pandasプログラムを
どうやって高速化するか？

2025/01/25

石坂一久

SciPyData 2025

自己紹介

石坂 一久

（NEC セキュアシステムプラットフォーム研究所所属）

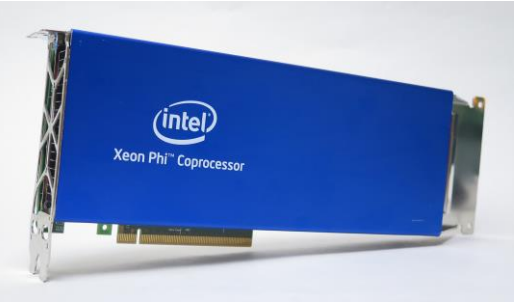
＜これまでの関わってきた主な領域＞

コンパイラ
並列処理
ハイパフォーマンスコンピューティング

現在はFireDucksを開発

（FireDucks: pandasの高速版ライブラリ）

Intel Xeon Phi
（メニコア）

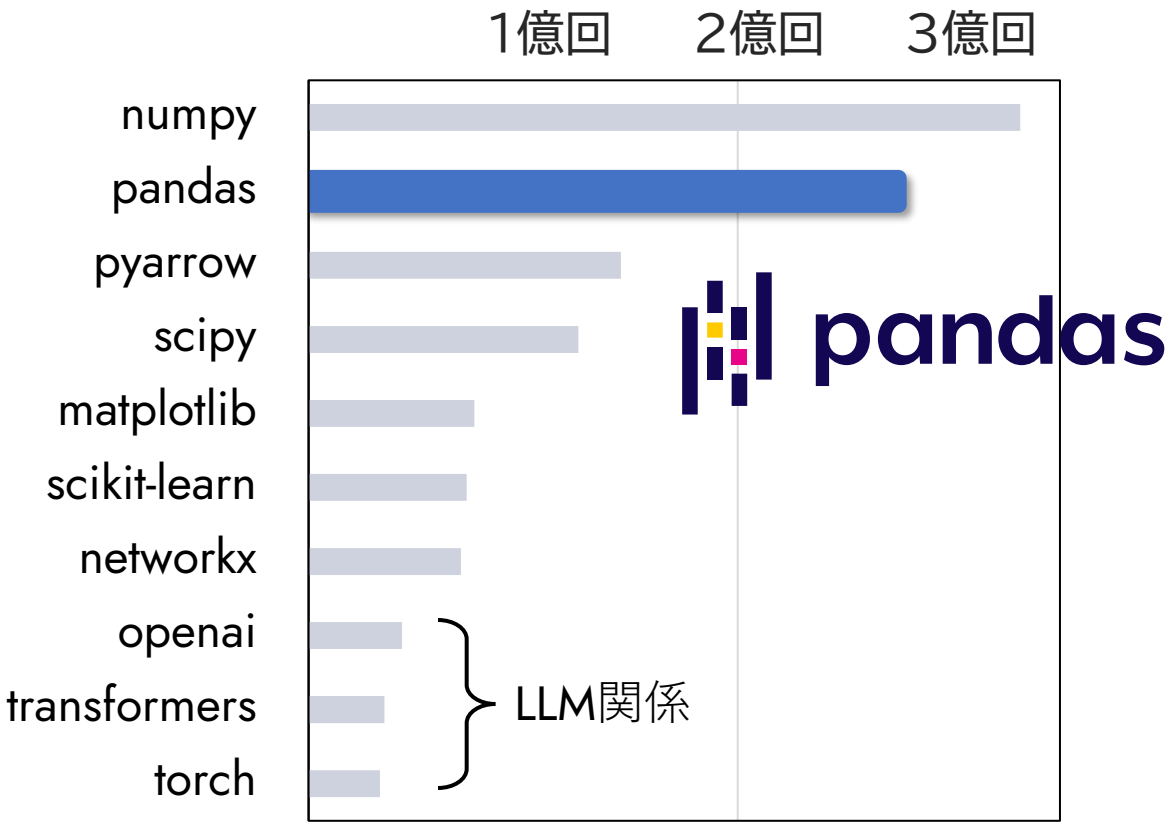


NEC SX-Aurora TSUBASA
（スパコン）

https://pc.watch.impress.co.jp/docs/news/yajiuma/1238340.html
https://jpn.nec.com/hpc/sxauroratsubasa/specification/index.html

