\Orchestrating a brighter world NEC

How compiler driven technologies can be useful to speedup data processing in python

Sep 20, 2024

Sourav Saha (NEC)

Agenda

- Icebreaking
- About Pandas
- Tips and Tricks of Optimizing Large-scale Data processing workload
- Compiler driven technologies to optimize the problems
- 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

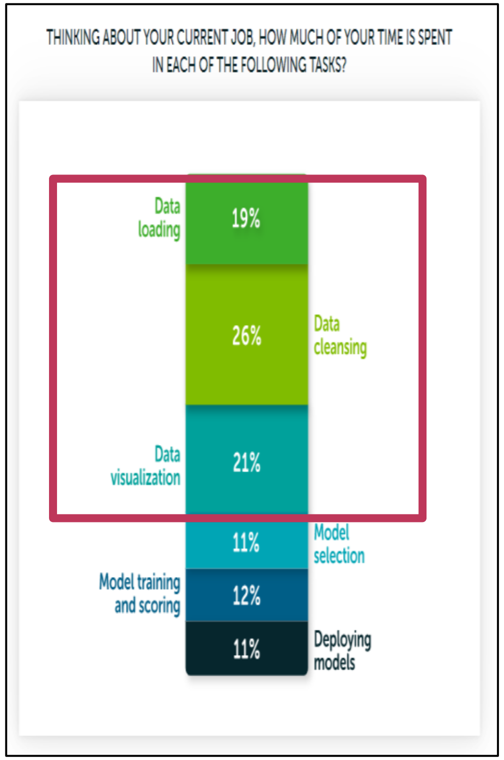
FireDucks

groupby | join | dropna | filter | sort | corr

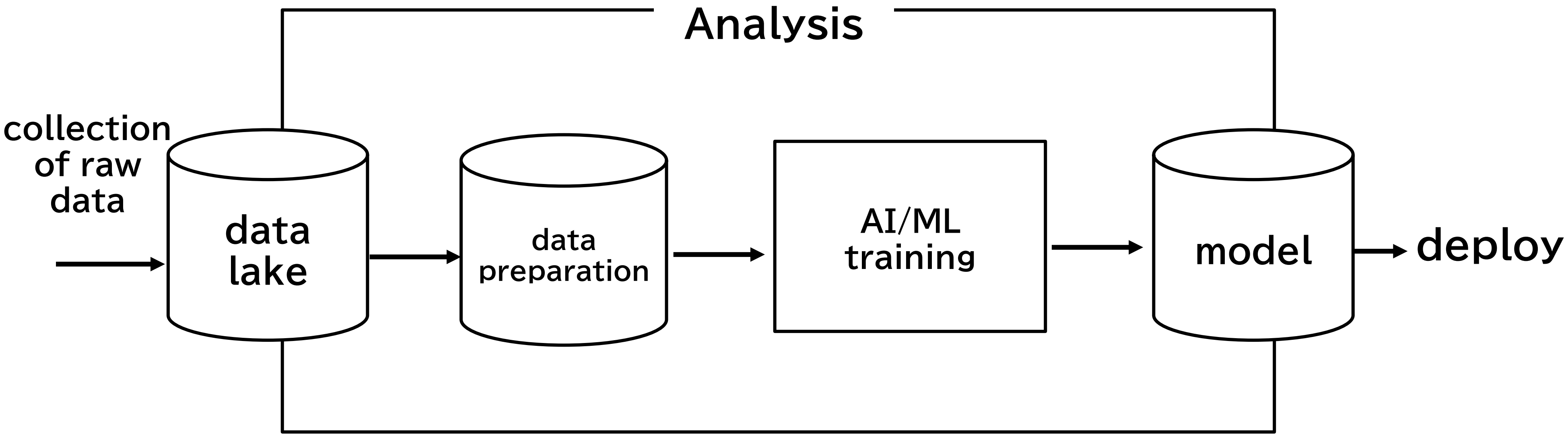
compiler technologies

Workflow of a Data Scientist

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

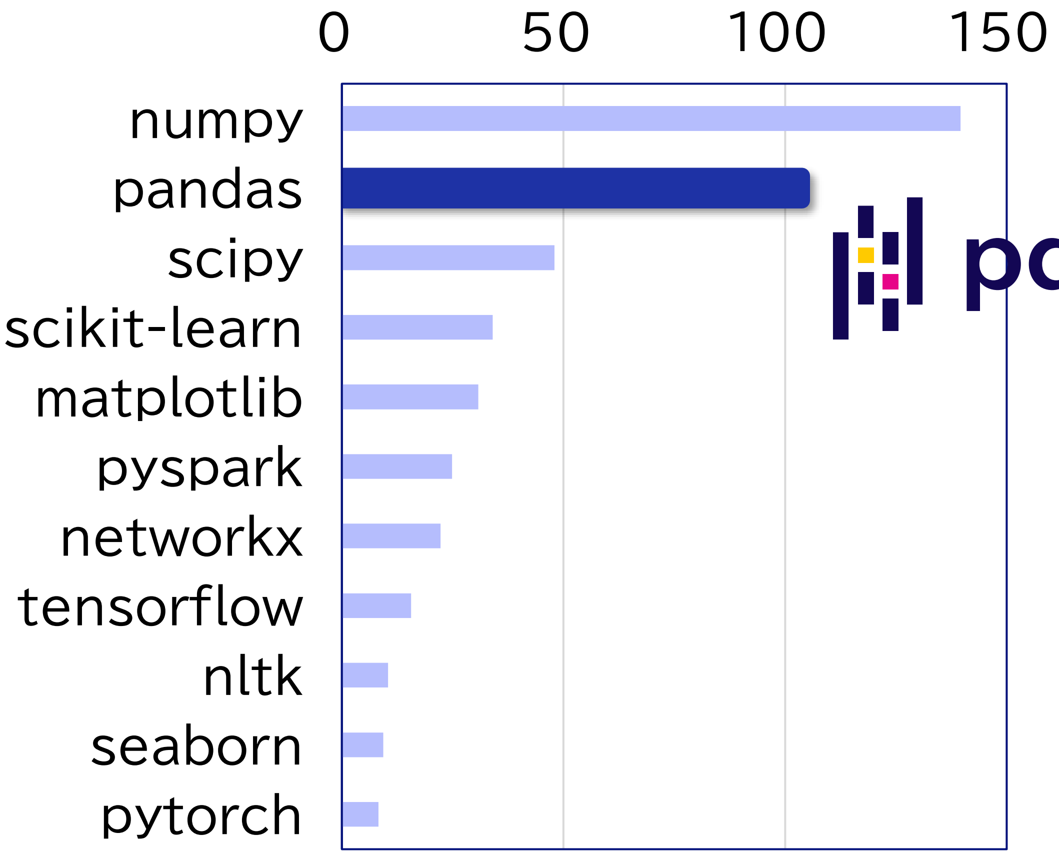


Anaconda:
The State of Data Science 2020



About Pandas (1/2)

