Introducing a high-performance compiler-accelerated DataFrame Library, FireDucks

July 12, 2024

Sourav Saha
Research Engineer, NEC

Quick Introduction!

SOURAV SAHA – Research Engineer @ NEC Corporation

Hey, 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

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

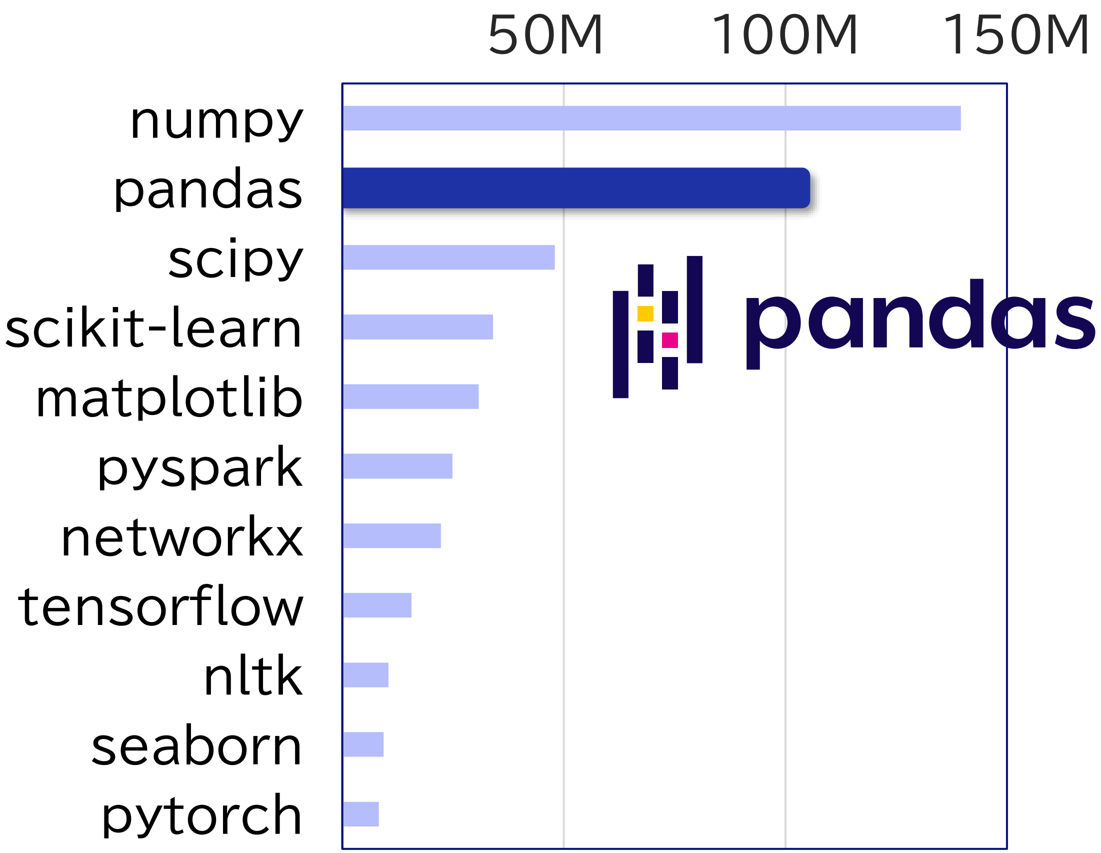
FireDucks

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

compiler technologies

Background: What is pandas?

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



Monthly download from pypi.org
(Data Analytics Libraries)

Books

Courses

https://www.udemy.com/ja/topic/pandas/

Drawback of pandas

- It (mostly) doesn’t support parallel computation.
- It doesn’t manage runtime memory well.

- It follows an eager execution model.
  - It doesn’t have any auto-optimization feature.
  - The implementation is not optimized for modern processors.

- There are many different methods of performing the same analysis in pandas.
  - The choice of APIs heavily impacts the performance of an application.

Need for Optimization

Improve in efficiency of Data Analysis

Reduction of cloud cost

Reduction of CO2 emission

Data Scientist

The amount spent on performing each simulation of an analytical task can be significantly reduced, resulting in more productive time for in-depth data analysis.

