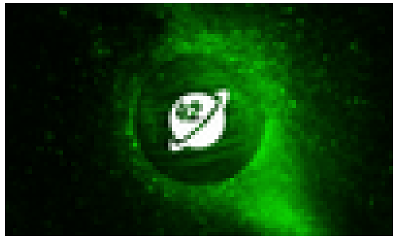
\Orchestrating a brighter world



Introducing FireDucks:
A Multithreaded DataFrame Library with JIT compiler

February 06, 2025

Sourav Saha (NEC)

Agenda

- About Pandas
- Tips and Tricks for Optimizing Large-scale Data processing workload
- FireDucks and Its Offerings
- FireDucks Optimization Strategy
- Evaluation Benchmarks
- Resources on FireDucks
- Test Drive
- FAQs

Quick Introduction!

SOURAV SAHA – Research Engineer @ NEC Corporation

https://www.linkedin.com/in/sourav-%E3%82%BD%E3%82%A6%E3%83%A9%E3%83%96-saha-%E3%82%B5%E3%83%8F-a5750259/

https://twitter.com/SouravSaha97589

Hello, I am a software professional with 11+ years of working experience across diverse areas of **HPC, Vector Supercomputing, Distributed Programming, Big Data and Machine Learning**. Currently, my team at NEC R&D Lab, Japan, is researching various data processing-related algorithms. Blending the mixture of different niche technologies related to compiler framework, high-performance computing, and multi-threaded programming, we have developed a Python library named FireDucks with highly compatible pandas APIs for DataFrame-related operations.

https://www.nec.com/en/global/solutions/hpc/sx/index.html

Mr. Kazuhisa Ishizaka (Primary Author)

we wanted to develop some library using compiler technology

we wanted to speed-up python

Data Scientists often face issues with slow performance of pandas

User Program

pandas API

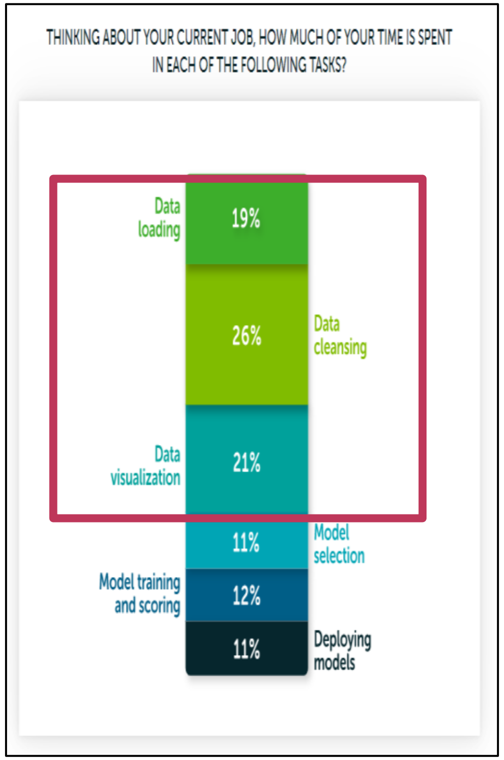
FireDucks

groupby | **join** | **dropna** | **filter** | **sort** | **corr**

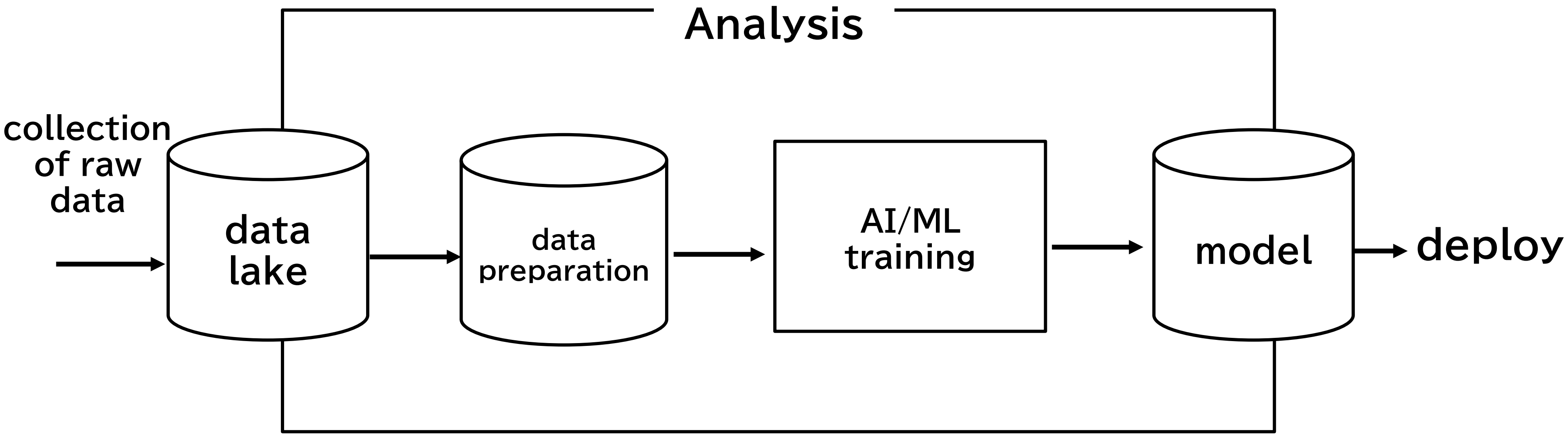
compiler technologies

Workflow of a Data Scientist

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

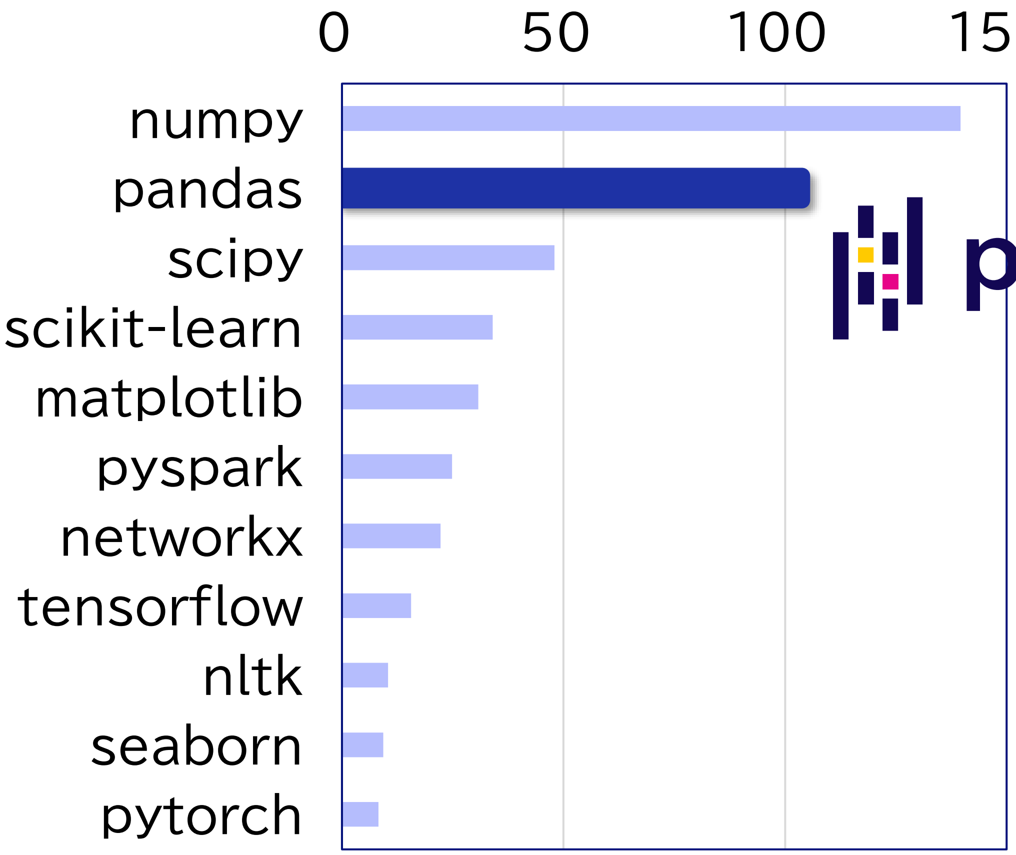


Anaconda:
The State of Data Science 2020



About Pandas (1/2)

