

Pandas 2, Polars or Dask?

ODSC 2023 Talk

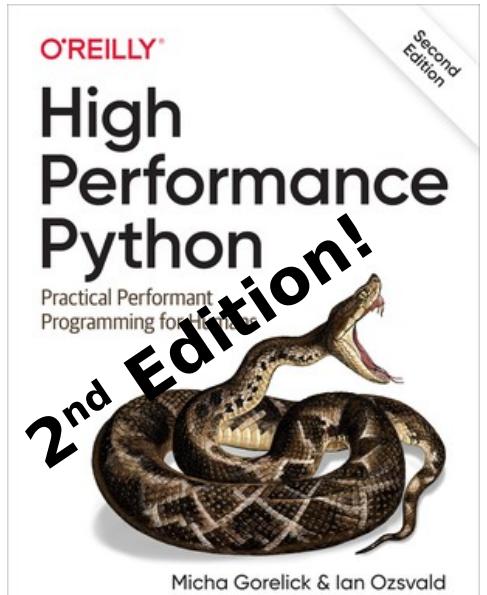


@IanOzsvald – ianozsvald.com

@GilesWeaver

We are Ian Ozsvald & Giles Weaver

- Interim Chief Data Scientist



Part of **PyData** – 219 groups [?](#)

PyData London Meetup

★★★★★ (1886) [?](#)

London, United Kingdom

12,346 members Public group [?](#)

Organized by NumFOCUS, Inc. and 21 others

- Data Scientist





3 interesting DataFrame libraries

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



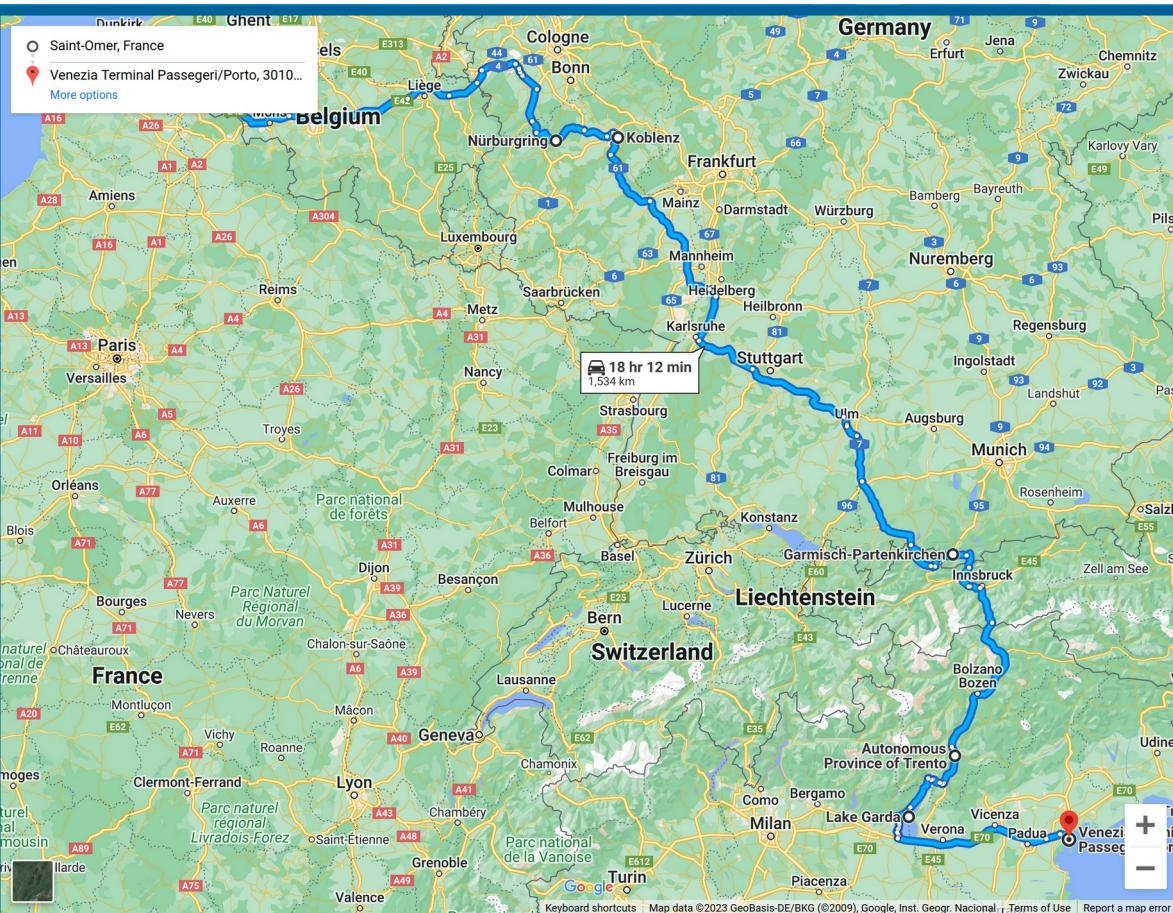
Motoscape Charity Rally

- Ian - “Let’s do something silly”
- September 2023 (4 mo)
- 2,000 mile round trip <£1k car
- Ideally it shouldn’t explode

