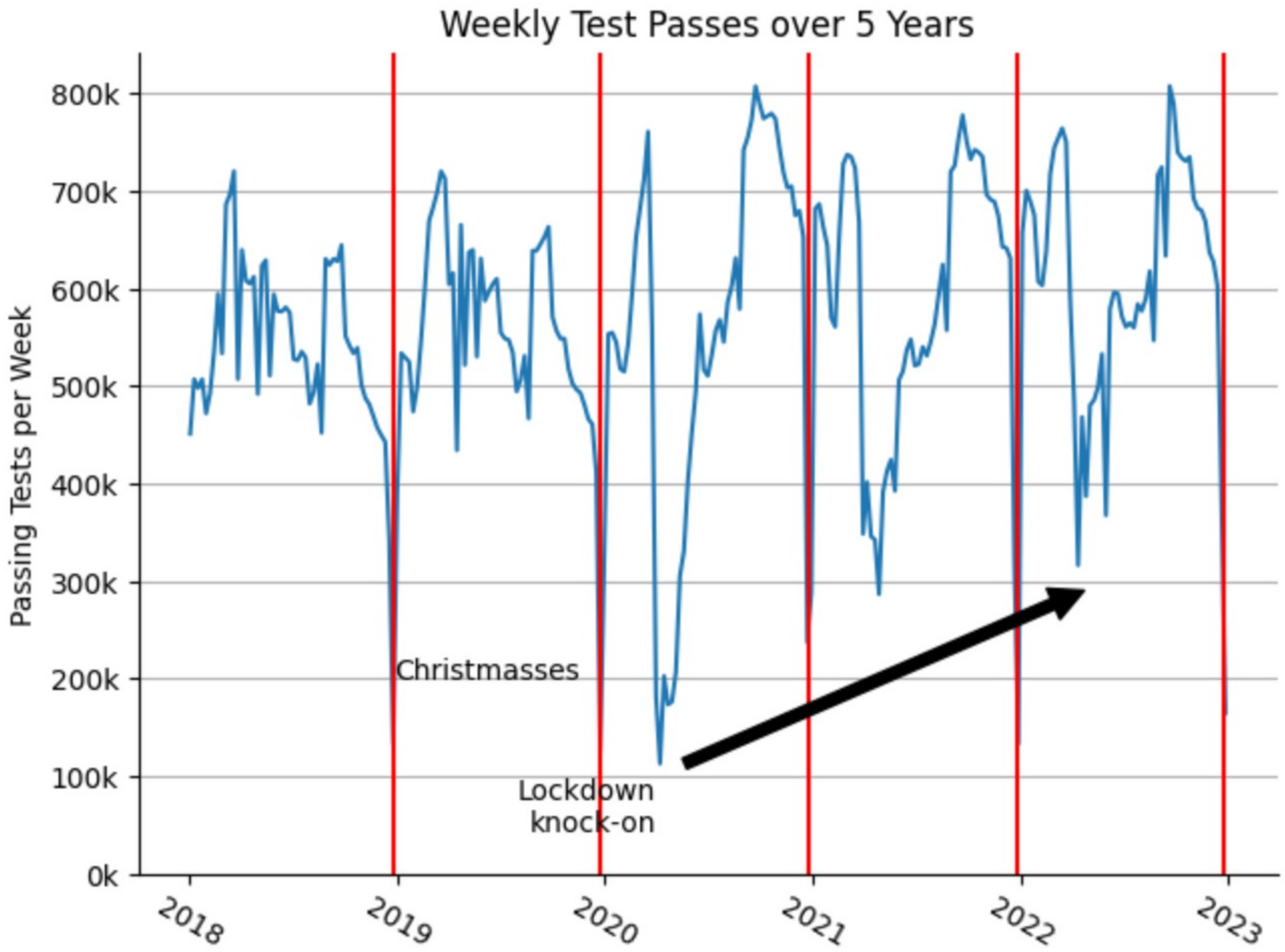
```
dfpll = pl.scan_parquet("../test_result.parquet/*.parquet")
dfpll.select(pl.count()).collect().item()
```

```
639506962
```

```
result_lz = (
    dfpll.filter(pl.col("test_date") > datetime.datetime(2018, 1, 1))
    .with_columns((pl.col("test_result") == pl.lit("P")).alias("passed"))
    .sort(pl.col("test_date"))
    .groupby_dynamic("test_date", every="1w")
    .agg(pl.col("passed").sum())
    .collect()
)
```