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

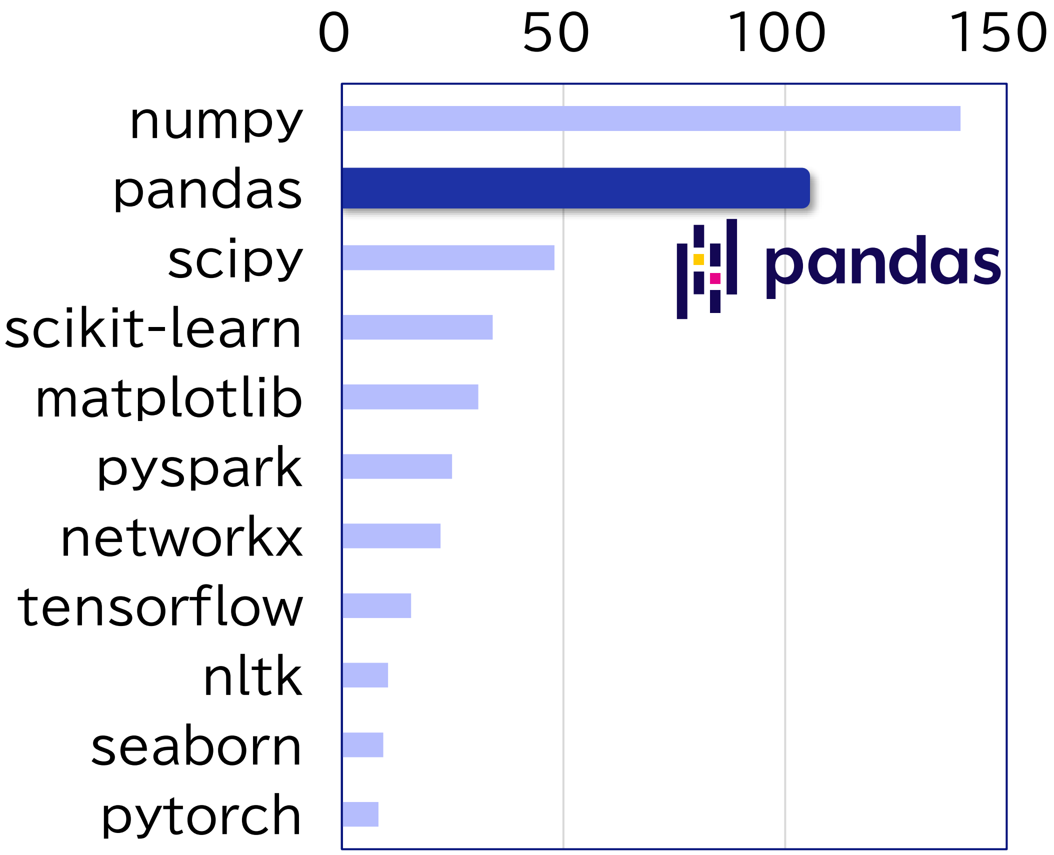


Monthly download from pypi.org
(Data Analytics Libraries)

- 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.

Ice-Breaking Session
(test your pandas skill)

Quick check on basic pandas operations (1/5)

- **Which one of the following is the right method of getting top-2 rows based on the column "A" from table "df"?**

1. df.sort("A", ascending=True).head(2)
2. df["A"].top_k(2)
3. df.sort("A", ascending=False).first(2)
4. df.sort_values("A", ascending=False).head(2)

	A	B
0	2	10
1	5	30
2	1	20
3	3	70
4	7	60
5	8	40
6	4	80

	A	B
5	8	40
4	7	60

Quick check on basic pandas operations (2/5)

- **Which ones of the following are the right methods of performing inner-join of table “df1” with table “df2” on common key-column “x”?**

1. pd.merge(df1, df2, on=“x”, how=“inner”)
2. df1.inner_join(df2, on=“x”)
3. df1.merge(df2, on=“x”, how=“inner”)
4. df1.merge(df2, on=“x”)

df1

	x	y
0	0	10
1	1	20

df2

	x	z
0	0	100
1	1	200
2	2	300

	x	y	z
0	0	10	100
1	1	20	200

Quick check on basic pandas operations (3/5)

◆ **Which one of the following is the right method to remove rows having a missing value?**

1. df.dropna()
2. df.dropna(how="any")
3. df[~df["A"].isnull()]
4. All of the above

	A	B
0	N	10
1	5	30
2	N	20
3	3	70
4	7	60
5	8	40
6	4	80

	A	B
1	5	30
3	3	70
4	7	60
5	8	40
6	4	80

Quick check on basic pandas operations (4/5)

- **Which one of the following is the right method of selecting columns “A”, “D” and “E” from table “df”?**

1. df[[“A”, “D”, “E”]]
2. df.loc[:, [“A”, “D”, “E”]]
3. df.iloc[:, [0, 3, 4]]
4. All of the above

	A	B	C	D	E
0	2	10	10	g	9
1	5	30	69	a	2
2	1	20	31	g	8
3	3	70	45	f	3
4	7	60	59	e	1
5	8	40	66	f	1
6	4	80	97	h	8

	A	D	E
0	2	g	9
1	5	a	2
2	1	g	8
3	3	f	3
4	7	e	1
5	8	f	1
6	4	h	8

Quick check on basic pandas operations (5/5)

- **Select the options for appending a new column “F“ by doubling the column “B” from table “df”.**

1. df[“F”] = df[“B”] * 2
2. df.assign(F=lambda x: x[“B”] * 2)
3. df.with_columns(df.col(“B”) * 2).alias(“F”)
4. df.insert(5, "F", df["B"]*2)

	A	B	C	D	E
0	2	10	10	g	9
1	5	30	69	a	2
2	1	20	31	g	8
3	3	70	45	f	3
4	7	60	59	e	1
5	8	40	66	f	1
6	4	80	97	h	8

	A	B	C	D	E	F
0	2	10	10	g	9	20
1	5	30	69	a	2	60
2	1	20	31	g	8	40
3	3	70	45	f	3	140
4	7	60	59	e	1	120
5	8	40	66	f	1	80
6	4	80	97	h	8	160

Performance Challenges & Best Practices to follow

(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

challenges with pandas APIs when writing chained expression

```
def foo(filename):
  df = pd.read_csv(filename)
  t1 = df.drop_duplicates()
  t2 = t1[t1["B"] > 0.20]
  t3 = t2.sort_values("B")
  t4 = t3.head(2)
  return t4
```

