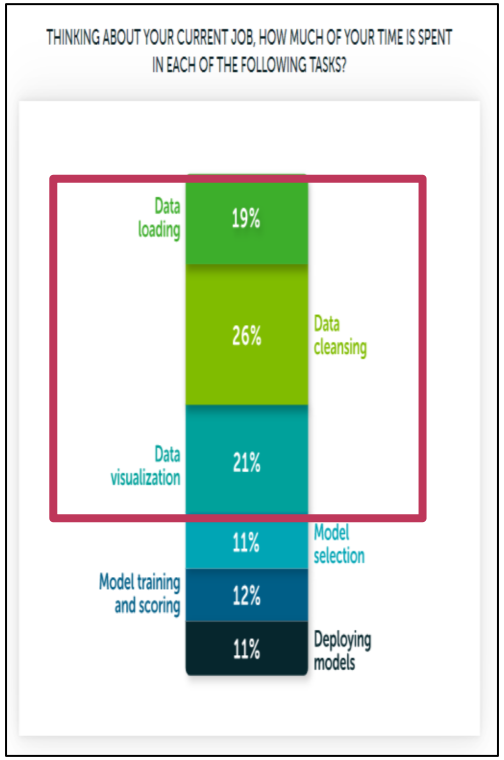
FireDucks: Pandas Accelerator using MLIR

September 28, 2024

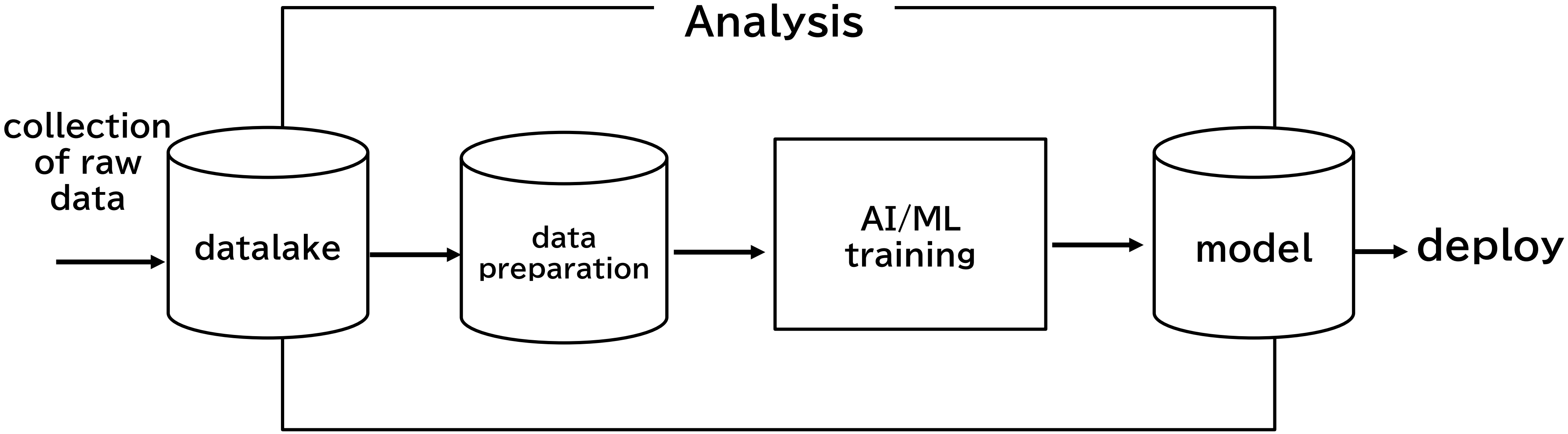
Ashu Thakur (NEC), Sourav Saha (NEC), Kazuhisa Ishizaka (NEC)

Workflow of a Data Scientist

almost 75% efforts of a Data Scientist spent on data preparation

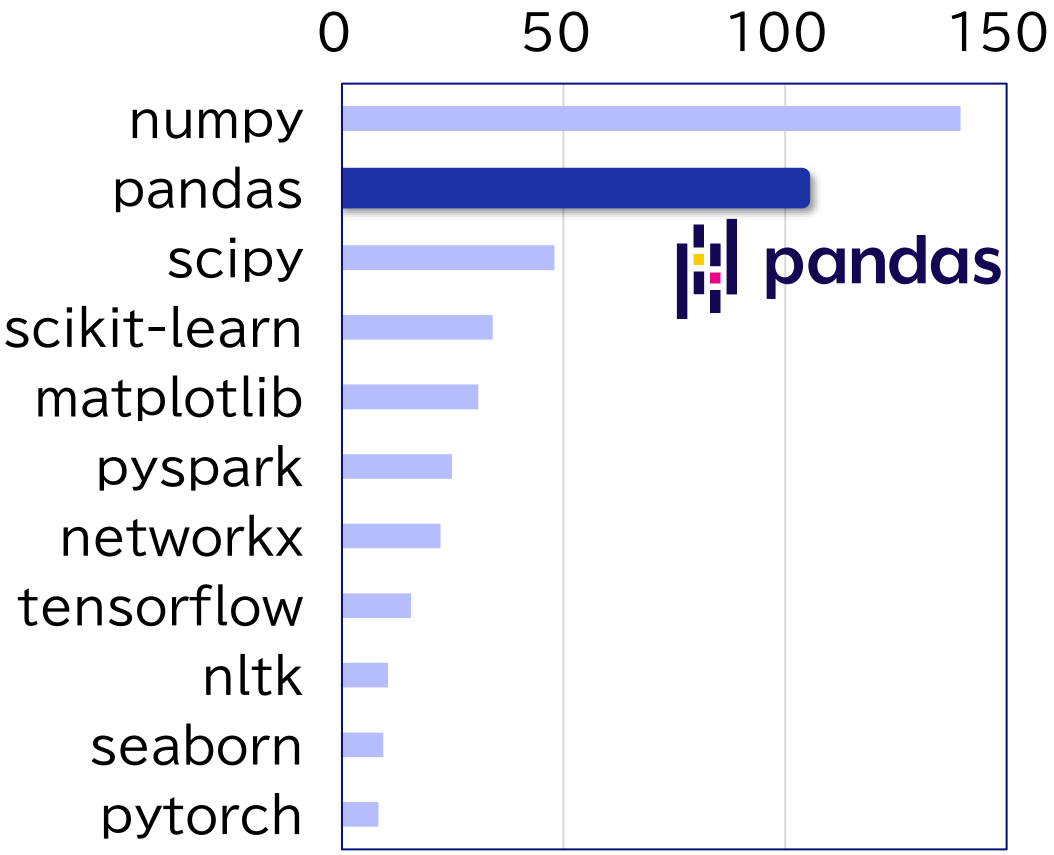


Anaconda:
The State of Data Science 2020



Pandas: Its Pros and Cons

◆ **Most popular Python library for data analytics.**



Monthly download from pypi.org
(Data Analytics Libraries)