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

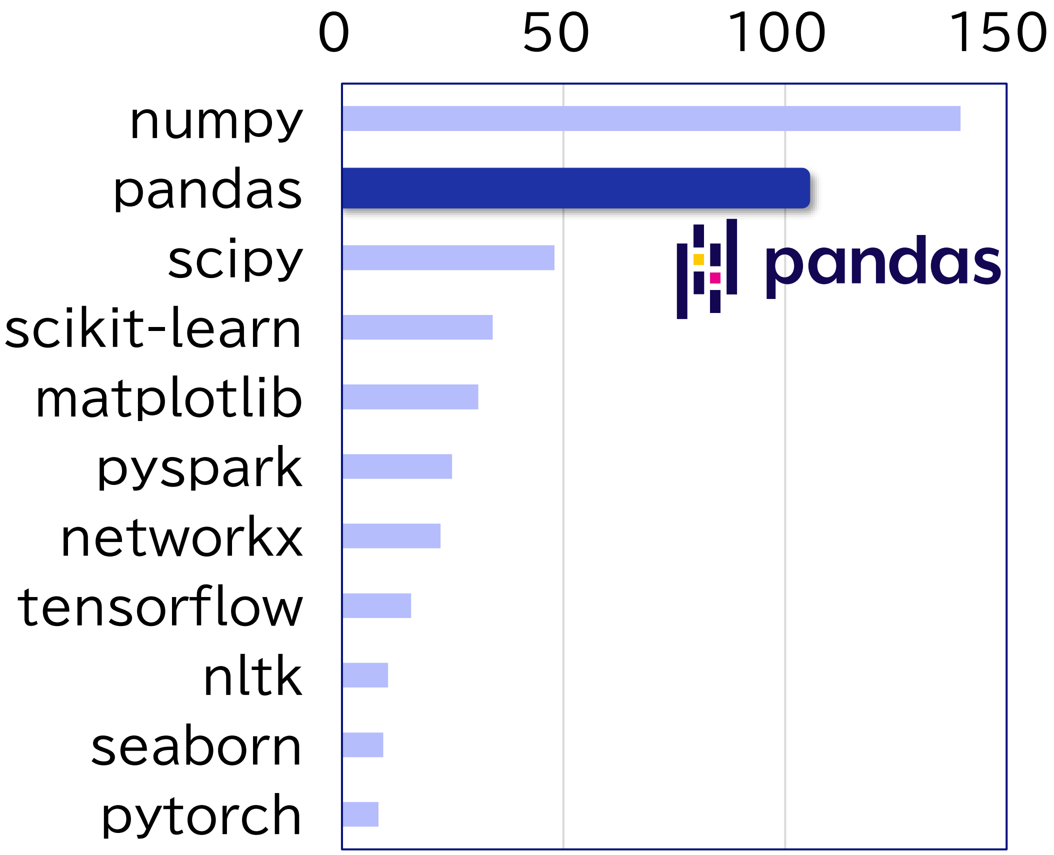


Monthly download from pypi.org
(Data Analytics Libraries)

- It (mostly) doesn't support parallel computation.
- It doesn't have any auto-optimization feature.
- Hence, it is not suitable for processing large datasets.
- Very slow execution reduces the efficiency of a data analyst.
- Long-running execution
  - produces higher cloud costs
  - attributes to higher CO2 emission

About Pandas (2/2)

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



Monthly download from pypi.org
(Data Analytics Libraries)

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

- We will discuss a couple of approaches to improve the performance related to computational time and memory of a query written in pandas, when processing large-scale data.
- We will also discuss how those approaches can be automated using compiler technologies.

Performance Challenges & Best Practices to follow

Quiz: Which one is a better code?

```
def foo(filename):
  df = pd.read_csv(filename)
  t1 = df.drop_duplicates()
  t2 = t1.sort_values(“B”)
  t3 = t2.head(2)
  return t3
```

OR

```
def foo(filename):
  return (
    pd.read_csv(filename)
      .drop_duplicates()
      .sort_values(“B”)
      .head(2)
  )
```

Best Practice (1): importance of chained expression

```
def foo(filename):
  df = pd.read_csv(filename)
  t1 = df.drop_duplicates()
  t2 = t1.sort_values("B")
  t3 = t2.head(2)
  return t3
```

df: ~16 GB

A	B	C
u	0.91	1
a	1.00	4
a	1.00	4
o	0.24	0
o	0.24	0
e	0.43	1
u	0.91	1
e	0.20	2
o	0.24	0
a	1.00	4

drop_duplicates

t1: ~8 GB

A	B	C
u	0.91	1
a	1.00	4
o	0.24	0
e	0.43	1
e	0.20	2

sort

t3: ~8 GB

A	B	C
a	1.00	4
u	0.91	1
e	0.43	1
o	0.24	0
e	0.20	2

head(2)

t4: ~x KB

A	B	C
a	1.00	4
u	0.91	1

re-write using chained expression

```
def foo(filename):
  return (
    pd.read_csv(filename)
      .drop_duplicates()
      .sort_values("B")
      .head(2)
  )
```

A	B	C
u	0.91	1
a	1.00	4
a	1.00	4
o	0.24	0
o	0.24	0
e	0.43	1
u	0.91	1
e	0.20	2
o	0.24	0
a	1.00	4

drop_duplicates

A	B	C
u	0.91	1
a	1.00	4
o	0.24	0
e	0.43	1
e	0.20	2

sort

A	B	C
a	1.00	4
u	0.91	1
e	0.43	1
o	0.24	0
e	0.20	2

head(2)

A	B	C
a	1.00	4
u	0.91	1

Quiz: Which one is a better code?

```
res = df.sort_values(by=“B”)[“A”].head()
```

OR

```
tmp = df[[“A”, “B”]]
res = tmp.sort_values(by=“B”)[“A”].head()
```

Domain Specific Optimization: Projection Pushdown

index	a	b	c	d	e	f	g	h	i	j
0	1	A	1	3	4	2	4	1	3	7
1	6	B	2	3	6	3	4	7	8	4
2	5	D	2	4	7	2	3	3	7	8
3	2	A	3	2	8	5	3	2	4	5
4	3	C	5	9	2	3	2	6	2	6
5	8	B	8	1	5	7	1	5	8	3

tmp = df[[“a”, “b”]]

index	a	b
0	1	A
1	6	B
2	5	D
3	2	A
4	3	C
5	8	B

sorted = df.sort_values(“b”)
-> sidx = [0, 3,1, 5, 4, 2] # get sorted index
-> sorted = df.take(sidx) # materialize result

index	a	b	c	d	e	f	g	h	i	j
0	1	A	1	3	4	2	4	1	3	7
3	2	A	3	2	8	5	3	2	4	5
1	6	B	2	3	6	3	4	7	8	4
5	8	B	8	1	5	7	1	5	8	3
4	3	C	5	9	2	3	2	6	2	6
2	5	D	2	4	7	2	3	3	7	8

Waste of computational memory and execution time

result = sorted[“a”]

index	a
0	1
3	2
1	6
5	8
4	3
2	5

sorted = tmp.sort_values(“b”)
-> sidx = [0, 3, 1, 5, 4, 2]
-> sorted = tmp.take(sidx)

index	a	b
0	1	A
3	2	A
1	6	B
5	8	B
4	3	C
2	5	D

result = sorted[“a”]

index	a
0	1
3	2
1	6
5	8
4	3
2	5

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

→

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

projection pushdown

Quiz: What is the performance issue with this data flow?

ID	E_Name	Gender	C_Code
1	A	Male	1
2	B	Male	1
3	C	Female	2
4	E	Male	2
5	F	Female	1
6	G	Female	2
7	H	Male	1
8	I	Female	2

employee

C_Code	C_Name
1	India
2	Japan

country

```
m = employee.merge(country, on="C_Code")
f = m[m["Gender"] == "Male"]
r = f.groupby("C_Name")["E_Name"].count()
print(r)
```

- sample case: **filter after merge operation**
  - merge is an expensive operation, as it involves data copy.
  - performing merge operation on a large dataset and then filtering the output would involve unnecessary costs in data-copy.

merge

ID	E_Name	Gender	C_Code	C_Name
1	A	Male	1	India
2	B	Male	1	India
3	C	Female	2	Japan
4	E	Male	2	Japan
5	F	Female	1	India
6	G	Female	2	Japan
7	H	Male	1	India
8	I	Female	2	Japan

filter

ID	E_Name	Gender	C_Code	C_Name
1	A	Male	1	India
2	B	Male	1	India
4	E	Male	2	Japan
7	H	Male	1	India

groupby-count

C_Name	E_Name
India	3
Japan	2

Domain Specific Optimization: Predicate Pushdown

ID	E_Name	Gender	C_Code
1	A	Male	1
2	B	Male	1
3	C	Female	2
4	E	Male	2
5	F	Female	1
6	G	Female	2
7	H	Male	1
8	I	Female	2

employee

C_Code	C_Name
1	India
2	Japan

country

filter

ID	E_Name	Gender	C_Code
1	A	Male	1
2	B	Male	1
4	E	Male	2
7	H	Male	1

merge

ID	Name	Gender	C_Code	C_Name
1	A	Male	1	India
2	B	Male	1	India
4	E	Male	2	Japan
7	H	Male	1	India

groupby-count

C_Name	E_Name
India	3
Japan	2

```
m = employee.merge(country, on="C_Code")
f = m[m["Gender"] == "Male"]
r = f.groupby("C_Name")["E_Name"].count()
print(r)
```