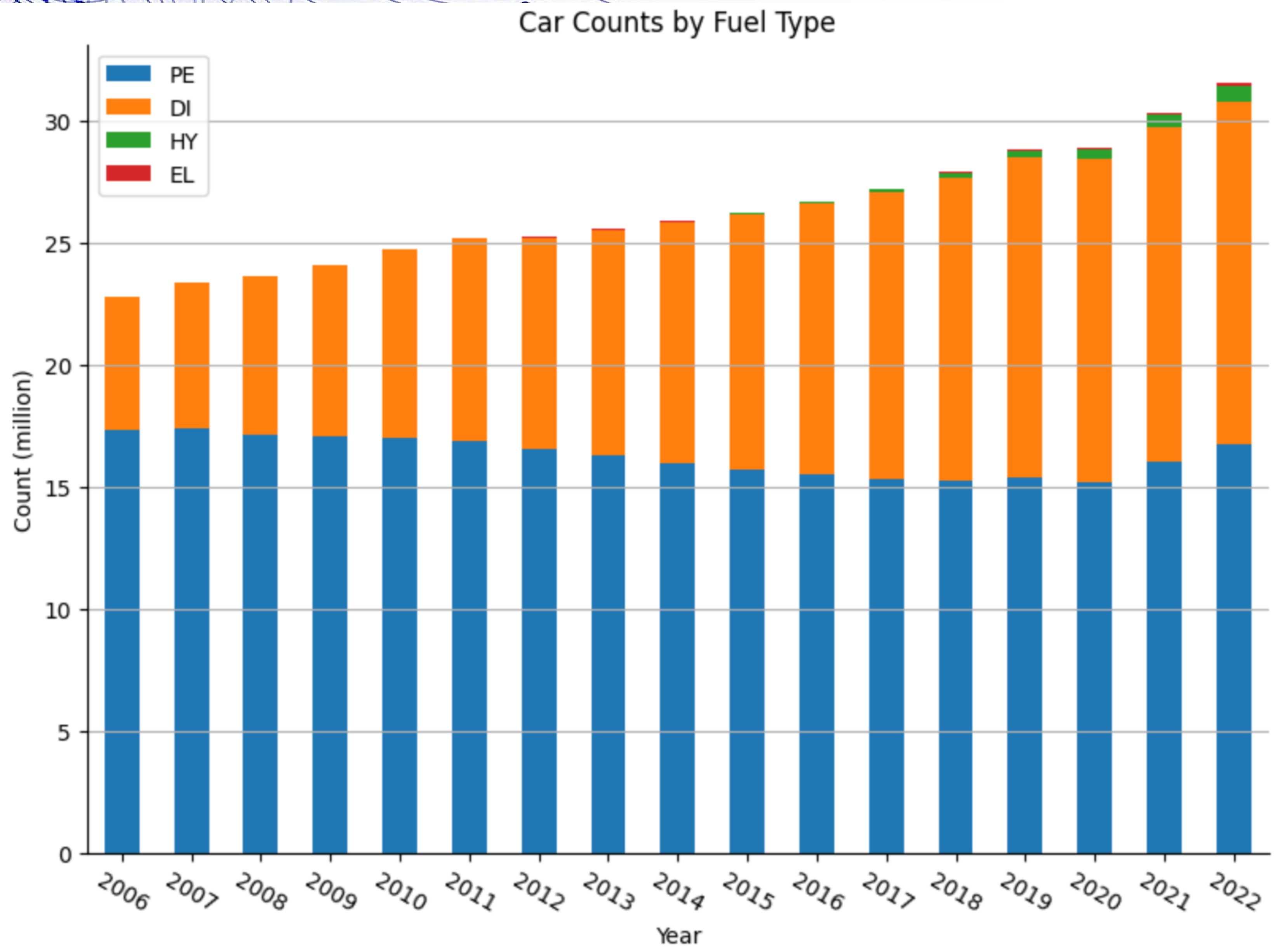
```
In [26] used -5.4 MiB RAM in 11.73s (system mean cpu 36%, single max cpu 100%)
```