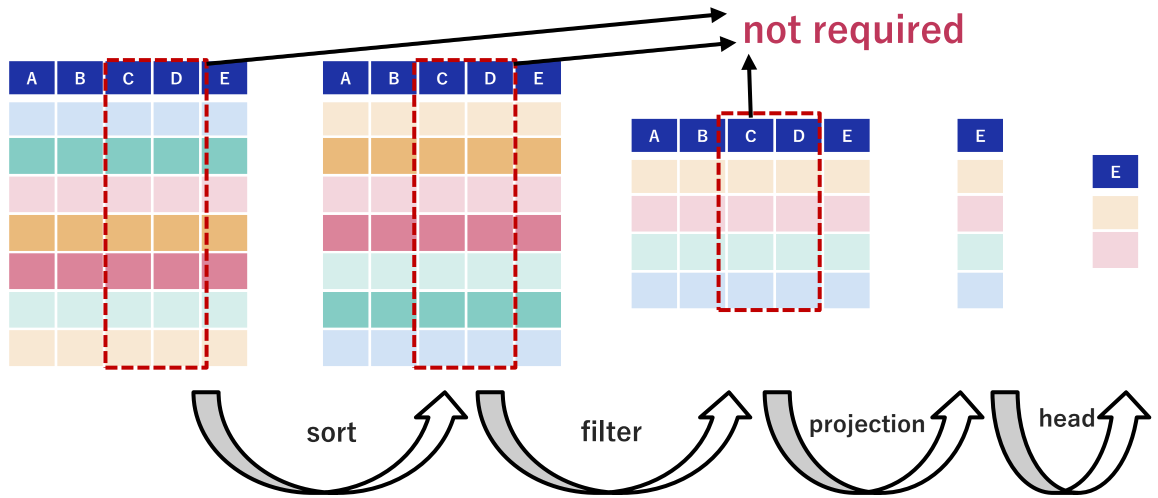
- pandas drawbacks:
  - It (mostly) doesn't support parallel computation.
  - The choice of API heavily impacts the performance of a pandas application.
  - Very slow execution reduces the efficiency of a data analyst.
  - Long-running execution
    - produces higher cloud costs
    - attributes to higher CO2 emission

The way of implementing a query in pandas-like library (that does not support query optimization) heavily impacts its performance!!

Execution order matters to boost the performance of a data analysis tool

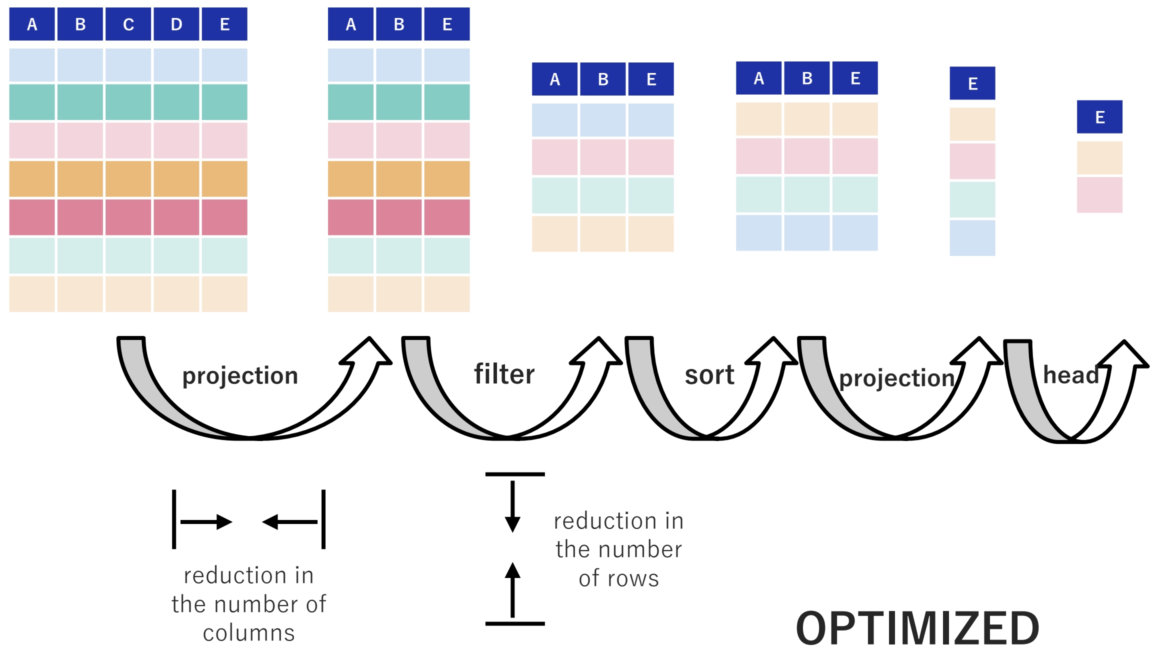
```
df.sort_values("A")
  .query("B > 1")["E"]
  .head(2)
```

※ sort-order: yellow->red->green->blue



SAMPLE QUERY

```
df.loc[:, ["A", "B", "E"]]
  .query("B > 1")
  .sort_values("A")["E"]
  .head(2)
```