pandas: データサイエンスの必須ツール

月間2億回以上ダウンロードされるデータ分析の標準的なpythonライブラリ



pypiの月間ダウンロード数
（データ分析関係のライブラリ 2024年10月）

https://www.amazon.co.jp/s?k=pandas+%E3%83%87%E3%83%BC%E3%82%BF&__mk_ja_JP=%E3%82%AB%E3%82%BF%E3%82%AB%E3%83%8A&crid=T44FOT0ODNPR&sprefix=pandas+%E3%83%87%E3%83%BC%E3%82%BF+%2Caps%2C165&ref=nb_sb_noss_2

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

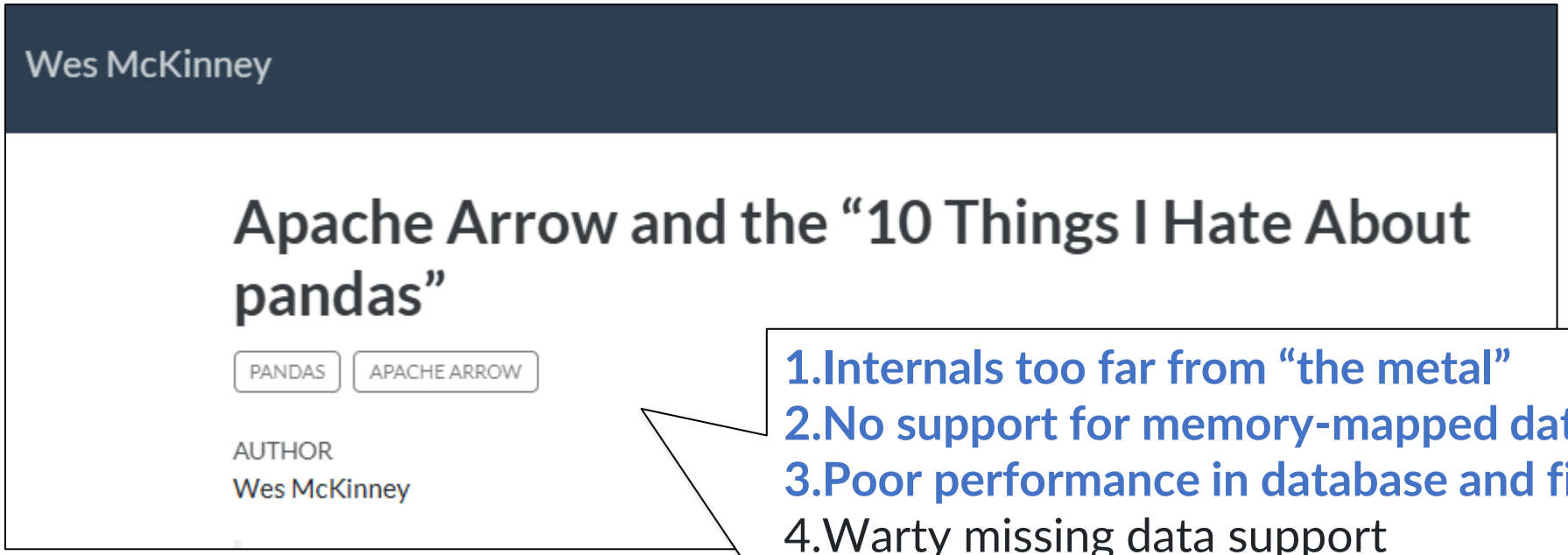
pandasへの不満

pandasは便利だけど...