<https://bit.ly/JustGivinglan>

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

 MOTOSCAPE BANGER RALLY 2023, 2 September 2023



Car Test Data (UK DVLA)

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



[Check another vehicle](#)

Colour
Blue

Fuel type
Diesel

Date registered
10 March 2005

MOT valid until
17 April 2024

Date tested
12 May 2022

PASS

Mileage
171,443 miles

Test location
[View test location](#)

MOT test number
9400 3587 8901

Expiry date
8 June 2023

| 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 |



Pandas 2 – what's new?

- Pandas 15 years old, NumPy based
- PyArrow first class alongside NumPy
- Internal clean-ups so less RAM used
- Copy on Write (off by default)



PyArrow vs NumPy – which to use?

Backend



String dtype

```
%timeit dfpda['make'].str.len()  
# e.g. TOYOTA, VOLKSWAGEN
```

1.09 s ± 130 ms per loop (mean ± std)



```
%timeit dfpdn['make'].str.len()
```

7.65 s ± 246 ms per loop (mean ± std)

NumPy strings expensive in RAM e.g.
82M rows 39GB NumPy, 11GB Arrow

Nullable integer dtype

```
%timeit dfpda['test_mileage'].mean()
```

60 ms ± 1.76 ms per loop (mean ± std)

```
%timeit dfpdn['test_mileage'].mean()
```

140 ms ± 1.18 ms per loop (mean ± std)

NumExpr & bottleneck both installed
Checks for identical results in notebook

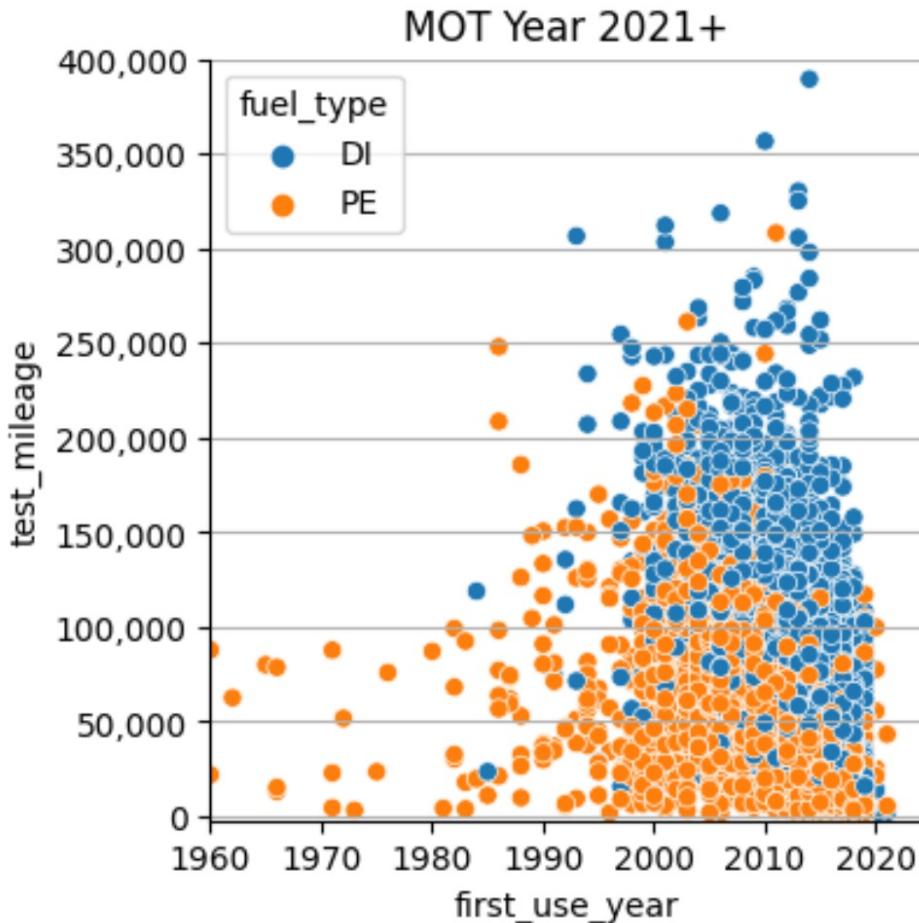
Pandas+Arrow, query, Seaborn

```
%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, 3 years old
- Arrow (not NumPy)
- Inherently multi-core and parallelised
- Eager and Lazy API (+Query Planner)
- Beta out-of-core (medium data) support



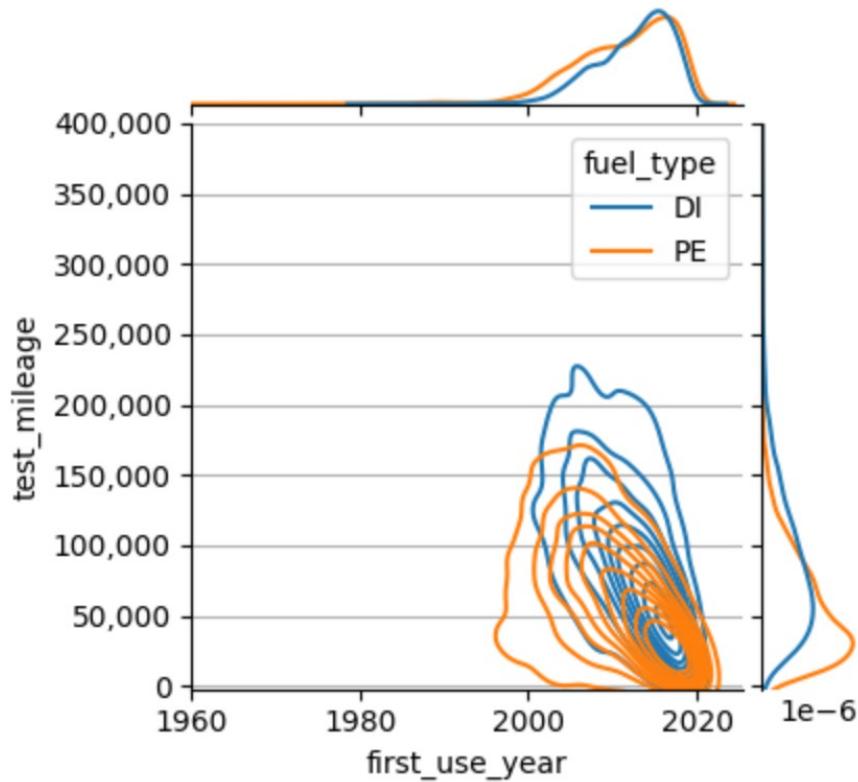
Polars – same query & Seaborn

