Unveiling the lazy execution benefit of FireDucks

A Multithreaded DataFrame Library with JIT compiler

March 07, 2025 (@SacPy)

Sourav Saha (NEC)

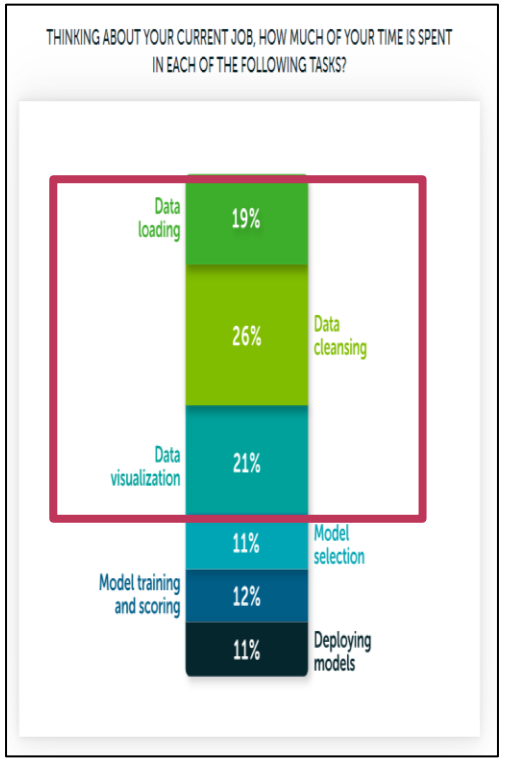
Senior Research Engineer

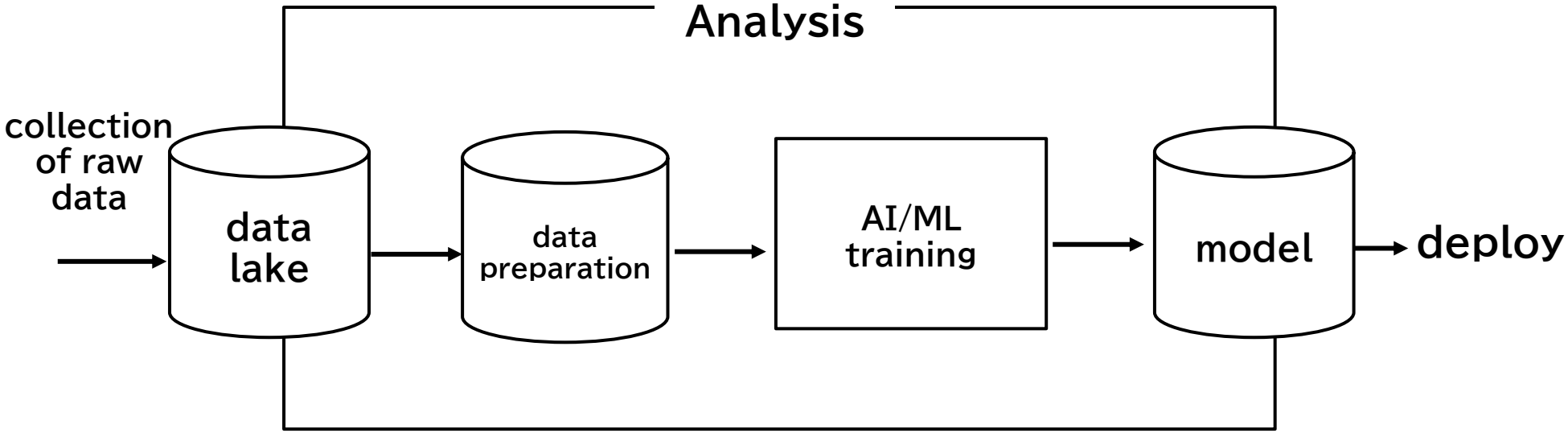
Agenda

- 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

Workflow of a Data Scientist

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

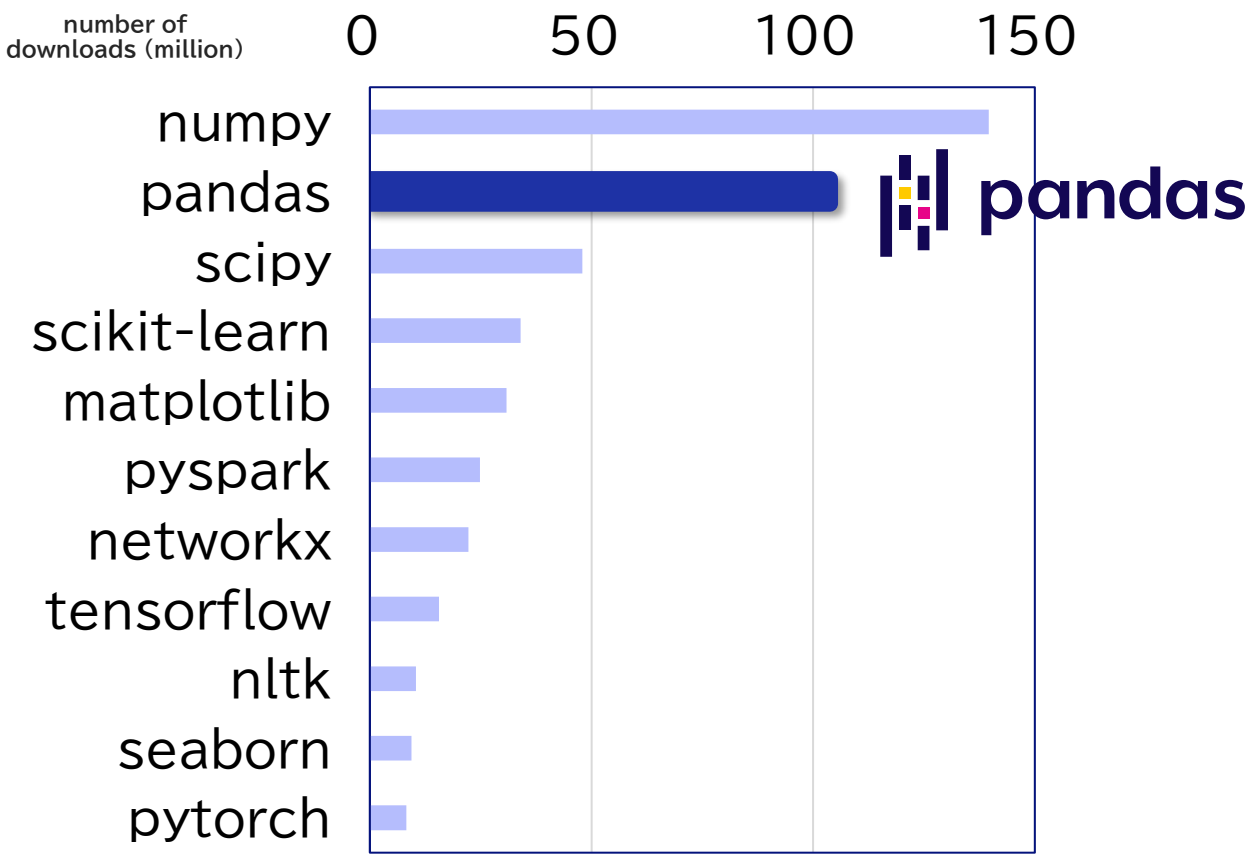




Anaconda:
The State of Data Science 2020

About Pandas (1/2)