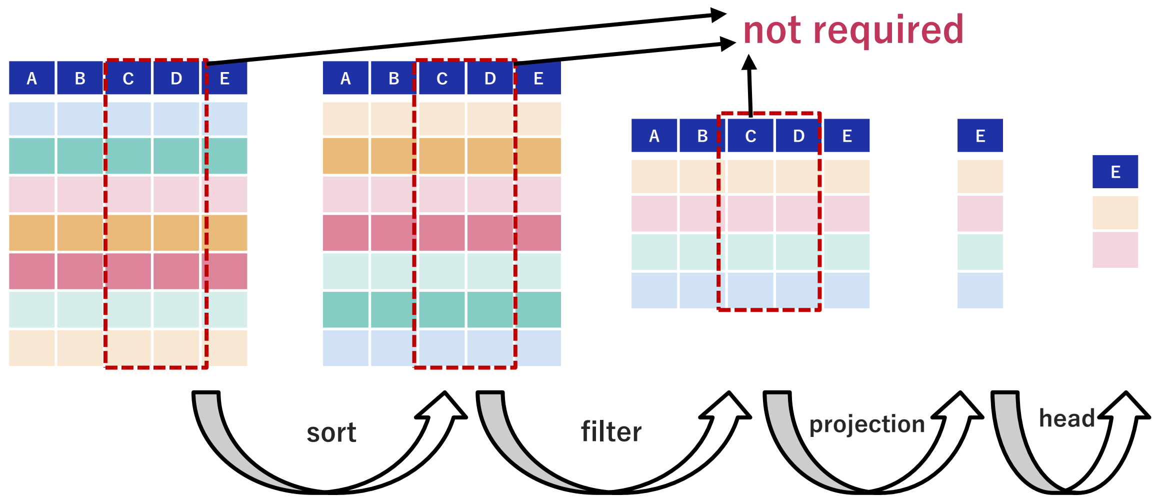
predicate pushdown

```
f = employee[employee["Gender"] == "Male"]
m = f.merge(country, on="C_Code")
r = m.groupby("C_Name")["E_Name"].count()
print(r)
```

Best Practice (2): importance of execution order

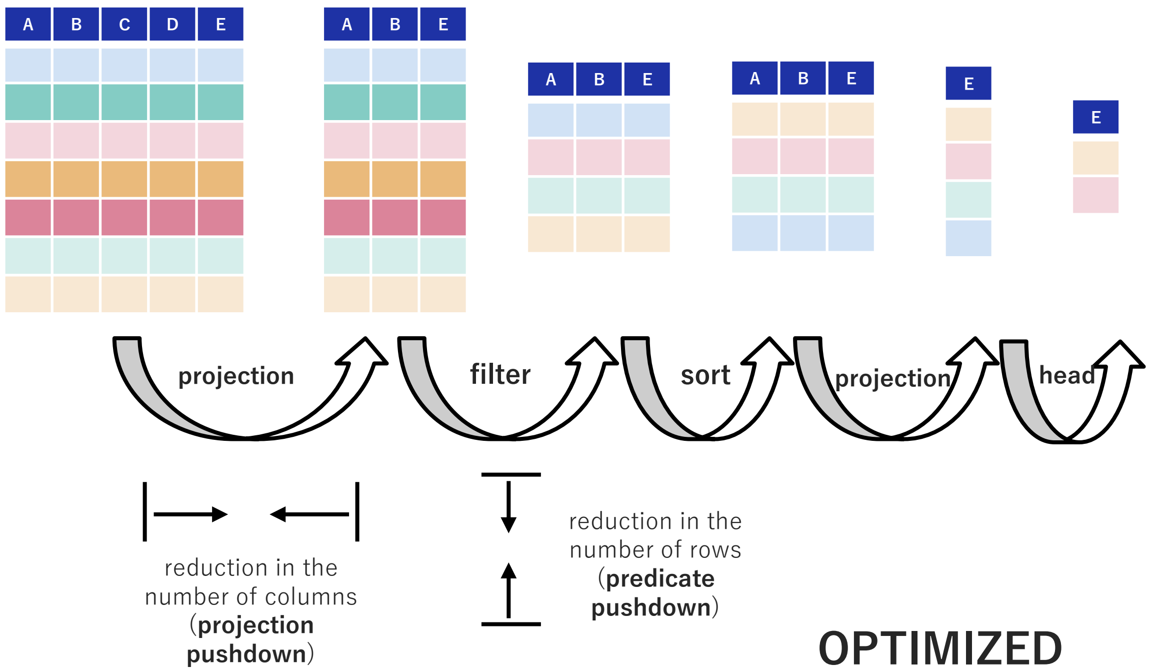
```
df.sort_values(“A”)
  .query(“B > 1”)[“E”]
  .head(2)
```

※ sort-order: yellow->red->green->blue
※ B=1for darker shade, B=2 for lighter shade



SAMPLE QUERY

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



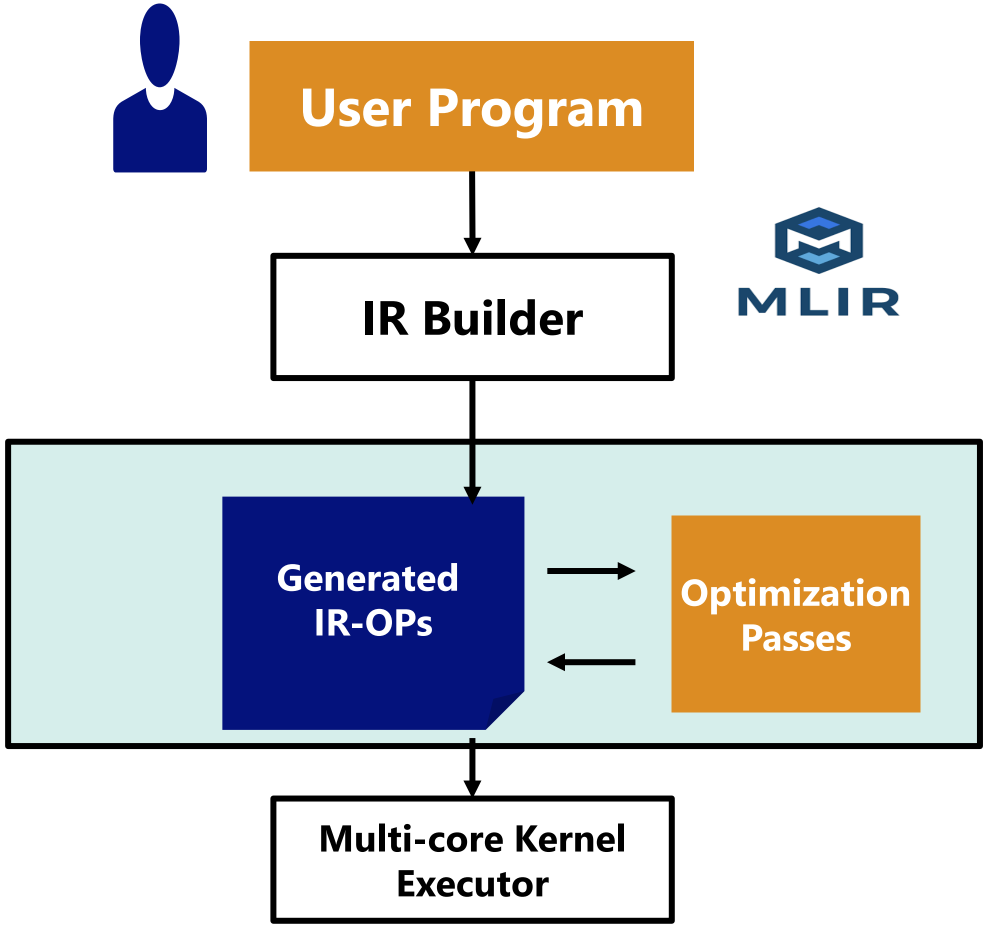
OPTIMIZED QUERY

Introducing FireDucks

Introducing FireDucks

※IR: Intermediate Representation

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




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

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

print (result)

```
%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)
```

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

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

Why FireDucks?