```
%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 df is even faster)



Manual Query Planning

```
%time dfpda.query('test_result=="P"');
CPU times: user 6.63 s, sys: 4.31 s, t...
Wall time: 10.9 s
In [23] used 0.0 MiB RAM in 11.13s (sys...
single max cpu 100%), peaked 9534.4 MiB
%%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 (sys...
single max cpu 100%), peaked 9178.8 MiB
```

```
%%time
pass_mask = dfpda["test_result"] == "P";
# make mask only
CPU times: user 391 ms, sys: 4.77 ms, t...
Wall time: 394 ms
In [47] used 0.0 MiB RAM in 0.49s (sys...
single max cpu 100%), peaked 19.3 MiB above
```

```
%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 (sys...
single max cpu 100%), peaked 2498.2 MiB
```

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 – your mileage will vary*

Should I buy Volvo V50 – 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:.0f}}th percentile for mileage"  
  
'80th percentile for mileage'
```

| 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

<https://bit.ly/JustGivinglan>



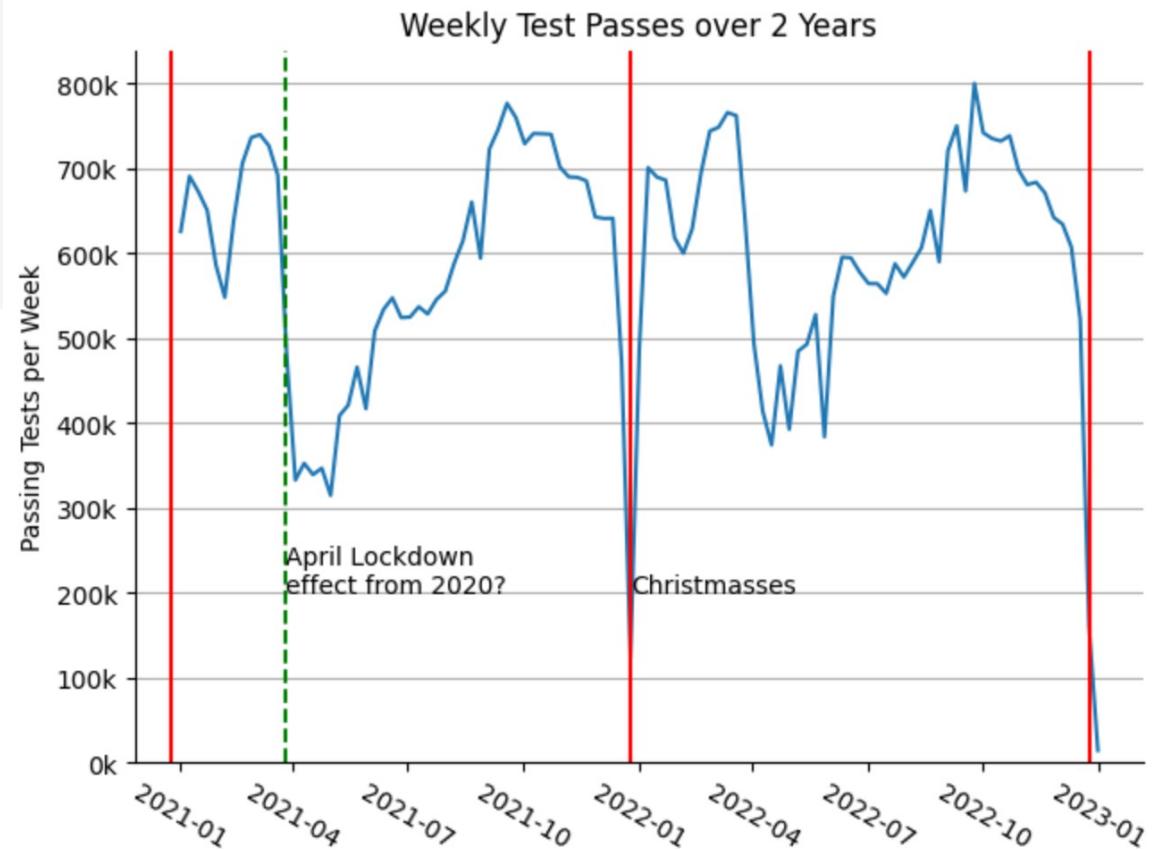
Resampling a timeseries



```
dfple = pl.read_parquet("../test_result_2021on.parquet")  
  
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())  
)
```

This dataset is in-RAM (2021-2022)

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

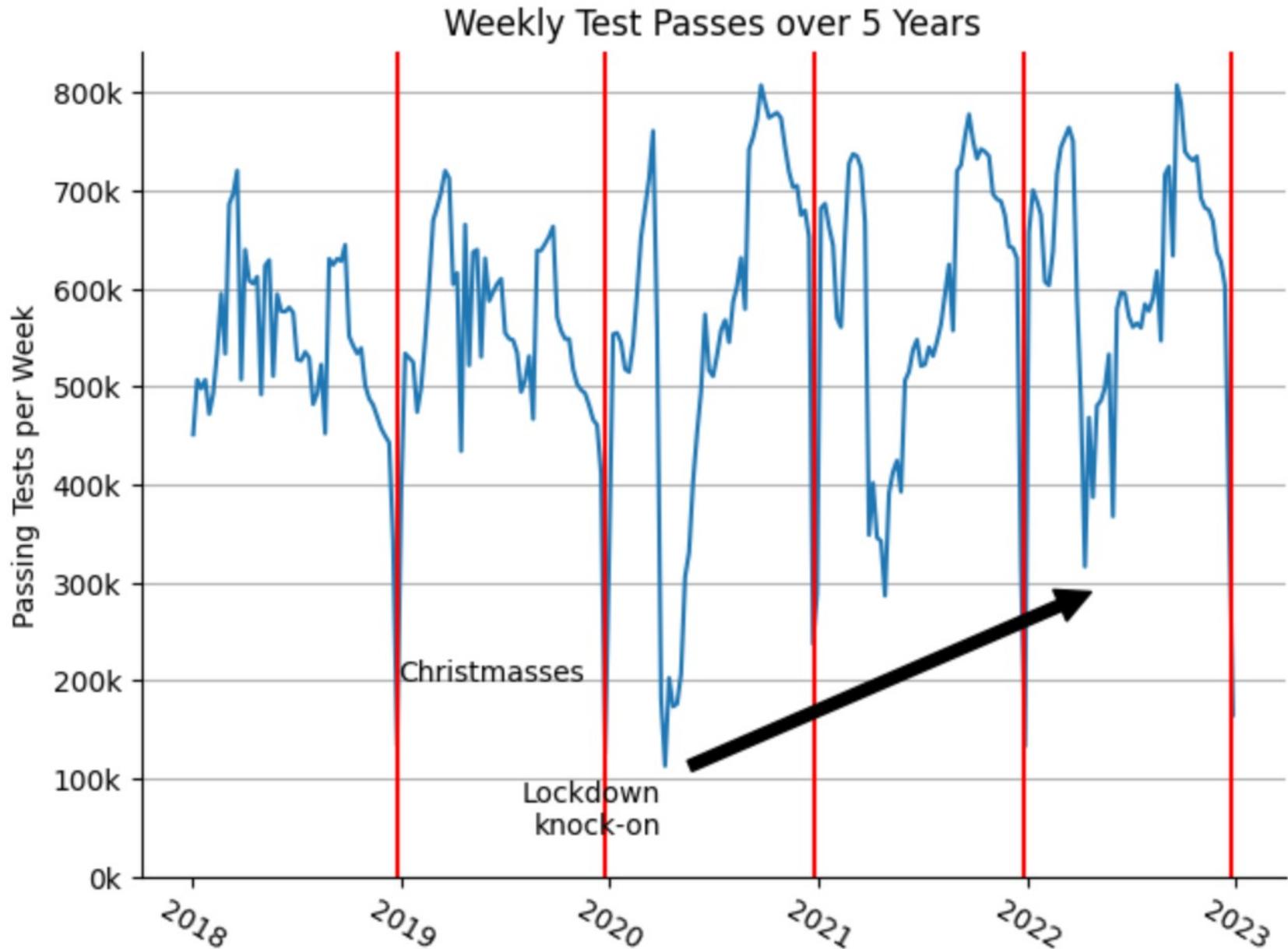
```
dfpll = pl.scan_parquet("../test_result.parquet/*.parquet") Implicit Lazy DataFrame
```