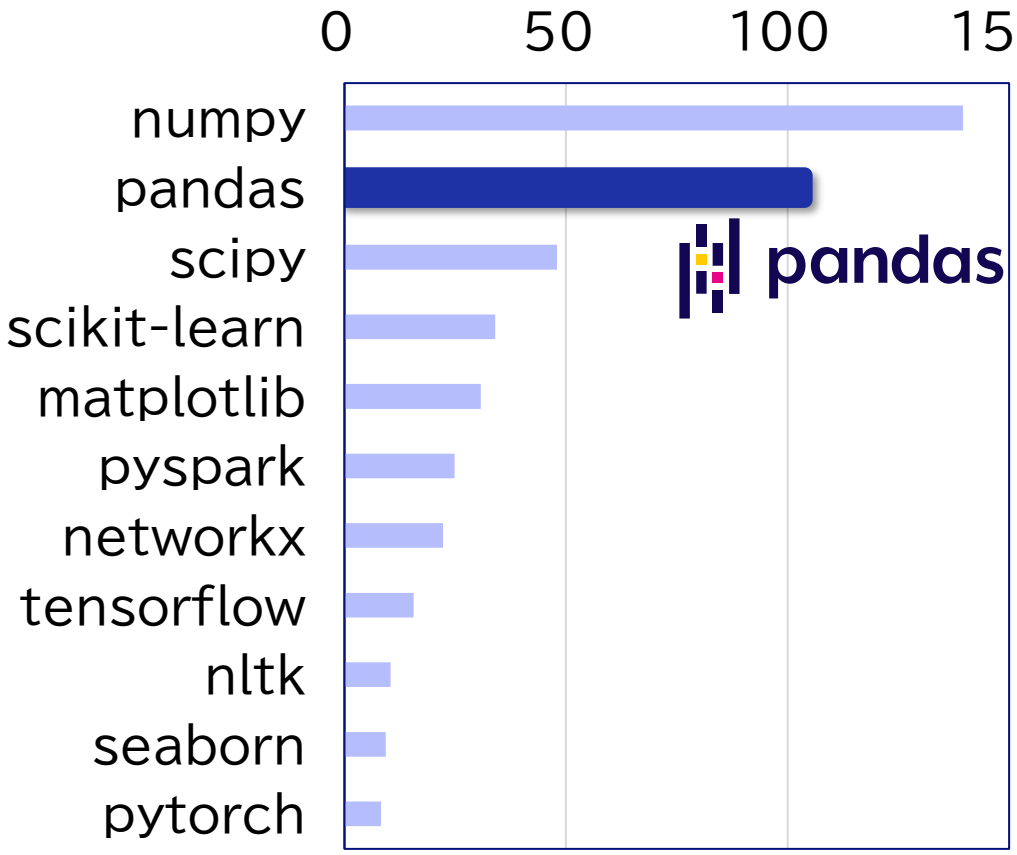
The most popular Python library for data processing



Monthly download from pypi.org (Data Analytics Libraries)

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 implementation?

```
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 bar(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

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

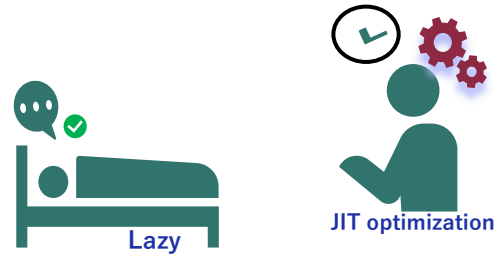
```
f = employee[employee["Gender"] == "Male"]
m = f.merge(country, on="C_Code")
r = m.groupby("C_Name")["E_Name"].count()
print(r)
```

Introducing FireDucks

About FireDucks

※IR: Intermediate Representation

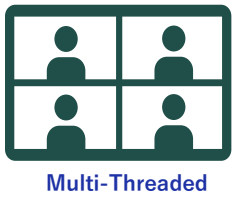
FireDucks (**F**lexible **IR** **E**ngine for DataFrame) is a high-performance compiler-accelerated DataFrame library with highly compatible pandas APIs.



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).

MLIR



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

Usage of FireDucks

※ Linux Only, Supported for Python 3.9 to Python 3.13

1. Explicit Import

easy to import

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

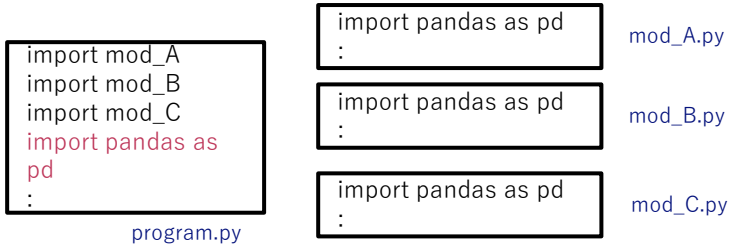
simply change the import statement

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: 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: 2.41126 sec
(65, 3)
Query #1 processing time: 2.88946 sec
total time taken: 5.30076 sec
```

~13x

```
$ python -mfireducks.pandas
nyc_demo.py
evaluation with fireducks.pandas:
(15435607, 5)
data-loading time: 0.35676 sec
(65, 3)
Query #1 processing time: 0.05789 sec
total time taken: 0.41471 sec
```

Why FireDucks is faster?

FireDucks delivers superior performance over Pandas due to the following three layers of optimizations:

✅ **Optimized Kernel Methods** → Efficient implementation of low-level code for faster execution.

