Pandas Should Go Extinct

Eddie Atkinson (he/him)

2025-04-04

Senior Software Engineer - Atlassian

Who's Who in the Zoo?

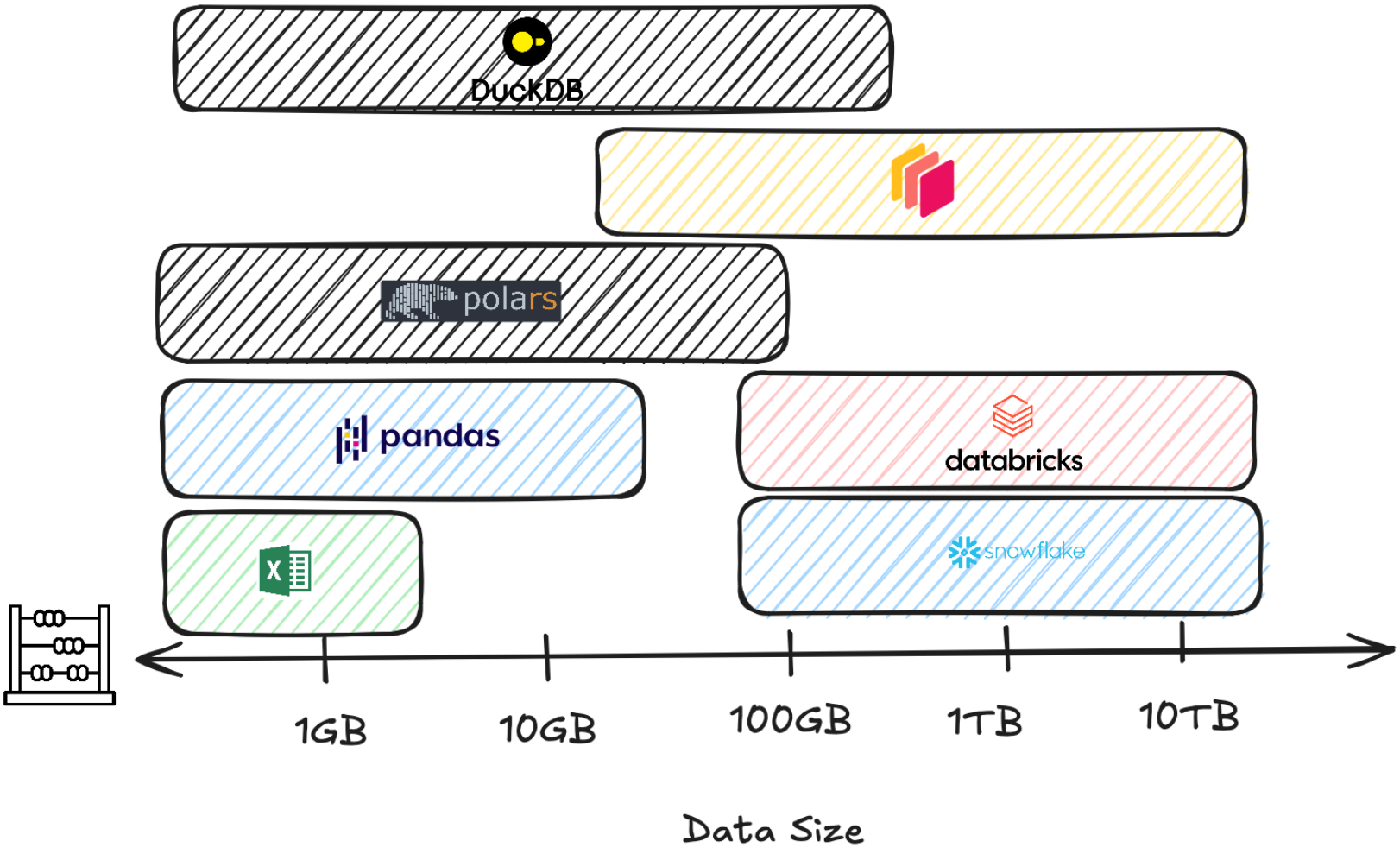


Figure 1: Rough Guide to Data Tool Adoption Curve by Data Size

Who's Who in the Zoo?