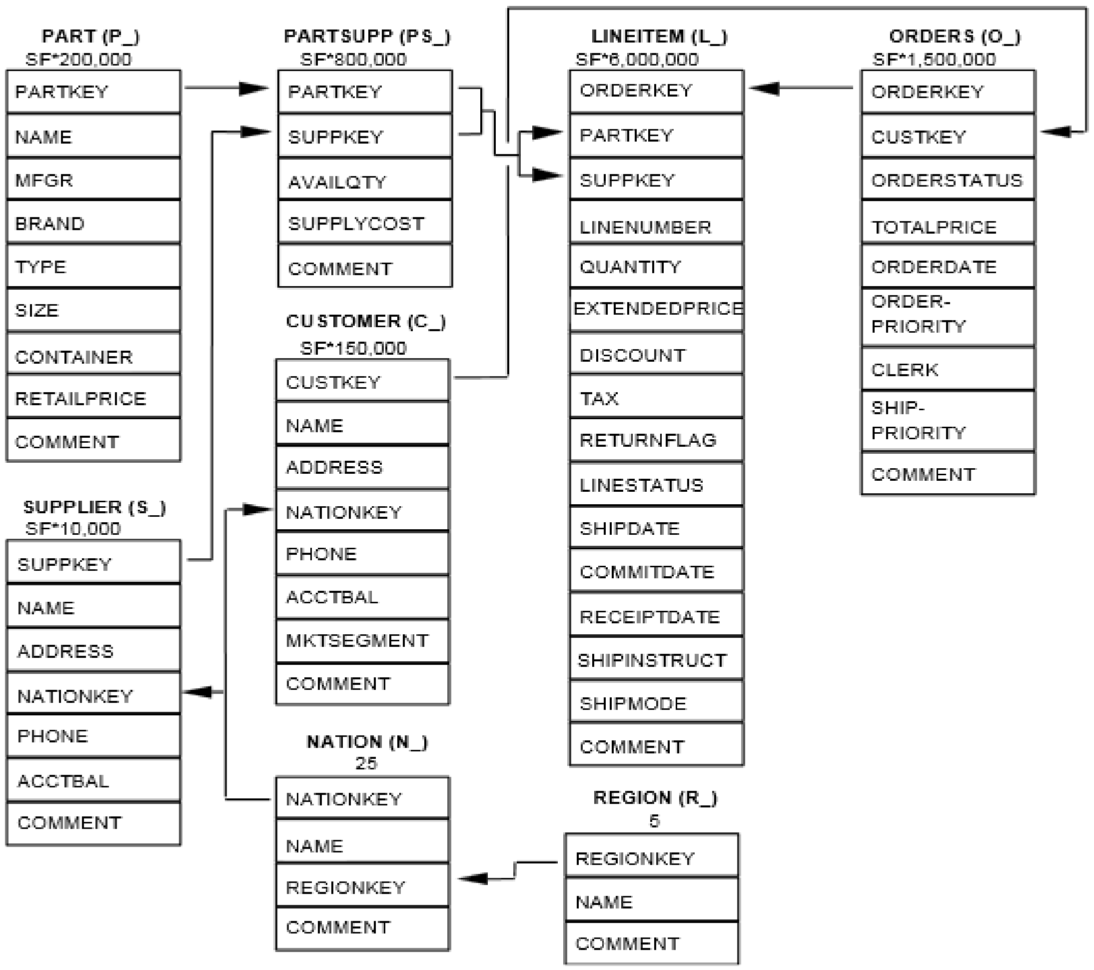
OPTIMIZED QUERY

Exercise: Query #3 from TPC-H Benchmark (SQL -> pandas)

- [query to retrieve the 10 unshipped orders with the highest value.]()

```
SELECT l_orderkey,
               sum(l_extendedprice * (1 - l_discount)) as revenue,
               o_orderdate,
               o_shippriority
FROM customer, orders, lineitem
WHERE
    c_mktsegment = 'BUILDING' AND
    c_custkey = o_custkey AND
    l_orderkey = o_orderkey AND
    o_orderdate < date '1995-03-15' AND
    l_shipdate > date '1995-03-15’
GROUP BY l_orderkey, o_orderdate, o_shippriority
ORDER BY revenue desc, o_orderdate
LIMIT 10;
```

```
rescols = ["l_orderkey", “revenue”, "o_orderdate", "o_shippriority“]
result = (
 customer.merge(orders, left_on="c_custkey", right_on="o_custkey")
  .merge(lineitem, left_on="o_orderkey", right_on="l_orderkey")
  .pipe(lambda df: df[df[“c_mktsegment"] == "BUILDING"])
  .pipe(lambda df: df[df["o_orderdate"] < datetime(1995, 3, 15)])
  .pipe(lambda df: df[df["l_shipdate"] > datetime(1995, 3, 15)])
  .assign(revenue=lambda df: df["l_extendedprice"] * (1 - df["l_discount"]))
  .groupby(["l_orderkey", "o_orderdate", "o_shippriority"], as_index=False)
  .agg({"revenue": "sum"})[rescols]
  .sort_values(["revenue", "o_orderdate"], ascending=[False, True])
  .head(10)
)
```



Exercise: Query #3 from TPC-H Benchmark (pandas -> optimized pandas)

```
rescols = ["l_orderkey", “revenue”, "o_orderdate", "o_shippriority“]
result = (
 customer.merge(orders, left_on="c_custkey", right_on="o_custkey")
  .merge(lineitem, left_on="o_orderkey", right_on="l_orderkey")
  .pipe(lambda df: df[df[“c_mktsegment"] == "BUILDING"])
  .pipe(lambda df: df[df["o_orderdate"] < datetime(1995, 3, 15)])
  .pipe(lambda df: df[df["l_shipdate"] > datetime(1995, 3, 15)])
  .assign(revenue=lambda df: df["l_extendedprice"] * (1 - df["l_discount"]))
  .groupby(["l_orderkey", "o_orderdate", "o_shippriority"], as_index=False)
  .agg({"revenue": "sum”})[rescols]
  .sort_values(["revenue", "o_orderdate"], ascending=[False, True])
  .head(10)
)
```