データがメモリに入りきらない

遅い（実行時間が長い）

pandasの著者も言っています



Sep. 21, 2017

1.Internals too far from “the metal”
2.No support for memory-mapped datasets
3.Poor performance in database and file ingest / export
4.Warty missing data support
5.Lack of transparency into memory use, RAM management
6.Weak support for categorical data
7.Complex groupby operations awkward and slow
8.Appending data to a DataFrame tedious and very costly
9.Limited, non-extensible type metadata
10.Eager evaluation model, no query planning
11.“Slow”, limited multicore algorithms for large datasets

https://wesmckinney.com/blog/apache-arrow-pandas-internals/

pandasが遅くて困ったとき

プログラムを最適化する

良いマシンを使う

ライブラリを変える

プログラム最適化する

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

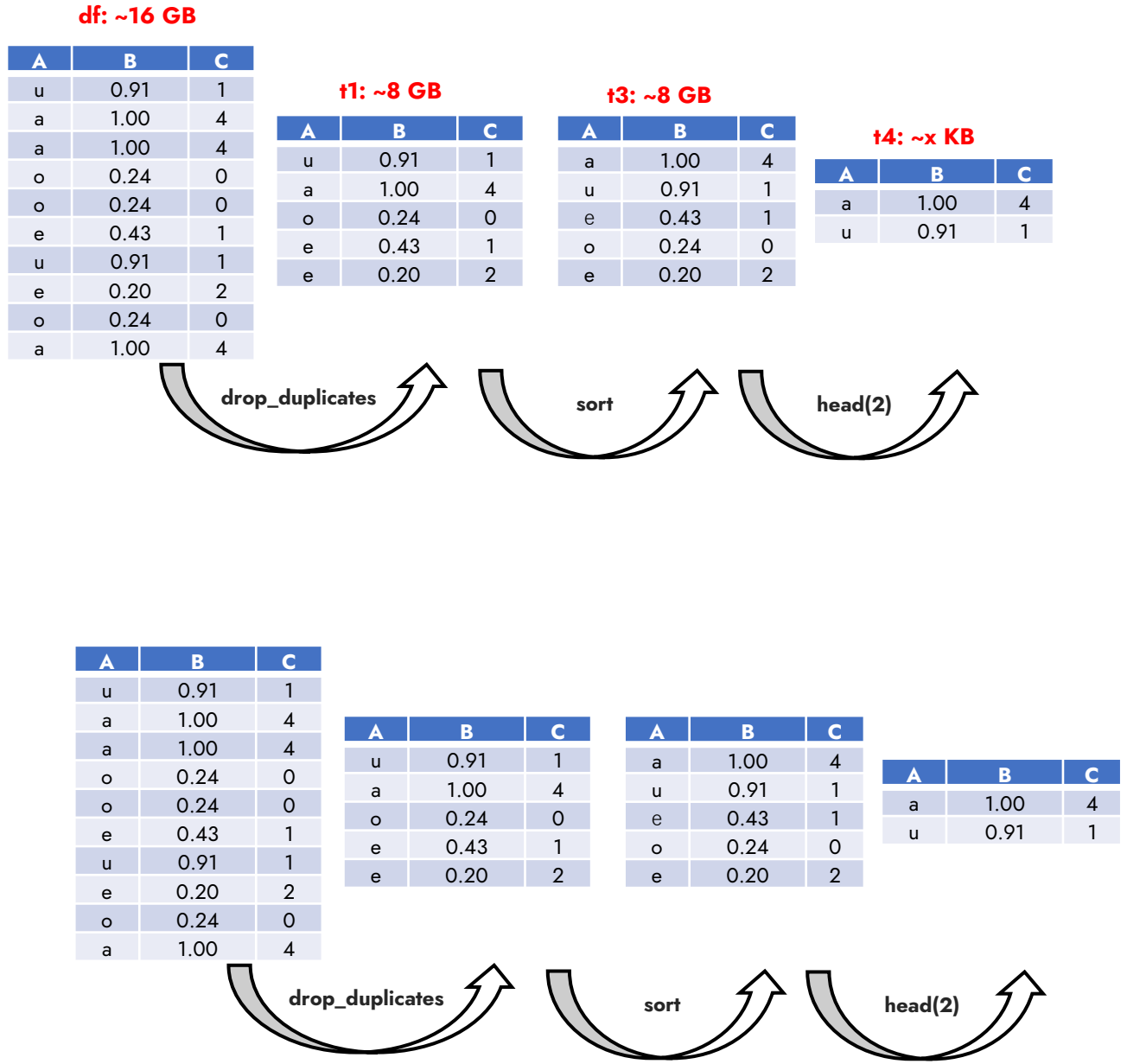
```
def foo(filename):
  return (
    pd.read_csv(filename)
      .drop_duplicates()
      .sort_values("B")
      .head(2)
  )
```