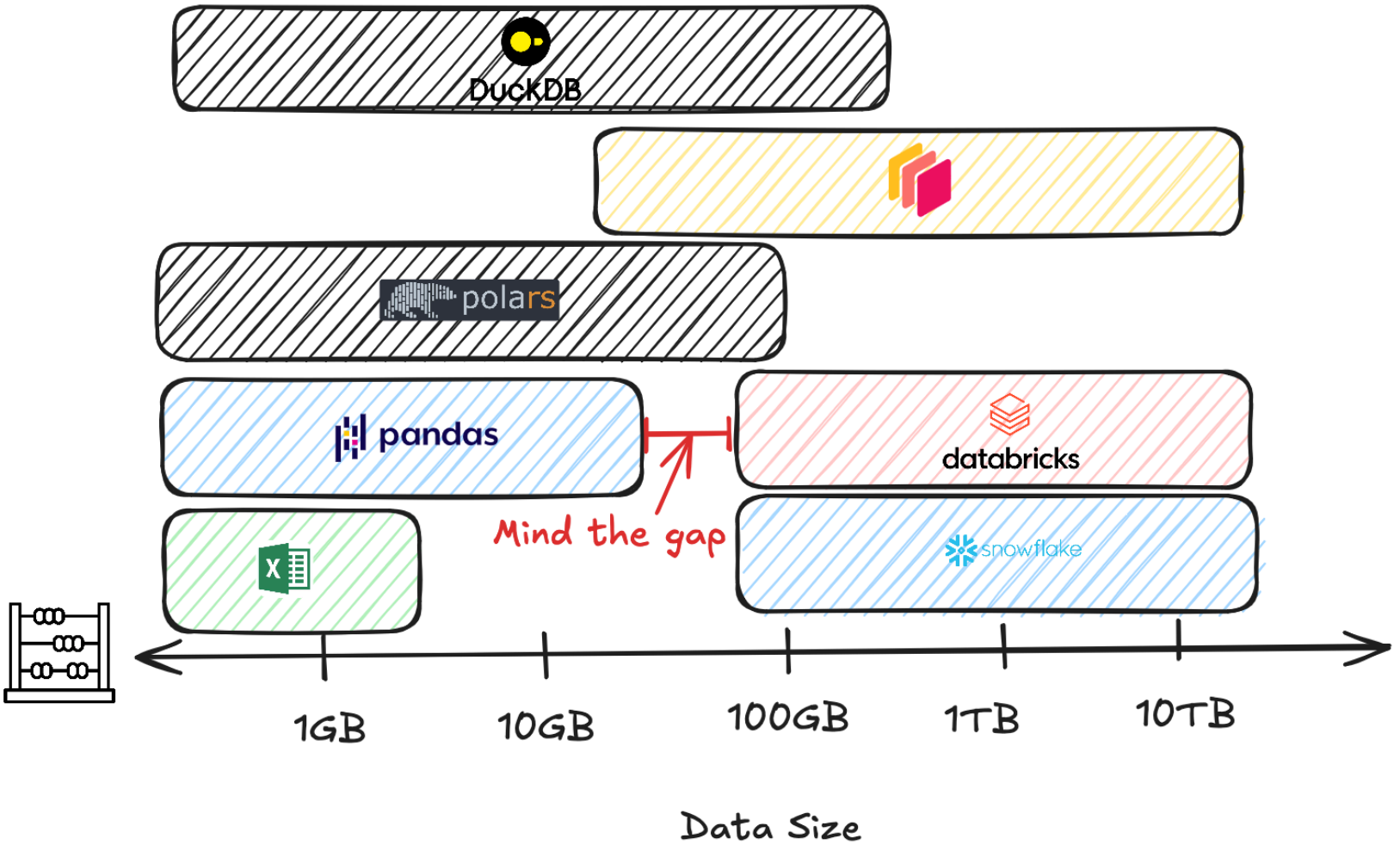


Figure 2: Pandas Data Size Cliff

Why do you care so much about ~100GB data sets?