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

filter

t2: ~8 GB

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

sort

t3: ~8 GB

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

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)
  )
```

```
def foo(filename):
  return (
    pd.read_csv(filename)
      .drop_duplicates()
      .query("B > 0.20")
      .sort_values("B")
      .head(2)
  )
```

query(): allows you to write SQL-like conditional expression, helping you to perform filter on the current state of the input frame, but its a little slower as it parses the input string to construct the filter mask.

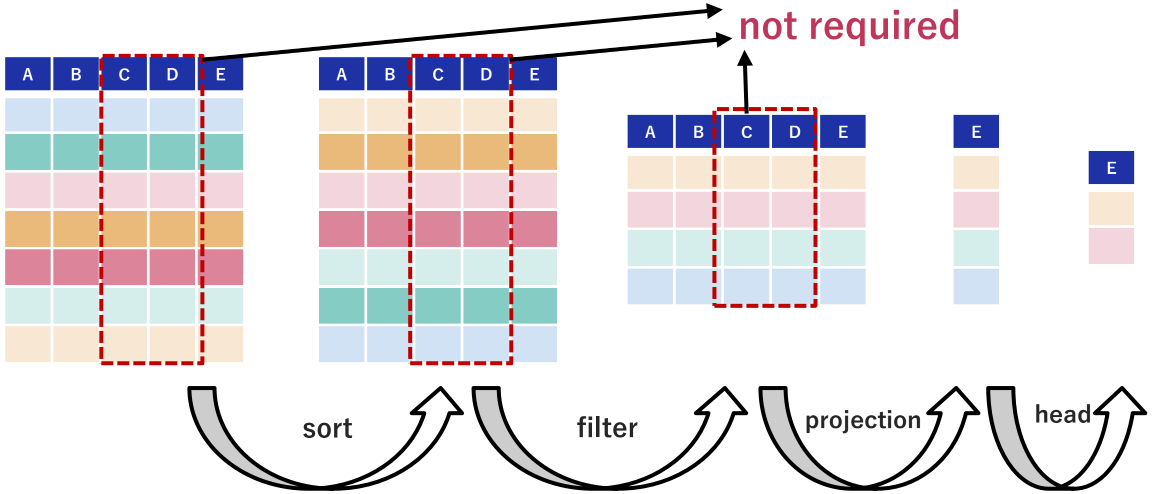
```
def foo(filename):
  return (
    pd.read_csv(filename)
      .drop_duplicates()
      .pipe(lambda tmp: tmp[tmp["B"] > 0.20]
      .sort_values("B")
      .head(2)
  )
```

pipe(): a convenient method allowing you to perform a given operation (like filter etc.) on the current state of the input frame without introducing computational overhead.

(2) importance of execution order

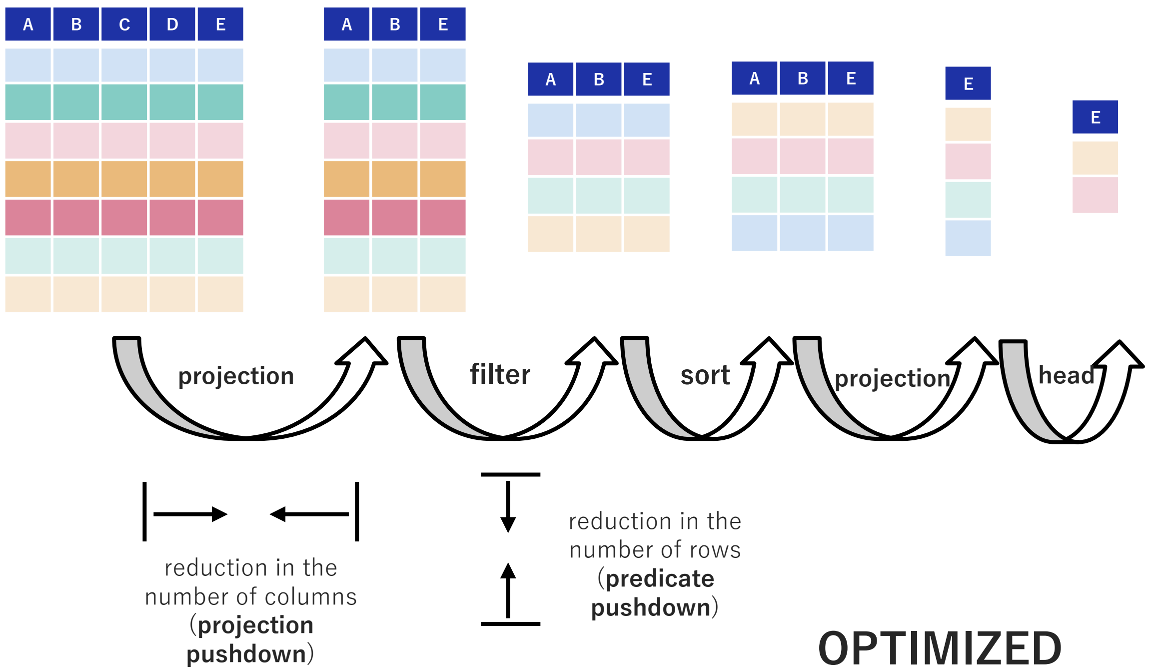
```
df.sort_values("A")
  .query("B > 1")["E"]
  .head(2)
```

※ 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)
```



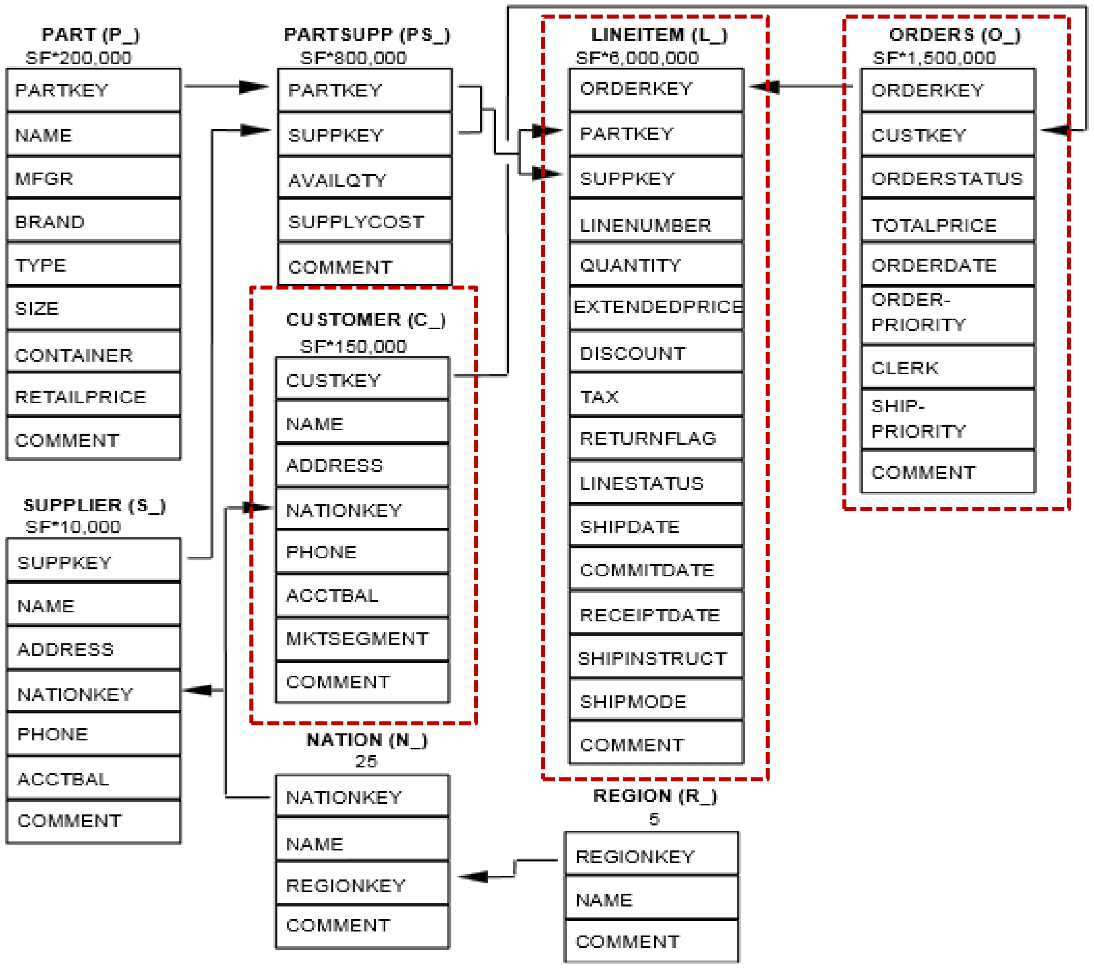
OPTIMIZED QUERY

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

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

```sql
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;
```

```python
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"]] # (2/8)
f_cust = cust[cust[“c_mktsegment"] == "BUILDING"]

# projection-filter: to reduce scope of “orders” table to be processe
ord = orders[[“o_custkey”, “o_orderkey”, “o_orderdate”, "o_shippriority"]] (4/9)
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"]] (4/16)
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)
)
```

