

Pandas 2, Polars or Dask? An update from June

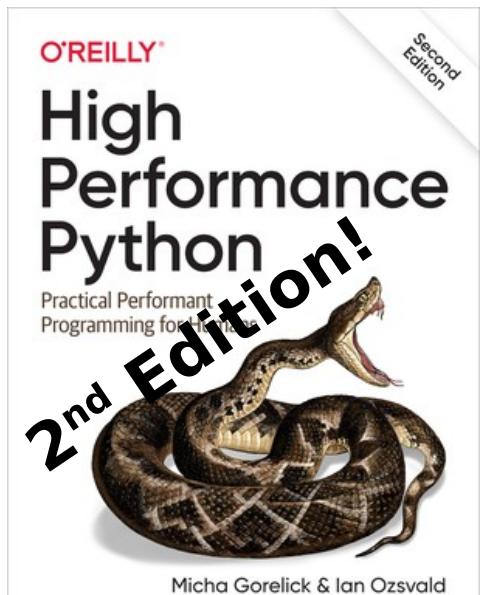
PyDataGlobal 2023 Talk

@ianOzsvald – ianozsvald.com

@GilesWeaver

We are Ian Ozsvald & Giles Weaver

- Interim Chief Data Scientist



Part of **PyData - 227 groups** ⓘ

PyData London Meetup

📍 London, 17, United Kingdom

👤 13,447 members Public group ⓘ

👤 Organized by NumFOCUS, Inc. and 15 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?**
- **Using Polars over 7 months**
- We benchmark. All benchmarks are lies



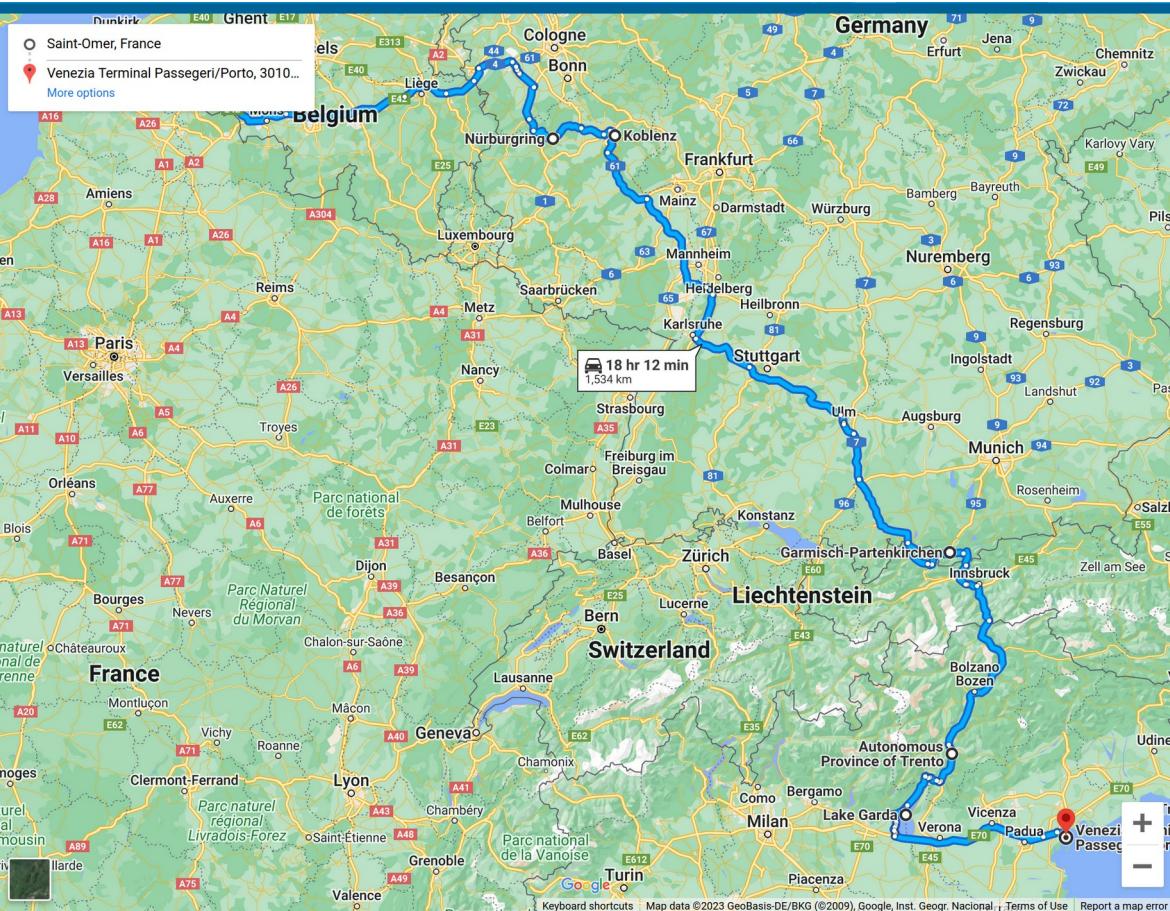
Motoscape Charity Rally

- 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

CN05 HJC
VOLVO V50

[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), faster & cheaper with it on



PyArrow vs NumPy – which to use?