Who's got Big Data™?

94.68%

Rowcount Bucket	Fleet	TPC-H	TPC-DS
$(10^0, 10^1]$	12.3%	12.5%	-
$(10^1, 10^2]$	10.8%	12.5%	25.0%
$(10^2, 10^3]$	12.0%	-	-
$(10^3, 10^4]$	12.9%	-	16.6%
$(10^4, 10^5]$	14.2%	-	12.5%
$(10^5, 10^6]$	13.5%	-	4.2%
$(10^6, 10^7]$	11.4%	-	4.2%
$(10^7, 10^8]$	7.6%	12.5%	8.3%
$(10^8, 10^9]$	3.5%	25.0%	12.5%
$\geq 10^9$	1.8%	37.5%	16.7%

x1KB/row = 100GB

Figure 3: Bucketed Total Row Counts for Amazon RedShift Fleet Over 1 Month [1]

Who's got Big Data™?

86.9%

Time bucket	% of queries			% of sum(runtime)		
	Fleet	TPC-H	-DS	Fleet	TPC-H	-DS
(0s, 10ms]	13.7	0	0	0.01	0	0
(10ms, 100ms]	48.3	0	0	0.4	0	0
(100ms, 1s]	24.9	0	22	2.3	0	2
(1s, 10s]	9.9	27	59	7.3	3	19
(10s, 1min]	2.2	55	13	13.3	30	25
(1min, 10min]	0.86	8	5	35.7	66	55
(10min, 1h]	8e-2	0	0	25.2	0	0
(1h, 10h]	8e-3	0	0	14.3	0	0
>=10h	9e-5	0	0	1.6	0	0

10 nodes @
8GB/s = 80GB

Figure 4: Bucketed Query Runtime for Amazon RedShift Fleet Over 1 Month [1]

No one stores or queries Big Data*

No one stores or queries Big Data*

*almost no one

Shut up and show me the code

Example: 1 Billion Row Challenge

Contest to create the fastest JVM implementation which could compute the min, mean, max for 1 billion rows of weather station data

Example: 1 Billion Row Challenge

Contest to create the fastest JVM implementation which could compute the min, mean, max for 1 billion rows of weather station data

```
Perth;12.0
Sydney;2.0
```

to

```
{Perth=12.0/12.0/12.0...}
```

Example: 1 Billion Row Challenge

Fastest JVM implementation: **1.535 seconds**

Example: 1 Billion Row Challenge

Fastest JVM implementation: **1.535 seconds**

Our Experimental Setup:

- 16GB CSV
- AWS M7a.8xlarge
  - 32 core AMD CPU
  - 128 GB of RAM
- 30 x repetitions (2 x warmups)

Example: 1 Billion Row Challenge

Fastest JVM implementation: **1.535 seconds**

Our Experimental Setup:

- 16GB CSV
- AWS M7a.8xlarge
  - 32 core AMD CPU
  - 128 GB of RAM
- 30 x repetitions (2 x warmups)

We won't benchmark output serialisation

Pandas

```
def do_1brc_pandas(file_path: str):
    df = (
        pd.read_csv(file_path, sep=";", names=["station", "measurement"])
        .groupby("station")
        .agg({"measurement": ["min", "mean", "max"]})
        .round(2)
    )
```

Polars

```python
def do_1brc_polars(file_path: str):
    df = (
        pl.scan_csv(
            file_path,
            separator=";",
            new_columns=["station", "measurement"],
            has_header=False,
        )
        .group_by("station")
        .agg(
            pl.col("measurement").min().round(2).alias("min"),
            pl.col("measurement").mean().round(2).alias("mean"),
            pl.col("measurement").max().round(2).alias("max"),
        )
        .sort(by="station")
        .collect(new_streaming=True) # Trigger evaluation in streaming mode
    )
```

