(Pretty) big data wrangling with DuckDB and Polars

With examples in R and Python

Grant McDermott

gmcd@amazon.com

Principal Economist, Amazon

December 15, 2025

Preliminaries

Agenda and expectations

These slides are mostly intended to serve as a road map.

- Most of what I'll (we'll) be doing is live coding and working through examples.
- I strongly encourage you try these examples on you own machines. Laptops are perfectly fine.

Note: All of the materials are available on my website:

- https://grantmcdermott.com/duckdb-polars

Preliminaries

Requirements

Important: If you'd like to follow along, please make sure that you have completed the requirements listed on the website.

- Install the required R and/or Python libraries.
- Download some NYC taxi data.

The data download step can take 15-20 minutes, depending on your internet connection.

Problem statement

Why this workshop?

It's a trope, but "big data" is everywhere. This is true whether you work in tech (like I do now), or in academic research (like I used to).

OTOH many of datasets that I find myself working with aren't at the scale of truly *huge* data that might warrant a Spark cluster.

- We're talking anywhere between 100 MB to 50 GB. (Max a few billion rows; often in the millions or less.)
- Can I do my work without the pain of going through Spark?

Another factor is working in polyglot teams. It would be great to repurpose similar syntax and libraries across languages…

Taster

DuckDB example

```r
library(duckdb)
library(arrow)
library(dplyr)

nyc = open_dataset(here::here("nyc-taxi"))
prettyNum(nrow(nyc), ",")
```

```
[1] "178,544,324"
```

```r
tic = Sys.time()

nyc_summ = nyc |>
  to_duckdb() |>
  summarise(
    mean_tip = mean(tip_amount),
    .by = passenger_count
  ) |>
  collect()

(toc = Sys.time() - tic)
```

```
Time difference of 0.912349 secs
```

Taster

DuckDB example (cont.)

We just read a ~180 million row dataset (from disk!) and did a group-by aggregation on it.

In < 1 second.

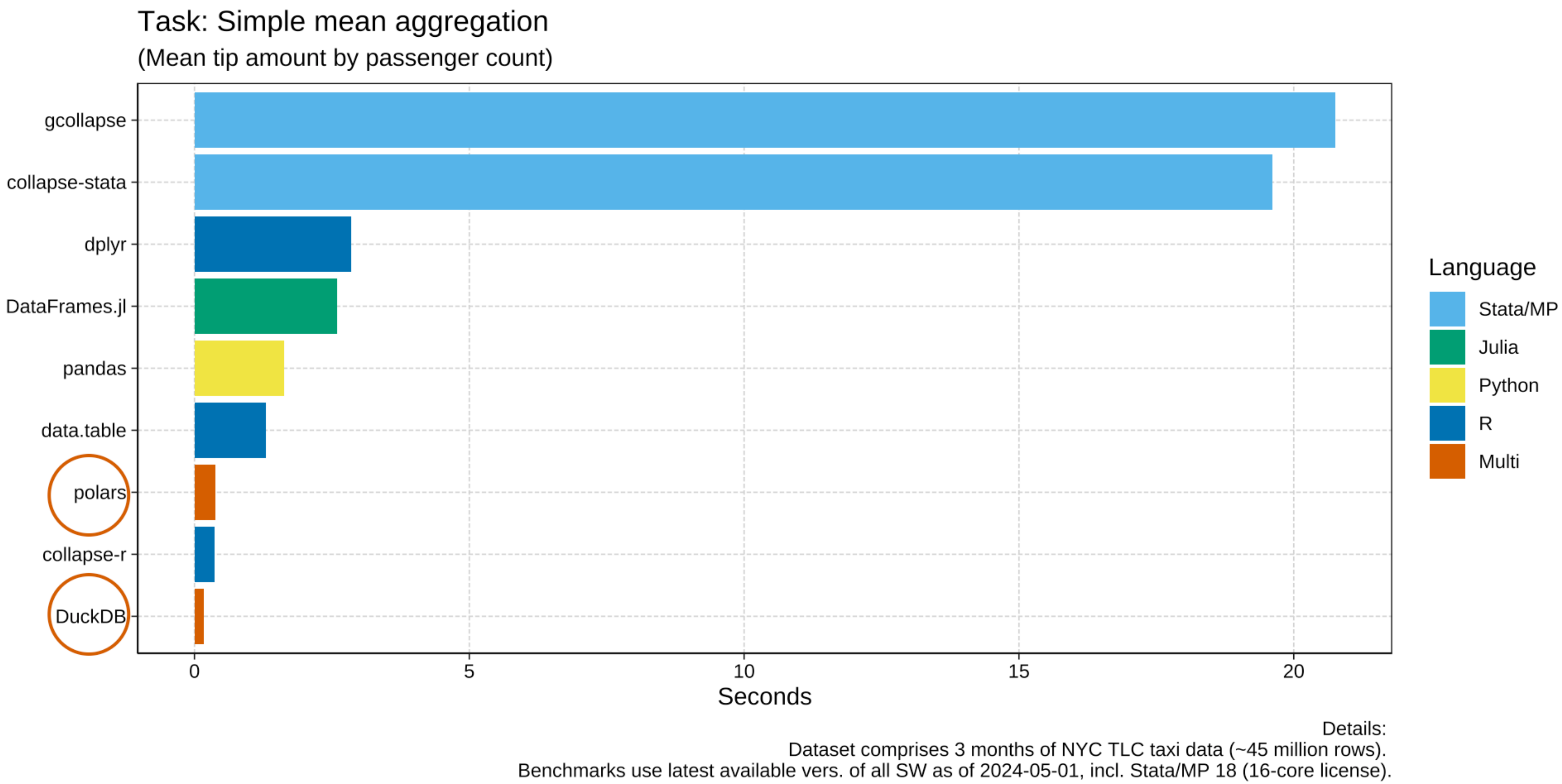
On a laptop.