Backend



String dtype

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

1.08 s ± 5.19 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)



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

8.18 s ± 44.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

GroupBy Arrow can be slower

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

✓ 20.0s

2.5 s ± 44.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

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

✓ 9.8s

1.23 s ± 12.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

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

NumExpr & bottleneck both installed
 Checks for identical results in notebook

```
pip install ipython-memory-usage
```



Default Copy on Write == False

```
df2 = dfpdn.rename(columns={'make': 'car_make'}). \  
    assign(is_petrol = dfpdn['fuel_type'] == 'PE'). \  
    drop(columns=['fuel_type'])
```

✓ 17.5s

```
In [4] used 8604.7 MiB RAM in 17.57s (system mean cpu 13%, single max cpu 100%)  
peaked 9657.1 MiB above final usage
```

```
assert df2.loc[0, 'test_mileage'] == 227219 # original value  
df2.loc[0, 'test_mileage'] = 99 # this column (only) copied  
assert df2.loc[0, 'test_mileage'] == 99  
assert dfpdn.loc[0, 'test_mileage'] == 227219 # original unchanged
```

✓ 0.2s

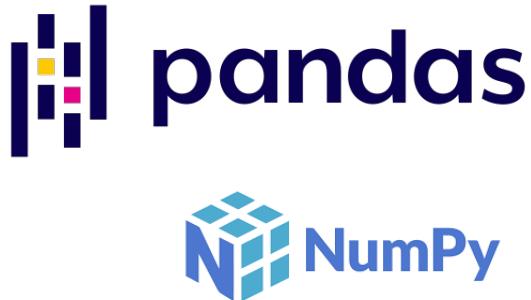
```
In [5] used 625.6 MiB RAM in 0.25s (system mean cpu 16%, single max cpu 100%)
```



3 defensive copies
Notably worse on NumPy than Arrow
17GB envelope over 17s

Not sure why +600MB here...

```
pd.options.mode.copy_on_write = True
```



Pandas Copy on Write == True

Common operations no longer trigger defensive copies

Copies made *when needed*

Code *may execute faster*

Less RAM may be used

```
df2 = dfpdn.rename(columns={'make': 'car_make'}). \  
    |           | assign(is_petrol = dfpdn['fuel_type'] == 'PE'). \  
    |           | drop(columns=['fuel_type'])
```

✓ 3.3s

```
In [4] used 79.5 MiB RAM in 3.35s (system mean cpu 0%, single max cpu 0%), peak
```

```
assert df2.loc[0, 'test_mileage'] == 227219 # original value  
df2.loc[0, 'test_mileage'] = 99 # this column (only) copied  
assert df2.loc[0, 'test_mileage'] == 99  
assert dfpdn.loc[0, 'test_mileage'] == 227219 # original unchanged
```

✓ 0.4s

```
In [5] used 1329.6 MiB RAM in 0.48s (system mean cpu 29%, single max cpu 100%)
```

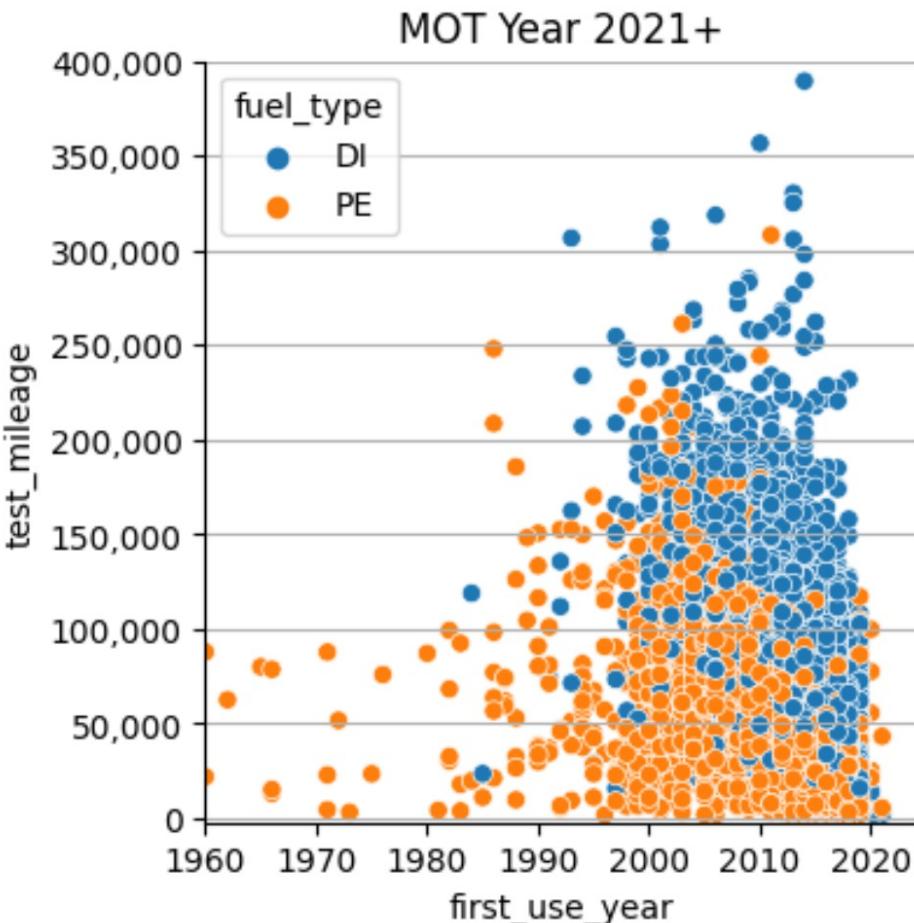
Pandas+Arrow, query, Seaborn

```
%%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']")
)
✓ 20.1s
```