Need for Optimization

Improve in efficiency of Data Analysis

Reduction of cloud cost

Reduction of CO2 emission

Data Engineer

If execution can be speed-up by 10x, Cloud cost can also be reduced up to **1/10 !**

Need for Optimization

Improve in efficiency of Data Analysis

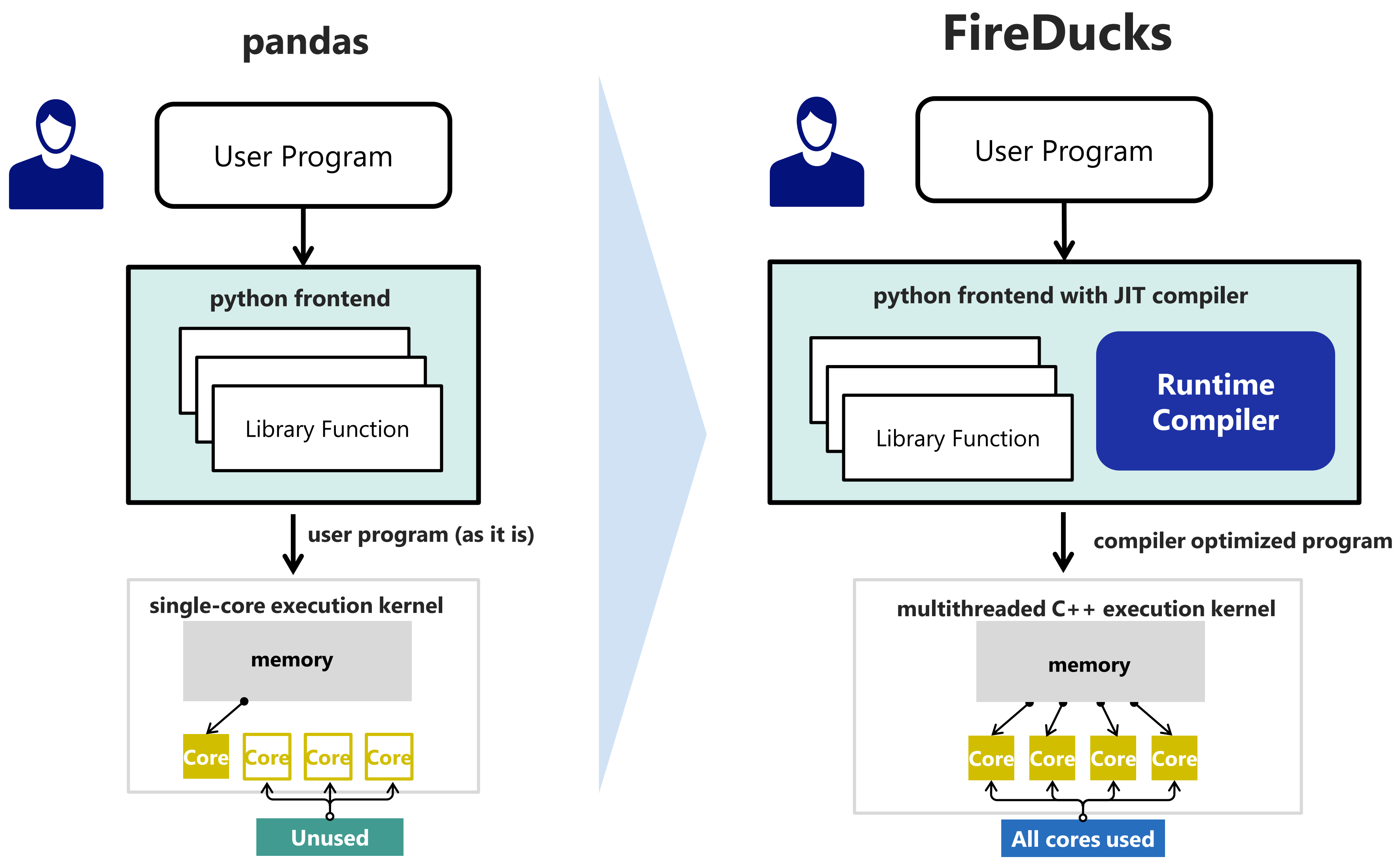
Reduction of cloud cost

Reduction of CO2 emission

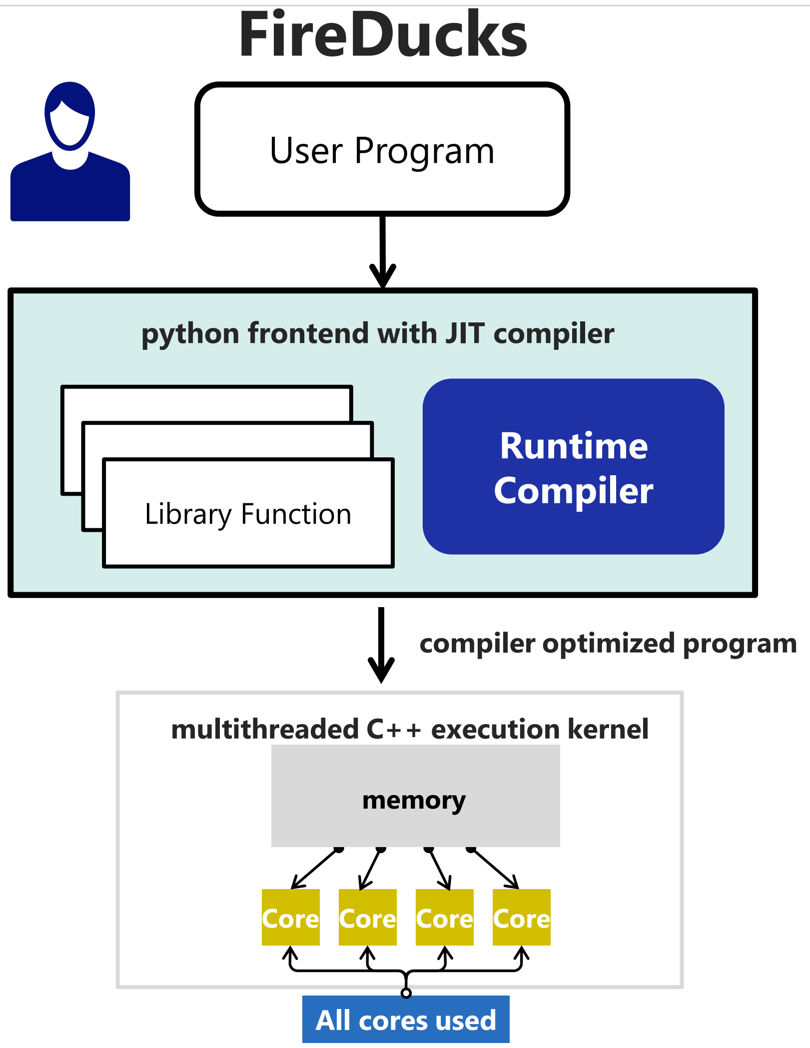
Performing long-running simulation on a cluster of computers might negatively impact the environment

Data Scientist

Execution model



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 Optimization (Example #1)

```
# Find the industry-wise average salary of an Indian employee
res = pd.DataFrame()

res["industry_wise_avg_sal"] = (
    employee[employee["country"] == "India"]
    .groupby("industry")["salary"]
    .mean()
)
```

```
# Find the industry-wise average salary of an Indian employee who is above 30
res["industry_wise_avg_sal_for_specific_age_group"] = (
    employee[(employee["country"] == "India") & (employee["age"] >= 30)]
    .groupby("industry")["salary"]
    .mean()
)
```

```
# To generate the required filtration masks in advance
cond1 = (employee["country"] == "India")
cond2 = (employee["age"] >= 30)
res = pd.DataFrame()
```

```
# Find the industry-wise average salary of an Indian employee
res["industry_wise_avg_sal"] = (
    employee[cond1]
    .groupby("industry")["salary"]
    .mean()
)
```

```
# Find the industry-wise average salary of an Indian employee who is above 30
res["industry_wise_avg_sal_for_specific_age_group"] = (
    employee[cond1 & cond2]
    .groupby("industry")["salary"]
    .mean()
)
```

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

Domain Specific Optimization (Example #1)

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

Domain Specific Optimization (Example #2) (1/2)

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 (Example #2) (2/2)

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

merge

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

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)
```

Pandas Specific Optimization (Example #1)

department-wise average salaries sorted in descending order

```
res = (                         groupby("department", sort=True)
   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