✅ **Lazy Execution with JIT Optimization** → Delays computation until necessary, allowing Just-In-Time (JIT) compilation to optimize queries dynamically.

✅ **Multi-Threaded Backend** → Utilizes multiple CPU cores efficiently, significantly improving processing speed for large datasets.

FireDucks three-layers of Optimization

Code	Pandas	FireDucks		

def foo_pandas():
(
  pd.read_csv("data.csv")
  .sort_values("amt")["ssn"]
  .head(10)
  .to_csv(“top_10_ssn.csv")
)
	46.1 s	**① Eager and single threaded**	**② Lazy and single threaded**	**③ Lazy and multi-threaded (default)**

		35.7 s	8.28 s	1.5 s

faster than pandas due to efficient algorithms, even when both optimization and parallelization is disabled

benefit of optimization

benefit of parallelization

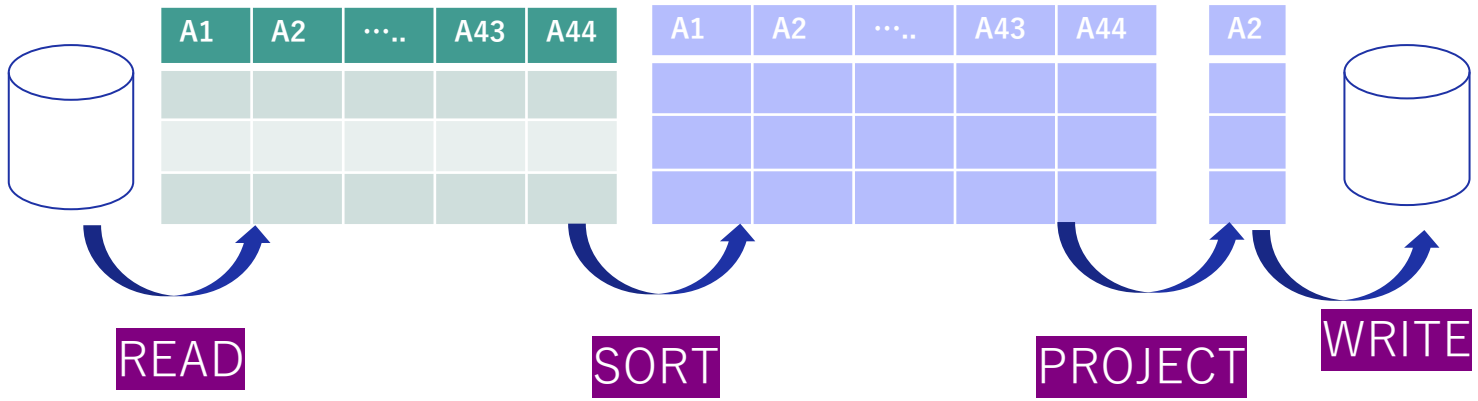
① $ FIREDUCKS_FLAGS=“--benchmark-mode” OMP_NUM_THREADS=1 python -mfireducks.pandas sample.py
② $ OMP_NUM_THREADS=1 python -mfireducks.pandas sample.py
③ $ python **-mfireducks.pandas** sample.py

Sample Query Optimization in Lazy Mode

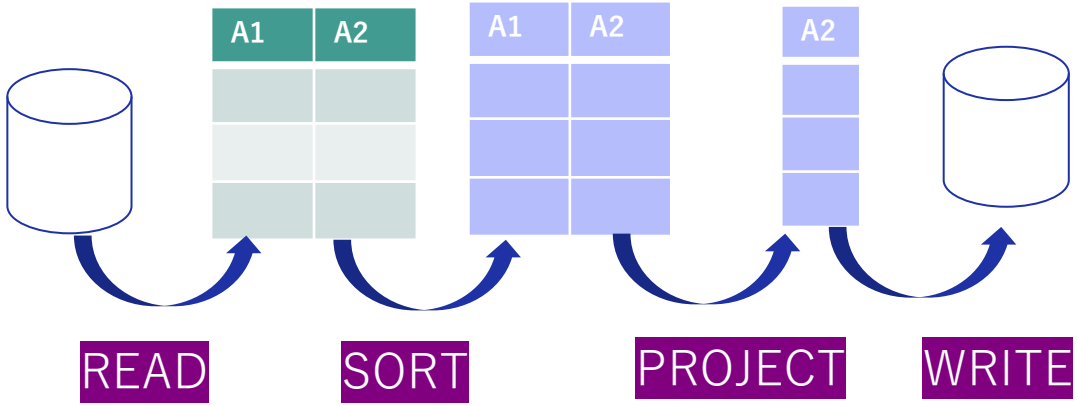
Code

```
def foo_pandas():
(
   pd.read_csv("data.csv")
      .sort_values("amt")["ssn"]
      .head(10)
      .to_csv(“top_10_ssn.csv")
)
```

pandas or FireDucks with eager execution mode



FireDucks with lazy execution mode



Real Use Case: 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

Real Use Case: Projection Pushdown, Predicate Pushdown (2/2)