Polars

```python
def do_1brc_polars(file_path: str):
    df = (
        pl.scan_csv(
            file_path,
            separator=";",
            new_columns=["station", "measurement"],
            has_header=False,
        )
        .group_by("station")
        .agg(
            pl.col("measurement").min().round(2).alias("min"),
            pl.col("measurement").mean().round(2).alias("mean"),
            pl.col("measurement").max().round(2).alias("max"),
        )
        .sort(by="station")
        .collect(new_streaming=True) # Trigger evaluation in streaming mode
    )
```

DuckDB

```
def do_1brc_duckdb(file_path: str):
    df = duckdb.read_csv(file_path, names=["station", "measurements"])

    src = duckdb.sql("""
          create table src as
          select
            station,
            min(measurements) min,
            max(measurements) max,
            cast(avg(measurements) as decimal(8, 1)) avg
          from df
          group by station
        """
    )
```

Results (M7a.8xlarge)

Library	Median Duration	Median Max CPU %	Median Max USS	Median Max Swap
Pandas	4m 28s	113.0%	38.12 GB	0 MB

Results (M7a.8xlarge)

Library	Median Duration	Median Max CPU %	Median Max USS	Median Max Swap
Pandas	4m 28s	113.0%	38.12 GB	0 MB
Polars	5.04s	3202.60%	18.02 GB	0 MB

Results (M7a.8xlarge)

Library	Median Duration	Median Max CPU %	Median Max USS	Median Max Swap
Pandas	4m 28s	113.0%	38.12 GB	0 MB
Polars	5.04s	3202.60%	18.02 GB	0 MB
DuckDB	5.19s	3174.64%	1.93 GB	0 MB

Results (11th Gen 8 Core i5, 16GiB RAM)

Library	Median Duration	Median Max CPU %	Median Max USS	Median Max Swap
Pandas	12m 15s	110.35%	15.67 GB	21.02 GB
Polars	39s	765.75%	15.22 GB	35.85 MB
DuckDB	47s	807.0%	546.87 MB	0 MB

What do these tools give us?

- Great performance on a **flawed benchmark**

What do these tools give us?

- Great performance on a **flawed benchmark**
- Painless multi-threading

What do these tools give us?

- Great performance on a **flawed benchmark**
- Painless multi-threading
- Streaming

What do these tools give us?

- Great performance on a **flawed benchmark**
- Painless multi-threading
- Streaming
- Spill to disk

What do these tools give us?

- Great performance on a **flawed benchmark**
- Painless multi-threading
- Streaming
- Spill to disk
- Lazy evaluation

I’m interested, but shiny tools have hurt me before

Painless Adoption with Arrow

- In-memory column oriented data format

Painless Adoption with Arrow

- In-memory column oriented data format
- Supported in Pandas since 2.0 (April 2023)

Painless Adoption with Arrow

- In-memory column oriented data format
- Supported in Pandas since 2.0 (April 2023)
  - Support is **getting there**

Painless Adoption with Arrow

- In-memory column oriented data format
- Supported in Pandas since 2.0 (April 2023)
  - Support is **getting there**
- **Polars and DuckDB support zero copy serialisation to and from Arrow-backed Pandas DataFrames**

Painless Adoption with Arrow

- In-memory column oriented data format
- Supported in Pandas since 2.0 (April 2023)
  - Support is **getting there**
- **Polars and DuckDB support zero copy serialisation to and from Arrow-backed Pandas DataFrames**
  - Otherwise you get a fresh copy

Practical Example: New York Taxi Data

- Dataset of every taxi trip taken in New York
- 1 Parquet file per month from 2009-

Practical Example: New York Taxi Data

- Dataset of every taxi trip taken in New York
- 1 Parquet file per month from 2009-
- Corporate wants us to find out if the Pandemic made cash less common
  - 2019-2022 (2.71GB of Parquet)

Practical Example: New York Taxi Data

- Dataset of every taxi trip taken in New York
- 1 Parquet file per month from 2009-
- Corporate wants us to find out if the Pandemic made cash less common
  - 2019-2022 (2.71GB of Parquet)
- We're using DuckDB with Pandas, also works with Polars and Pandas