Exec-time: 68.55 s

Scale Factor: 10

6.5x

Exec-time: 10.33 s

```
# projection-filter: to reduce scope of “customer” table to be processed
cust = customer[[“c_custkey”, “c_mktsegment"]]
f_cust = cust[cust[“c_mktsegment"] == "BUILDING"]

# projection-filter: to reduce scope of “orders” table to be processed
ord = orders[[“o_custkey”, “o_orderkey”, “o_orderdate”, "o_shippriority"]]
f_ord = ord[ord["o_orderdate"] < datetime(1995, 3, 15)]

# projection-filter: to reduce scope of “lineitem” table to be processed
litem = lineitem[["l_orderkey“, "l_shipdate“, "l_extendedprice“, "l_discount"]]
f_litem = litem[litem["l_shipdate"] > datetime(1995, 3, 15)]

rescols = ["l_orderkey", “revenue”, "o_orderdate", "o_shippriority“]
result = ( f_cust.merge(f_ord, left_on="c_custkey", right_on="o_custkey")
  .merge(f_litem, left_on="o_orderkey", right_on="l_orderkey")
  .assign(revenue=lambda df: df["l_extendedprice"] * (1 - df["l_discount"]))
  .pipe(lambda df: df[rescols])
  .groupby(["l_orderkey", "o_orderdate", "o_shippriority"], as_index=False)
  .agg({"revenue": "sum"})[rescols]
  .sort_values(["revenue", "o_orderdate"], ascending=[False, True])
  .head(10)
)
```

Idea #1

- **Can such optimization be automated?**
  - Yes, we can define specialized intermediate representation (IR) for each pandas API using LLVM/MLIR.
  - we can implement define-by-run mechanism to generate the IRs from the pandas APIs.
  - the IRs can then be optimized to implement different domain-specific optimizations, such as projection pushdown, predicate pushdown, etc.
  - the optimized IRs can be translated back to the pandas API.

```
df.sort_values(“A”)
  .query(“B > 1”)[“E”]
  .head(2)
```

pandas API to intermediate representation (IR)

```
%v2 = “sort_values_op"(%v1, “A")
%v3 = "filter_op"(%v2, “B > 1”)
%v4 = “project_op”(%v3, [“E”])
%v5 = “slice_op”(%v4, 2)
```

MLIR

JIT optimization

input IR

optimized IR

```
%t1 = “project_op”(%v1, [“A”, “B”, “E”])
%t2 = "filter_op"(%t1, “B > 1”)
%t3 = “sort_values_op"(%t2, “A")
%t4 = “project_op”(%t3, [“E”])
%t5 = “slice_op”(%t4, 2)
```

intermediate representation (IR) to pandas API

```
df.loc[:, [“A”, “B”, “E”]]
  .query(“B > 1”)
  .sort_values(“A”)[“E”]
  .head(2)
```

Idea #2

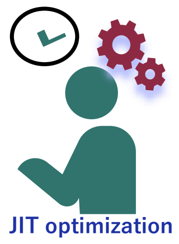
- Pandas methods are slow due to poor memory utilization and single-core computation.
- But pandas is one of the most popular data manipulation tools.
- **How can we solve the core performance issue in pandas while keeping the same API for users?**
  - Well, we can
    - have a frontend with pandas API that generates IR.
    - develop our own library parallelizing the workload of DataFrame-related methods as a backend.
    - translate the optimized IRs to the **backend library API** (instead of pandas API).

```
df.sort_values(“A”)
  .query(“B > 1”)[“E”]
  .head(2)
```

frontend pandas API to intermediate representation (IR)

```
%v2 = “sort_values_op"(%v1, “A")
%v3 = "filter_op"(%v2, “B > 1”)
%v4 = “project_op”(%v3, [“E”])
%v5 = “slice_op”(%v4, 2)
```

MLIR



input IR

optimized IR

```
%t1 = “project_op”(%v1, [“A”, “B”, “E”])
%t2 = "filter_op"(%t1, “B > 1”)
%t3 = “sort_values_op"(%t2, “A")
%t4 = “project_op”(%t3, [“E”])
%t5 = “slice_op”(%t4, 2)
```

intermediate representation (IR) to backend API

```
t1 = backend::project_columns(df, {“A”, “B”, “C”});
t2 = backend::filter_rows(t1, “B > 1”);
t3 = backend::sort_values(t2, “A”);
t4 = backend::project_columns(t3, {“E”});
t5 = backend::slice_rows(t4, 2);
```