CPU times: user 13.4 s, sys: 7.03 s, total: 20.4 s

Wall time: 20.1 s

In 2023-06 this took 14s, so now it is slower





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



```
pip install ipython-memory-usage
```

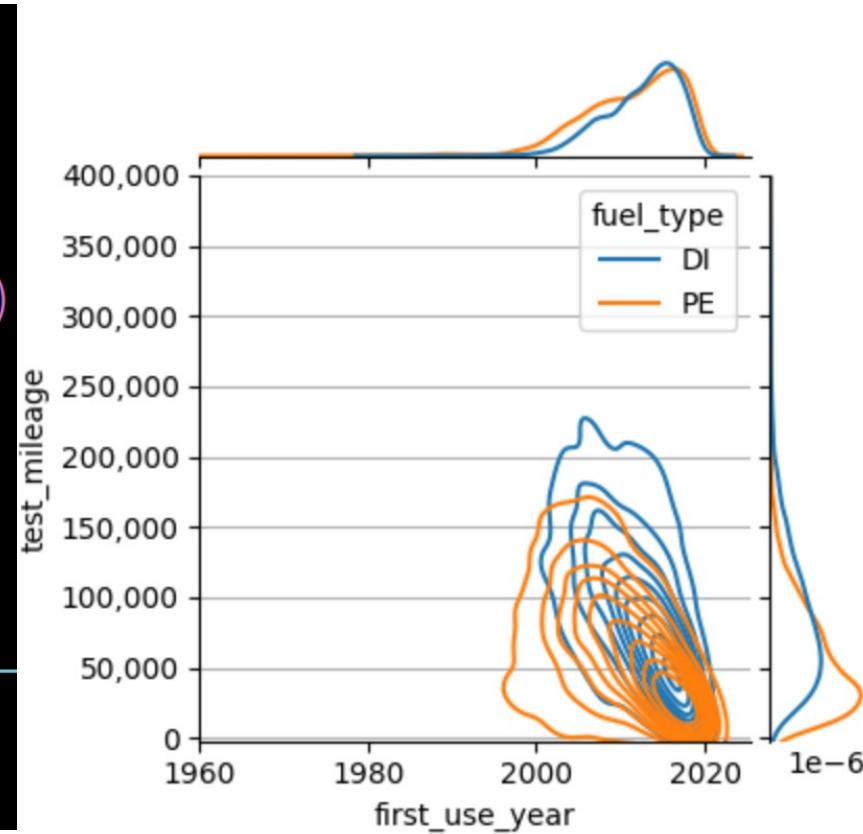


Polars – same query & Seaborn

```
%%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 10.3 s, sys: 5.76 s, total: 16 s

Wall time: **2.69 s**



50% faster than in 2023-06, the Lazy DataFrame can be even faster

By [ian]@ianozsvald[.com] and giles@wvr-consulting.co.uk @gilesweaver

Ian Ozsvald

A more advanced query

```
%%time
result = (
    dfpda.dropna(subset=["cylinder_capacity"])
    .groupby("make")["cylinder_capacity"]
    .agg(["median", "count"])
    .query("count > 10")
    .sort_values("median")
)
#"RuntimeWarning: Engine has switched to 'python'
# because numexpr does not support extension
# array dtypes"
result
```

CPU times: user 14.8 s, sys: 12.6 s, total: 27.3 s
 Wall time **27.4 s**

Pandas+NumPy takes 13s using numexpr

50% slower than in 2023-06
 for Pandas+Arrow, the
 numexpr warning is new