Automatic Optimization

Idea #1

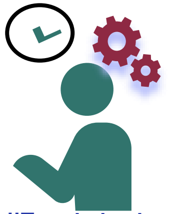
- **Can such optimization be automated?**
  - Yes, using LLVM/MLIR define-by-run mechanism we can build specialized intermediate representation for each pandas API.
  - The generated IRs can be parsed to implement different domain-specific optimizations, such as projection pushdown, predicate pushdown, 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



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

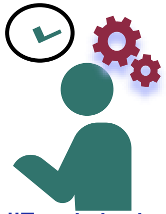
- Pandas methods are slow due to poor memory utilization and single-core computation.
- But pandas is one of the most popular data manipulation tools.
- **How can we solve the core performance issue in pandas while keeping the same API for users?**
  - Well, we can
    - have a frontend with pandas API 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



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 backend API

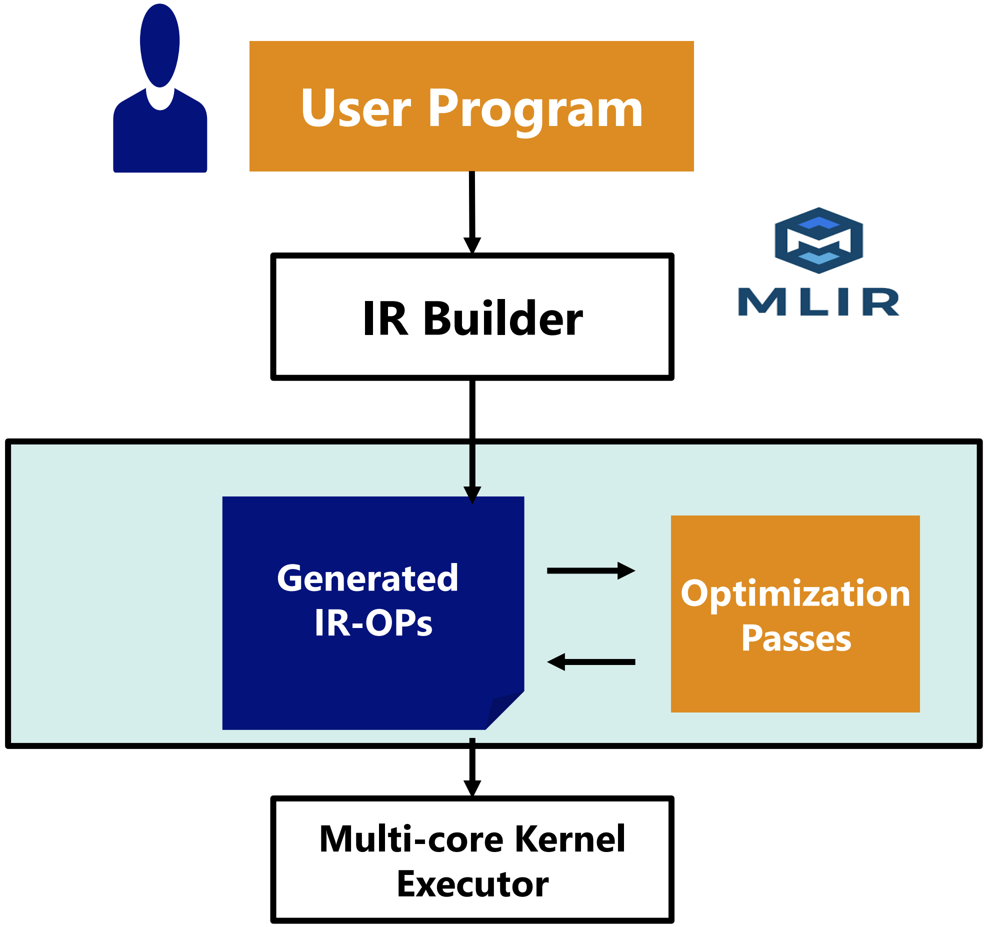
```
t1 = backend::project_columns(df, {“A”, “B”, “C”});
t2 = backend::filter_rows(t1, “B > 1”);
t3 = backend::sort_values(t2, “A”);
t4 = backend::project_columns(t3, {“E”});
t5 = backend::slice_rows(t4, 2);
```

Introducing FireDucks

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

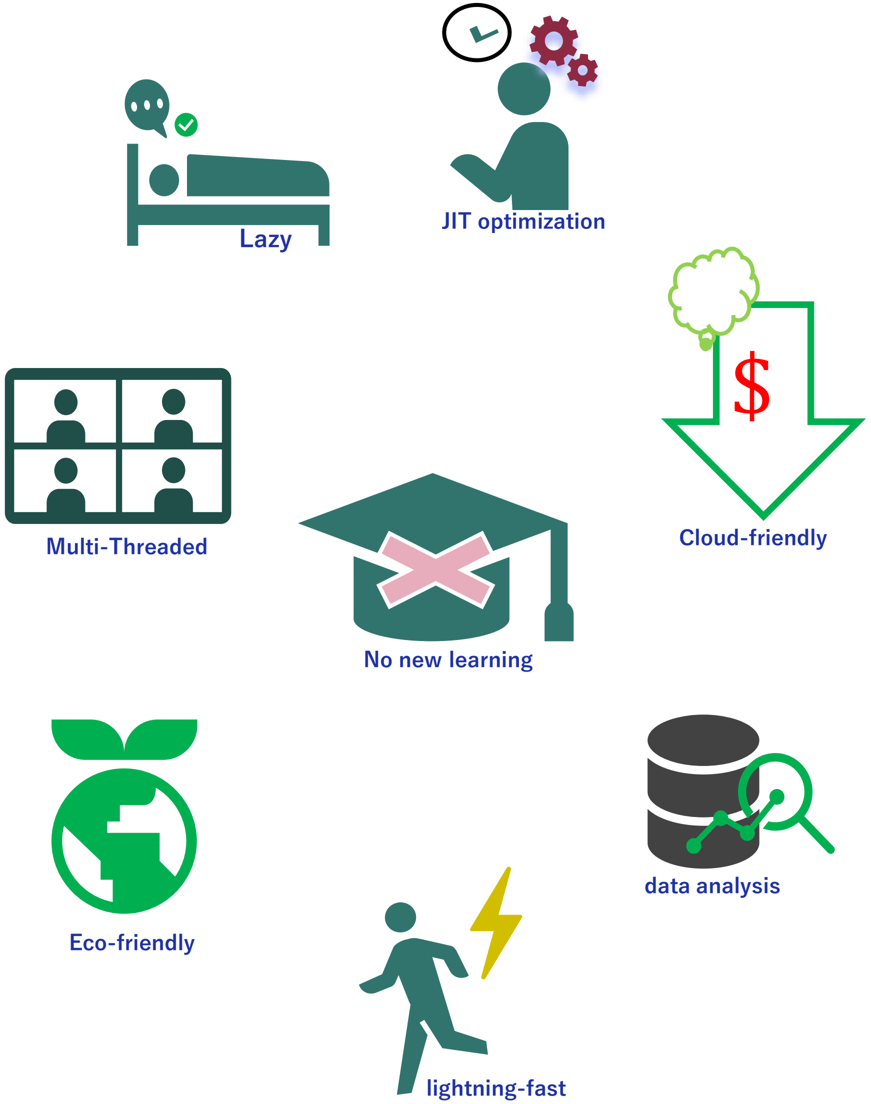
FireDucks (**F**lexible **IR** **E**ngine for DataFrame) is a high-performance compiler-accelerated DataFrame library with highly compatible pandas APIs.