Pandas Reading

```python
def read_data_pandas(folder: Path) -> pd.DataFrame:
    df = None
    for entry in folder.iterdir():
        if not entry.name.endswith("parquet"): continue
        temp = pd.read_parquet(entry, columns=COLUMNS, dtype_backend="pyarrow")
        if df is not None:
            df = pd.concat([df, temp])
        else:
            df = temp
    df = df[df["tpep_pickup_datetime"] > pd.to_datetime(MIN_DATETIME)]
    df = df[df["tpep_pickup_datetime"] < pd.to_datetime(MAX_DATETIME)]

    df["month"] = df["tpep_pickup_datetime"].dt.month
    df["year"] = df["tpep_pickup_datetime"].dt.year
    df = df.drop(["tpep_pickup_datetime"], axis="columns")
    return df
```

Pandas Reading

```python
def read_data_pandas(folder: Path) -> pd.DataFrame:
    df = None
    for entry in folder.iterdir():
        if not entry.name.endswith("parquet"): continue
        temp = pd.read_parquet(entry, columns=COLUMNS, dtype_backend="pyarrow")
        if df is not None:
            df = pd.concat([df, temp])
        else:
            df = temp
    df = df[df["tpep_pickup_datetime"] > pd.to_datetime(MIN_DATETIME)]
    df = df[df["tpep_pickup_datetime"] < pd.to_datetime(MAX_DATETIME)]

    df["month"] = df["tpep_pickup_datetime"].dt.month
    df["year"] = df["tpep_pickup_datetime"].dt.year
    df = df.drop(["tpep_pickup_datetime"], axis="columns")
    return df
```

Pandas Compute

```
def calculate_cash_pandas(df: pd.DataFrame):
    df = (
        df.groupby(["year", "month", "payment_type"])
        .agg({"payment_type": "count"})
        .unstack(fill_value=0, level=2)["payment_type"]
        .reset_index()
    )

    df["total_payments"] = df.iloc[:, 2:8].sum(axis=1)
    df["cash_pct"] = (df[CASH] / df["total_payments"]) * 100
    return df
```

DuckDB Reading

```
def read_data_duck(folder: str):
    return duckdb.sql(f"""
        select
          datepart('year', tpep_pickup_datetime) year,
          datepart('month', tpep_pickup_datetime) month,
          payment_type
        from '{folder}/*.parquet'
        where
          tpep_pickup_datetime > '{MIN_DATE}'
          and tpep_pickup_datetime < '{MAX_DATE}'""")
      )
```

DuckDB Reading

```
def read_data_duck(folder: str):
    return duckdb.sql(f"""
        select
          datepart('year', tpep_pickup_datetime) year,
          datepart('month', tpep_pickup_datetime) month,
          payment_type
        from '{folder}/*.parquet'
        where
          tpep_pickup_datetime > '{MIN_DATE}'
          and tpep_pickup_datetime < '{MAX_DATE}'""")
      )
```

DuckDB Reading

```
def read_data_duck(folder: str):
    return duckdb.sql(f"""
        select
          datepart('year', tpep_pickup_datetime) year,
          datepart('month', tpep_pickup_datetime) month,
          payment_type
        from '{folder}/*.parquet'
        where
          tpep_pickup_datetime > '{MIN_DATE}'
          and tpep_pickup_datetime < '{MAX_DATE}'"""
      )
```

DuckDB Compute

```
def calculate_cash_duck(data):
    return duckdb.sql(f"""
        with total as (
            select year, month, count(payment_type) payments from df
            group by year, month
        ),
        total_cash as (
            select year, month, count(payment_type) cash from df
            where payment_type={CASH}
            group by year, month
        )
        select total.*, cash, (cash / total) * 100 cash_pct from total
        join total_cash
          on total.year=total_cash.year and total.month=total_cash.month
        order by total.year, total.month
    """).df()
```