Introducing FireDucks

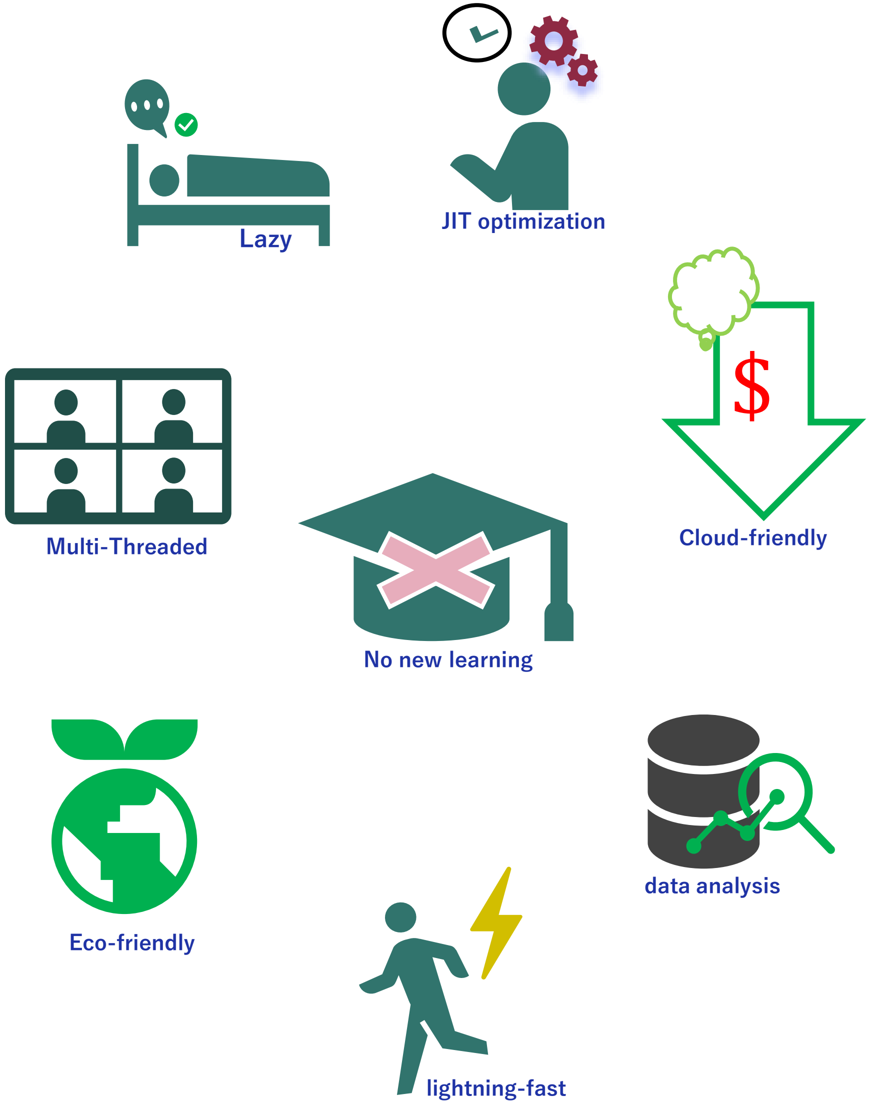
FireDucks (**F**lexible **IR E**ngine for DataFrame) is a high-performance compiler-accelerated DataFrame library with highly compatible pandas APIs.

Speed: significantly faster than pandas

- FireDucks is multithreaded to fully exploit the modern processor
- Lazy execution model with Just-In-Time optimization using a defined-by-run mechanism supported by MLIR (a subproject of LLVM).
    - supports both lazy and non-lazy execution models without modifying user programs (same API).

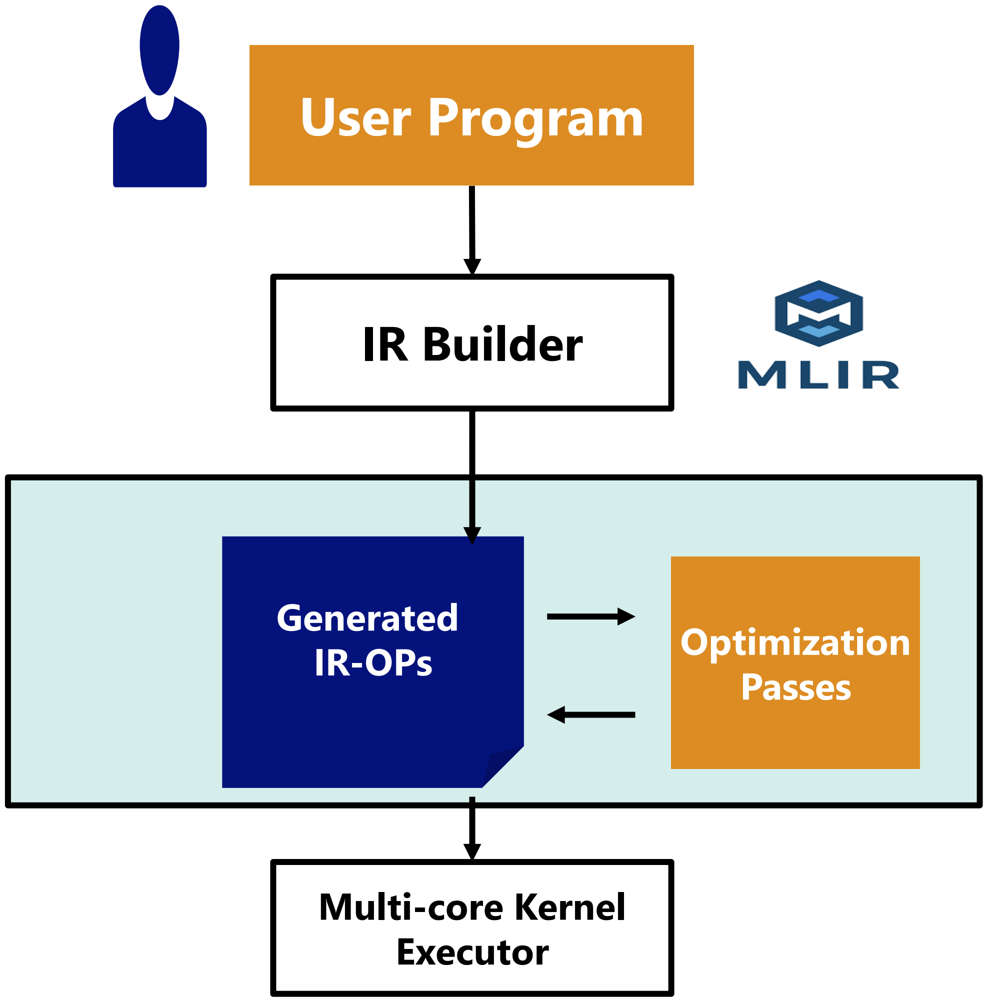
Ease of use: drop-in replacement of pandas

- FireDucks is highly compatible with pandas API
    - seamless integration is possible not only for an existing pandas program but also for any external libraries (like seaborn, scikit-learn, etc.) that internally use pandas dataframes.
- No extra learning is required
- No code modification is required



How does FireDucks work?