※IR: Intermediate Representation

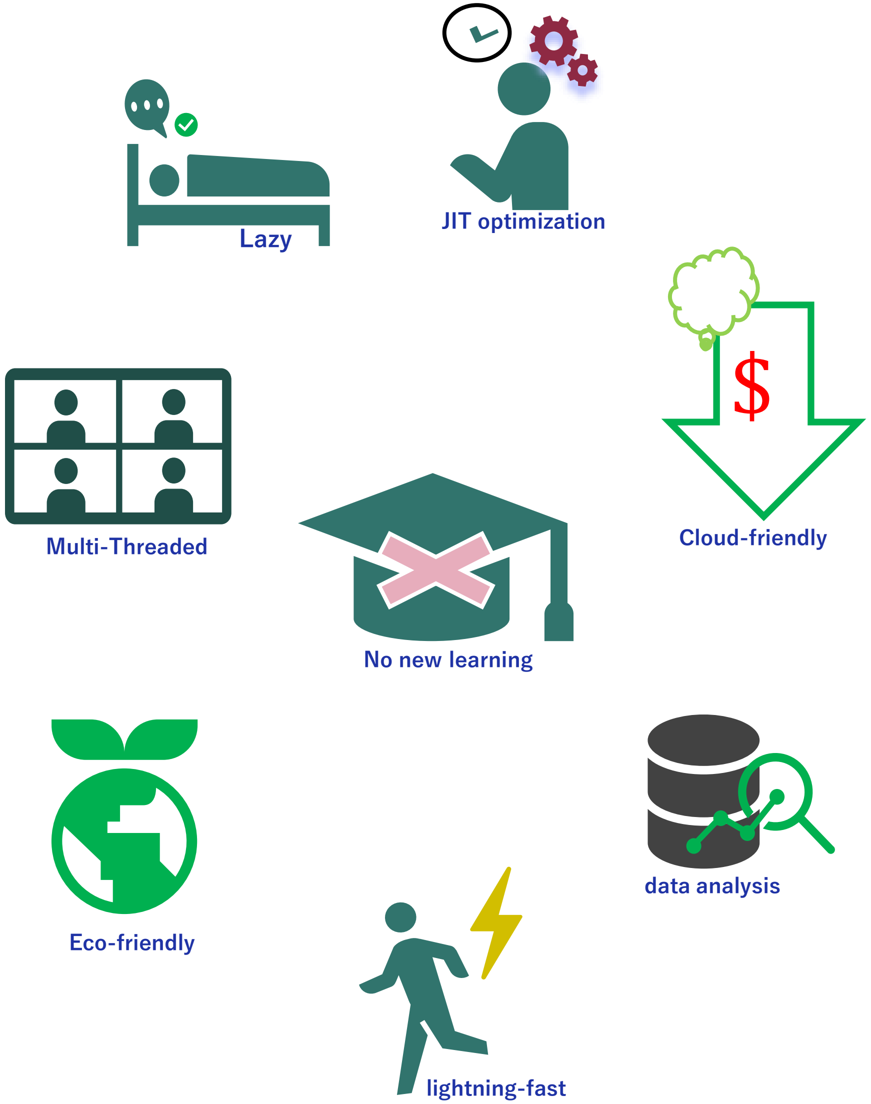
FireDucks (**F**lexible **IR** **E**ngine for DataFrame) is a high-performance compiler-accelerated DataFrame library with highly compatible pandas APIs.

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

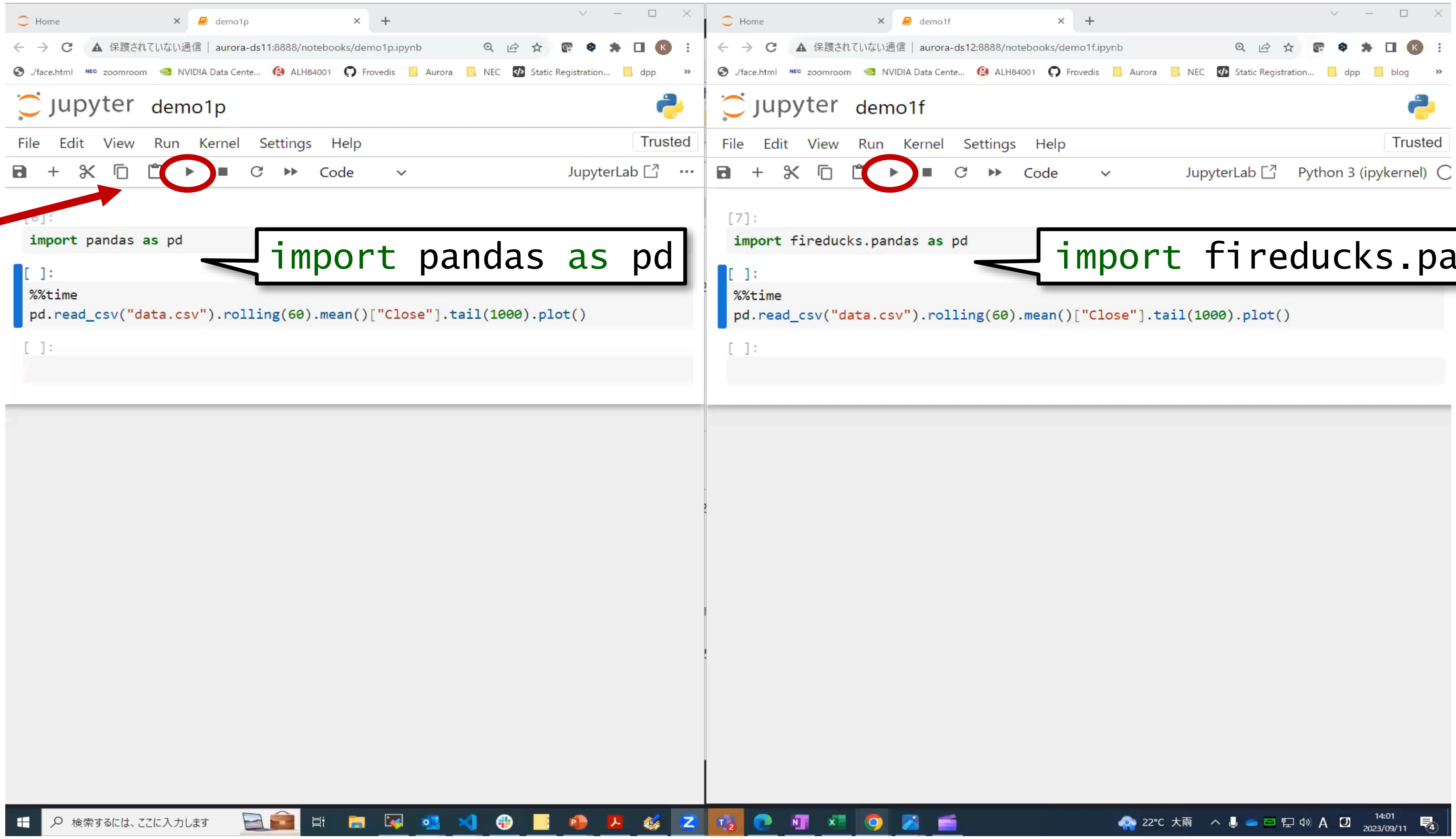


Let’s Have a Quick Demo!

```
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**




