# CHOOSING YOUR DATA CHAMPION:

A Side-by-Side Look at **FireDucks** and **Polars**

## Introduction:

When it comes to data analysis, the choice of tools is critical for peak performance and workflow optimization. FireDucks emerges as a formidable DataFrame library, with its memory-efficient architecture that not only promises but delivers enhanced performance. Compared to other libraries like Polars, FireDucks stands out for its superior handling of large datasets, ensuring swift and efficient processing from loading to complex aggregations. This blog post dives into a comprehensive performance comparison, focusing on memory usage and execution speed. We'll demonstrate through rigorous benchmarks how FireDucks consistently outperforms its peers, particularly when managing voluminous data, making it the preferred choice for data analysts aiming for both speed and precision. Witness the capabilities of FireDucks as we place it head-to-head with Polars and discover why it could be the game-changer for your data analysis endeavours.

# Analysing Efficiency: FireDucks vs. Polars in Memory Management and Execution Speed

The match-up of FireDucks versus Polars is more than just a comparison—it's a detailed examination of efficiency and speed. When it comes to data analysis, the choice of DataFrame library can make all the difference. Both FireDucks and Polars have their merits, with strong features and robust performance. Yet, the true test lies in their management of memory and execution times when pitted against each other. We're going to explore this by looking at specific code examples, providing a balanced comparison of these two contenders.

## Benchmarking Environment:

- **Evaluation environment details:** Ubuntu 18.04.6 LTS (GNU/Linux 5.4.0-150-generic x86_64)
- **System Memory:** 64GB
- **CPU:** Count-16, Model-Intel(R) Xeon(R) Gold 6252 CPU @ 2.10GHz
- **Dataset:** Synthetic sales dataset with 160000000 rows, each containing product ID, category, region, and sales amount.
- **Operations:** Loading, selection, filtering, aggregation (mean, sum)
- **Data Source:** https://sftpweb.nectechnologies.in/web/client/pubshares/sUv3vivCDAZVASorFyAmqd

# Memory Footprint:

homepage

## FireDucks:

FireDucks matches Polars stride for stride in peak memory utilization, distinguishing itself with a strategic allocation approach. Thanks to its 'lazy evaluation model,' FireDucks conserves resources by allocating memory exclusively on demand (in this example memory is allocated only when '_evaluate()' method is called).
Note: As explained in the FireDucks homepage, the '_evaluate()' method has been used in the below example to measure the actual execution time of read_csv() by disabling the lazy execution mode of FireDucks in our benchmark program.

**Python:**

```
import fireducks.pandas as fd
GB = 1000000000

df_fireduck = fd.read_csv(sales_data.csv)._evaluate()
print("Fireduck memory usage:", df_fireduck.memory_usage(index=True).sum()/GB, " GBs")
```

```
Fireduck memory usage: 5.120000128  GBs
```

## Polars:

Polars operates by loading data in memory immediately, which shows up as increased memory allocation at the start of the function. This in-memory strategy, while resource-intensive, can often result in quicker data processing times for subsequent operations.

**Python:**

```
import polars as pl
GB = 1000000000

df_polar = pl.read_csv("sales_data.csv")
print("Polars memory usage:", df_polar.estimated_size()/GB, " GBs")
```

```
Polars memory usage: 8.96  GBs
```

# Running Time: Running Time comparison for the following

- Loading
- Selection & Filtering
- Aggregation

We can use the following Pandas code for evaluation. No changes would be required in this code for FireDucks, we would be executing same Pandas code using execution hooks from FireDucks.
However, changes would be required in Polars, as outlined in the subsequent section.

## #Pandas Sample Code

```
1   import time
3   import pandas as pd
4
5   target = f"{pd.__name__}_ver_{pd.__version__}"
6
7   stime = time.time()
9
10  t0 = time.time()
11  df = pd.read_csv('sales_data.csv')
12  read_t = time.time() - t0
13  print(f"[{target}] read-csv-time: {read_t} sec")
14
15  t0 = time.time()
16  fdf = df[(df['region'] == 'east') & (df['category'] == 'Electronics') & (df['sales_amount'] > 550)]
17  filter_t = time.time() - t0
18  print(f"[{target}] data-filtration-time: {filter_t} sec")
19
20  t0 = time.time()
21  avg = df["sales_amount"].mean()
22  avg_t = time.time() - t0
23  print(f"[{target}] avg-calc-time: {avg_t} sec")
24
25  t0 = time.time()
26  grp_sum = df.groupby('region').sum()
27  grp_sum_t = time.time() - t0
28  print(f"[{target}] group-wise-sum-calc-time: {grp_sum_t} sec")
29
30  total_t = time.time() - stime
31  print(f"[{target}] total-execution-time: {total_t} sec")
32
```