```
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()
)
```

11 seconds, 640M rows, circa 850 partitions (files)

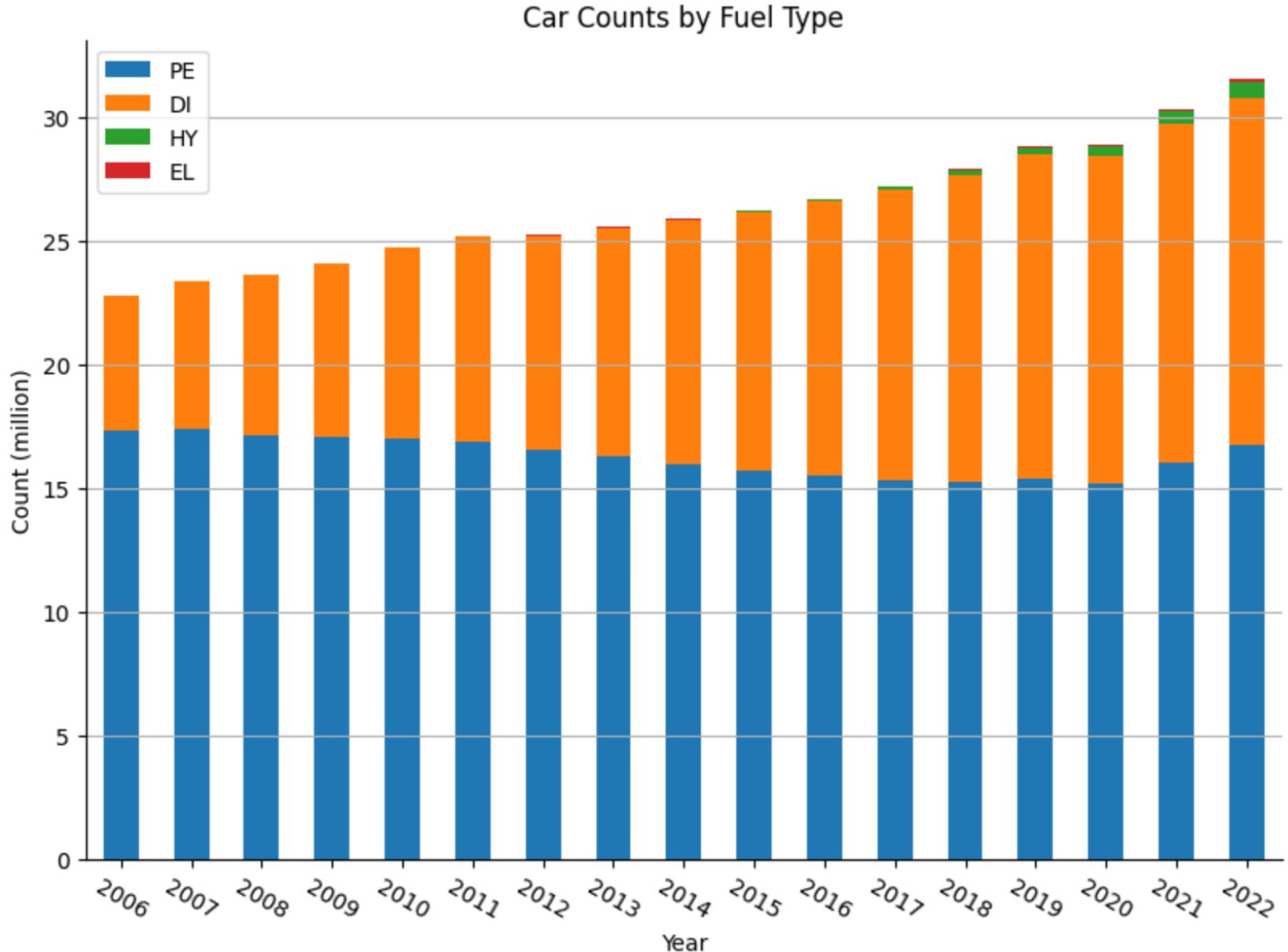


April drop
was due to
lockdown

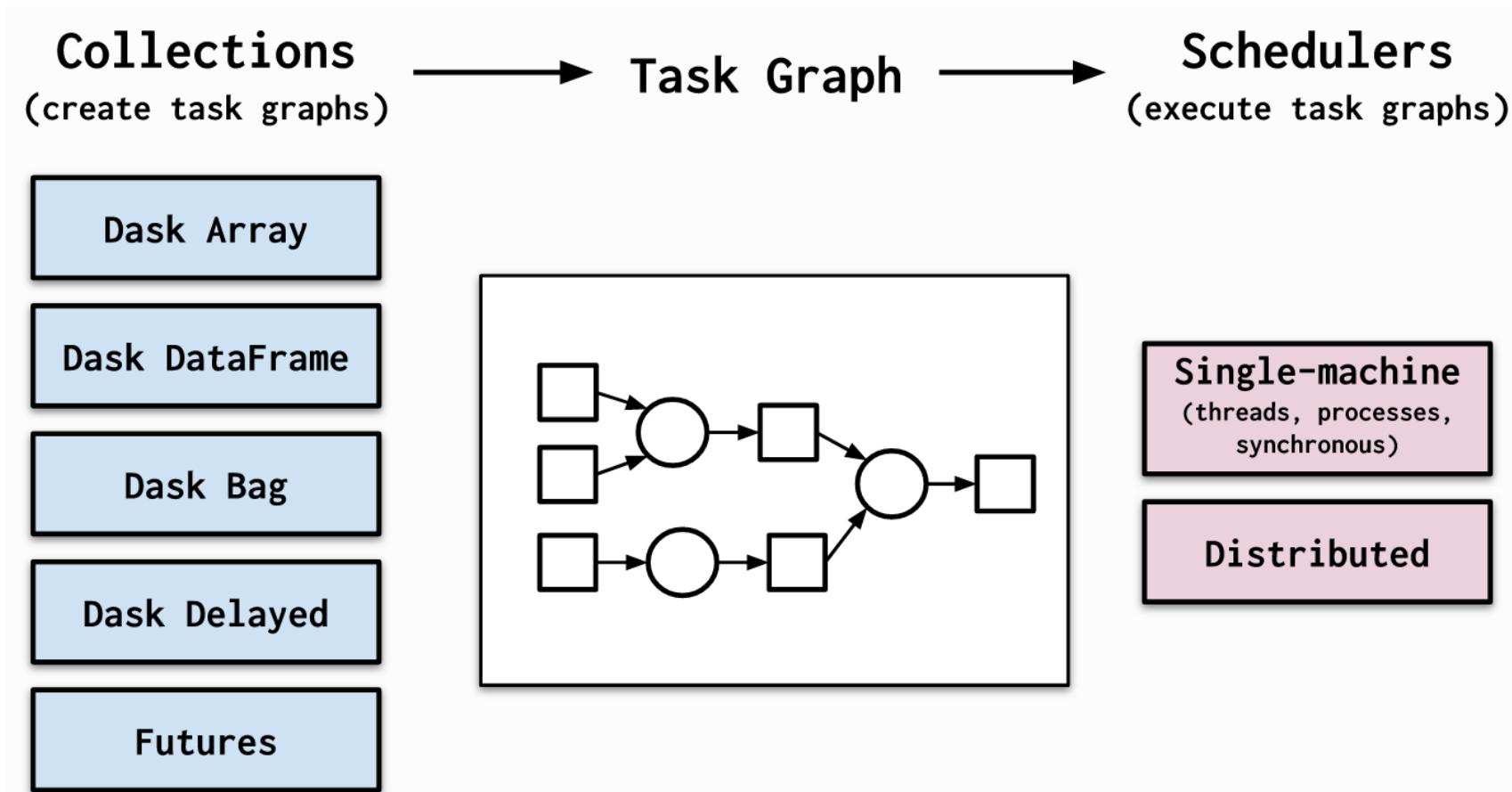


Vehicle ownership increases, Hybrids growing

We have to touch all parquet files, so we can't easily use Pandas
MOT after 3 years of age for all vehicles



Dask scales Pandas (and lots more)



| Dask | |
|--------------------|---|
| | |
| Original author(s) | Matthew Rocklin |
| Developer(s) | Dask |
| Initial release | January 8, 2015; 8 years ago ↗ |
| Stable release | 2022.05.02 / May 2, 2022; 12 months ago |



dask



polars