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
```

re-write using chained expression

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



Quiz: Which one is a better code?

```
res = df.sort_values(by="B")["A"]
```

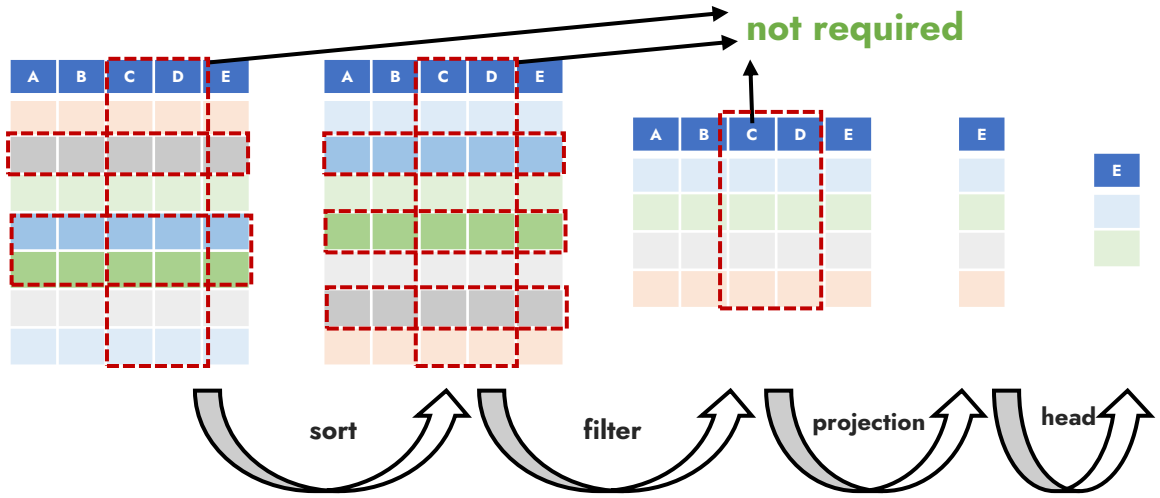
OR