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

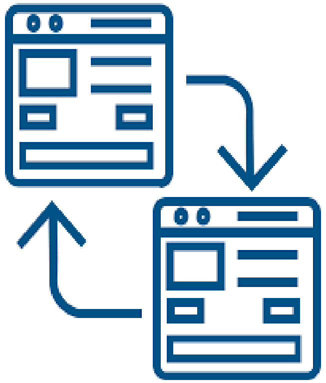
- FireDucks is highly compatible with pandas API
  - seamless integration is possible not only for an existing pandas program but also for any external libraries (like seaborn, scikit-learn, etc.) that internally use pandas dataframes.
- No extra learning is required
- No code modification is required

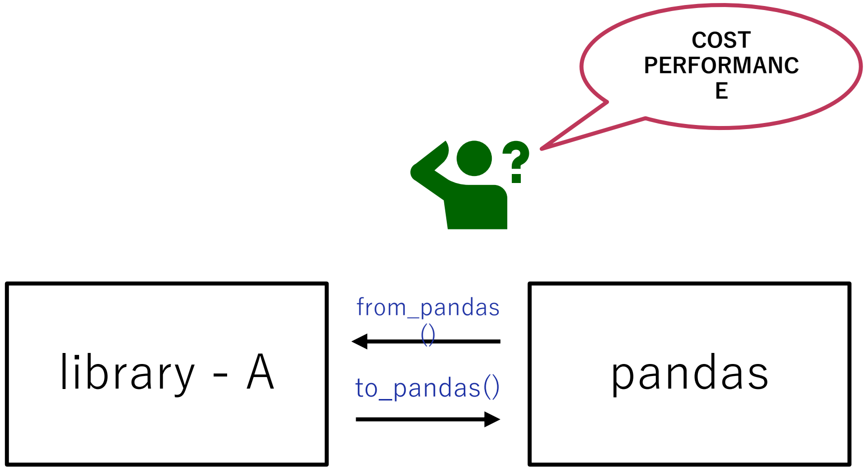


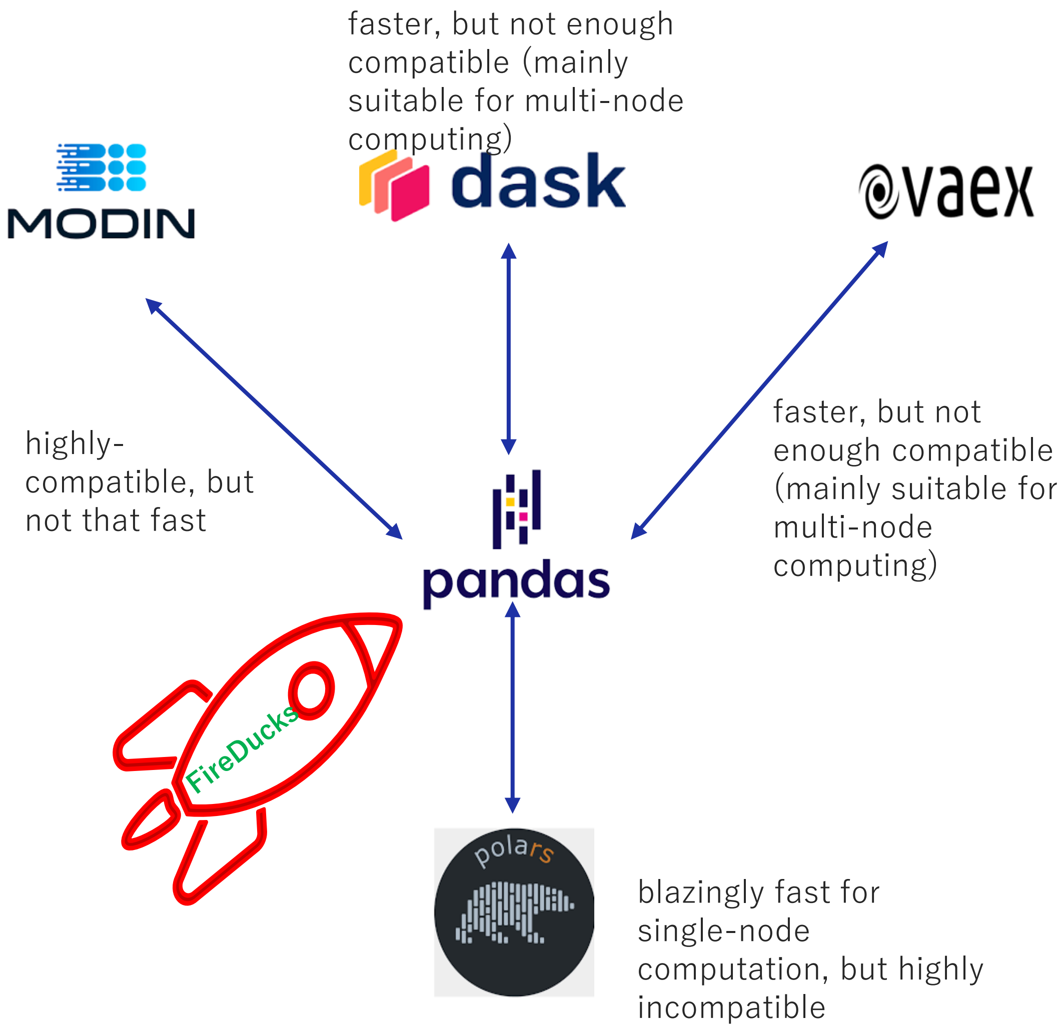
Seamless Integration with pandas

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.