Test Cases

```python
def pure_pandas(folder: Path):
  data = read_data_pandas(folder)
  df = calculate_cash_pandas(data)

def duck_reads_panda_thinks(folder: Path):
  data = read_data_duck(folder).df()
  df = calculate_cash_pandas(data)

def panda_reads_duck_thinks(folder: Path):
  data = read_data_pandas(folder)
  df = calculate_cash_duck(data)

def pure_duck(folder: Path):
    data = read_data_duck(folder)
    df = do_taxi_duck_compute(data)
```

Results (11th Gen 8 Core i5, 16GiB RAM)

Approach	Median Duration	Median Max CPU %	Median Max USS	Median Max Swap
Pure Pandas	41.88s	146.10%	14.52 GB	1.92 GB
Duck Reads Panda Thinks	28.39s	793.7%	14.79 GB	1.22 GB
Panda Reads Duck Thinks	29.25s	765.4%	12.39 GB	0 MB
Pure DuckDB	21.70s	814.95%	216.76 MB	0 MB

So…did cash usage fall over the Pandemic?

- Yes
- Correlation !== causality, don’t @ me

So...did cash usage fall over the Pandemic?

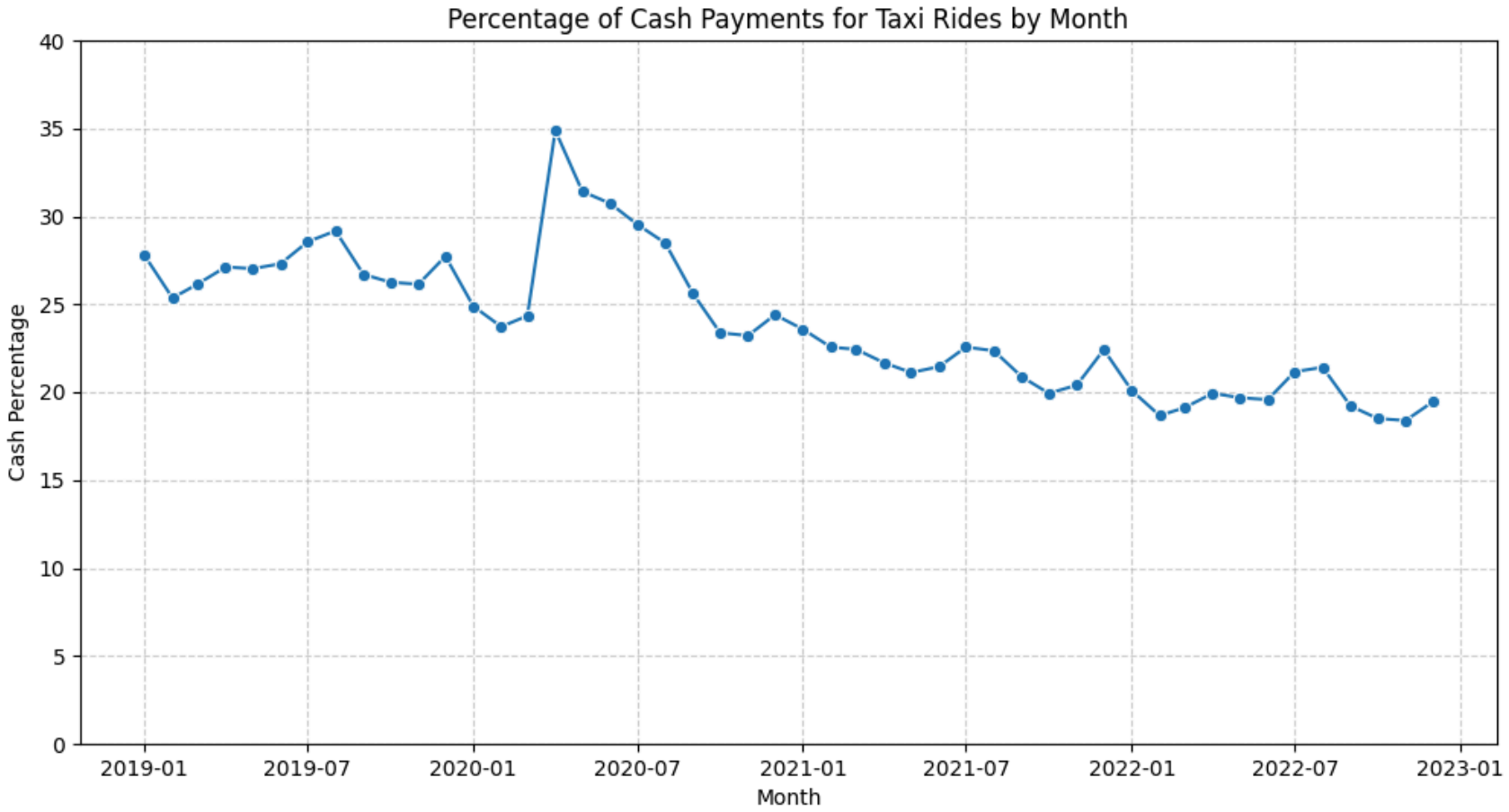


Figure 5: Percentage of cash payments for taxis by month

OK, why shouldn’t I listen to you?

Reasons NOT to listen to me

Reasons NOT to listen to me

1. I'm just a guy with a laptop - do your own research

Reasons NOT to listen to me

1. I'm just a guy with a laptop - do your own research
2. You're super deeply integrated into the Pandas ecosystem

Reasons NOT to listen to me

1. I'm just a guy with a laptop - do your own research
2. You're super deeply integrated into the Pandas ecosystem
   - e.g. Geospatial

Reasons NOT to listen to me

1. I'm just a guy with a laptop - do your own research
2. You're super deeply integrated into the Pandas ecosystem
   - e.g. Geospatial
3. Let Pandas cook

Reasons NOT to listen to me

1. I'm just a guy with a laptop - do your own research
2. You're super deeply integrated into the Pandas ecosystem
   - e.g. Geospatial
3. Let Pandas cook
   - Progress on integrating PyArrow nicely has been…slow

Reasons NOT to listen to me

1. I'm just a guy with a laptop - do your own research