```
%%time
result = (
    dfple.lazy()
    .filter(pl.col("cylinder_capacity").is_not_null())
    .group_by(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 25.3 s, sys: 2.69 s, total: 28 s
 Wall time: **3.48 s**

Polars eager (no “lazy() / collect()” call) takes
 4.5s



First conclusions

- Pandas+Arrow *maybe better than* Pandas+NumPy
- Polars seems to be faster than Pandas+Arrow
- Pandas Copy on Write seems like a nice optimisation
- *All benchmarks are lies – your mileage will vary*

Volvo v50 lasts <24 hours

<https://bit.ly/JustGivinglan>



Resampling a timeseries

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

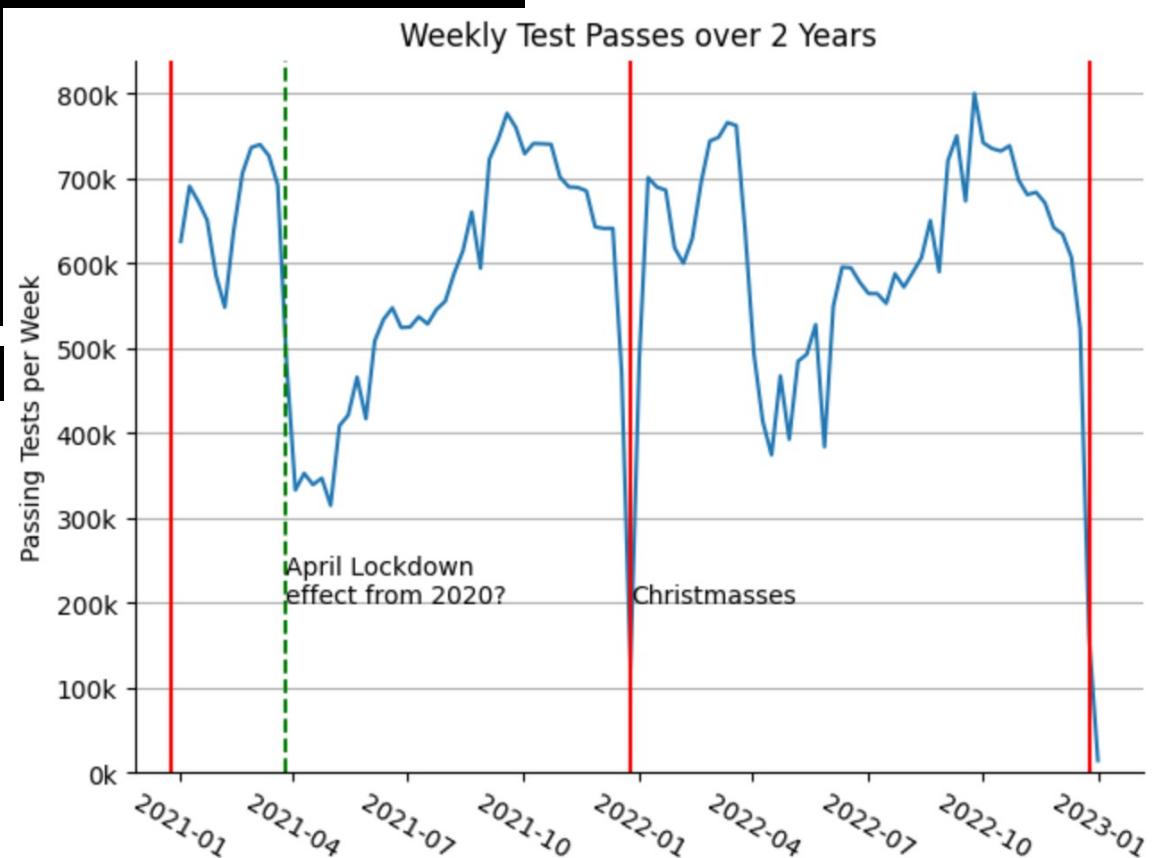


```
dfple = dfple.with_columns(pl.col("test_result") == pl.lit("P")). \
    |    |    |    |    alias("passed"))
result = (
    dfple.sort(pl.col("test_date"))
    .group_by_dynamic("test_date", every="1w")
    .agg(pl.col("passed").sum())
)
```

used 60.4 MiB RAM in 6.76s (system mean cpu 55%, single max cpu 100%)