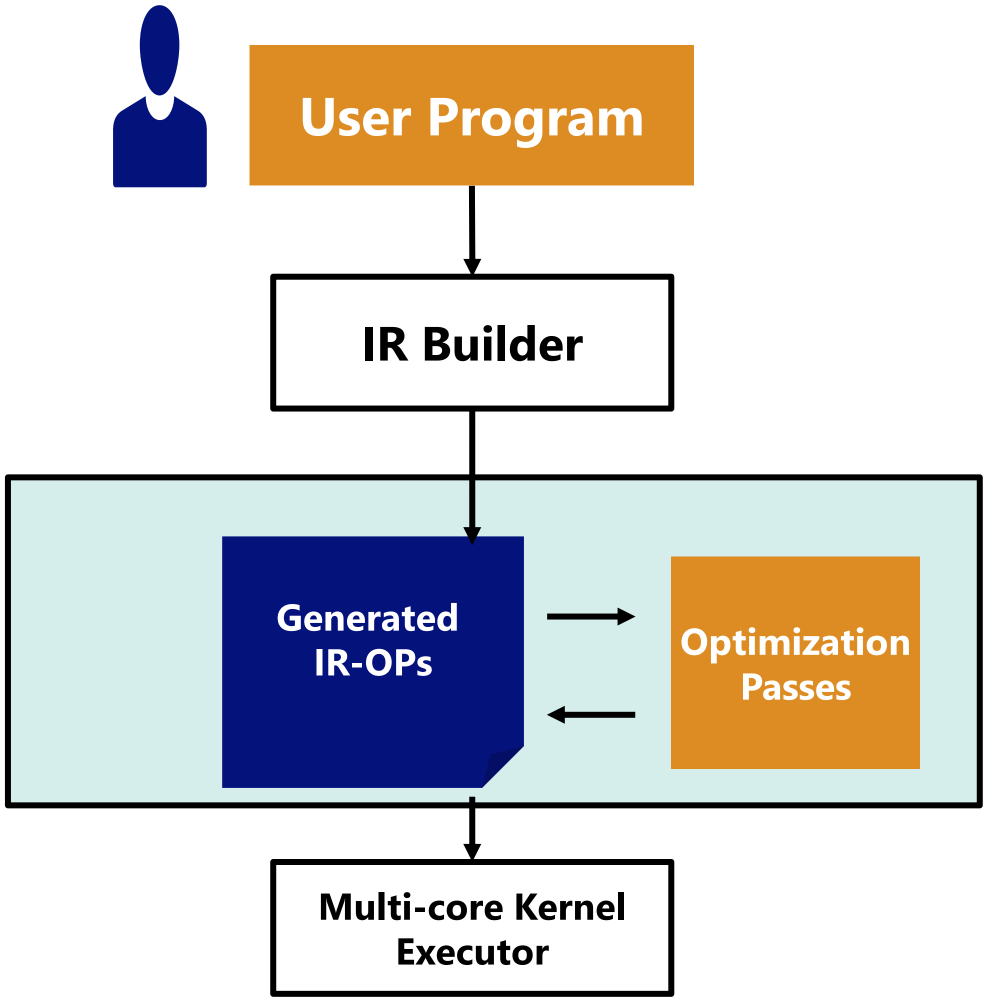
department	salary (USD)
Admin	60,000
Corporate	78,000
Finance	97,500
IT	83,000
Sales	80,000

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"

How does FireDucks Work?




```
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"]
```

Usage of FireDucks

1. Import Hook

FireDucks provides command line option to automatically replace pandas with FireDucks