```
tmp = df[["A", "B"]]
res = tmp.sort_values(by="B")["A"]
```

Pushdown最適化

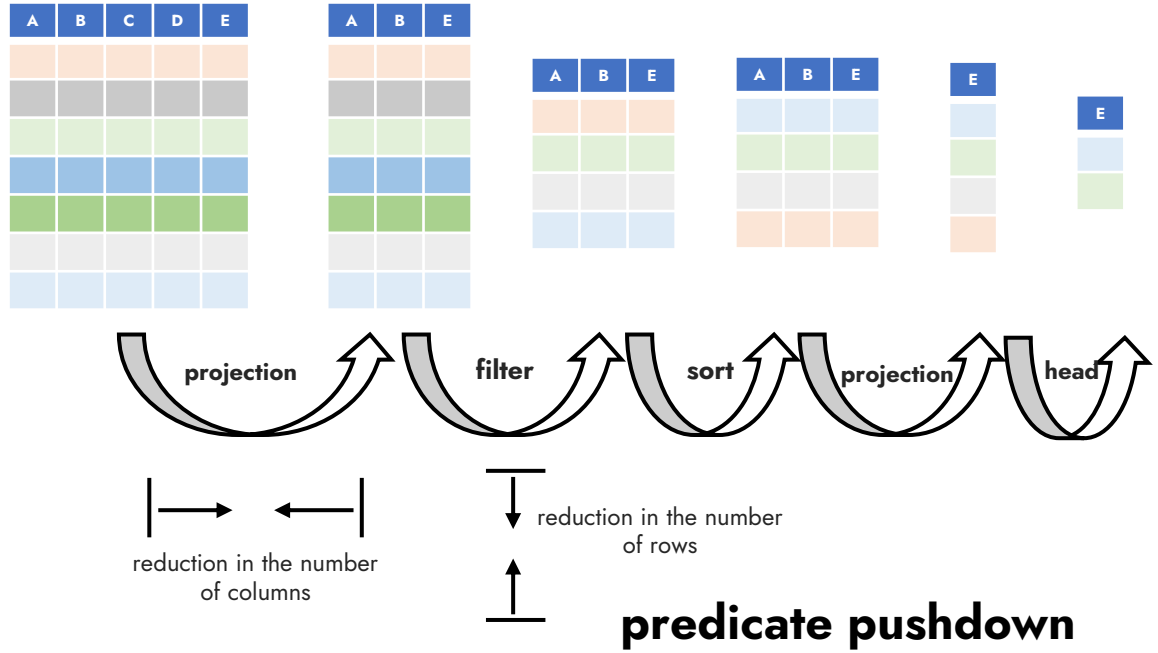
SAMPLE QUERY

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



OPTIMIZED QUERY

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



Pushdown最適化の効果

例: TPC-H Q3

SQLの参照実装をそのままpandasにしたプログラム

```
pd.read_parquet(os.path.join(datapath, "customer.parquet"))
  .merge(pd.read_parquet(os.path.join(datapath, "orders.parquet")),
         left_on="c_custkey", right_on="o_custkey")
  .merge(pd.read_parquet(os.path.join(datapath, "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(os.path.join("q3_result.parquet"))
```

ファイル読み込み

対象行の抽出

グループ毎の売上の合計を掲載

Pushdown最適化の効果

最適化したプログラム

```
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(os.path.join(datapath, "customer.parquet"), columns=req_customer_cols)
lineitem =  pd.read_parquet(os.path.join(datapath, "lineitem.parquet"), columns=req_lineitem_cols)
orders =  pd.read_parquet(os.path.join(datapath, "orders.parquet"), columns=req_orders_cols)

f_cust = customer[customer["c_mktsegment"] == "BUILDING"]
f_ord = orders[orders["o_orderdate"] < datetime.date(1995, 3, 15)]
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(os.path.join("opt_q3_result.parquet"))
```

必要な列だけを読み出し

行フィルタを先に実行

Pushdown最適化の効果

最適化したプログラム