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

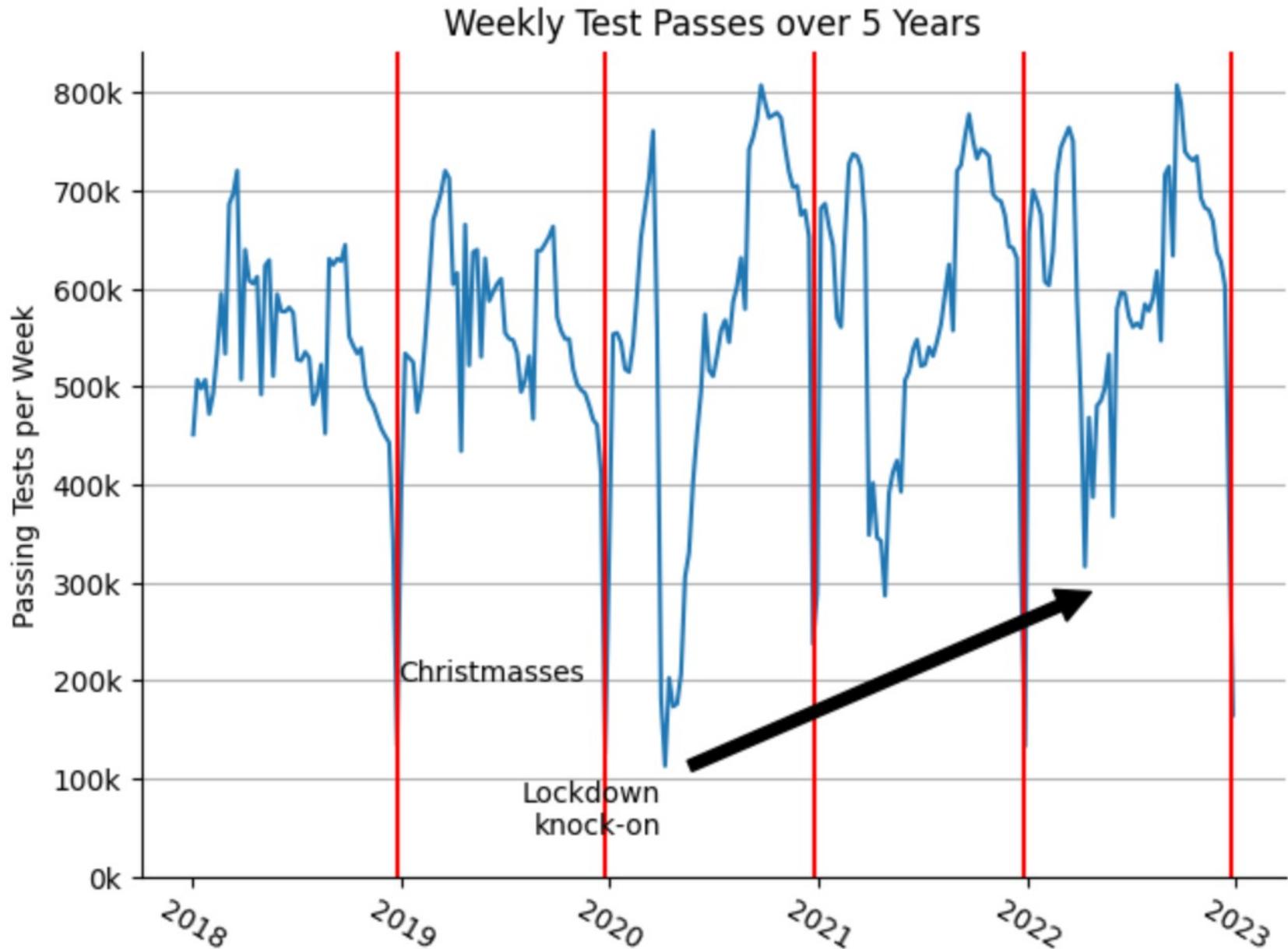
```
dfpll = pl.scan_parquet("../test_result.parquet/*.parquet") Implicit Lazy DataFrame
```

```
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"))
    .group_by_dynamic("test_date", every="1w")
    .agg(pl.col("passed").sum())
    .collect()
)
```