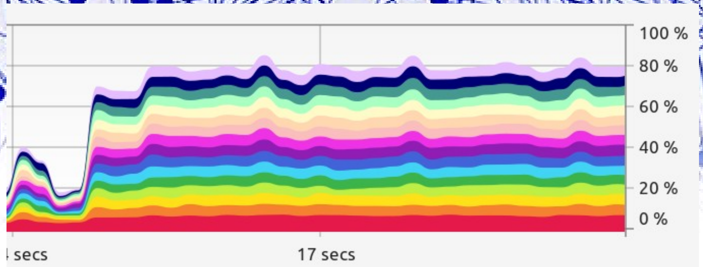
April drop was due to lockdown

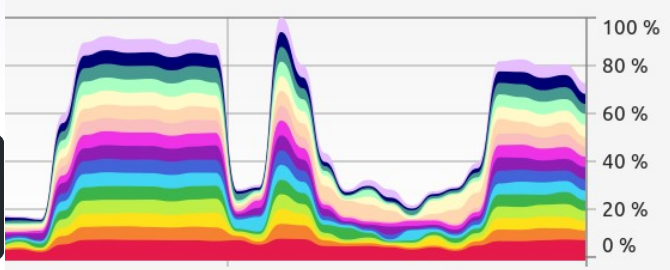


Count vehicles by major fuel types

We have to touch all parquet files, so we can't easily use Pandas








```
%%time
fuel_type_ddf = (
    dd.read_parquet(
        "../../test_result.parquet",
        dtype_backend="pyarrow",
        columns=["test_result", "test_date", "fuel_type"],
    )
    .query('test_result == "P"')
    .replace({"fuel_type": {"Hybrid Electric (Clean)": "HY",
                             "Electric": "EL"}})
    .assign(Year=lambda x: x.test_date.dt.year)
    .groupby(["Year", "fuel_type"])
    .agg({"test_result": "count"})
    .rename(columns={"test_result": "vehicle_count"})
    .compute()
)
```

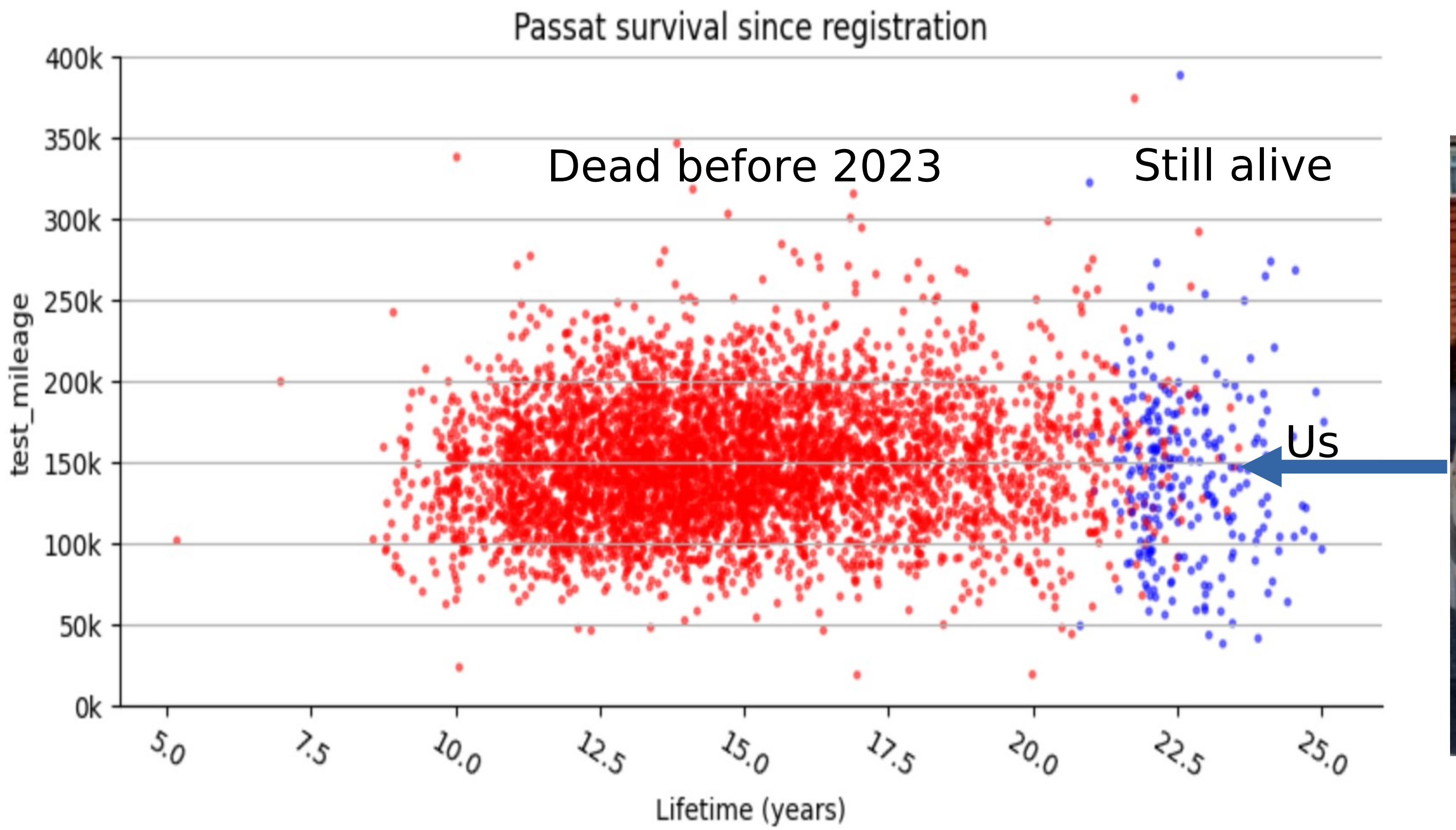
```
CPU times: user 6.77 s, sys: 1.26 s, total: 8.02 s
Wall time: 49.8 s
```