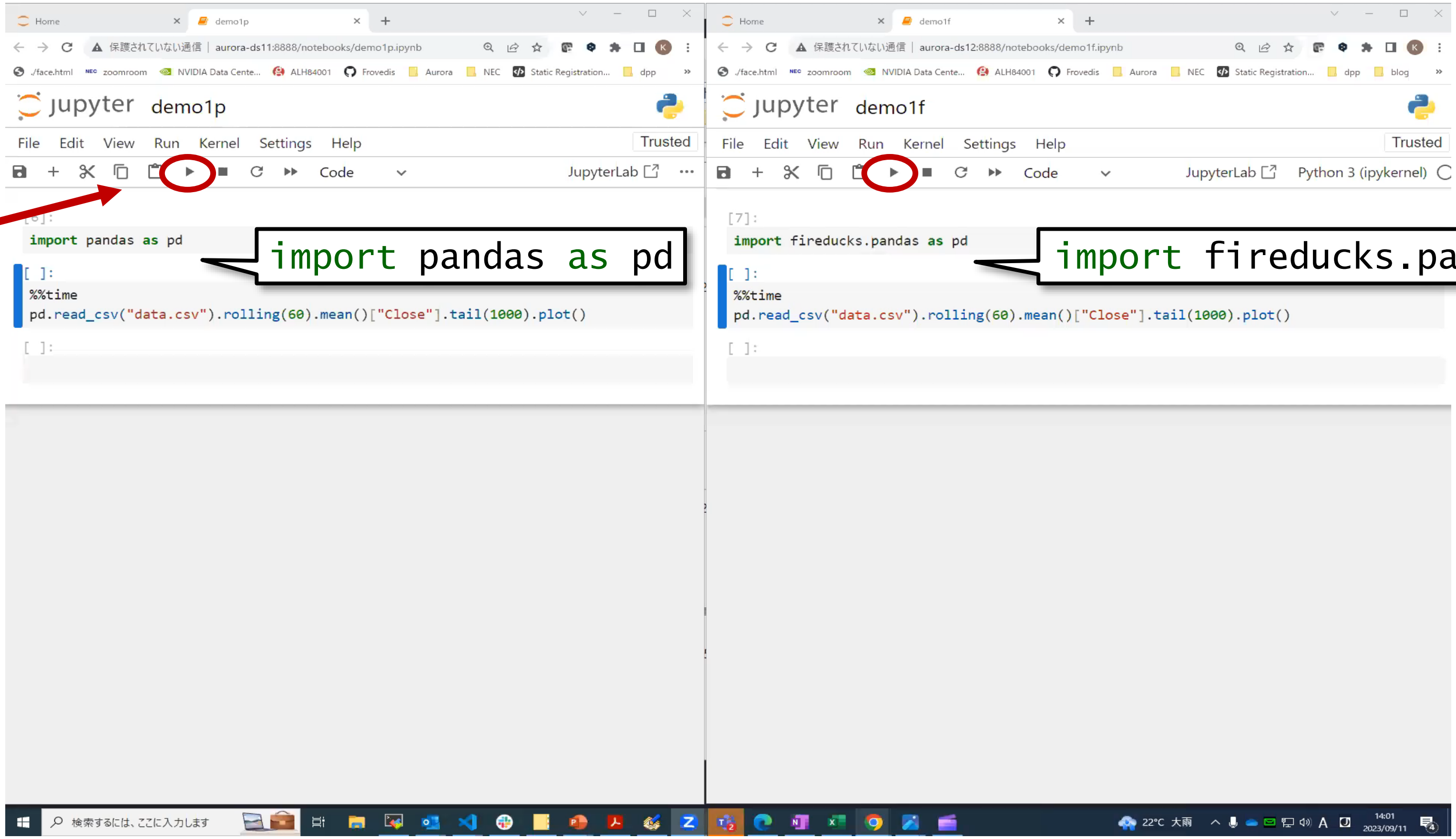
Let’s Have a Quick Demo!

```
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`

pandas: 4.06s

~15x

FireDucks: 275ms

data.csv:
[Bitcoin Historical Data]

Usage of FireDucks

1. Explicit Import

easy to import

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

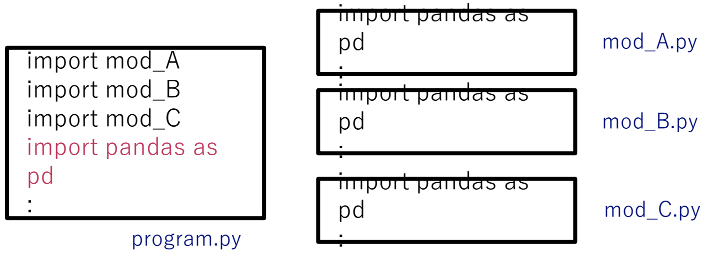
simply change the import statement

2. Import Hook

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

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

zero code modification



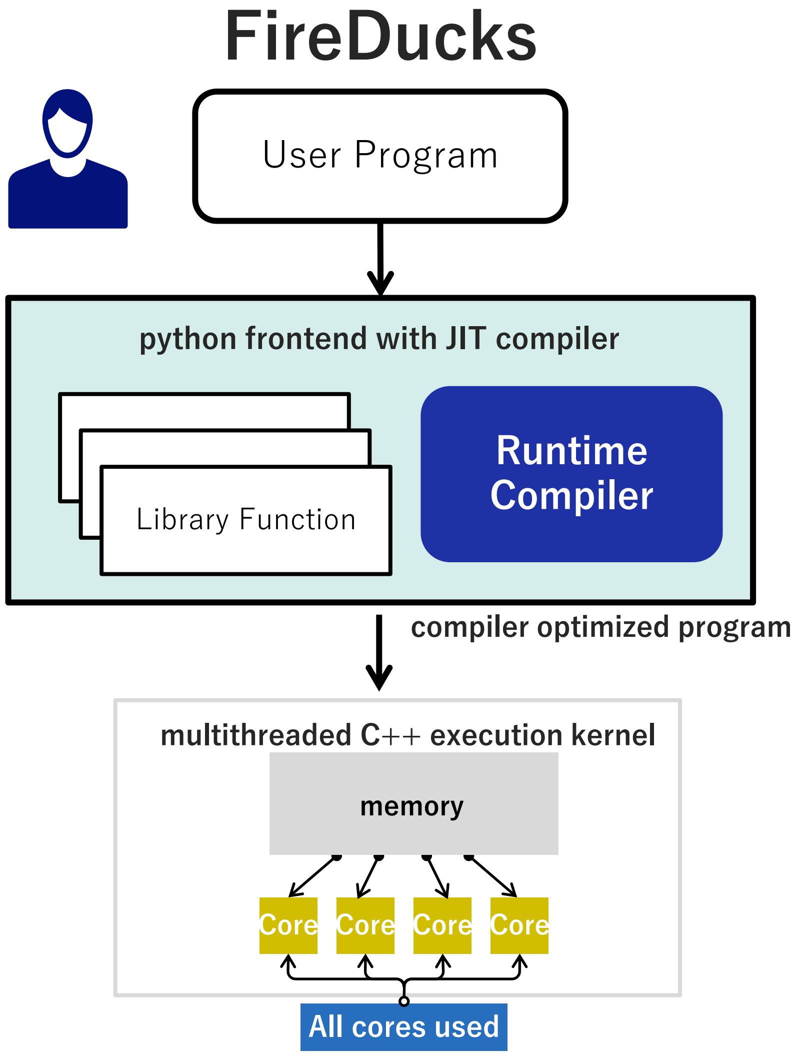
3. Notebook Extension

FireDucks provides simple import extension for interative notebooks.

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

simple integration in a notebook

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.

IR-driven Lazy-execution addresses memory issue with intermediate tables