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

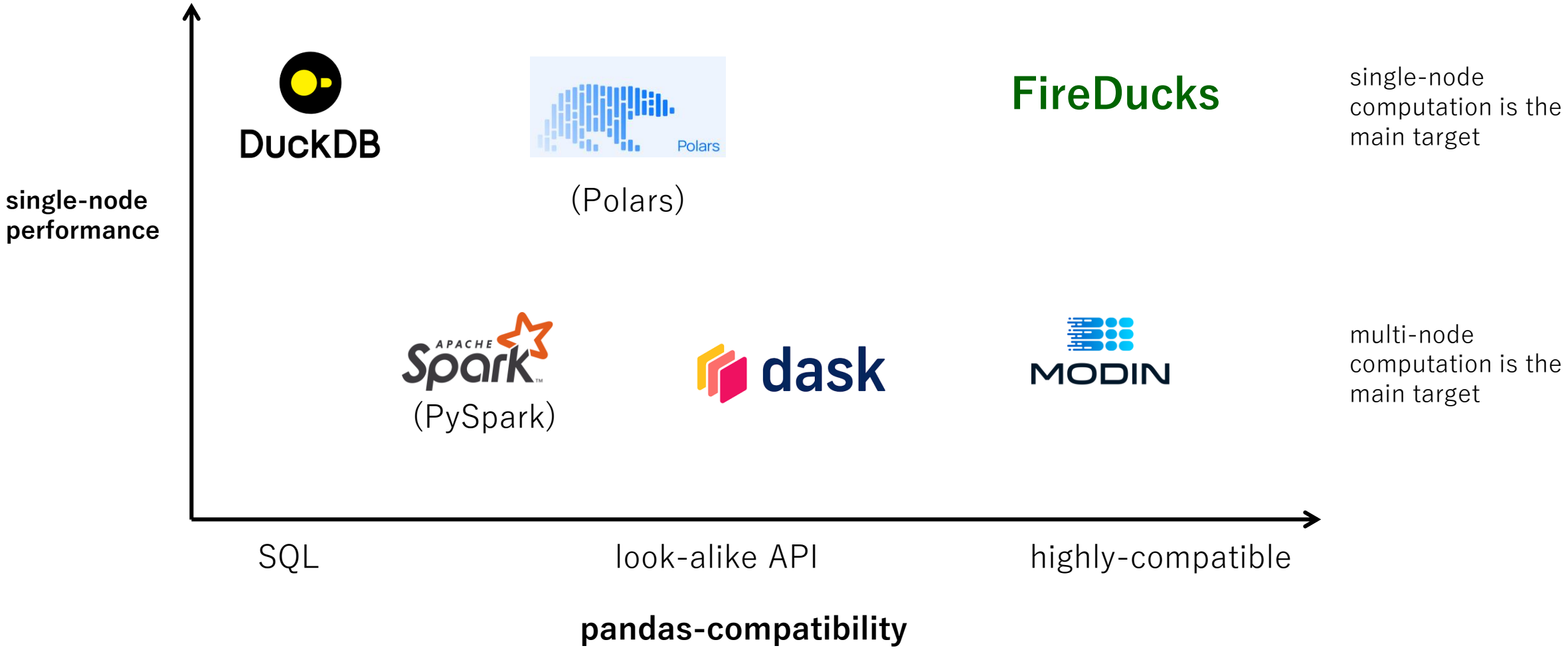
manual optimization

```
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

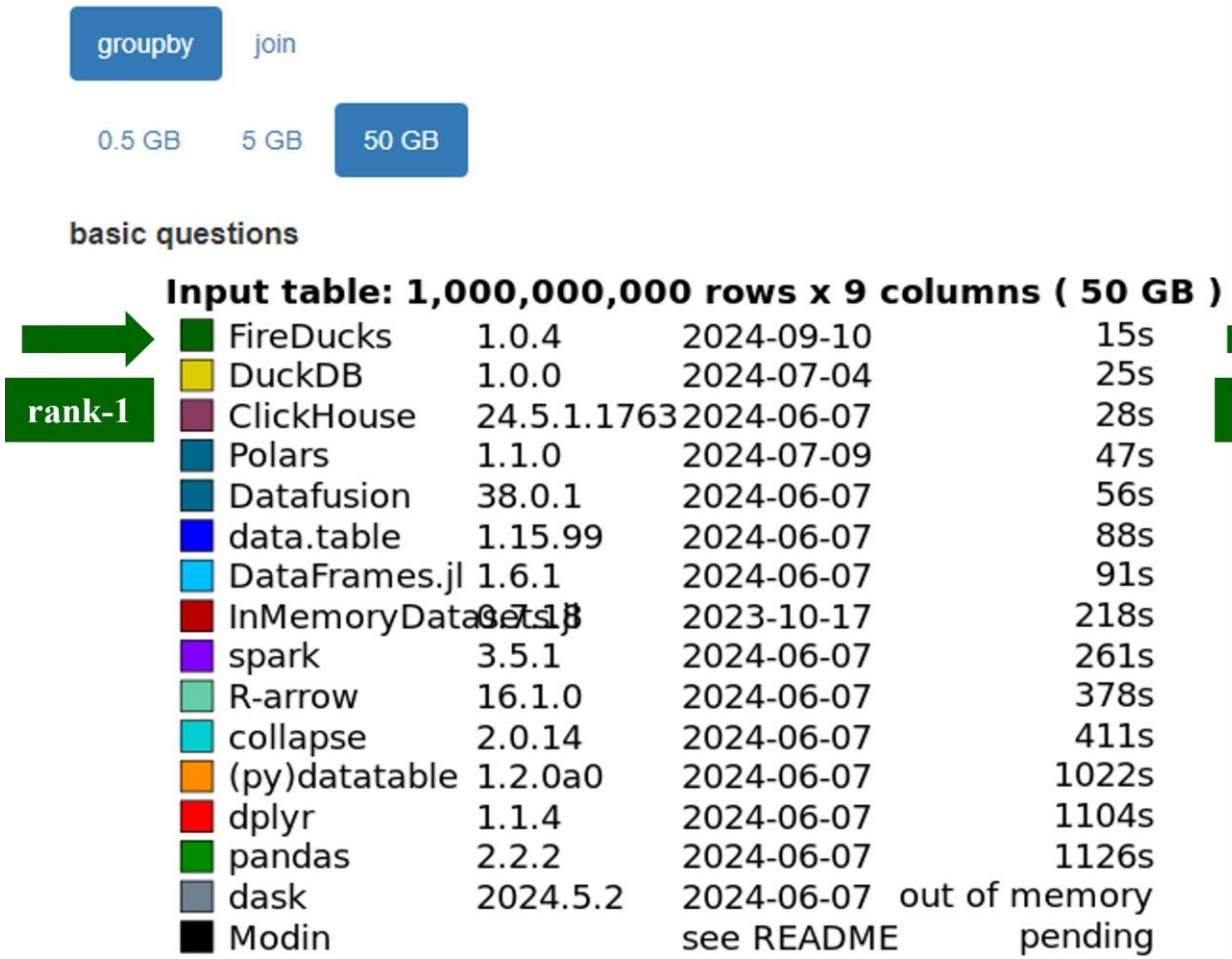
$ python –**mfireducks.pandas** opt_q3.py:
exec-time: 4.8 seconds;
memory consumption: 3.4 GB

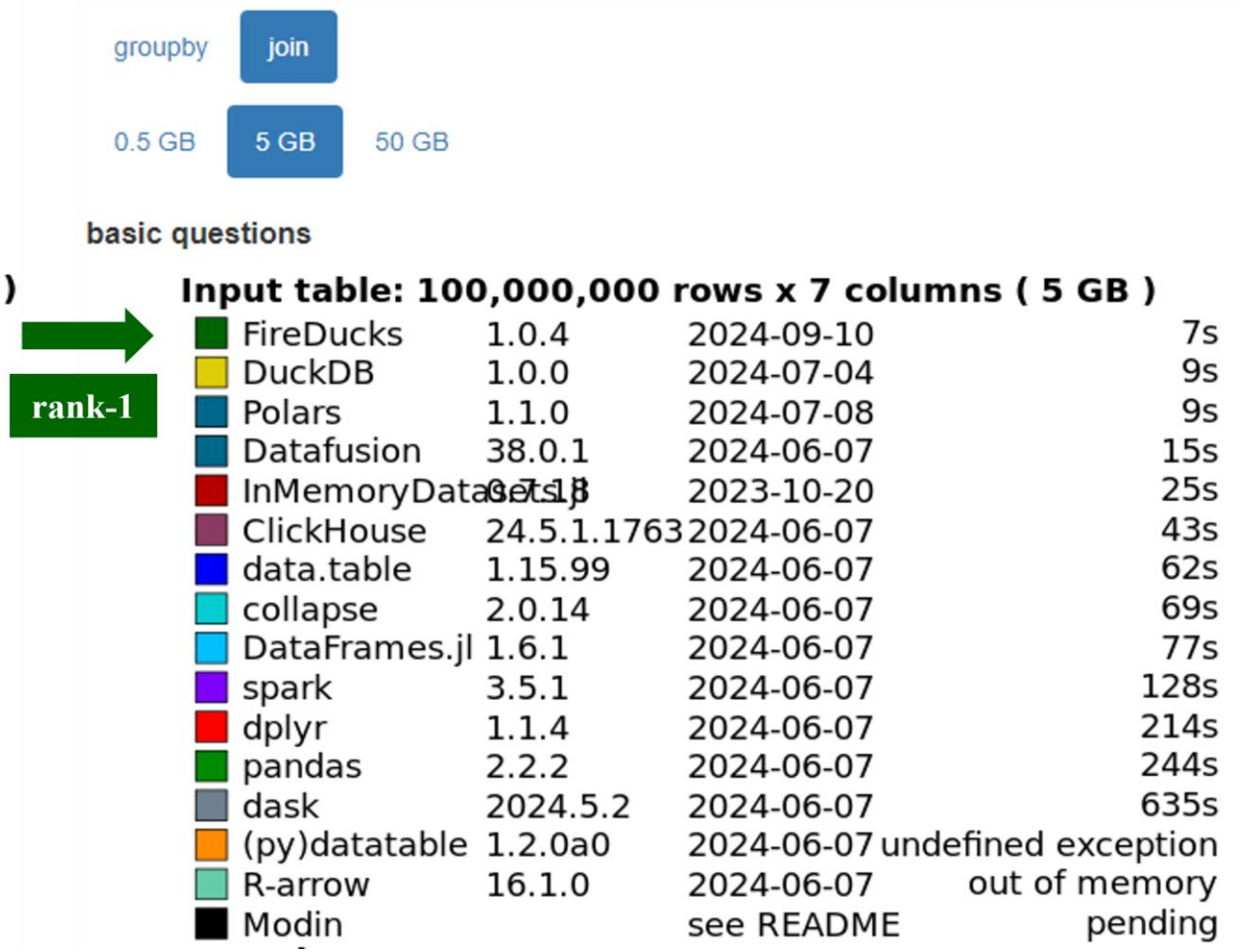
General Overview: DataFrame Libraries



Benchmark (1): DB-Benchmark

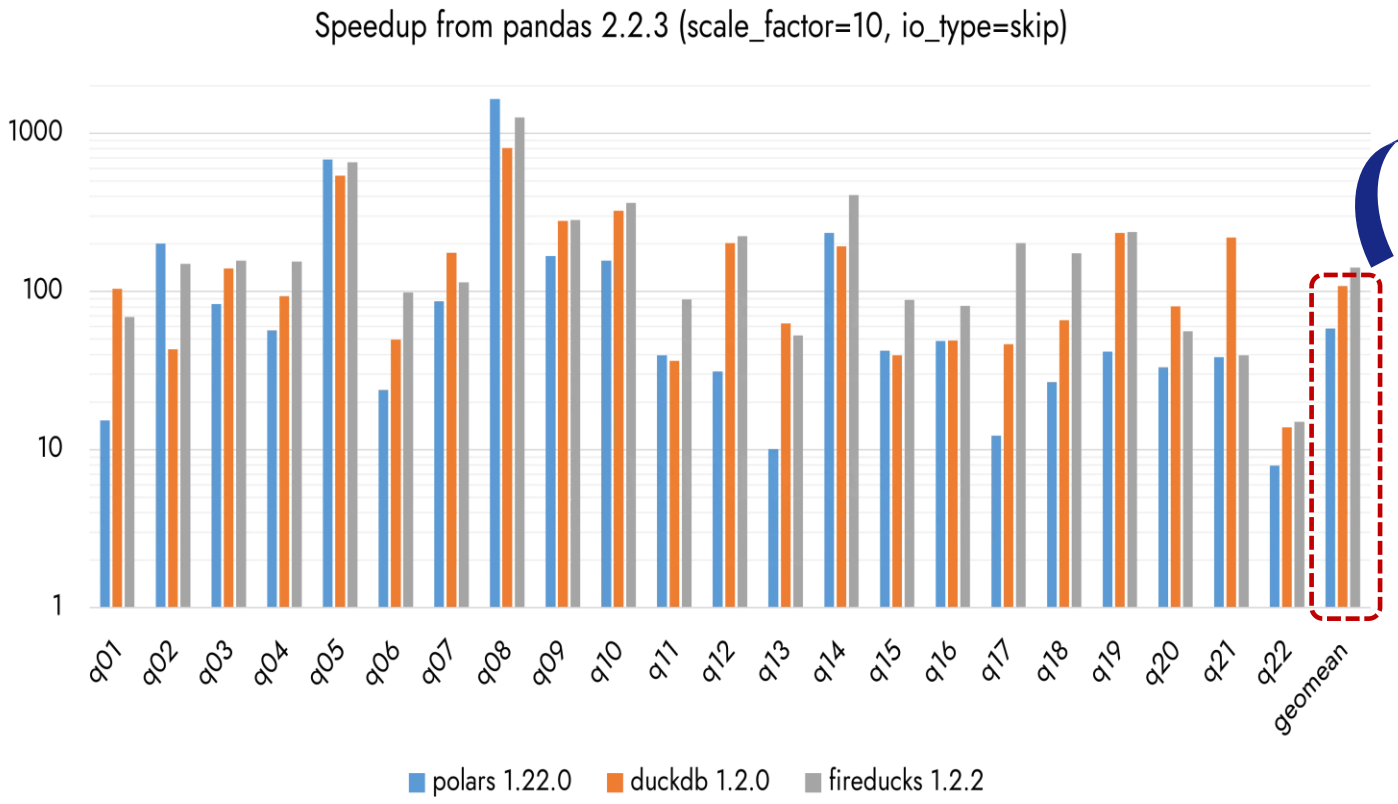
Database-like ops benchmark (https://duckdblabs.github.io/db-benchmark)





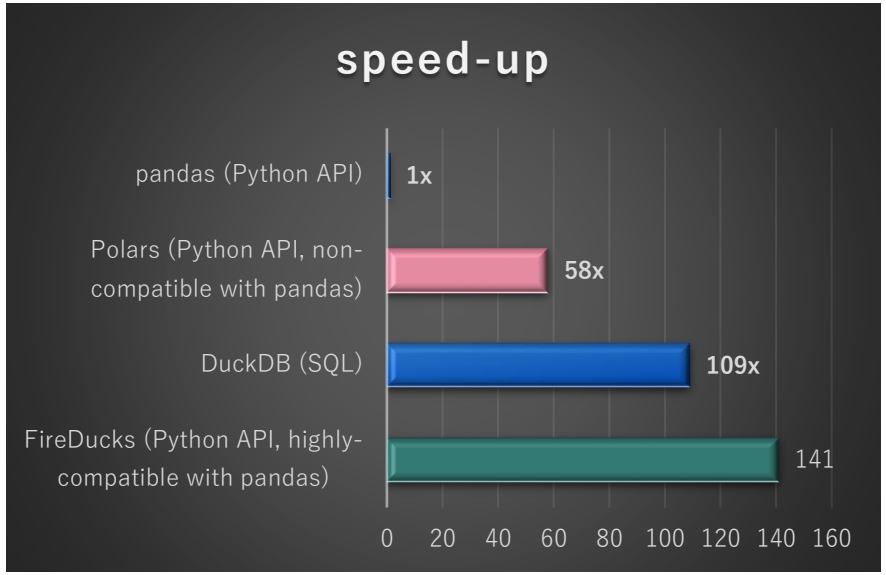
Benchmark (2): Speedup from pandas in TPC-H benchmark

Speedup over 1000x is also possible



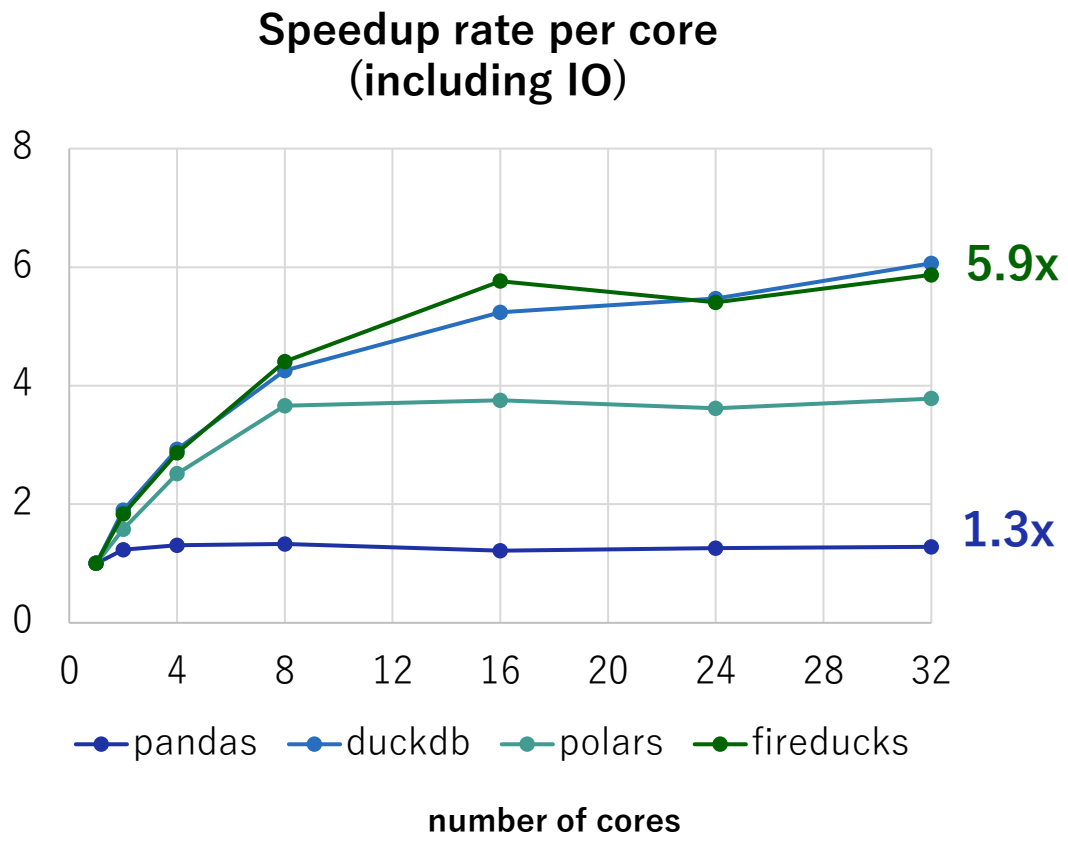
Server Specification

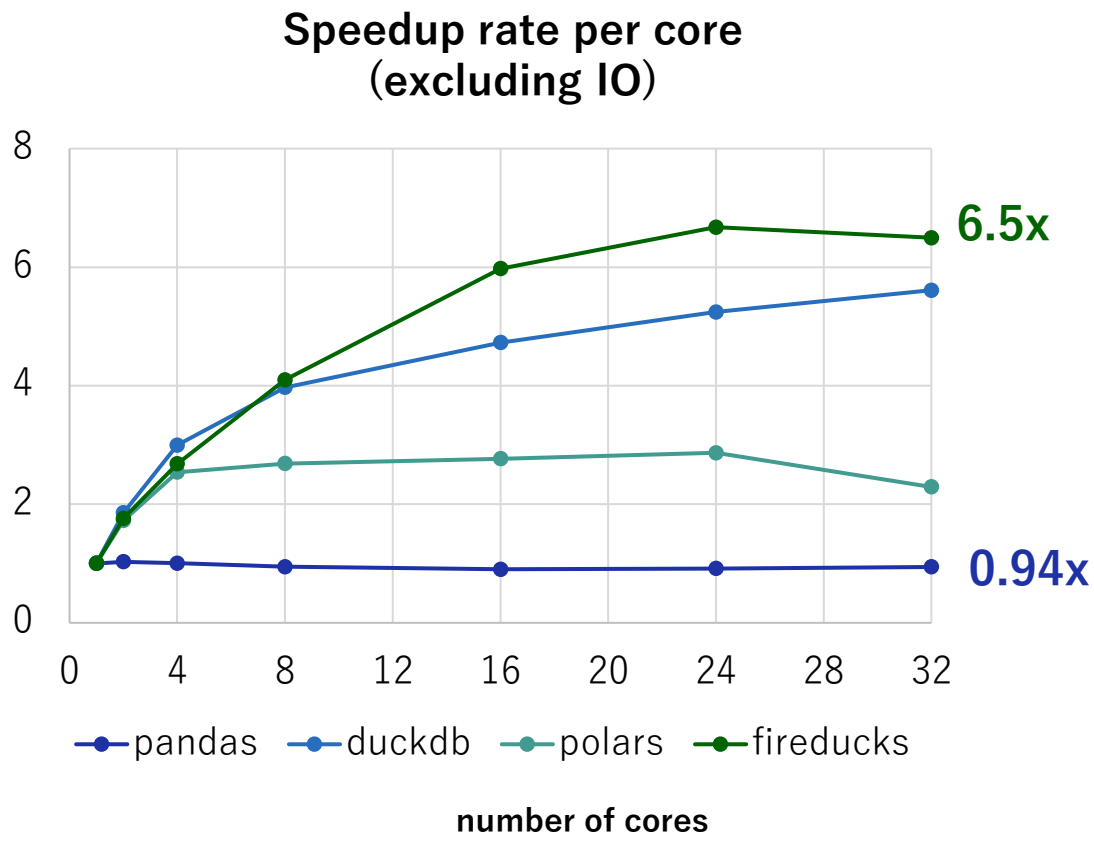