```
req_customer_c
req_lineitem_c
req_orders_co

customer = pd.
lineitem =  pd
orders =  pd.r

f_cust = custo
f_ord = orders
f_litem = line

f_cust.merge(f
      .merge(f
      .assign(
      .groupby
      .agg({"r
      .sort_va
      .reset_i
      .head(10)
      .to_parquet(os.path.join("opt_q3_result.parquet"))
```

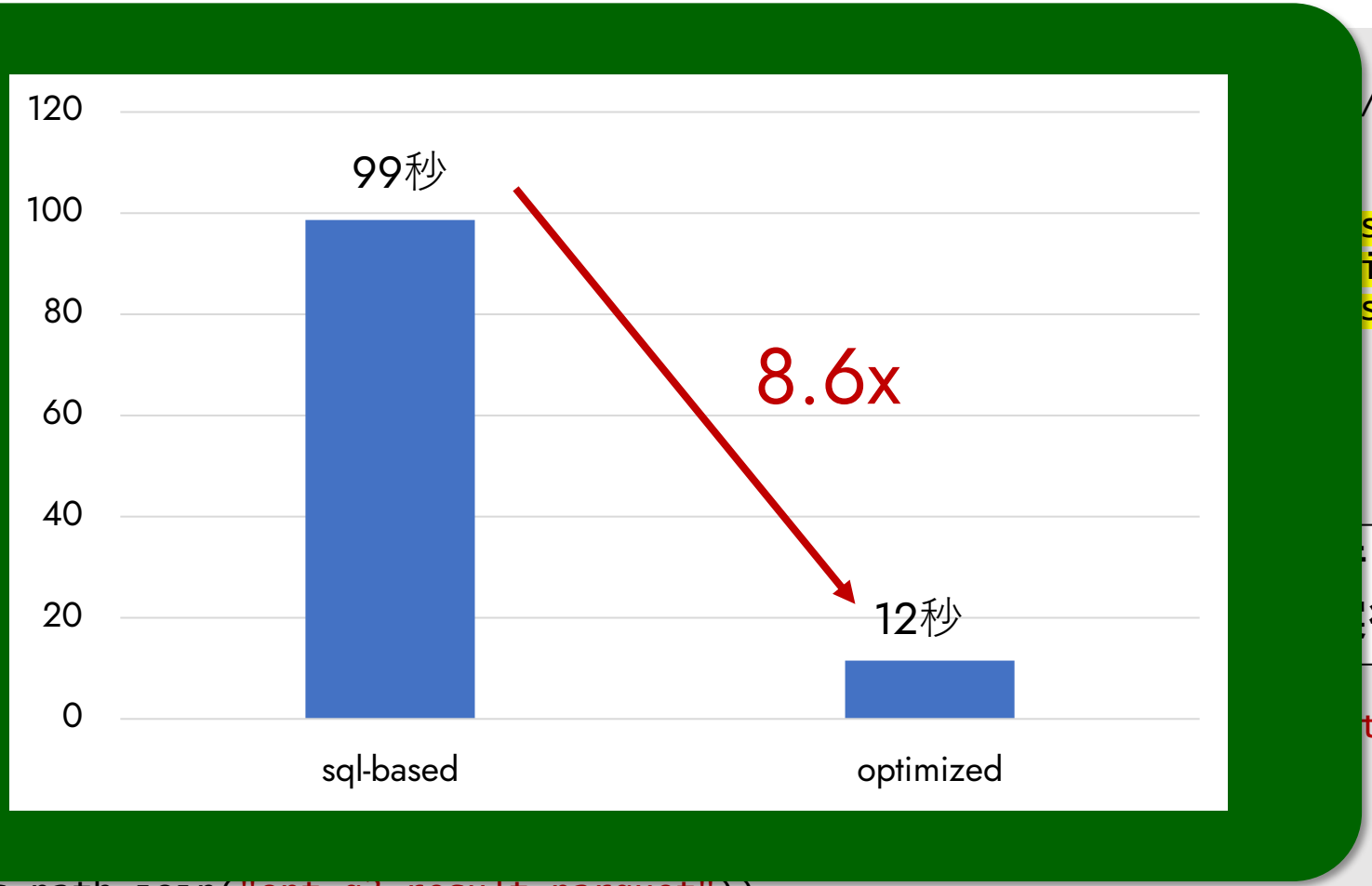
```
/16)


stomer_cols)
ineitem_cols)
s_cols)