```
def foo(filename):
  df = pd.read_csv(filename)
  t1 = df.drop_duplicates()
  t2 = t1[t1["B"] > 0.20]
  t3 = t2.sort_values("B")
  t4 = t3.head(2)
  return t4

ret = foo("data.csv")
print(ret.shape)
```

example without chained expression

```
def foo(filename):
  return (
    pd.read_csv(filename)
      .drop_duplicates()
      .query("B > 0.20")
      .sort_values("B")
      .head(2)
  )

ret = foo("data.csv")
print(ret.shape)
```

example with chained expression

```
%t3 = read_csv_with_metadata('dummy.csv', ...)
%t4 = drop_duplicates(%t3, ...)
%t5 = project(%t4, 'B')
%t6 = gt.vector.scalar(%t5, 0.20)
%t7 = filter(%t4, %t6)
%t8 = sort_values(%t7, ['B'], [True])
%t9 = slice(%t8, 0, 2, 1)
%v10 = get_shape(%t9)
return(%t9, %v10)
```

IR Generated by FireDucks
(can be inspected when setting environment variable FIRE_LOG_LEVEL=3)

Compiler Specific Optimizations

- **same operation on the same data repeatedly**

```
# Find year and month-wise average sales
s = pd.Series(["2020-01-01", "2021-01-01", "2022-01-01"])

df = pd.DataFrame()
df["year"] = pd.to_datetime(s).dt.year
df["month"] = pd.to_datetime(s).dt.month
df["sales"] = [100, 200, 500]
r = df.groupby(["year", "month"])["sales"].mean()
print(r)
```

```
%t8 = to_datetime(%t7, None)
%t9 = datetime_extract(%t8, 'year')
%t10 = setitem(%t6, 'year', %t9)
%t11 = to_datetime(%t7, None)
%t12 = datetime_extract(%t11, 'month')
%t13 = setitem(%t10, 'month', %t12)
%t14 = from_pandas.frame.metadata(%arg4, %arg5)
%t15 = setitem(%t13, 'sales', %t14)
%t16 = groupby_select_agg(%t15, ['year', 'month'], ['mean'], [], [], 'sales')
%v17 = get_shape(%t16)
return(%t16, %v17)
```

Common Sub-expression Elimination

```
# Find year and month-wise average sales
s = pd.Series(["2020-01-01", "2021-01-01", "2022-01-01"])
tmp = pd.to_datetime(s).

df = pd.DataFrame()
df["year"] = tmp.dt.year
df["month"] = tmp.dt.month
df["sales"] = [100, 200, 500]
r = df.groupby(["year", "month"])["sales"].mean()
print(r)
```

```
%t8 = to_datetime(%t7, None)
%t9 = datetime_extract(%t8, 'year’)
%t11 = setitem(%t6, 'year', %t9)
%t12 = datetime_extract(%t8, 'month’)
%t14 = setitem(%t11, 'month', %t12)
%t15 = from_pandas.frame.metadata(%arg4, %arg5)
%t16 = project(%t14, ['year', 'month'])
%t17 = setitem(%t16, 'sales', %t15)
%t18 = groupby_select_agg(%t17, ['year', 'month'], ['mean'], [], [], 'sales')
%v19 = get_shape(%t18)
return(%t18, %v19)
```

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")
```

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

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

```
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.

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

```
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)
```

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

Domain Specific Optimization

```
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"]] # (2/8)
f_cust = cust[cust[“c_mktsegment"] == "BUILDING"]

# projection-filter: to reduce scope of “orders” table to be processe
ord = orders[[“o_custkey”, “o_orderkey”, “o_orderdate”, "o_shippriority"]] (4/9)
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"]] (4/16)
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)
)
```

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

Pandas Specific Optimization – Auto-selection of optimized method

```
# Datetime Extractor

year = date.dt.strftime(“%Y”).astype(int)
month = date.dt.strftime(“%m”).astype(int)
day = date.dt.strftime(“%d”).astype(int)
```

```
# Datetime Extractor

year = date.dt.year
month = date.dt.month
day = date.dt.day
```

Pandas Specific Optimization – Optimization on Index