AWS EC2 m7i.8xlarge:
INTEL(R) XEON(R) GOLD 6526Y (32cores), 512 GB



Scalability: DuckDB vs Polars vs FireDucks (TPC-H)

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





Resource on FireDucks

Web site (User guide, benchmark, blog)

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

X(twitter) (Release information)

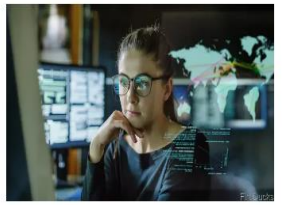
https://x.com/fireducksdev

Github (Issue report)

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

Q/A, communication

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



FireDucks

Compiler Accelerated DataFrame Library for Python with fully-compatible pandas API

Get Started

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

News

Release fileducks-0.12.4 (Jul 09, 2024)

Have you ever thought of speeding up your data analysis in pandas with a compiler?(blog) (Jul 03, 2024)

Evaluation result of Database-like ops benchmark with FireDucks is now available. (Jun 18, 2024)

Accelerate pandas without any manual code changes

Do you have a pandas-based program that is slow? FireDucks can speed-up your programs without any manual code changes. You can accelerate your data analysis without worrying about slow performance due to single-threaded execution in pandas.

Thank You!

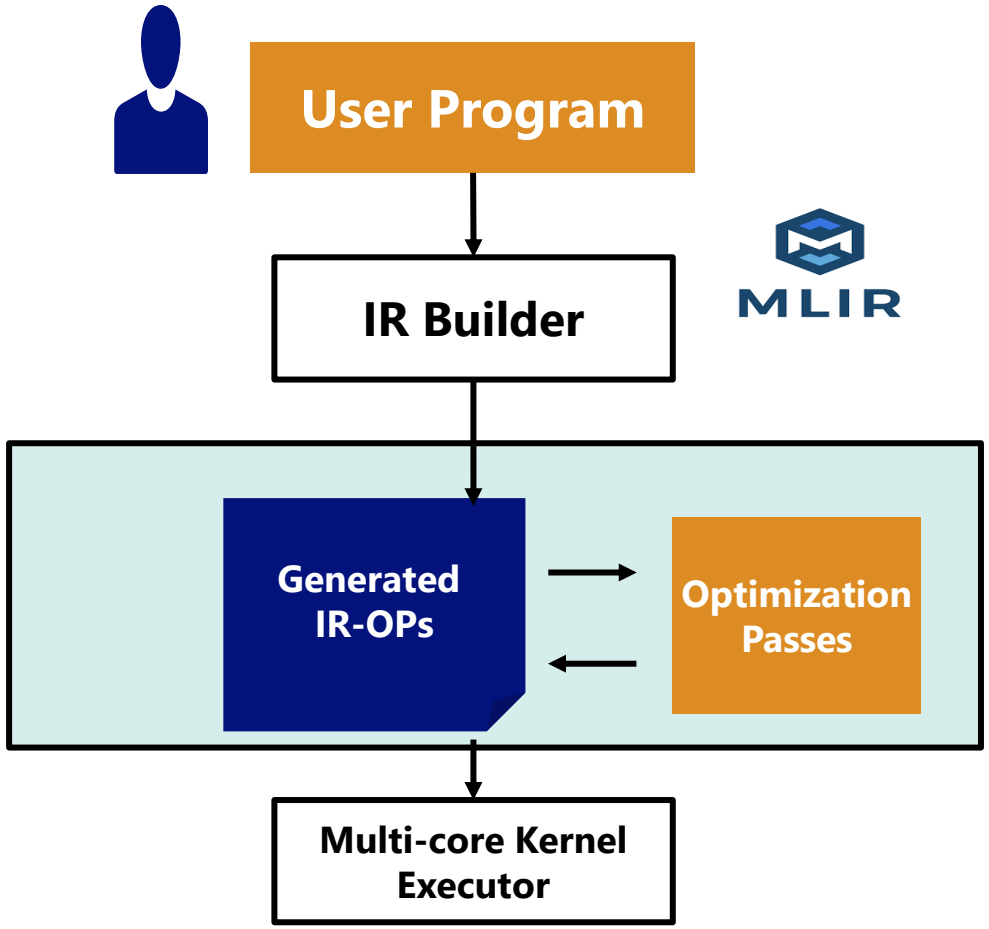
- Focus more on in-depth data exploration using “**pandas**”.
- Let the “**FireDucks**” take care of the optimization for you.
- Enjoy Green Computing!

Frequently Asked Questions

How does FireDucks work?

※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)
```