```
import pandas as pd
```

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

button to start execution

pandas: 4.06s

~15x

FireDucks: 275ms

data.csv:
[Bitcoin Historical Data]

Usage of FireDucks

※ Linux Only, Supported for Python 3.9 to Python 3.12

1. Explicit Import

easy to import

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

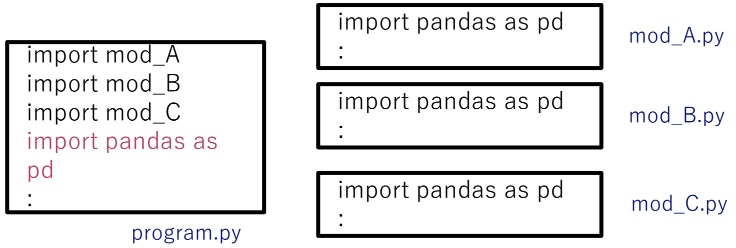
simply change the import statement

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.

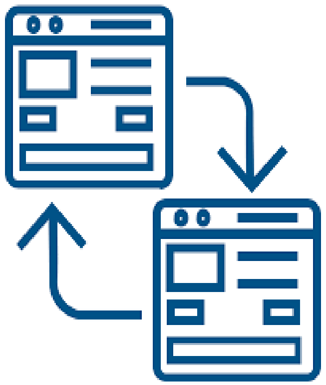
```
%load_ext fireducks.pandas
import pandas as pd
```

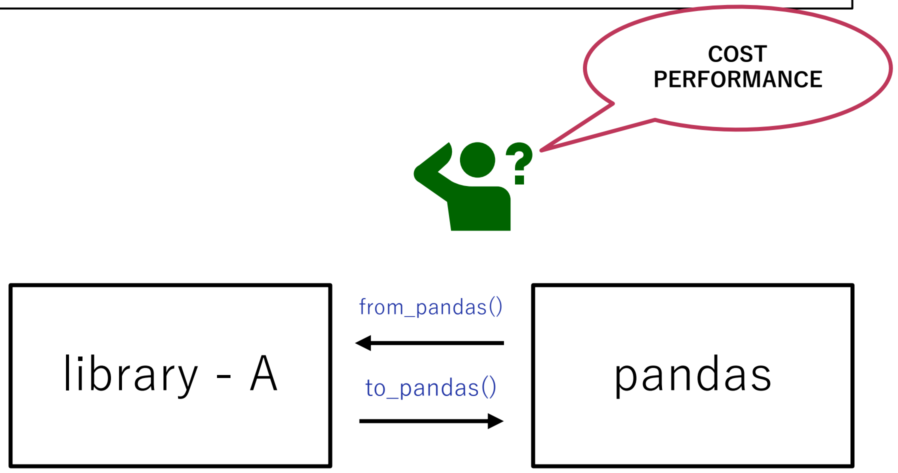
simple integration in a notebook

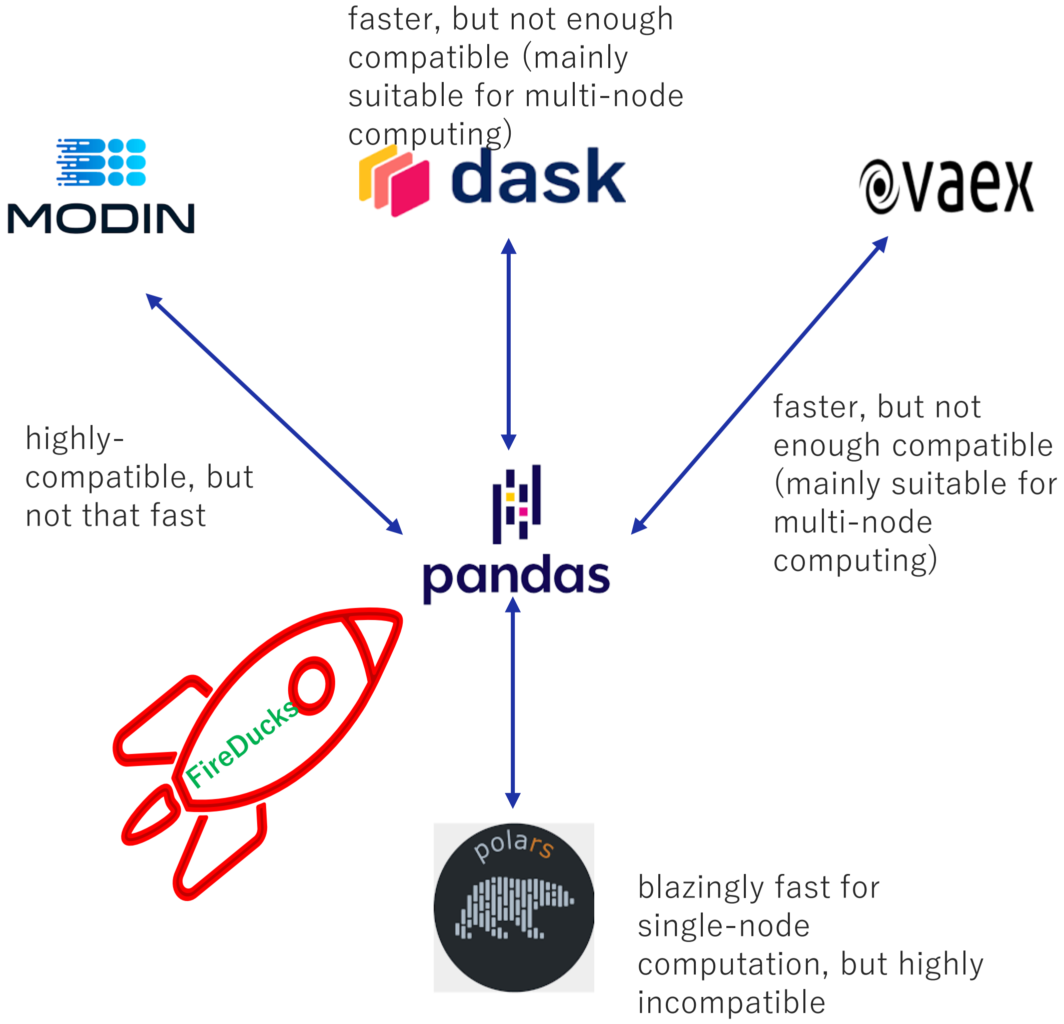
Seamless Integration with pandas: Challenge

Three most common challenges in switching from pandas:

- Needs to learn new library and their interfaces.
- Manual fallback to pandas when the target library doesn't support a method used in an existing pandas application.
- Performance can be evaluated, and results can be tested after the migration is completed.







Seamless Integration with pandas: Demo

Refer: https://github.com/fireducks-dev/fireducks/blob/main/notebooks/nyc_demo/fireducks_pandas_nyc_demo.ipynb

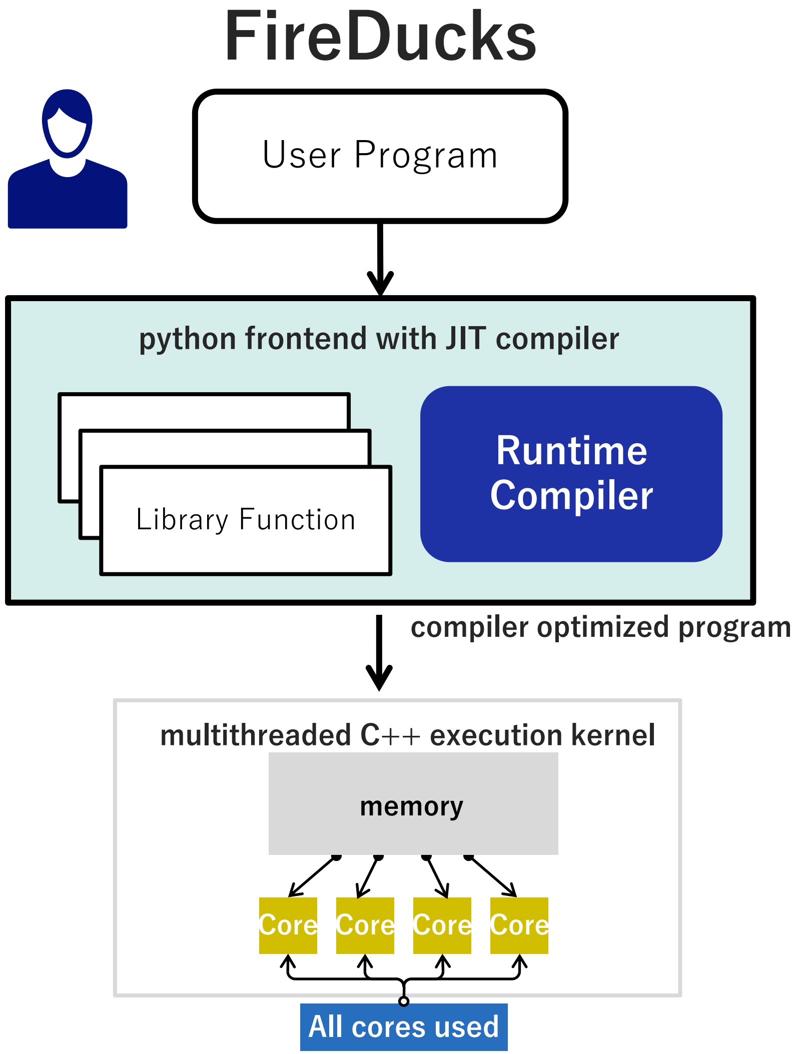
```
import pandas as pd
print(f"evaluation with {pd.__name__}")

start = time.time()
# Data Loading
t1 = time.time()
df = pd.read_parquet(
    "nyc_parking_violations_2022.parquet",
    columns=["Registration State", "Violation Description",
             "Vehicle Body Type", "Issue Date", "Summons Number"]
)
print(df.shape)
print(f"data-loading time: {time.time() - t1} sec")


# Q1: Which parking violation is most commonly committed by vehicles from various U.S states?
t2 = time.time()
r1 = (df[["Registration State", "Violation Description"]]
 .value_counts()
 .groupby("Registration State")
 .head(1)
 .sort_index()
 .reset_index()
)
print(r1.shape)
print(f"Query #1 processing time: {time.time() - t2} sec")


end = time.time()
print(f"total time taken: {end - start} sec")
```

```
$ python nyc_demo.py
evaluation with pandas:
(15435607, 5)
data-loading time: 2.4112608432769775 sec
(65, 3)
Query #1 processing time: 2.8894600868225098 sec
total time taken: 5.300761699676514 sec
```

~8x

```
$ python -mfireducks.pandas nyc_demo.py
(15435607, 5)
data-loading time: 0.3567678928375244 sec
(65, 3)
Query #1 processing time: 0.05789780616760254 sec
total time taken: 0.4147005081176758 sec
```

Optimization Features



1. **Compiler Specific Optimizations**: Common Sub-expression Elimination, Dead-code Elimination, Constant Folding etc.
2. **Domain Specific Optimization**: Optimization at query-level: reordering instructions etc.
3. **Pandas Specific Optimization**: selection of suitable pandas APIs, selection of suitable parameter etc.

1. **Multi-threaded Computation**: Leverage all the available computational cores.
2. **Efficient Memory Management**: Data Structures backed by Apache Arrow
3. **Optimized Kernels**: Patented algorithms for Database like kernel operations: like sorting, join, filter, groupby, dropna etc. developed in C++ from scratch.

Compiler Specific Optimizations

- **Common mistakes often found in Kaggle notebooks**
  - same operation on the same data repeatedly
  - computation without further usage

The in-built compiler of FireDucks can auto-detect such issues and optimize at runtime.

```
# Find year and month-wise average sales
df["year"] = pd.to_datetime(df["time"]).dt.year
df["month"] = pd.to_datetime(df["time"]).dt.month
r = df.groupby(["year", "month"])["sales"].mean()
```

Common **S**ub-expression **E**limination

```
s = pd.to_datetime(df["time"])
df["year"] = s.dt.year
df["month"] = s.dt.month
r = df.groupby(["year", "month"])["sales"].mean()
```

```
def func(x: pd.DataFrame, y: pd.DataFrame):
    merged = x.merge(y, on="key")
    sorted = merged.sort_values(by="key")
    return merged.groupby("key").max()
```

Dead **C**ode **E**limination

```
def func(x: pd.DataFrame, y: pd.DataFrame):
    merged = x.merge(y, on="key")
    return merged.groupby("key").max()
```

[Have you ever thought of speeding up your data analysis in pandas with a compiler?]

Domain Specific Optimizations: Projection Pushdown, Predicate Pushdown (1/2)

Scale Factor: 10
Number of logical cores: 96

※ Shipping Priority Query (Q3) from TPC-H benchmark:
This query retrieves the 10 unshipped orders with the highest value.