🤯

Let’s do a quick horesrace comparison (similar grouped aggregation, but on a slightly smaller dataset)…

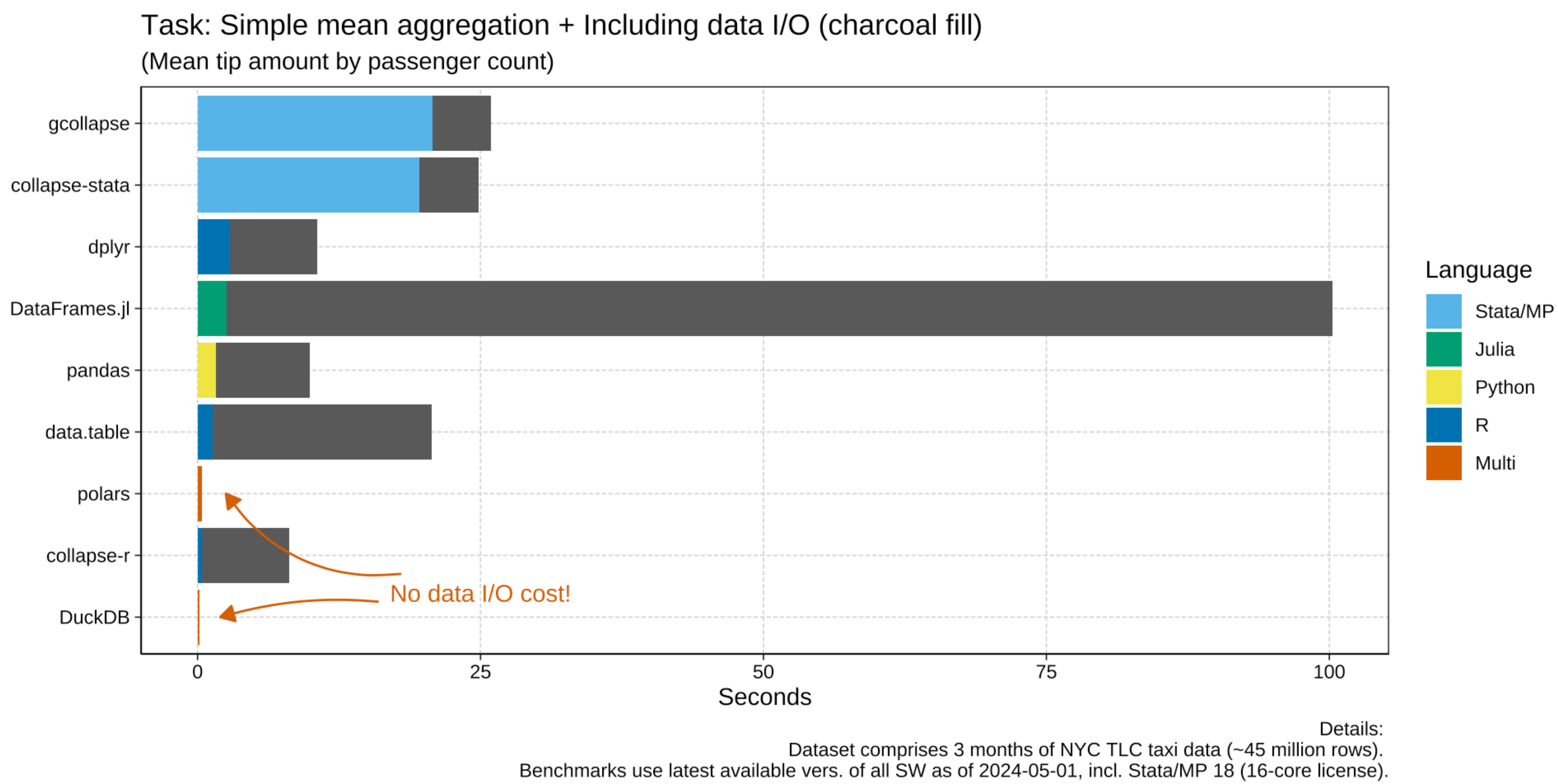
Simple benchmark: Computation time only

DuckDB and Polars are already plenty fast…



Simple benchmark: Computation time + data I/O

… but are even more impressive once we account for data import times



Wait. How??