2. You're super deeply integrated into the Pandas ecosystem
   - e.g. Geospatial
3. Let Pandas cook
   - Progress on integrating PyArrow nicely has been…slow
   - There are benefits beyond performance

Reasons NOT to listen to me

1. I'm just a guy with a laptop - do your own research
2. You're super deeply integrated into the Pandas ecosystem
   - e.g. Geospatial
3. Let Pandas cook
   - Progress on integrating PyArrow nicely has been…slow
   - There are benefits beyond performance
     - Ergonomics / API grokkability

Reasons NOT to listen to me

1. I'm just a guy with a laptop - do your own research
2. You're super deeply integrated into the Pandas ecosystem
   - e.g. Geospatial
3. Let Pandas cook
   - Progress on integrating PyArrow nicely has been…slow
   - There are benefits beyond performance
     - Ergonomics / API grokkability
     - Transferable skill set

You've suggested two tools, which is better?

Tool Showdown

- Depends on your preference - SQL vs Code
  - As an engineer I prefer Polars for unit testing
- DuckDB goes well beyond dataframes
  - e.g. Local first analytics embedded in an Android app
  - e.g. Compile to WASM and do SQL queries for visualisation on the…frontend(?)

Takeaways

- You can't avoid complex distributed querying systems, but **you can defer them (potentially indefinitely) by avoiding Pandas**

Takeaways

- You can't avoid complex distributed querying systems, but **you can defer them (potentially indefinitely) by avoiding Pandas**
- We are experiencing a Cambrian explosion in dataframe tools - try them out

Shameless Self-Promotion

- GitHub: https://github.com/eddie-atkinson/pandas-should-go-extinct
- LinkedIn: https://www.linkedin.com/in/eddie-atkinson
- Blog: https://eddie.codes

Bibliography

[1] A. van Renen *et al.*, "Why TPC is not enough: An analysis of the Amazon Redshift fleet," *Proceedings of the VLDB Endowment*, vol. 17, no. 11, pp. 3694–3706, 2024.