## #Pandas Execution Time

[pandas_ver_1.5.3] read-csv-time: 120.5727264881134 sec

[pandas_ver_1.5.3] data-filtration-time: 16.80004382133484 sec

[pandas_ver_1.5.3] avg-calc-time: 0.4351644515991211 sec

[pandas_ver_1.5.3] group-wise-sum-calc-time: 9.41200041770935 sec

[pandas_ver_1.5.3] total-execution-time: 147.22016096115112 sec

## #FireDucks Sample Code

```
1
3   We have executed the same program written for pandas as follows to evaluate the same for
4   FireDucks:
5   FIREDUCKS_FLAGS="--benchmark-mode" python -mfireducks.imhook sample1.py
6
7   The --benchmark-mode flag is specified to disable its lazy execution mode to benchmark each
9   API correctly. We don't even need to modify the import statement by specifying
10  -mfireducks.imhook option.
11
12
13
15
16
17
18
19
20
```

## #FireDucks Execution Time

[fireducks.pandas_ver_0.10.3] read-csv-time: 5.116232633590698 sec

[fireducks.pandas_ver_0.10.3] data-filtration-time: 0.473590612411499 sec

[fireducks.pandas_ver_0.10.3] avg-calc-time: 0.05000877380371094 sec

[fireducks.pandas_ver_0.10.3] group-wise-sum-calc-time: 0.42878246307373047 sec

[fireducks.pandas_ver_0.10.3] total-execution-time: 6.069048643112183 sec

## #Polars Sample Code

```
1
3   #For Polars make the following two changes:
4   # 1. Change in import statement:
5   #     import pandas as pd -> import polars as pl
6   # 2. Change in filter statement:
7   #       fdf = df[(df['region'] == 'east') & (df['category'] == 'Electronics') & (df['sales_amount'] >
9   550)]
10              -> fdf = df.filter((pl.col('region') == 'east') & (pl.col('category') == 'Electronics') &
11  (pl.col('sales_amount') > 550))
12  # 3. Change groupby to group_by
13
15
16
17
18
19
20
```

## #Polars Execution Time

[polars_ver_0.20.7] read-csv-time: 4.136218309402466 sec

[polars_ver_0.20.7] data-filtration-time: 3.3097283840179443 sec

[polars_ver_0.20.7] avg-calc-time: 0.15314102172851562 sec

[polars_ver_0.20.7] group-wise-sum-calc-time: 2.512341022491455 sec

[polars_ver_0.20.7] total-execution-time: 10.111685276031494 sec

# Key Takeaways:

When it comes to evaluating FireDucks and Polars for data manipulation and analysis tasks, several key insights have emerged:

- When it comes to **memory management**, FireDucks consistently shows greater efficiency, a trait that becomes particularly advantageous with the increase in dataset size.
- In the realm of **data loading**, FireDucks displays remarkable speed, especially notable when dealing with voluminous datasets.
- For tasks involving **data selection and filtering**, both FireDucks and Polars perform adeptly, yet FireDucks leads with a noticeably superior edge.
- With **aggregations**, FireDucks not only surpasses Polars but also demonstrates a widening advantage as the intricacy of the operations escalates.
- FireDucks can readily execute an existing Pandas code but Polars requires syntactical changes to run Pandas code. This **high-level of API compatibility** makes it very comfortable for a pandas user since no manual effort is required to speed-up an existing pandas application using FireDucks.

| API | FireDucks speedup over Pandas | FireDucks speedup over Polars |
|---|---|---|
| read-csv speedup | 23.57x | 0.81x |
| data-filtration speedup | 35.74x | 7.33x |
| avg-calc speedup | 8.7x | 3.06x |
| group-wise-sum-calc speedup | 21.99x | 5.87x |
| total-execution speedup | 24.26x | 1.67x |

# Conclusion: Choosing the Right Tool

Your optimal selection hinges on your particular requirements. Should memory efficiency be paramount, especially with expansive datasets, FireDucks emerges as the prime candidate. Also if priority is speed, then Fireducks performs better than Polars, and this improves further with increased complexity of operations. It's essential to weigh the nature of your routine operations and rigorously evaluate both libraries with your specific data. FireDucks, with its robust performance in memory management and processing, often becomes the library of choice for those seeking to strike a balance between speed and efficient memory usage.