Better disk storage 🤝Better memory representation

Two coinciding (r)evolutions enable faster, smarter computation:

1. Better on-disk storage

- E.g. Hive-partitioned Parquet files.
- Columnar storage format allows better compression (much smaller footprint) and efficient random access to selected rows or columns (don't have to read the whole dataset *a la* CSVs).

2. Better in-memory representation

- Standardisation around the Apache Arrow format + columnar representation. (Allows zero copy, fewer cache misses, etc.)
- OLAP + materialisation. (Rather than "eagerly" executing each query step, we can be "lazy" and optimise queries before executing them.)

Query optimization

Key concepts

Three key optimizations work together:

- **Lazy Materialization:** *When* computation happens
- **Predicate Pushdown:** *Where* filtering happens
- **Projection Pushdown:** *Which* columns are read

Lazy Materialization

When computation happens

- Build query plan first, execute later
- Only compute when results are needed (`.collect()`, `.show()`)
- Enables global optimization across entire pipeline

Example: `SELECT` operations are bumped to the top of an (optimzed) query to avoid unecessary work

Predicate Pushdown

Which rows are read

- Push `WHERE` conditions to the storage layer
- Filter at file/partition level before reading into memory
- Dramatically reduces I/O by skipping irrelevant data

Example: `WHERE month = 3` → Only scan March parquet files

Projection Pushdown

Which columns are read

- Only read columns actually needed for the query
- Skip unnecessary columns at the storage layer
- Reduces memory usage and I/O bandwidth

Example: `SELECT tip_amount` → Only read tip column, skip other 29 columns

Scaling up

Even moar benchmarks

Question: Do these benchmarks hold and scale more generally?
Answer: Yes. See *Database-like ops benchmark*.

Moreover—and I think this is key—these kinds of benchmarks normally exclude the data I/O component… and the associated benefits of not having to hold the whole dataset in RAM.

- There are some fantastically fast in-memory data wrangling libraries out there. (My personal faves: data.table and collapse.) But “in-memory” means that you always have to keep the full dataset in, well, memory. And this can be expensive.
- Libraries like DuckDB and Polars sidestep this problem, effectively supercharging your computer’s data wrangling powers.

DuckDB

- Embedded C++ analytical database (no server needed)
- Multiple language frontends (R, Python, Julia, etc.)
- SQL interface with “friendly” extensions
- Excellent for out-of-memory operations

Polars

- Embedded Rust-based DataFrame library
- Python/R bindings (multiple language support)
- DataFrame interface with lazy evaluation
- Built on Apache Arrow memory format

Examples

Live coding sessions

Let’s head back to the website to work through some notebooks.

DuckDB

- DuckDB SQL
- DuckDB + dplyr (R)
- DuckDB + Ibis (Python)

Polars

- Polars from R and Python

What didn't we cover?

Other cool features

- **S3 I/O**
  - → DuckDB & Polars can both read/write directly from/to S3. You just need to provision your AWS credentials. [Ex. 1, 2, 3]
  - → Note: I prefer/recommend the workflow we practiced today—first download to local disk via `aws cli`—to avoid network + I/O latency.
- **Geospatial**
  - → IMO the next iteration of geospatial computation will be built on top of the tools we've seen today (and related libs).
  - → DuckDB provides an excellent spatial extension (works with dplyr). See also the GeoParquet, GeoArrow, & GeoPolars initiatives.

What didn’t we cover?

Other cool features (cont.)

- **Streaming**
  - → Streaming is the feature that enables working with bigger-than-RAM data.
  - → Very easy to use and/or adjust our workflow to these cases…
  - → DuckDB: Simply specify a disk-backed database when you first fire up your connection from Python or R, e.g.

```
con = dbConnect(duckdb(), dbdir = "nyc.dbb")
```

  - → Polars: Simply specify streaming when collecting, e.g.

```
some_query.collect(streaming=True)
```

What didn’t we cover?

Other cool features (cont.)

- **Modeling**
  - → The modeling part *used* to be less tightly integrated with DuckDB/Polars workflows. But that’s changing rapidly, e.g.: **dbreg**, **duckreg**, and **duckdb-mlpack**.
  - → FWIW being able to quickly I/O parts of large datasets makes it very easy to iteratively run analyses on subsets of your data. E.g., I often pair with **fixest** for exceptional in-memory performance.
  - → You can also run bespoke models via UDFs and/or predictions on database backends. [Ex. 1, 2, 3]

Resources

Learning more

DuckDB

- DuckDB homepage. Includes a very informative blog and standalone documentation for the client APIs (Python, R, and many others).
- Also check out Harlequin for a cool, shell-based DuckDB IDE.

Polars

- Polars GitHub Repo. Contains links to the standalone documentation for the client APIS (Python, R, etc.)
- Side-by-side code comparisons (versus pandas, dplyr, etc.) are available in *Modern Polars (in Python)* and *Codebook for Polars in R*.