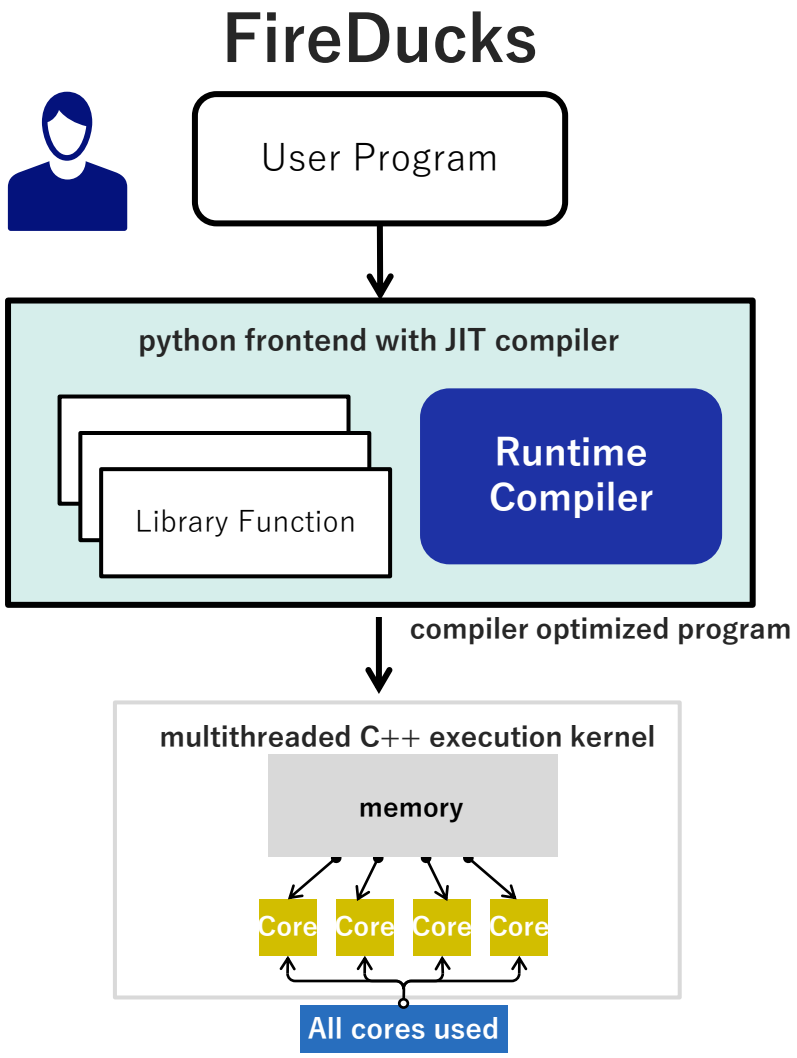
print (result)

```
%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: Write Once, Execute Anywhere

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?

Pandas Specific Optimization – Parameter Tuning

parameter tuning in pandas

department-wise average salaries sorted in descending order

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

groupby("department", sort=True)