```
sorted = df.sort_values(“a”).reset_index(drop=True)
```

```
sorted = df.sort_values(“a”, ignore_index=True)
```

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.jl	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

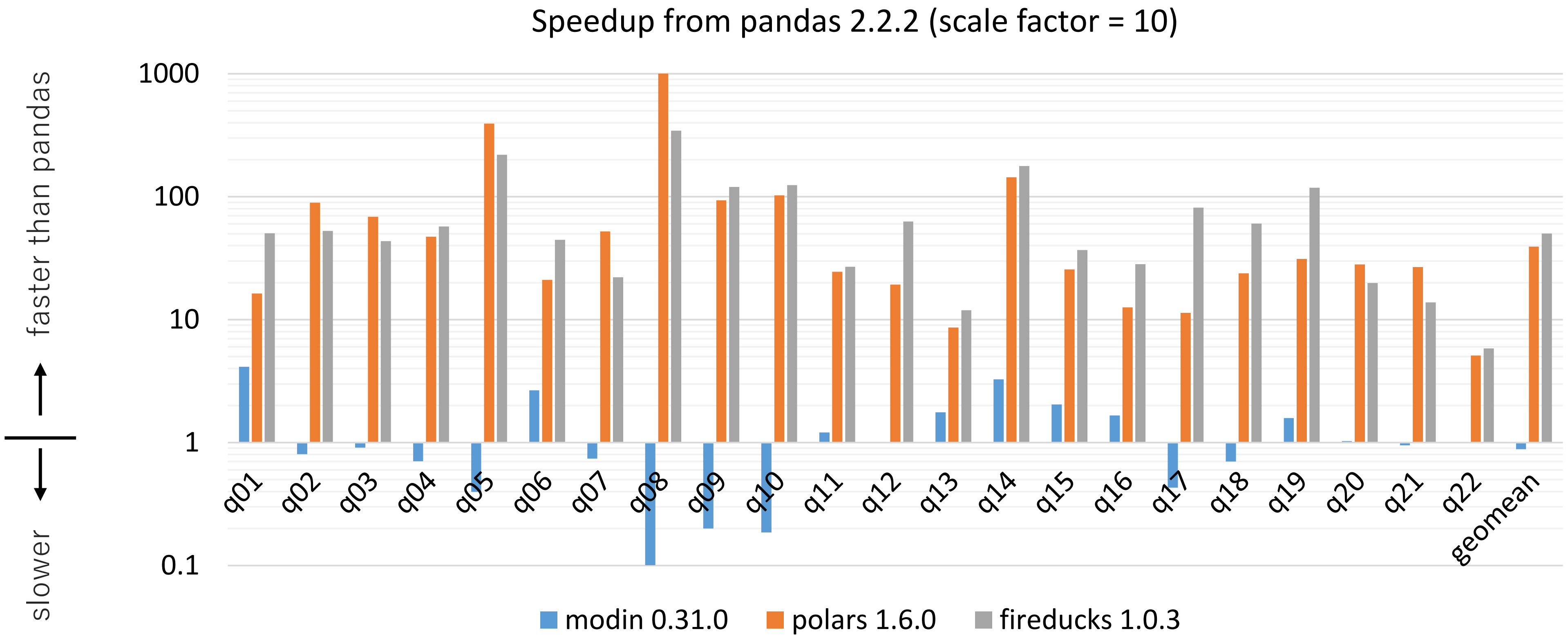
FireDucks	1.0.4	2024-09-10	7s
DuckDB	1.0.0	2024-07-04	9s
Polars	1.1.0	2024-07-08	9s
Datafusion	38.0.1	2024-06-07	15s
InMemoryDataSets.jl	0.7.18	2023-10-20	25s
ClickHouse	24.5.1.1763	2024-06-07	43s
data.table	1.15.99	2024-06-07	62s
collapse	2.0.14	2024-06-07	69s
DataFrames.jl	1.6.1	2024-06-07	77s
spark	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

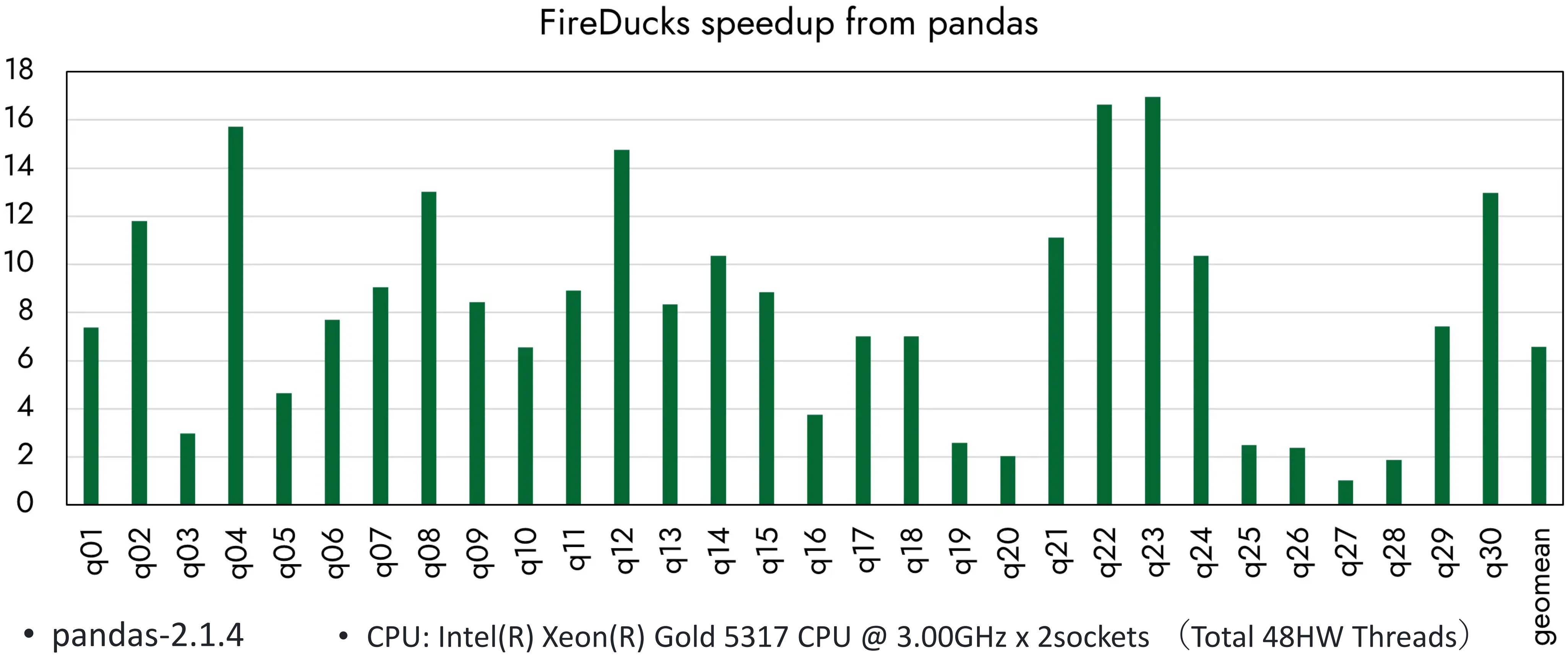


Comparison of DataFrame libraries (average speedup)

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)

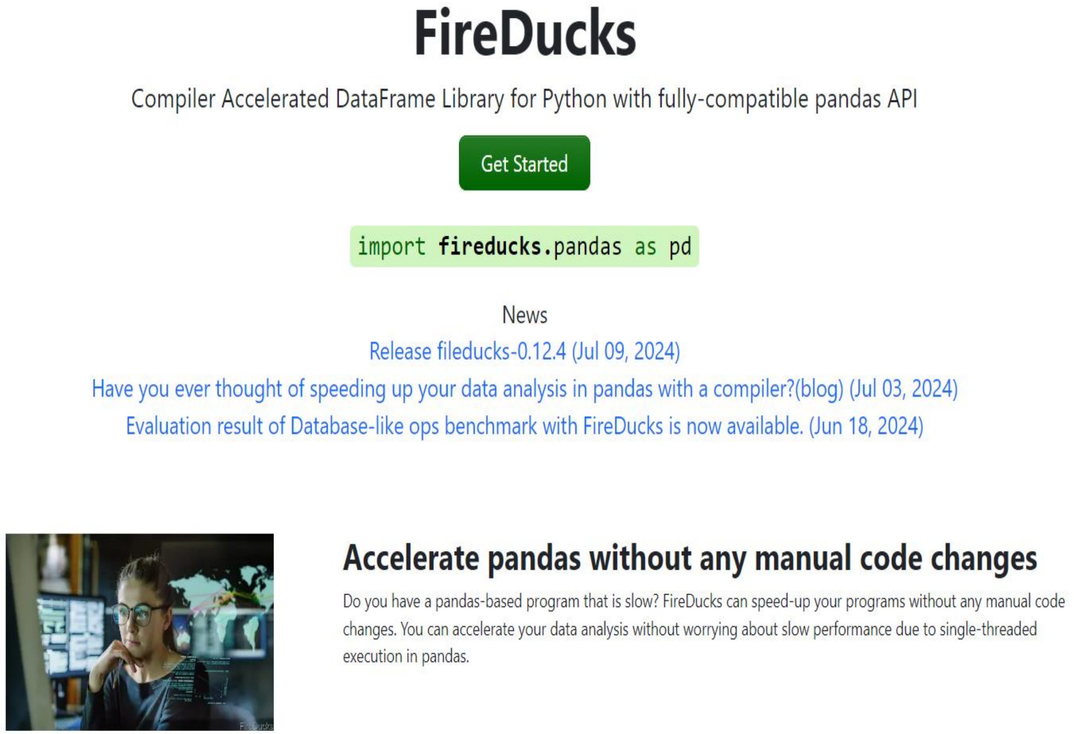
https://x.com/fireducksdev

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!

Frequently Asked Questions

FAQ: Why FireDucks is highly compatible with pandas?

library - A
from_pandas()
to_pandas()
pandas

FireDucks

Python Frontend (Pandas API)
fallback
pandas
IR Builder

```
%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