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

Zero code modification

2. Explicit Import

User replaces import statement

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

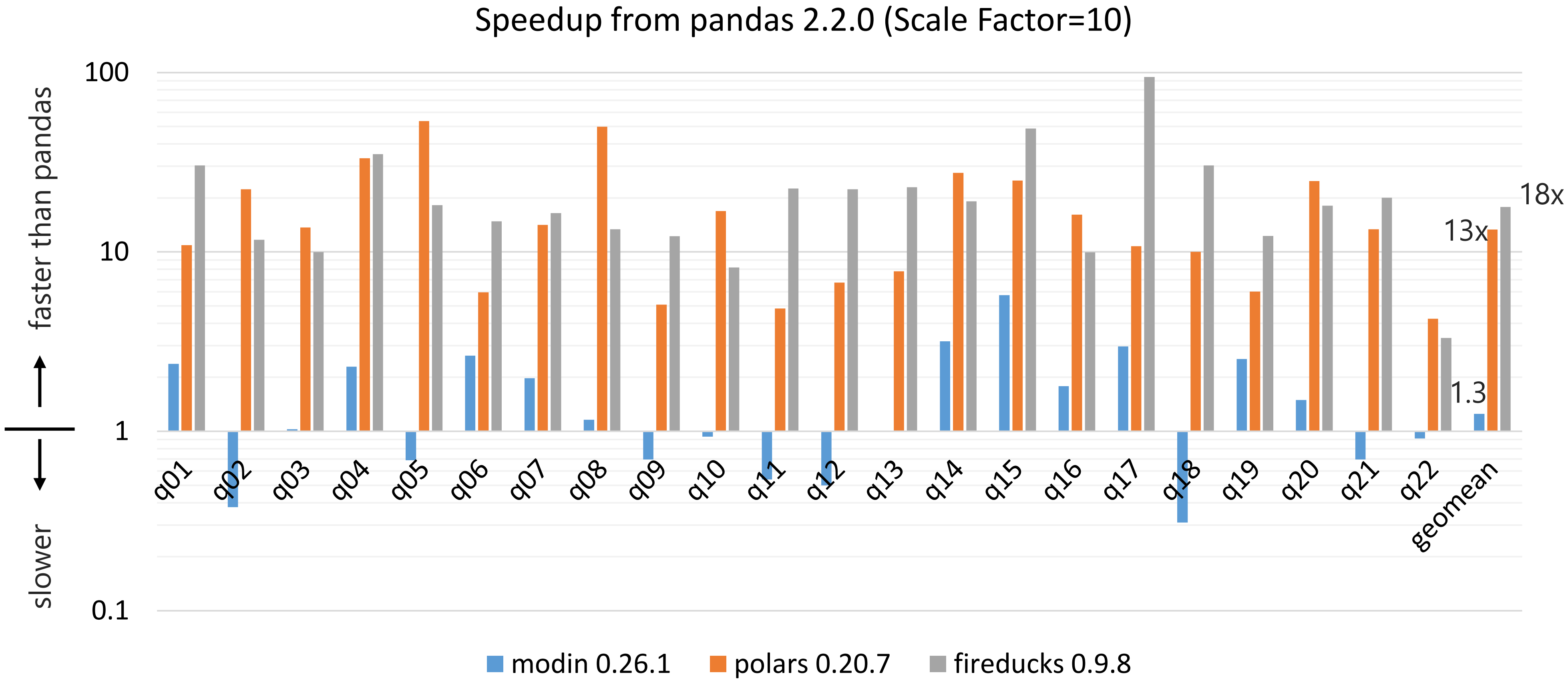
single line modification (convenient with Jupyter notebook)