```
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=False)
```

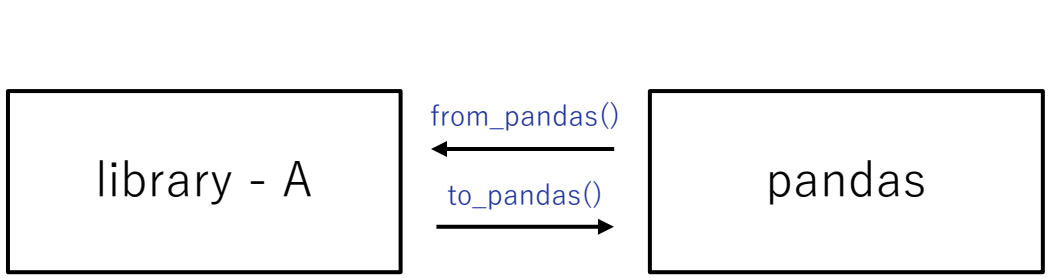
~50 sec

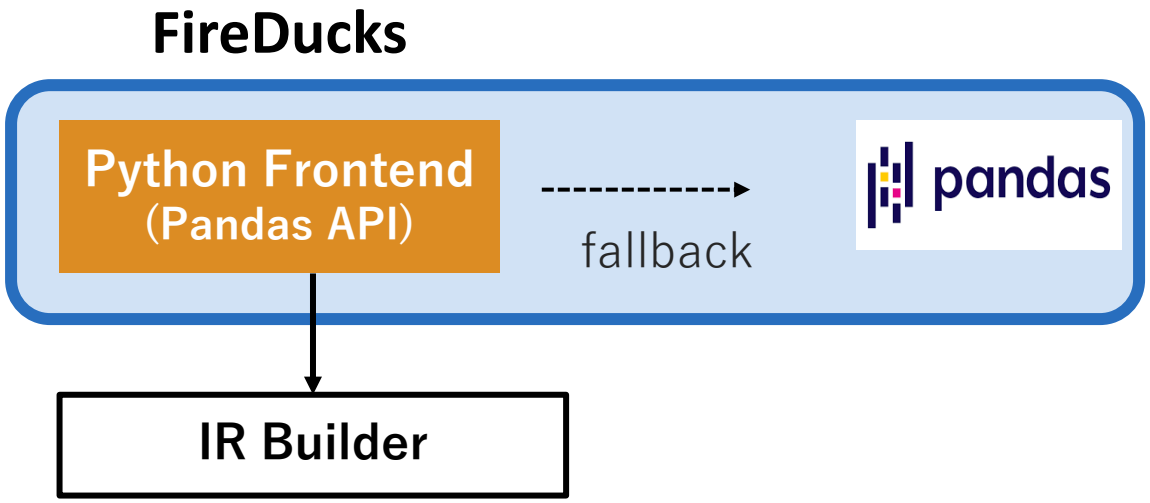
```
df.groupby(["A", "B"],
sort=False)["C"]
  .mean()
  .sort_values(ascending=False)
```

~30 sec

100M samples with high-cardinality

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)
)
r1["val"] = r1["val"].cumsum()
r1.describe()
```

pd.from_pandas(r1["val"].to_pandas().cumsum())

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"

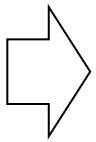
Raise feature request when you encounter some expensive fallback hindering your program performance!

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
```

fireducks time: 0.0000123 sec



IR Builder

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

```
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