ty"]]
```

必要な列だけを読み出し

フィルタを先に行



共通部分式の削除，デッドコード削除

共通部分式

```
# 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)
```

デッドコード

```
def demo_dce(df1, df2, is_eager_mode = False):
    merged_df = df1.merge(df2, on="a")
    sorted_df = merged_df.sort_values(by="a")
    # print(sorted_df)
    return merged_df.groupby("c")["d"].sum()
```

pandas固有のパターン最適化

特定のメソッドの組み合わせをより良い組み合わせに変換する

df[“timestamp”].dt.strftime(“%Y”).astype(int)　# timestamp列から年の取り出し

➡ df[“timestamp”].**dt.year**

groupby(“a”).sum().sort_values(“b”)　# groupby結果をb列でソート

➡ groupby(“a”, **sort=False**).sum().sort_values(“b”)

sort=Falseを追加

ループやapplyを避ける

Close列がOpen列より大きい行は"up",
それ例外は"down"という列を作りたい

Open	Close
100	110
108	107
112	103
103	120

result
up
down
down
up

ループ

```
def by_loop(df)
    result = []
    for _, row in df.iterrows():
        if row["Close"] > row["Open"]:
            result += ["up"]
        else:
            result += ["down"]
    return pd.Series(result, index=df.index)
```

apply

```
def by_apply(df):
    def func(row):
       if row["Close"] > row["Open"]:
           return "up"
       return "down"

    return df.apply(func, axis=1)
```

API

```
def by_getitem(df)
    result = pd.Series("down", index=df.index)
    result[df["Close"] > df["Open"]] = "up"
    return result
```

ループやapplyを避ける

```
def by_loop(df)
    result = []
    for _, row in df.iterrows():
        if row["Close"] > row["Open"]:
            result += ["up"]
        else:
            result += ["down"]
    return pd.Series(result, index=df.index)
```

65秒

bitcoin価格データ
(485万行)

```
def by_apply(df):
    def func(row):
       if row["Close"] > row["Open"]:
           return "up"
       return "down"

    return df.apply(func, axis=1)
```

18秒

```
def by_getitem(df)
    result = pd.Series("down", index=df.index)
    result[df["Close"] > df["Open"]] = "up"
    return result
```

35ミリ秒

1887x from loop
524x from apply

csvからparquetに変える

CSV: テキストフォーマット
Parquet: バイナリフォーマット

（ファイルサイズ）