Benchmark: Speedup from pandas in TPC-H benchmark

FireDucks is 95x faster than pandas at max

Server

Xeon Gold 5317 x2
(24 cores), 256GB



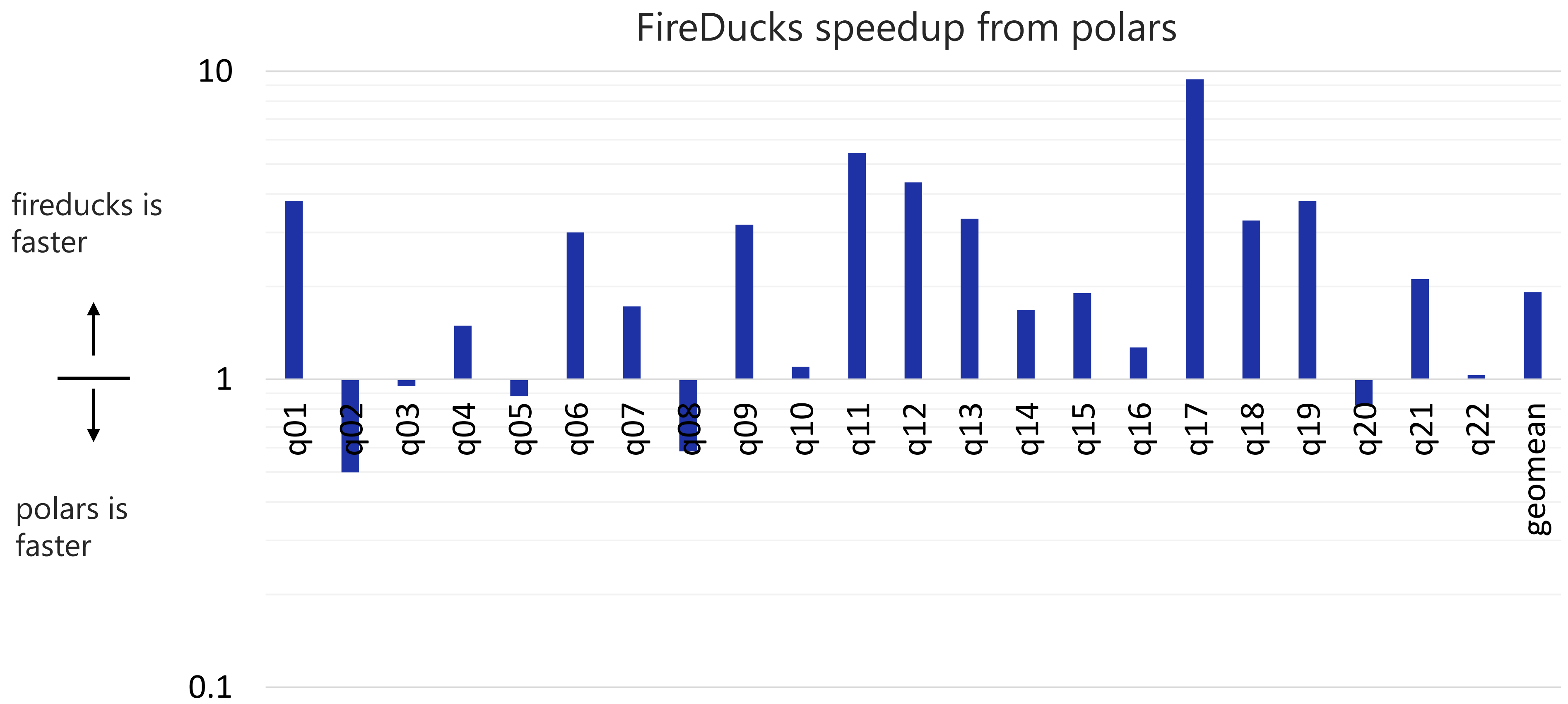
Comparison of DataFrame libraries (average speedup)

FireDucks	**18x**
Polars	13x
Modin	1.3x

Benchmark: FireDucks and Polars

FireDucks is faster than polars 12x at max (1.9x in average)

Polars: faster DataFrame library with own API (not compatible with pandas)



Xeon Gold 5317
(12 core x 2s)
Memory: 256GB
OS: Linux

pandas 2.2.0
polars 0.20.7
FireDucks 0.10.1

Benchmark: DB-Benchmark

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

groupby join groupby2014

0.5 GB 5 GB 50 GB

basic questions

rank-1

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

FireDucks	0.12.1	2024-06-17	14s
duckdb-latest	0.9.1.1	2023-10-26	24s
DuckDB	0.8.1.3	2023-10-26	25s
ClickHouse	23.10.4.25	2023-11-30	29s
Polars	0.19.8	2023-10-17	32s
DataFrames.jl	1.6.1	2023-10-17	84s
data.table	1.14.9	2023-10-17	89s
Datafusion	31.0.0	2023-10-24	133s
InMemoryDatasets	0.7.18	2023-10-17	218s
collapse	2.0.3	2023-10-26	233s
spark	3.5.0	2023-10-24	297s
R-arrow	13.0.0.1	2023-10-17	511s
dask	2023.10.0	2023-11-29	544s
pandas	2.1.1	2023-10-17	773s
(py)datatable	1.1.0a0	2023-10-17	993s
dplyr	1.1.3	2023-10-17	1022s
Modin		see README	pending

Groupby

groupby join groupby2014

0.5 GB 5 GB 50 GB

basic questions

rank-3

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

DuckDB	0.8.1.3	2023-10-20	8s
duckdb-latest	0.9.1.1	2023-10-25	8s
FireDucks	0.12.1	2024-06-17	8s
Polars	0.19.8	2023-10-20	14s
Datafusion	31.0.0	2023-10-25	22s
InMemoryDatasets	0.7.18	2023-10-20	25s
ClickHouse	23.10.4.25	2023-11-30	42s
data.table	1.14.9	2023-10-20	55s
collapse	2.0.3	2023-10-26	65s
DataFrames.jl	1.6.1	2023-10-20	71s
spark	3.5.0	2023-10-24	129s
dask	2023.10.0	2023-11-29	179s
dplyr	1.1.3	2023-10-20	225s
pandas	2.1.1	2023-10-20	265s
(py)datatable	1.1.0a0	2023-10-20	5699s
R-arrow	13.0.0.1	2023-10-20	out of memory
Modin		see README	pending