※IR: Intermediate Representation




```
sorted = df.sort_values("b")
result = sorted["a"]
```

```
%v2 = "fireducks.sort_values"(%v1,"b")
%v3 = "fireducks.project"(%v2,["a"])
```

print (result)

```
%v11 = "fireducks.project"(%v1,["a","b"])
%v2 = "fireducks.sort_values"(%v11,"b")
%v3 = "fireducks.project"(%v2,["a"])
```

```
tmp = df[["a","b"]]
sorted = tmp.sort_values("b")
result = sorted["a"]
```

Primary Objective: <u>Write Once, Execute Anywhere</u>

Let’s Have a Quick Demo!

```
pd.read_csv(“data.csv”).rolling(60).mean()[“Close”].tail(1000).plot()
```

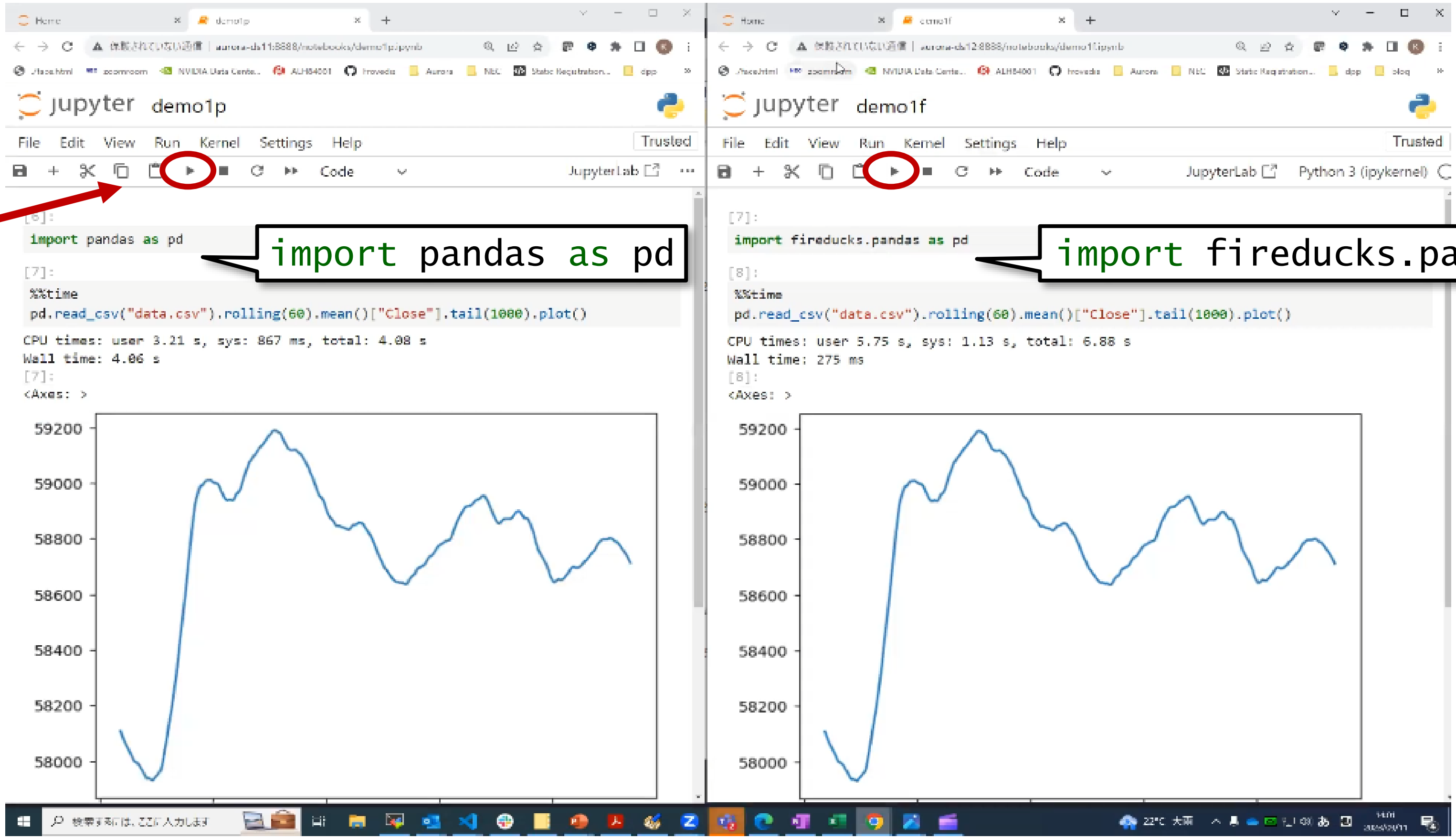
Program to calculate moving average

pandas the difference is only in the import **FireDucks**

button to start execution

```
import pandas as pd
```

```
import fireducks.pandas as pd
```



data.csv: [Bitcoin Historical Data]

pandas: 4.06s

~15x

FireDucks: 275ms

Usage of FireDucks

1. Explicit Import

easy to import

```
# import pandas as pd
import fireducks.pandas as pd
```

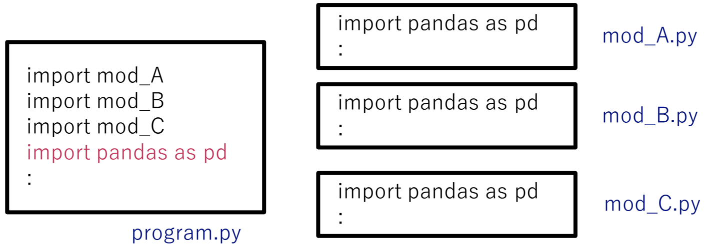
simply change the import statement

2. Import Hook

FireDucks provides command line option to automatically replace “**pandas**” with “**fireducks.pandas**”

```
$ python -m fireducks.pandas program.py
```

zero code modification



3. Notebook Extension

FireDucks provides simple import extension for interative notebooks.

```
%load_ext fireducks.pandas
import pandas as pd
```

simple integration in a notebook

Benchmark (1): DB-Benchmark

Database-like ops benchmark (https://duckdblabs.github.io/db-benchmark)

groupby join

0.5 GB 5 GB 50 GB

basic questions

Input table: 1,000,000,000 rows x 9 columns (50 GB)