Adding columns=[...] saves 20s

```
%%time
fuel_type_edf = (
    pl.scan_parquet("../../test_result.parquet/*")
    .select(["test_result", "test_date", "fuel_type"])
    .filter(pl.col("test_result") == "P")
    .with_columns(
        pl.col("fuel_type")
        .map_dict(
            {"Hybrid Electric (Clean)": "HY", "Electric": "EL"},
            default=pl.first()
        )
        .cast(str),
        pl.col("test_date").dt.year().alias("Year"),
    )
    .groupby(["Year", "fuel_type"])
    .agg(pl.col("test_result").count().alias("vehicle_count"))
    .collect(streaming=True)
)
```

```
PARTITIONED DS
CPU times: user 3min 20s, sys: 49.8 s, total: 4min 10s
Wall time: 36 s
```

For the rally we bought a ‘99 Passat



dask

Giles had to sort the Parquet (6 mins) & change groupby agg shuffle, else performance much worse

```
%%time
vehicle_summary_ddf = (
    dd.read_parquet(
        path="test_result_sorted.parquet",
        dtype_backend="pyarrow",
        index="vehicle_id",
        calculate_divisions=True,
    )
    .query(
        ('make in ["VOLVO", "VOLKSWAGEN", "ROVER"] & '
        'model in ["V50", "PASSAT", "200", "200 VI"]')
    )
    .dropna()
    .pipe(vehicle_grouper, groupby_sort=True, agg_shuffle="p2p")
    .compute()
)
```

```
CPU times: user 8.52 s, sys: 2.43 s, total: 11 s
Wall time: 1min 7s
```

3min+ with default 4 workers (*4 threads)
1min with 12 works (*1 thread) – hand tuned

polars

```
%%time
ldf = (
    pl.scan_parquet("../../test_result.parquet/*")
    .filter(pl.col("make").is_in(["VOLVO", "ROVER", "VOLKSWAGEN"]))
    .filter(pl.col("model").is_in(["V50", "200", "PASSAT"]))
    .groupby("vehicle_id")
    .agg(
        pl.col(
            "make", "model", "fuel_type", "cylinder_capacity", "first_use_date"
        ).last(),
        pl.col("test_date").max().alias("last_test_date"),
        pl.col("test_mileage").max().alias("last_known_mileage"),
    )
    .collect(streaming=True)
)
```

```
PARTITIONED DS: estimated cardinality: 0.9952009 exceeded the boundary: 0.4, run
CPU times: user 2min 20s, sys: 38.6 s, total: 2min 58s
Wall time: 16.8 s
```

Issues encountered

Surprisingly high Swap usage with `read_parquet` - much higher than eventual RAM usage bug python