Join

Resource on FireDucks

Web site (User guide, benchmark, blog)

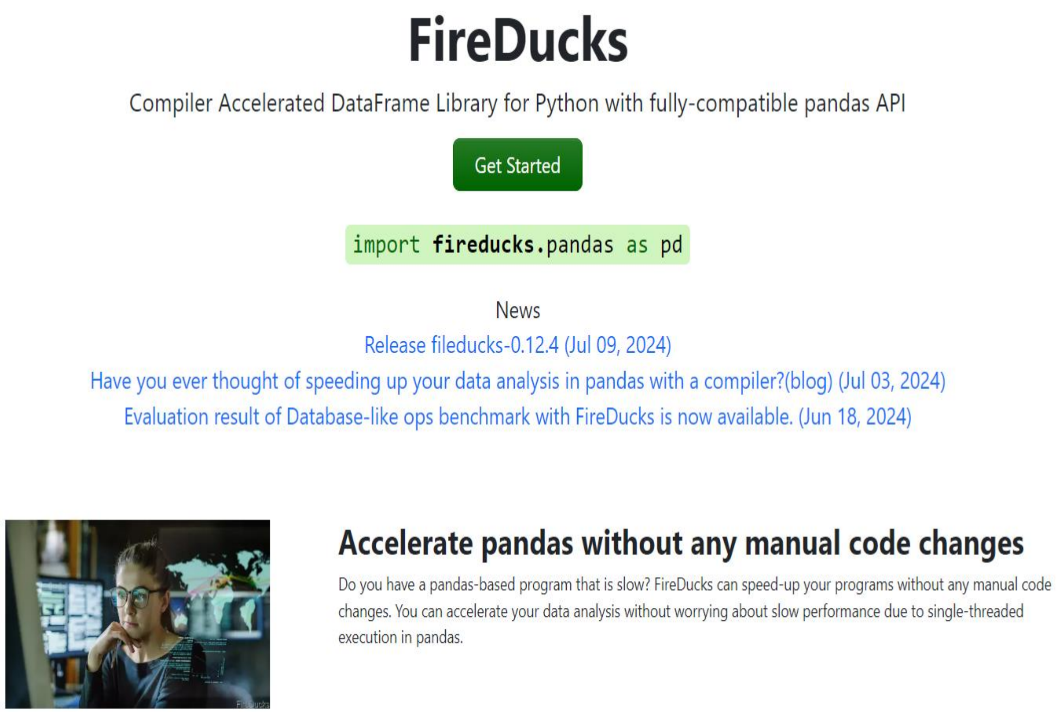
https://fireducks-dev.github.io/

X(twitter) (Release information)

https://x.com/fireducksdev

Github (Issue report)