```
import datetime
import pandas as pd

def tpch_q3():
    (
        pd.read_parquet("customer.parquet")
          .merge(pd.read_parquet("orders.parquet"), left_on="c_custkey", right_on="o_custkey")
          .merge(pd.read_parquet("lineitem.parquet"), left_on="o_orderkey", right_on="l_orderkey")
          .pipe(lambda df: df[df["c_mktsegment"] == "BUILDING"])
          .pipe(lambda df: df[df["o_orderdate"] < datetime.date(1995, 3, 15)])
          .pipe(lambda df: df[df["l_shipdate"] > datetime.date(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"})[["l_orderkey", "revenue", "o_orderdate", "o_shippriority"]]
          .sort_values(["revenue", "o_orderdate"], ascending=[False, True])
          .reset_index(drop=True)
          .head(10)
          .to_parquet("result.parquet")
    )
```

$ python q3.py:
exec-time: 203 seconds;
memory consumption: 60 GB

$ python –**mfireducks.pandas** q3.py:
exec-time: 4.24 seconds;
memory consumption: 3.3 GB

Domain Specific Optimizations: Projection Pushdown, Predicate Pushdown (2/2)

Refer: https://github.com/fireducks-dev/fireducks/blob/main/notebooks/tpch-query3-pandas-fireducks-cudf.ipynb

manual optimization

```
import datetime
import pandas as pd

def tpch_optimized_q3():
    # load only required columns from respective tables
    req_customer_cols = ["c_custkey", "c_mktsegment"] # (2/8)
    req_lineitem_cols = ["l_orderkey", "l_shipdate", "l_extendedprice", "l_discount"] #(4/16)
    req_orders_cols = ["o_custkey", "o_orderkey", "o_orderdate", "o_shippriority"] #(4/9)
    customer = pd.read_parquet("customer.parquet", columns = req_customer_cols)
    lineitem = pd.read_parquet("lineitem.parquet", columns = req_lineitem_cols)
    orders = pd.read_parquet("orders.parquet", columns = req_orders_cols)

    # advanced-filter: to reduce scope of “customer” table to be processed
    f_cust = customer[customer["c_mktsegment"] == "BUILDING"]

    # advanced-filter: to reduce scope of “orders” table to be processed
    f_ord = orders[orders["o_orderdate"] < datetime.date(1995, 3, 15)]

    # advanced-filter: to reduce scope of “lineitem” table to be processed
    f_litem = lineitem[lineitem["l_shipdate"] > datetime.date(1995, 3, 15)]

    (
        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"]))
              .groupby(["l_orderkey", "o_orderdate", "o_shippriority"], as_index=False)
              .agg({"revenue": "sum"})[["l_orderkey", "revenue", "o_orderdate", "o_shippriority"]]
              .sort_values(["revenue", "o_orderdate"], ascending=[False, True])
              .reset_index(drop=True)
              .head(10)
              .to_parquet("result.parquet")
    )
```

$ python opt_q3.py:
exec-time: 13 seconds;
memory consumption: 5.5 GB

$ python –**mfireducks.pandas** opt_q3.py:
exec-time: 4.8 seconds;
memory consumption: 3.4 GB

Pandas Specific Optimization – Parameter Tuning

parameter tuning in pandas

department-wise average salaries sorted in descending order

groupby("department", sort=True)

```
res = (
   employee.groupby("department")["salary"]
                .mean()
                .sort_values(ascending=False)
)
```

```
res = (
   employee.groupby("department", sort=False)["salary"]
                .mean()
                .sort_values(ascending=False)
)
```

department	salary (USD)
IT	85,000
Admin	60,000
Finance	100,000
IT	81,000
Finance	95,000
Corporate	78,000
Sales	80,000

employee table

department	salary (USD)
IT	85,000
IT	81,000

department	salary (USD)
Admin	60,000

department	salary (USD)
Finance	100,000
Finance	95,000

department	salary (USD)
Corporate	78,000

department	salary (USD)
Sales	80,000

creating groups

department	salary (USD)
IT	83,000
Admin	60,000
Finance	97,500
Corporate	78,000
Sales	80,000

group-wise average-salary

department	salary (USD)
Admin	60,000
Corporate	78,000
Finance	97,500
IT	83,000
Sales	80,000

group-wise average-salary
sorted by "department"

department	salary (USD)
Finance	97,500
IT	83,000
Sales	80,000
Corporate	78,000
Admin	60,000

group-wise average-salary
sorted by "department"

```
df.groupby(["A", "B"])["C"]
  .mean()
  .sort_values(ascending=Fals
e)
```

~50 sec