#8925 opened 2 weeks ago by ianozsvald 2 tasks done

OOM errors with scan_parquet > limit > collect bug python

#9001 by gwvr was closed last week 2 tasks done

polars

Support `value_counts` on `LazyFrame`? enhancement

#8933 opened 2 weeks ago by ianozsvald

`read_csv` with `include_path_column` and `dtype_backend='pyarrow'` generates a mix of String and Categorical which hurts Parquet usage dataframe enhancement

#10302 opened 2 weeks ago by ianozsvald

dask

Thoughts on our testing

- Haven't checked to_numpy(), Numba, apply, rolling, writing partitioned Parquet (Polars)
- NaN / Missing behaviour different Polars/Pandas
- sklearn partial support (sklearn assumes Pandas API) – but maybe Pandas+Arrow has copy issues too?
- Arrow Polars/Dask/Pandas timeseries vs NumPy?

Medium-data conclusions

- Dask ddf and Polars can perform similarly
- Dask learning curve harder, especially for performance
- Dask does a lot more (e.g. Bag, ML, NumPy, clusters, diagnostics)
- Polars easy to start with, easy for medium data

Summary

Beaumont

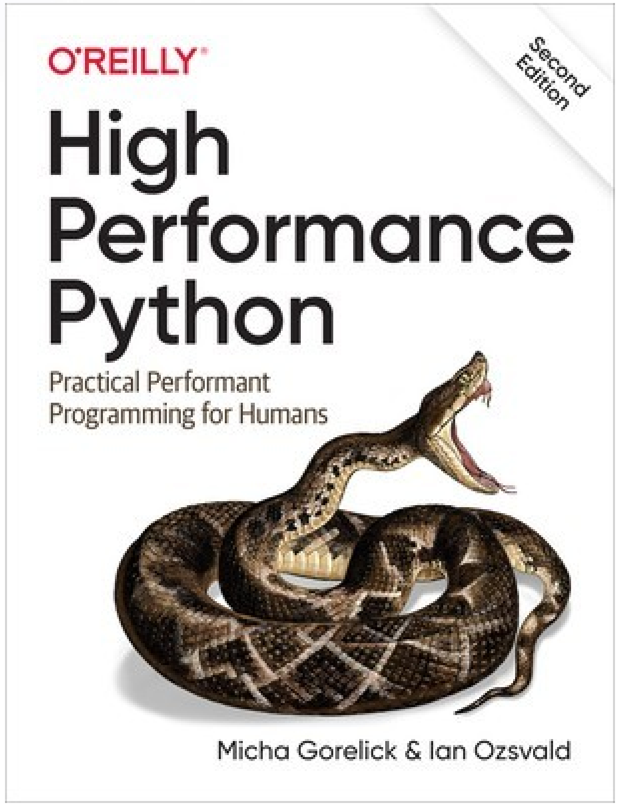
11:00 AM 40min — Executives at PyData — Ian Ozsvald

- Experiment, we have options!

NotANumber.email

A Pythonic Data Science Newsletter

- **I love receiving postcards (email me)**
- Sponsor Ian (JustGiving)
- *Join after for our discussion*



Robert Hansen

Rob, Edd and Ian's fundraiser for Parkinson's UK

Appendix

Manual Query Planning

```
%time dfpda.query('test_result=="P"');
```

```
CPU times: user 6.63 s, sys: 4.31 s, to
Wall time: 10.9 s
In [23] used 0.0 MiB RAM in 11.13s (sys
ingle max cpu 100%), peaked 9534.4 MiB
```

```
%%time
pass_mask = dfpda["test_result"] == "P";
# make mask only
```

```
CPU times: user 391 ms, sys: 4.77 ms, to
Wall time: 394 ms
In [47] used 0.0 MiB RAM in 0.49s (syste
ngle max cpu 100%), peaked 19.3 MiB abov
```

```
%%time
cols = ["test_mileage", "fuel_type",
        "first_use_date"]
dfpda[pass_mask][cols];
# select all columns after mask
```

```
CPU times: user 5.88 s, sys: 4.29 s, t
Wall time: 10.1 s
In [26] used 0.0 MiB RAM in 10.27s (sy
ingle max cpu 100%), peaked 9178.8 MiB
```

```
%time dfpda[cols][pass_mask];
# select all columns before mask
```

```
CPU times: user 1.48 s, sys: 870 ms,
Wall time: 2.34 s
In [27] used 0.0 MiB RAM in 2.48s (sy
ngle max cpu 100%), peaked 2498.2 MiB
```