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

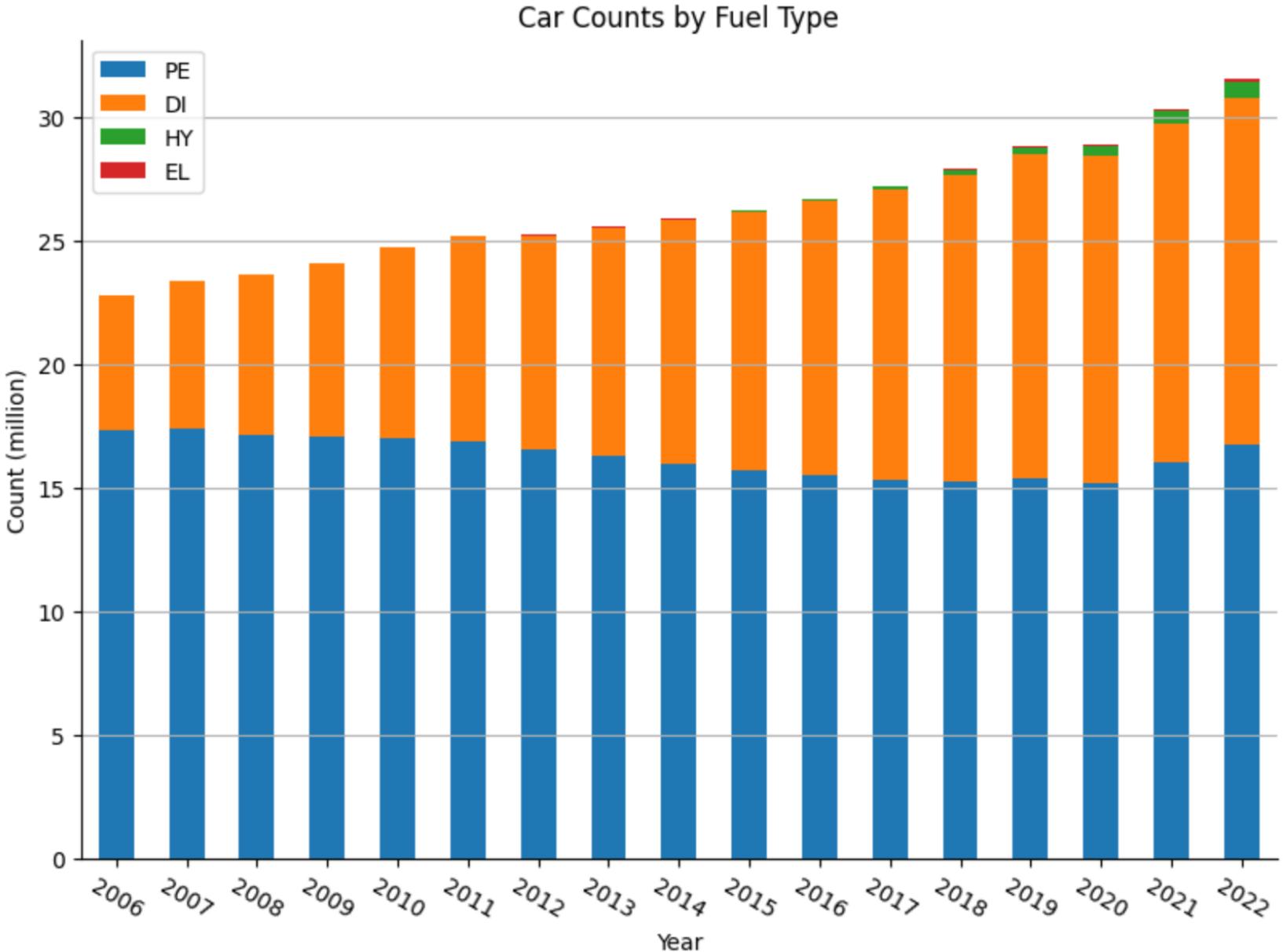


April drop
was due to
lockdown

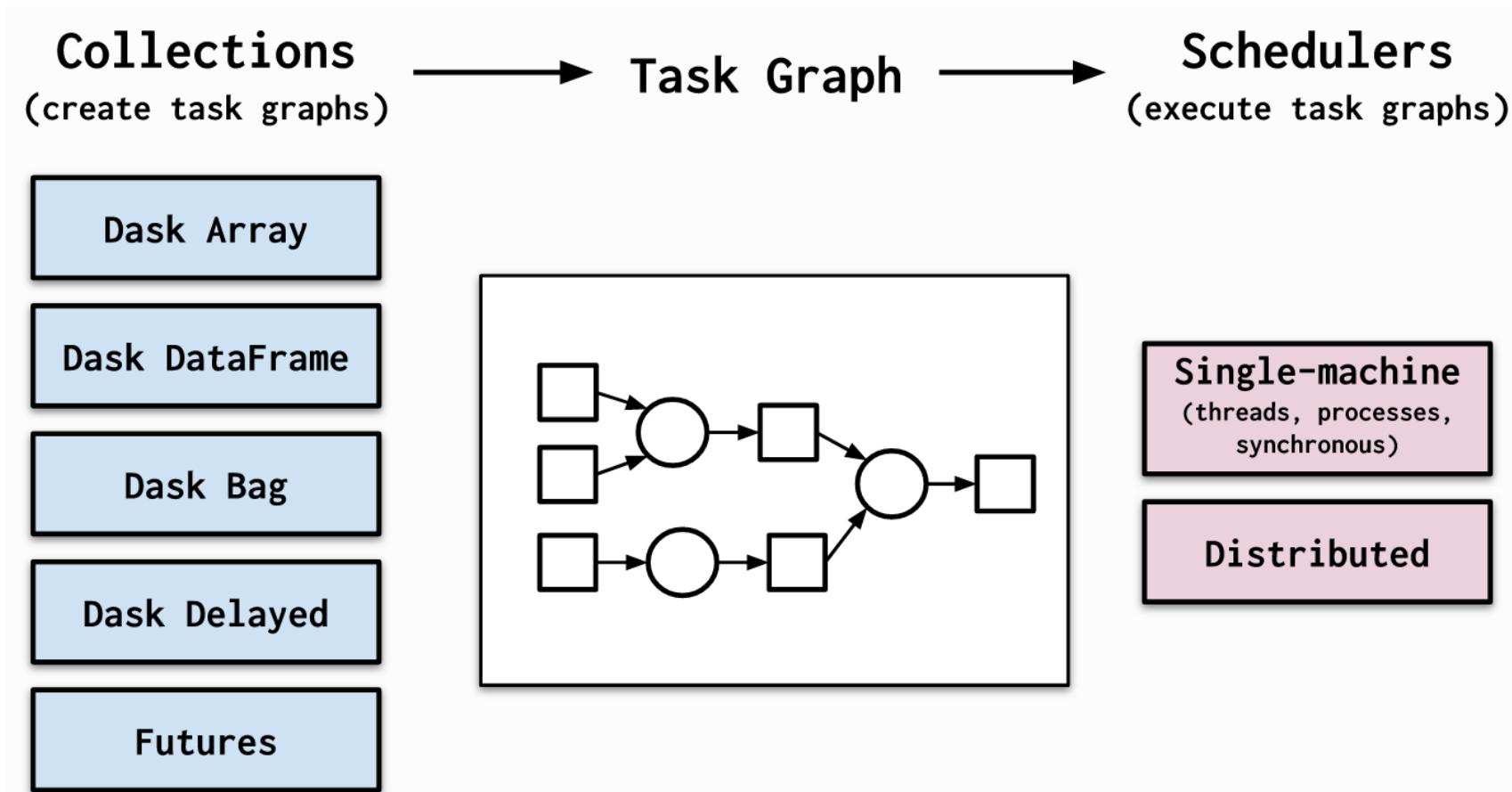


Vehicle ownership increases, Hybrids growing

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



```
fuel_type_ddf = (
    dd.read_parquet(parquet_path, dtype_backend="pyarrow")
    .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()
)
```

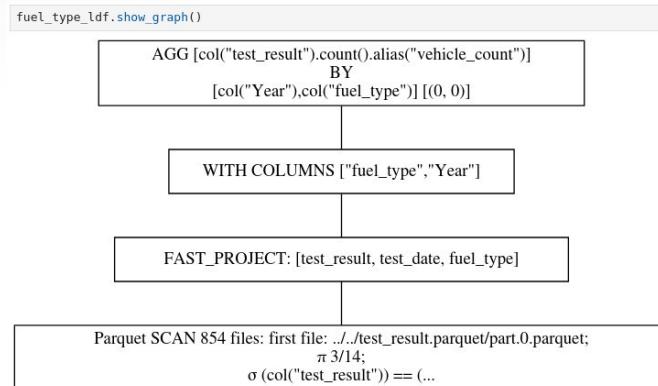
Last executed in 42.73s

```
ddf = dx.read_parquet(parquet_path, dtype_backend="pyarrow",)
fuel_type_ddf = (
    ddf[(ddf['test_result'] == "P")]
    #.query('test_result == "P"') # not in dask-expr 0.2.4
    .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"})
    .optimize()
    .compute()
)
```

Last executed in 26.61s

“Basic” Dask, looks similar to Pandas
But lacks a query planner, so does unnecessary work

Dask Expressions only 6 months old, builds on existing DDF, undergoing rapid improvement
This includes a query planner



```

fuel_type_ddf = (
    dd.read_parquet(parquet_path,
                    dtype_backend="pyarrow",
                    columns=["test_result", "test_date", "fuel_type"],
                    filters=[("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()
)
  
```

Last executed in 23.26s

Manual “Predicate & Projection Pushdown” to Parquet reader improves performance (but Expressions should do as well as this)
Dask faster in last 6 months too

```

fuel_type_edf = (
    pl.scan_parquet(parquet_path)
    .filter(pl.col("test_result") == "P")
    .with_columns(
        pl.col("fuel_type")
        .replace({"Hybrid Electric (Clean)": "HY", "Electric": "EL"},
                 default=pl.first())
        .cast(str),
        pl.col("test_date").dt.year().alias("Year"),
    )
    .group_by(["Year", "fuel_type"])
    .agg(pl.col("test result").count().alias("vehicle count"))
    .collect(streaming=True) # streaming required to prevent OOM
)
  
```

Last executed in 24.08s

Streaming allows bigger-than-RAM, still early-stage, required on 32GB laptop for this example but not on 64GB laptop



Thoughts on our testing

- Haven't checked lots of things!
- Numba doesn't compile Arrow extension array
- **NaN / Missing behaviour different Polars/Pandas**
- **sklearn partial support** (sklearn assumes Pandas API)
 - Polars working on dataframe interchange protocol
- Arrow timeseries/str **different** to Pandas NumPy?