```
df.groupby(["A", "B"],
sort=False)["C"]
  .mean()
  .sort_values(ascending=False)
```

~30 sec

100M samples with high-cardinality

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

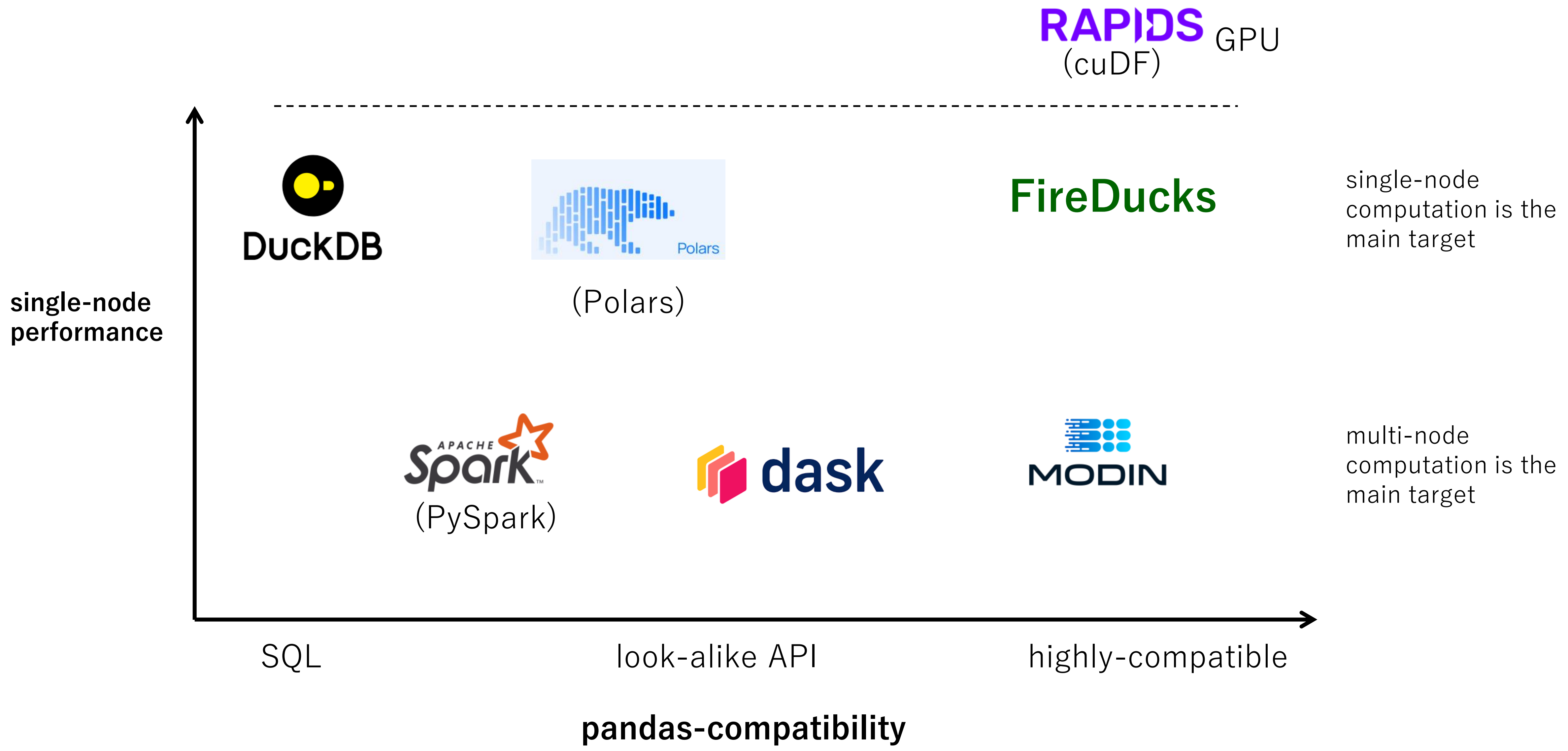
basic questions

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

General Overview: DataFrame Libraries

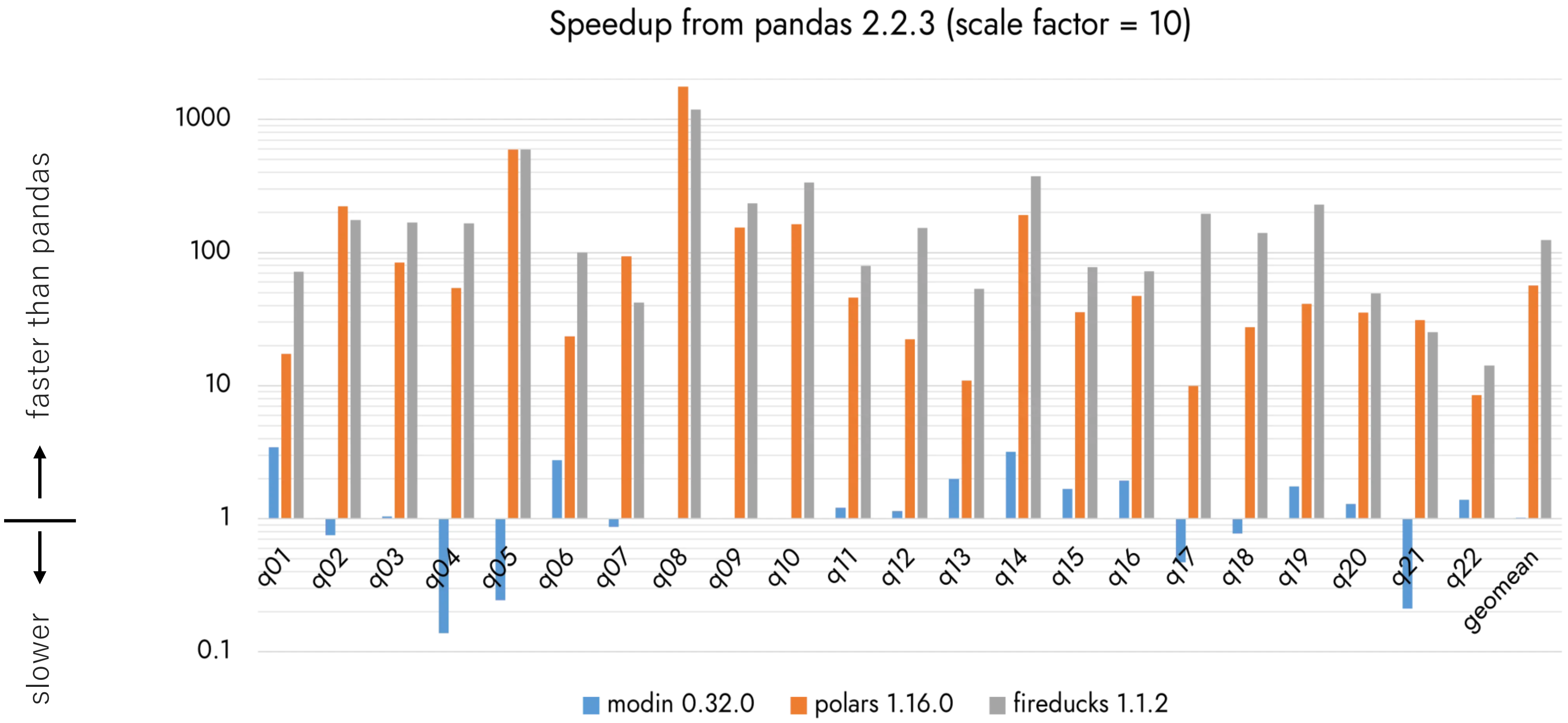


Benchmark (2): Speedup from pandas in TPC-H benchmark

FireDucks is >1000x faster than pandas at max

Server

AWS EC2 m7i.8xlarge: Intel(R) Xeon(R) Platinum 8488C (32cores), 128 GB

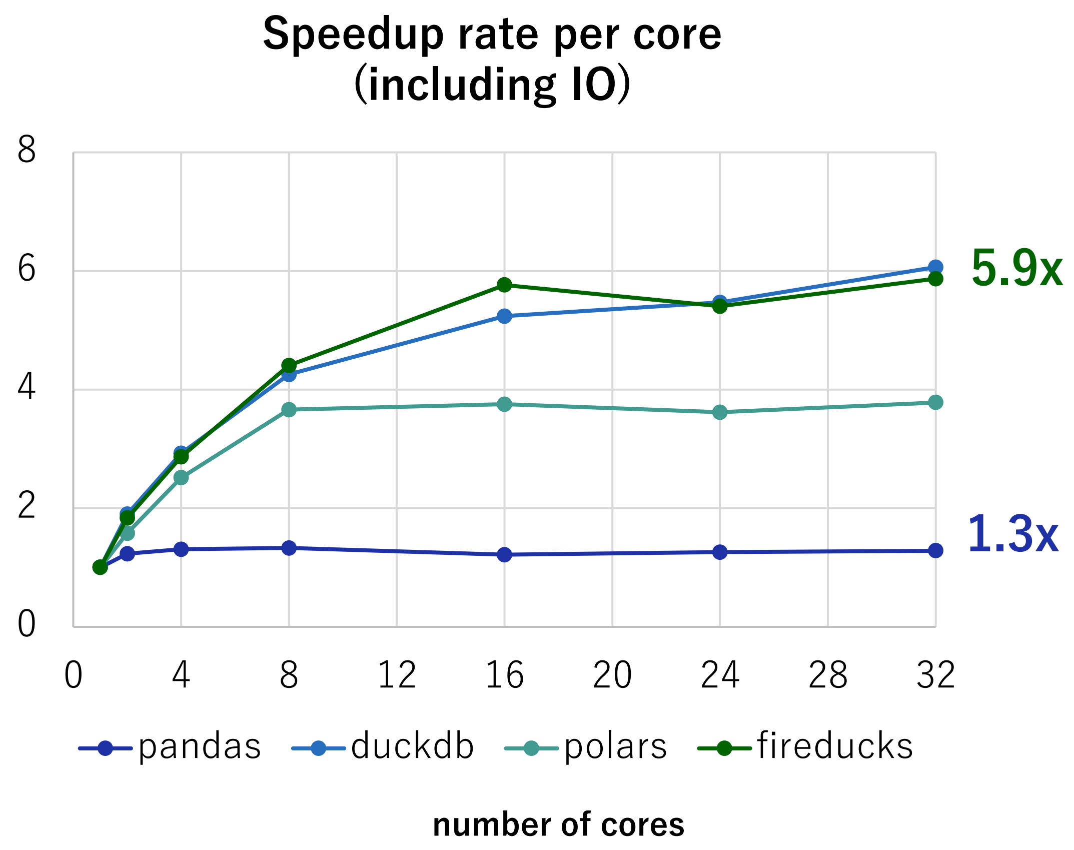


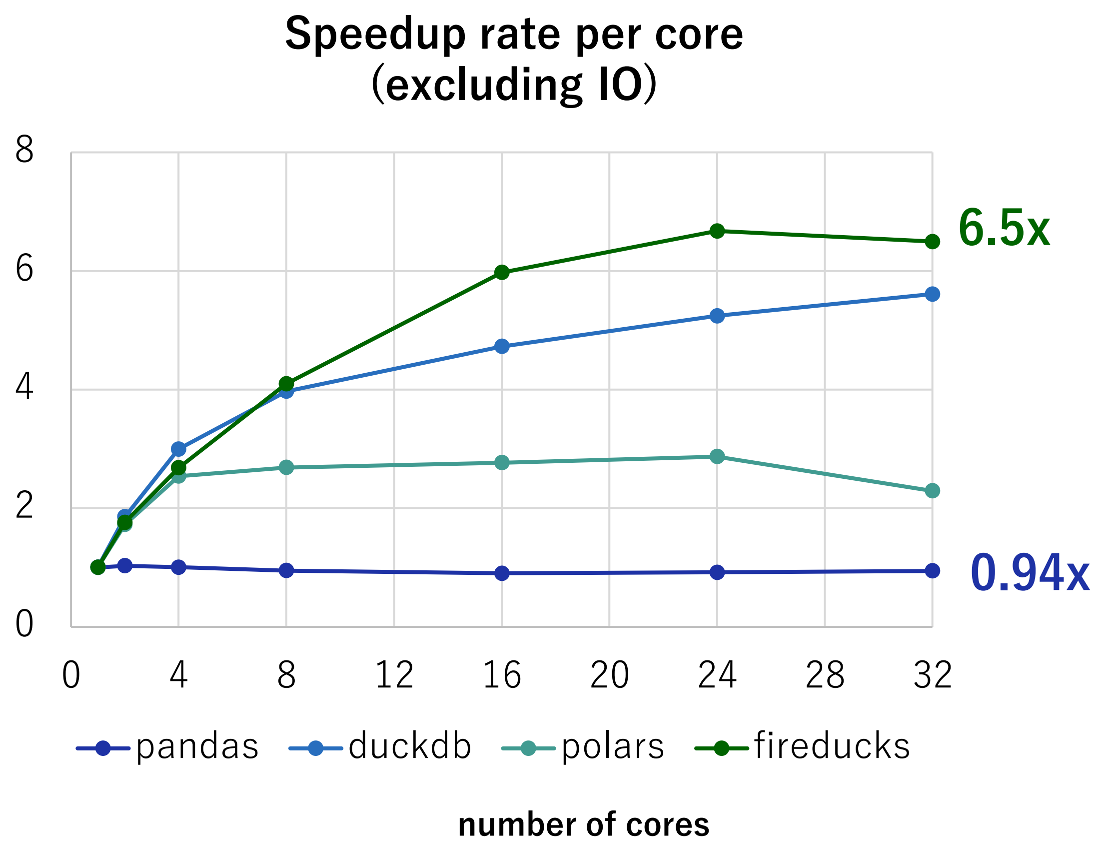
Comparison of DataFrame libraries (average speedup)

FireDucks	**125x**
Polars	57x
Modin	1x

Scalability: DuckDB vs Polars vs FireDucks

Libraries that support multi-threading will benefit from a good machine





Resource on FireDucks

Web site (User guide, benchmark, blog)

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

X(twitter) (Release information)

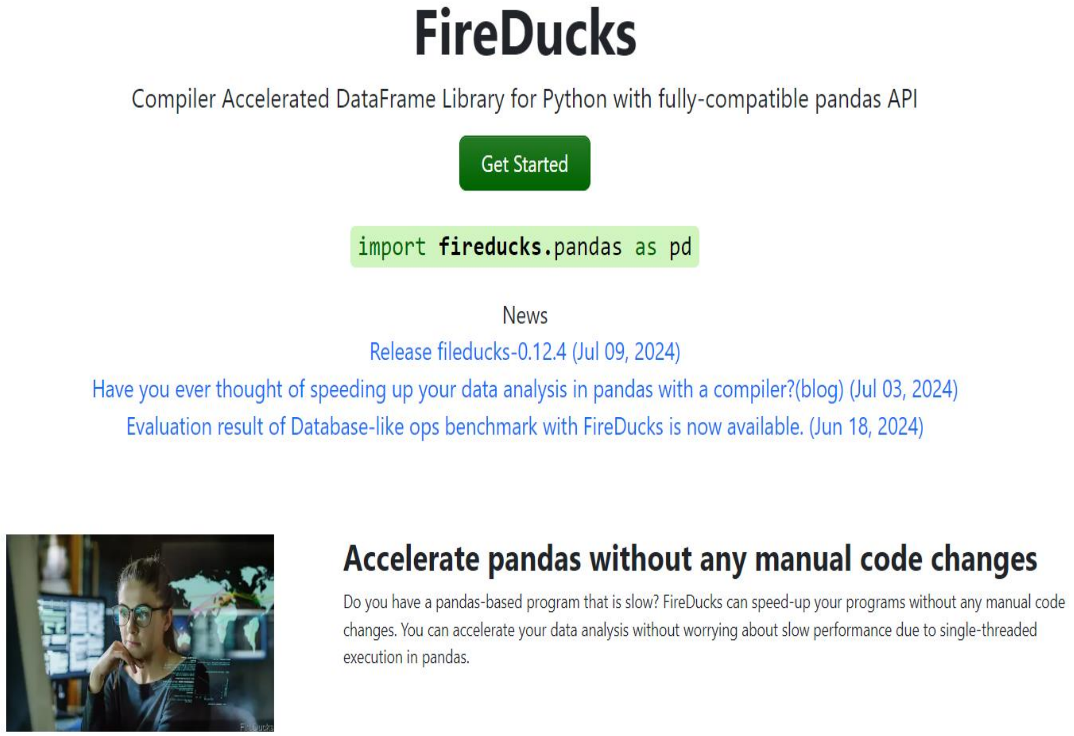
https://x.com/fireducksdev

Github (Issue report)

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

slack Q/A, communication

https://join.slack.com/t/fireducks/shared_invite/zt-2j4lucmtj-IGR7AWIXO62Lu605pnBJ2w

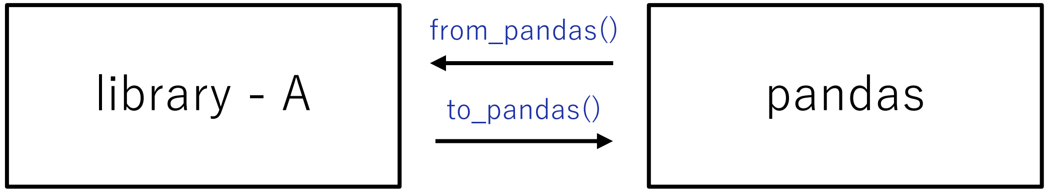


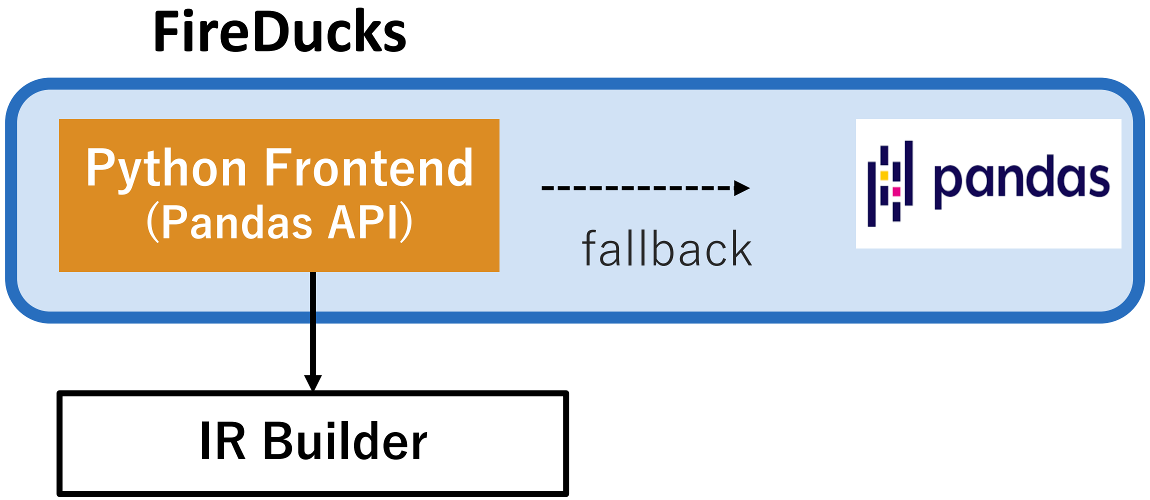
Thank You!

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

Frequently Asked Questions

FAQ: Why FireDucks is highly compatible with pandas?