```
%%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

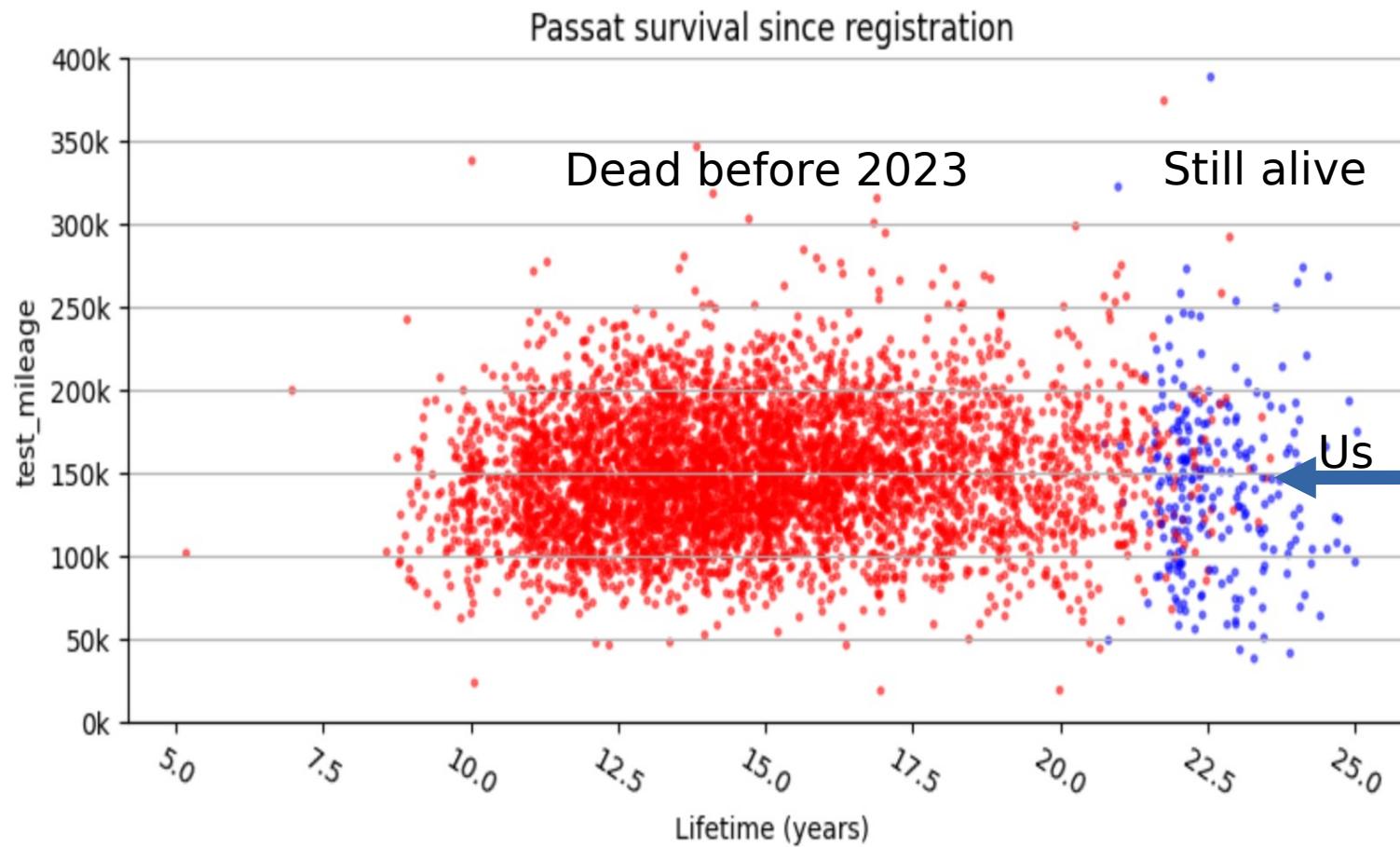
```
%%time
fuel_type_edf = (
    pl.scan_parquet("../test_result.parquet/*")
    .select(["test_result", "test_date", "fuel_type"])
    .filter(pl.col("test_result") == "P")
    .with_columns(
        pl.col("fuel_type")
        .map_dict(
            {"Hybrid Electric (Clean)": "HY", "Electric": "EL"},
            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



<https://bit.ly/JustGivinglan>



Giles had to push directives to
the Arrow read, set shuffle on
set_index and agg