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 (Dask has API differences))
- 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)

We won the “rally”! ££→Parkinsons



Next year - vehicle telematics?

Summary

- Experiment, we have options!

NotANumber.email 

A Pythonic Data Science Newsletter

- I love receiving postcards (email me)

Second
Edition

O'REILLY®

High Performance Python

Practical Performant
Programming for Humans

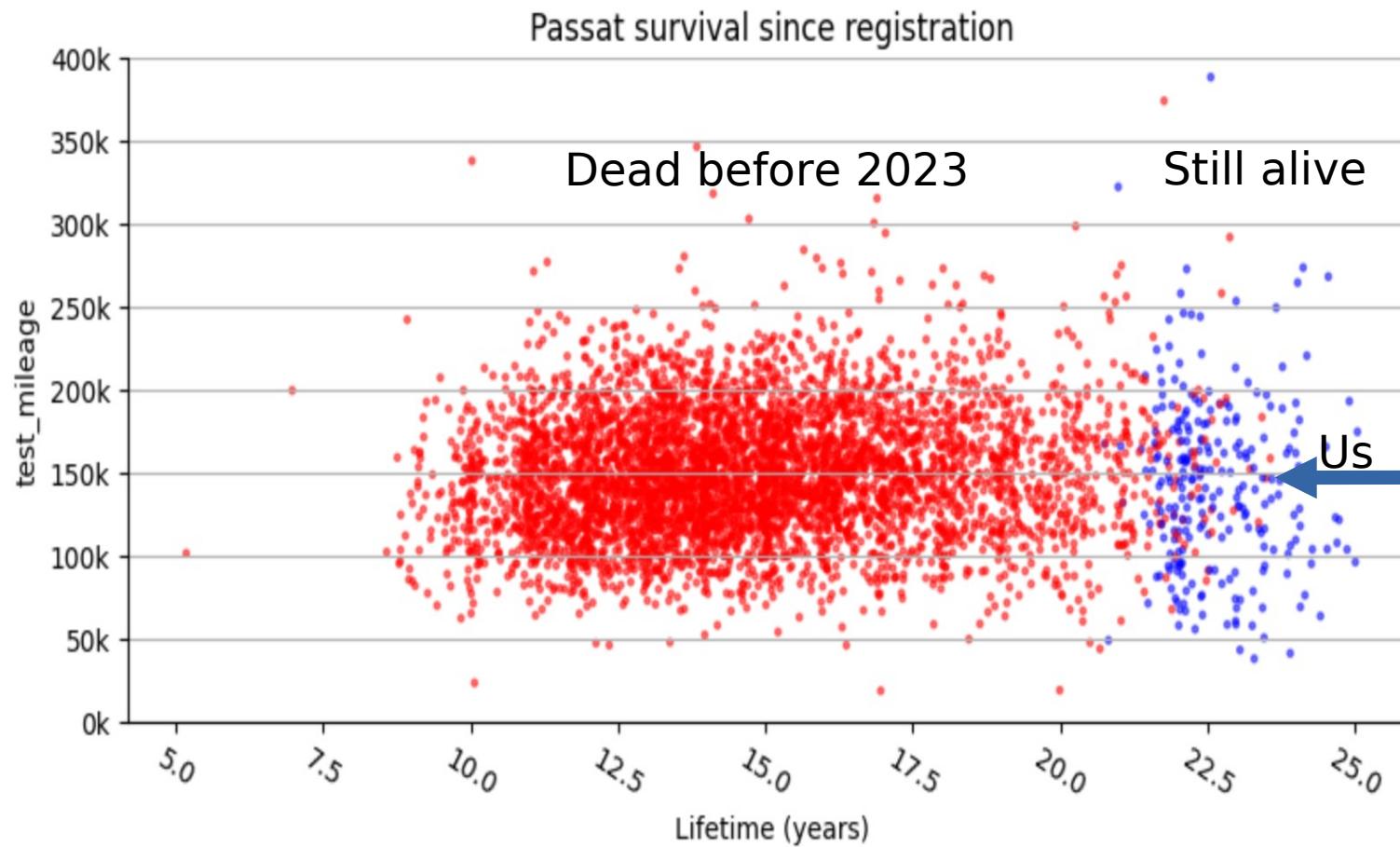


Micha Gorelick & Ian Ozsvald



Appendix

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
```



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

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
```