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

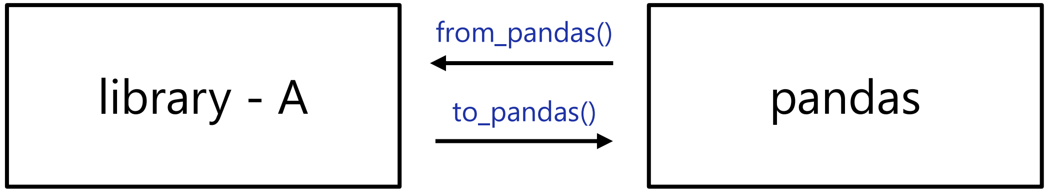


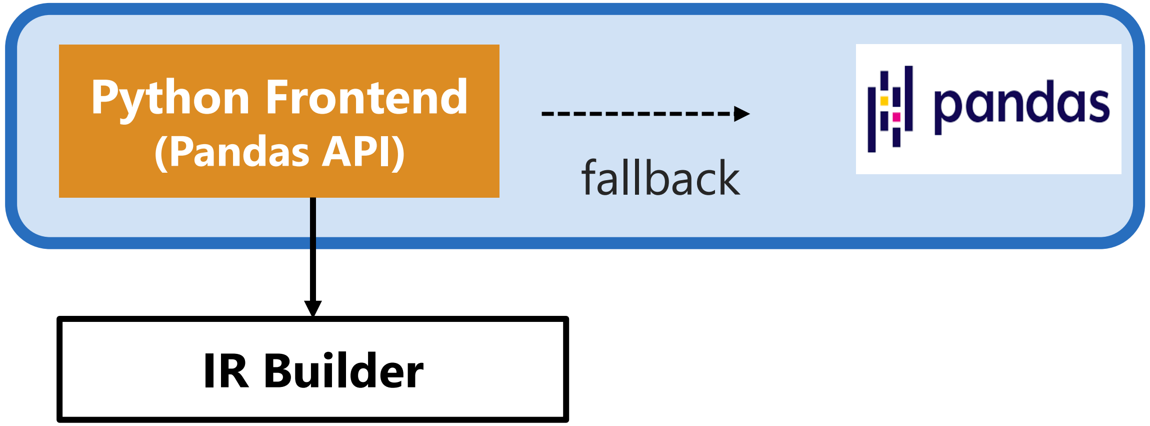
User feedback

Due to a significant reduction in execution time, I can now focus more on in-depth data analysis.

Easy integration in an existing application in just 30 mins!

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)
)

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!

Demo

https://colab.research.google.com/drive/1qpej-X7CZsleOqKuhBg4kq-cbGuJf1Zp?usp=sharing

Summary

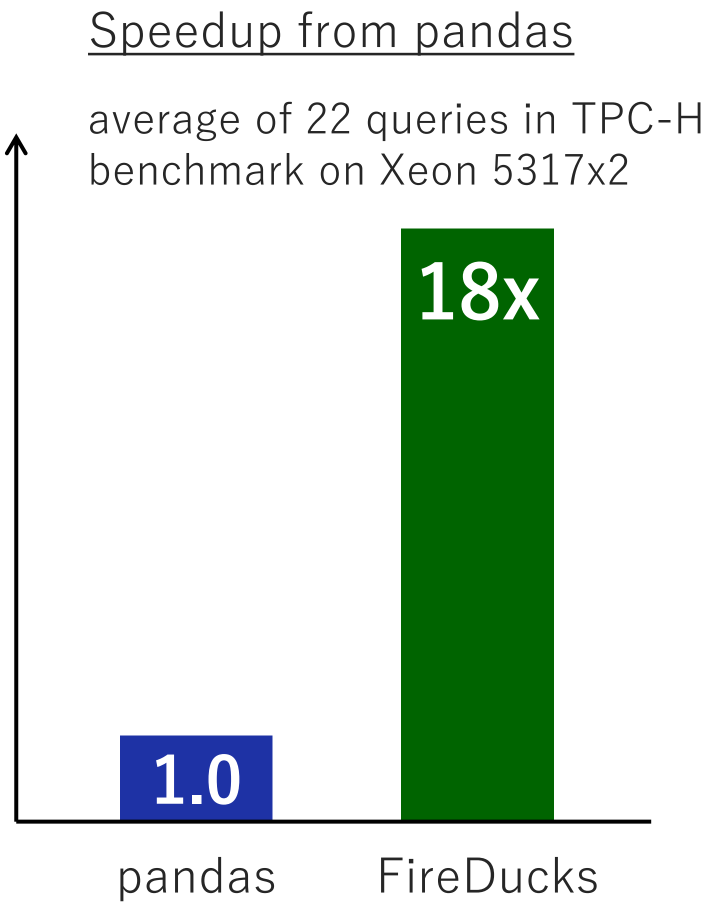
FireDucks is a high-performance compiler-accelerated DataFrame library with highly compatible pandas APIs.

Speed: significantly faster than pandas

- FireDucks is multithreaded to fully exploit modern processor
- FireDucks optimizes user program at runtime by embedded **runtime compiler**

Ease of use: drop-in replacement of pandas

- FireDucks is highly compatible with pandas API
- No extra learning is required
- No code modification is required



\Orchestrating a brighter world

NEC creates the social values of safety, security,
fairness and efficiency to promote a more sustainable world
where everyone has the chance to reach their full potential.

\Orchestrating a brighter world

NEC