```
%load_ext fireducks.pandas      ← notebook extension for importhook
import pandas as pd
import numpy as np
```

```
%%fireducks.profile             ← notebook specific profiler
df = pd.DataFrame({
    "id": np.random.choice(list("abcdef"), 10000),
    "val": np.random.choice(100, 10000)
})

r1 =(
    df.sort_values("id")
      .groupby("id")
      .head(2)
      .reset_index(drop=True)
)
              pd.from_pandas(r1["val"].to_pandas().cumsum())
r1["val"] = r1["val"].cumsum()
r1.describe()
```

profiling-summary:: total: 42.4832 msec (fallback: 1.1448 msec)

	name	type	n_calls	duration (msec)
0	groupby_head	kernel	1	16.696805
1	sort_values	kernel	1	16.684564
2	from_pandas.frame.metadata	kernel	2	3.641694
3	to_pandas.frame.metadata	kernel	2	2.237987
4	describe	kernel	1	2.021135
5	DataFrame._repr_html_	fallback	1	1.021662
6	Series.cumsum	fallback	1	0.111802
7	setitem	kernel	1	0.010280
8	get_metadata	kernel	1	0.009650
9	reset_index	kernel	1	0.008050

When running a python script/program, you may like to set the environment variable to get fallback warning logs:
FIREDUCKS_FLAGS="-Wfallback"

Raise feature request when you encounter some expensive fallback hindering your program performance!

Directly **communicate** with us over our slack channel for any performance or API related queries!

FAQ: How to evaluate Lazy Execution?

```
def foo(employee, country):
    stime = time.time()
    m = employee.merge(country, on="C_Code")
    r = m[m["Gender"] == "Male"]
    print(f"fireducks time: {time.time() – stime} sec")
    return r
```

IR Builder

create_data_op(…)
merge_op(…)
filter_op(…)

fireducks time: 0.0000123 sec

```
def foo(employee, country):
    employee._evaluate()
    country._evaluate()
    stime = time.time()
    m = employee.merge(country, on="C_Code")
    r = m[m["Gender"] == "Male"]
    r._evaluate()
    print(f"fireducks time: {time.time() – stime} sec")
    return r
```

fireducks time: 0.02372143 sec

FIREDUCKS_FLAGS="--benchmark-mode"

Use this to disable lazy-execution mode when you do not want to make any changes in your existing application during performance evaluation.

FAQ: How to configure number of cores to be used?

OMP_NUM_THREADS=1

Use this to stop parallel execution, or configure this with the intended number of cores to be used

Alternatively, you can use the Linux taskset command to bind your program with specific CPU cores.

\Orchestrating a brighter world

NECは、安全・安心・公平・効率という社会価値を創造し、
誰もが人間性を十分に発揮できる持続可能な社会の実現を目指します。

\Orchestrating a brighter world

NEC