rank-1

FireDucks	1.0.4	2024-09-10	15s
DuckDB	1.0.0	2024-07-04	25s
ClickHouse	24.5.1.1763	2024-06-07	28s
Polars	1.1.0	2024-07-09	47s
Datafusion	38.0.1	2024-06-07	56s
data.table	1.15.99	2024-06-07	88s
DataFrames.jl	1.6.1	2024-06-07	91s
InMemoryDatasets	0.7.18	2023-10-17	218s
spark	3.5.1	2024-06-07	261s
R-arrow	16.1.0	2024-06-07	378s
collapse	2.0.14	2024-06-07	411s
(py)datatable	1.2.0a0	2024-06-07	1022s
dplyr	1.1.4	2024-06-07	1104s
pandas	2.2.2	2024-06-07	1126s
dask	2024.5.2	2024-06-07	out of memory
Modin		see README	pending

groupby join

0.5 GB 5 GB 50 GB

basic questions

Input table: 100,000,000 rows x 7 columns (5 GB)

rank-1

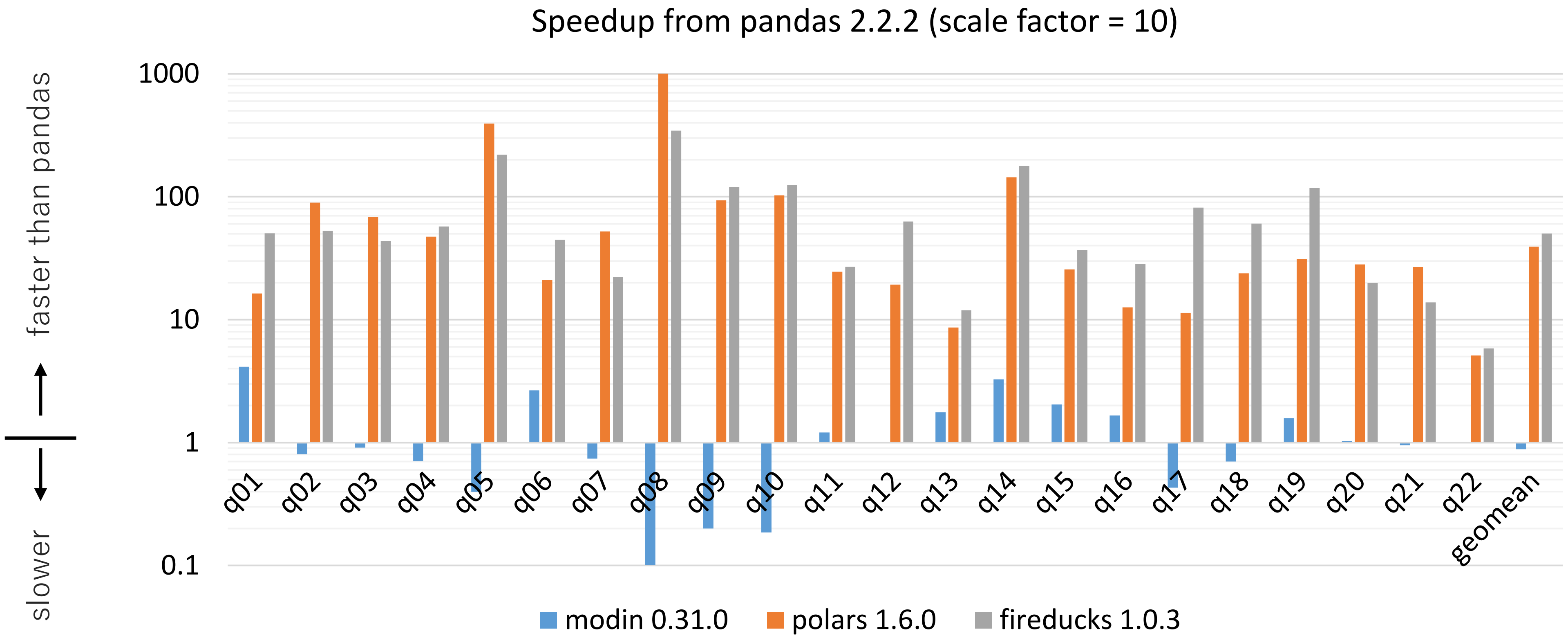
FireDucks	1.0.4	2024-09-10	7s
DuckDB	1.0.0	2024-07-04	9s
Polars	1.1.0	2024-07-08	9s
Datafusion	38.0.1	2024-06-07	15s
InMemoryDatasets	0.7.18	2023-10-20	25s
ClickHouse	24.5.1.1763	2024-06-07	43s
data.table	1.15.99	2024-06-07	62s
collapse	2.0.14	2024-06-07	69s
DataFrames.jl	1.6.1	2024-06-07	77s
spark	3.5.1	2024-06-07	128s
dplyr	1.1.4	2024-06-07	214s
pandas	2.2.2	2024-06-07	244s
dask	2024.5.2	2024-06-07	635s
(py)datatable	1.2.0a0	2024-06-07	undefined exception
R-arrow	16.1.0	2024-06-07	out of memory
Modin		see README	pending

Benchmark (2): Speedup from pandas in TPC-H benchmark

FireDucks is ~345x faster than pandas at max

Server

Xeon Gold 5317 x2
(24 cores), 256GB



Comparison of DataFrame libraries (average speedup)