```
%%time
vehicle_summary_df = (
    dd.read_parquet(
        path="test_result.parquet",
        dtype_backend="pyarrow",
        columns=['vehicle_id', 'make', 'model', 'fuel_type',
                 'cylinder_capacity', 'first_use_date',
                 'test_date', 'test_mileage'],
        filters=[
            ("make", "in", ["VOLVO", "VOLKSWAGEN"]),
            ("model", "in", ["V50", "PASSAT"]),
        ],
    ).set_index('vehicle_id', npartitions=96, shuffle='tasks')
    .groupby("vehicle_id")
    .agg({"make": "last", "model": "last", "fuel_type": "last",
          "cylinder_capacity": "last", "first_use_date": "last",
          "test_date": "max", "test_mileage": "max"}, shuffle='tasks')
    .compute()
)

CPU times: user 8.46 s, sys: 1.5 s, total: 9.96 s
Wall time: 1min 6s
```

3min+ with default 4 workers (*4 threads)
1min with 12 workers (*1 thr.) hand tuned



```
%%time
edf = (
    pl.scan_parquet("test_result.parquet/*")
    .filter(pl.col("make").is_in(["VOLVO", "VOLKSWAGEN"]))
    .filter(pl.col("model").is_in(["V50", "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()
)
```

PARTITIONED DS: estimated cardinality: 0.9911628 exceeded the boundary: 0.4, run

```
CPU times: user 1min 51s, sys: 12.9 s, total: 2min 4s
Wall time 11.7 s
```

Issues encountered

- Surprisingly high Swap usage with `read_parquet` - much higher than eventual RAM usage bug python
#8925 opened 2 weeks ago by ianozsvard 2 tasks done
- **OOM errors with `scan_parquet > limit > collect`** bug python
#9001 by gwvr was closed last week 2 tasks done
- Support `value_counts` on `LazyFrame`? enhancement
#8933 opened 2 weeks ago by ianozsvard
- `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 ianozsvard





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 timeseries/str **different** to Pandas NumPy?



Arrow RAM usage great!

```
dfpda = pd.read_parquet("../test_result_2021on.parquet",  
                        dtype_backend="pyarrow")
```

```
show_rows_memory(dfpda)
```

```
'Pandas df with 82,012,245 rows, 11.19 GB bytes'
```



Polars is similar
as it uses Arrow

```
dfpdn = pd.read_parquet("../test_result_2021on.parquet",  
                        dtype_backend="numpy_nullable")
```

```
show_rows_memory(dfpdn, deep=True)
```

```
'Pandas df with 82,012,245 rows, 39.66 GB bytes'
```





Pandas vs Polars conclusions

- Polars easy to use, Pandas we all know
- **Arrow in both is great** (fast+low RAM footprint)
- Differences in Polars API (day of week starts at 1 not 0,
no `sample` on LazyDF, different verb names)
- Clear Polars API design **makes thinking easier**



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)

Summary

- Experiment, we have options!

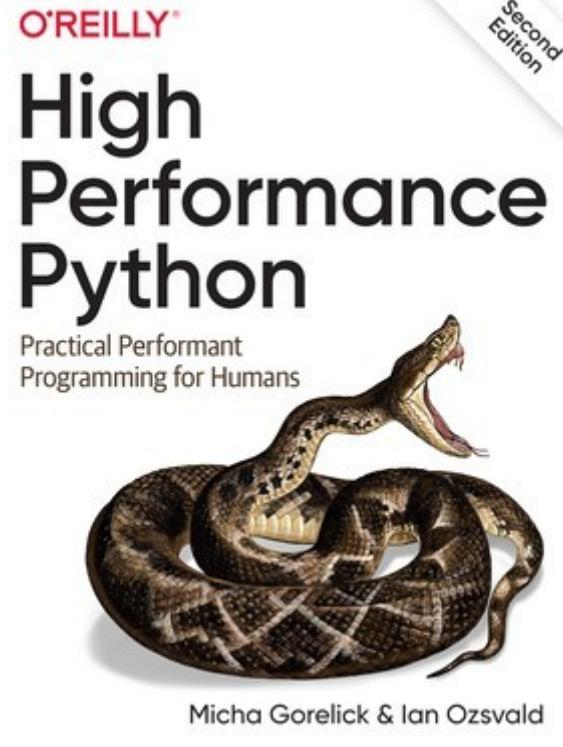
NotANumber.email 

A Pythionic Data Science Newsletter

- I love receiving postcards (email me)
- Follow our journey->
- *I'm happy to discuss after*

Robert Hansen

<https://bit.ly/JustGivinglan>
Rob, Edd and Ian's fundraiser for
Parkinson's UK





Appendix



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

```
%%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