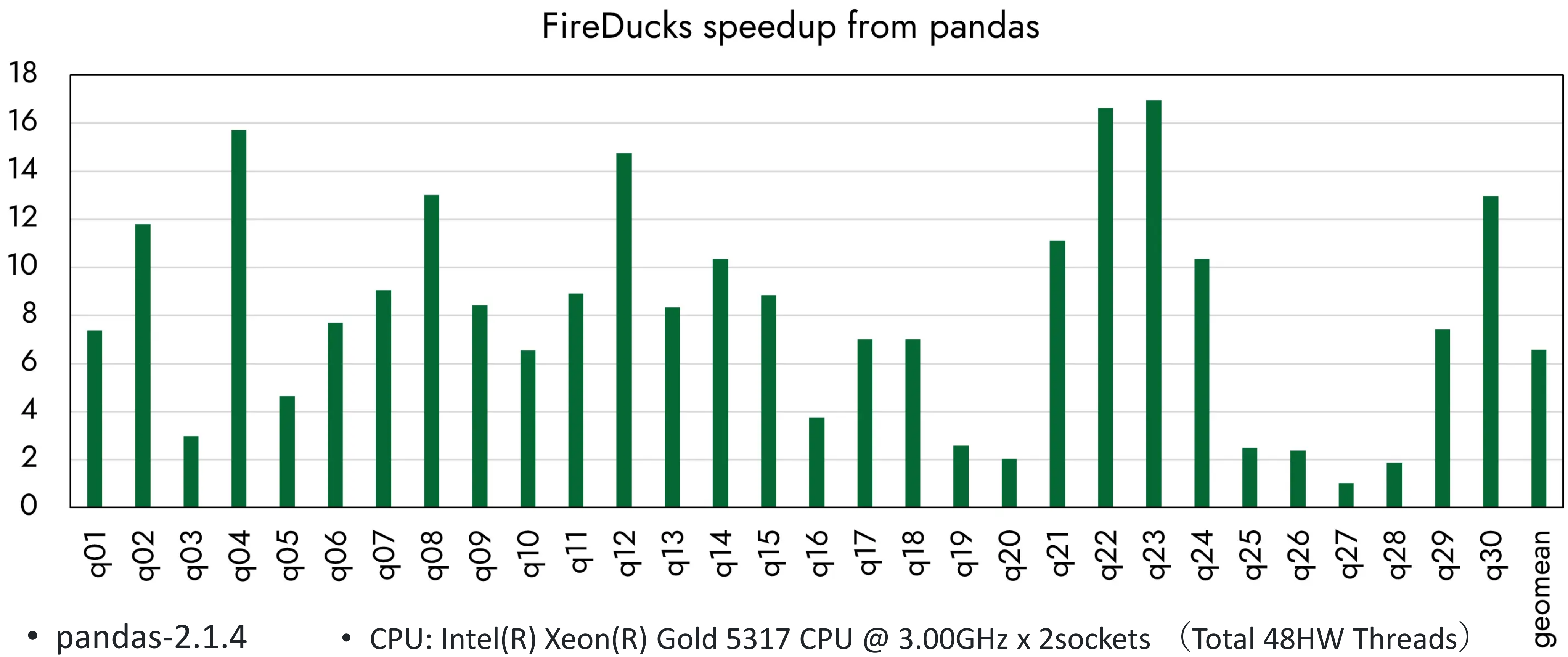
FireDucks 50x

Polars 39x

Modin 0.9x

Benchmark (3): Speedup from pandas in TPCx-BB benchmark

ETL(Extract, Transform, Load) and ML Workflow



- pandas-2.1.4
- fireducks-0.9.3

- CPU: Intel(R) Xeon(R) Gold 5317 CPU @ 3.00GHz x 2sockets （Total 48HW Threads）
- Main memory: 256GB

Resource on FireDucks

Web site (User guide, benchmark, blog)

https://fireducks-dev.github.io/

X(twitter) (Release information)

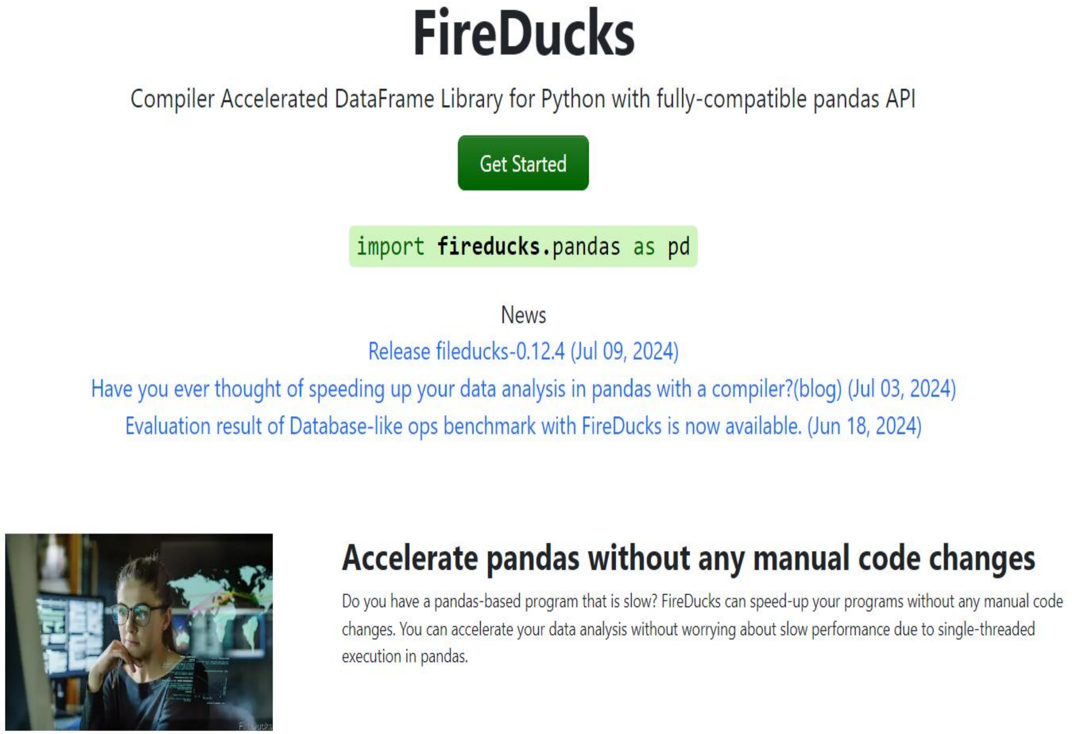
https://x.com/fireducksdev

Github (Issue report)

https://github.com/fireducks-dev/fireducks

Q/A, communication

https://join.slack.com/t/fireducks/shared_invite/zt-2j4lucmtj-IGR7AWIXO62Lu605pnBJ2



Let’s go for a test drive!

https://colab.research.google.com/drive/1qpej-X7CZsIeOqKuhBg4kq-cbGuJf1Zp?usp=sharing

Thank You!

- Focus more on in-depth data exploration using “**pandas**”.
- Let the “**FireDucks**” take care of the optimization for you.
- Enjoy Green Computing!