```
pd.read_csv(data.csv)
```

725MB

6.7秒

```
pd.read_parquet(data.parquet)
```

2.5秒 **2.7x**

203MB

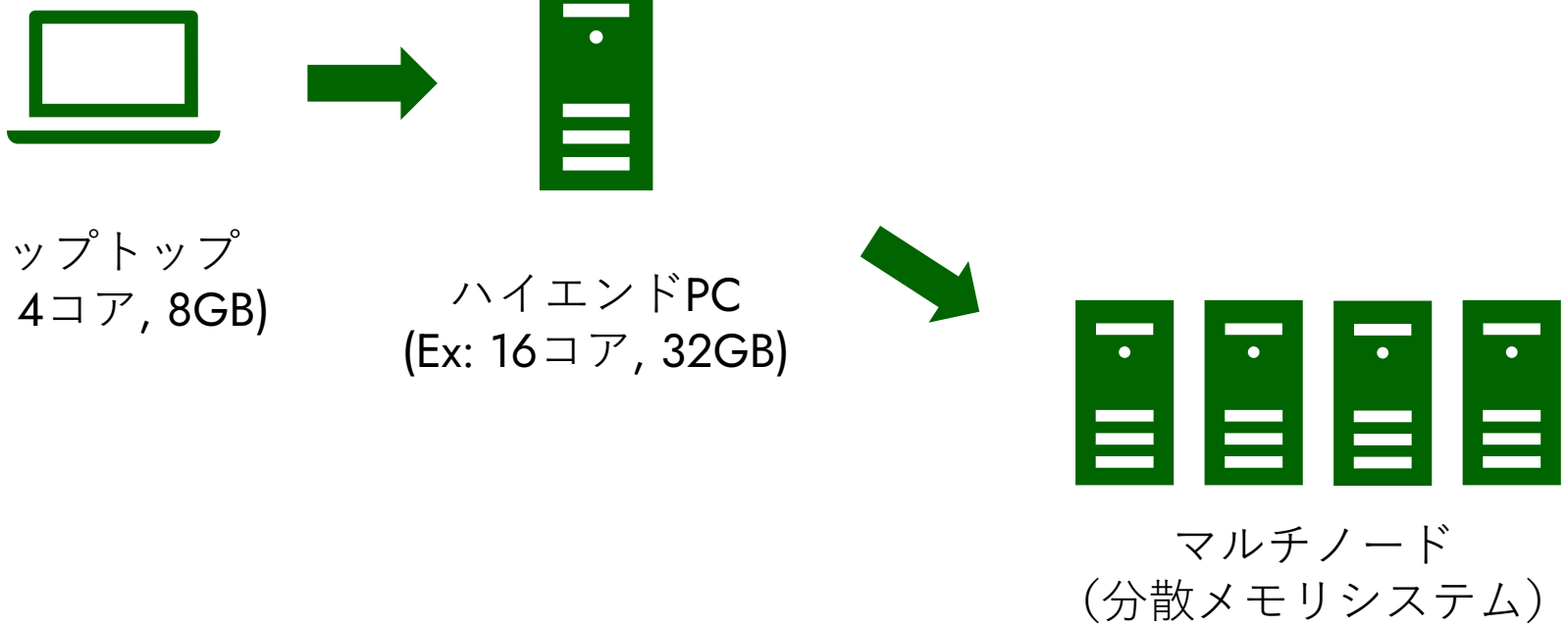
```
pd.read_parquet(data.parquet,
                dtype_backend="pyarrow")
```

0.35秒 **19x**

TPC-Hのlineitemテーブルでの測定(sf=1)
*ファイルキャッシュに載った状態での測定

良いマシンを使う

良いマシンを使う



マルチスレッド処理
（ライブラリ内部で対応しやすい）

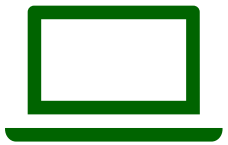
メモリに入るなら速度は優位

分散処理
（ユーザーの関与が必要）

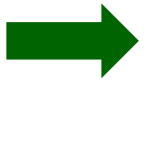
数十TB～のデータであれば必須

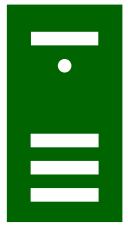
良いマシンを使う

本日はこちら
中心



ラップトップ
(Ex: 4コア, 8GB)



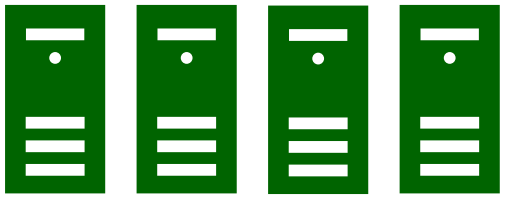


ハイエンドPC
(Ex: 16コア, 32GB)

ハイエンドサーバー
(Ex: 64コア, 1TB)

マルチスレッド処理
（ライブラリ内部で対応しやすい）

メモリに入るなら速度は優位



マルチノード
(分散メモリシステム)

分散処理
（ユーザーの関与が必要）

数十TB～のデータであれば
必須

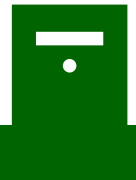
良いマシンを使う

本日はこちら中心

ハイエンドサーバー
(Ex: 64コア, 1TB)

マルチスレッド処理
（ライブラリ内部で対応しやすい）

メモリに入るなら速度は優位



ラップトッ
(Ex: 4コア,

残念ながらpandasは**速度での**恩恵はあまりない
（シングルスレッドなので）

良いマシンを使うならライブラリも変えたい

あれば

ライブラリを変える

データフレームライブラリ

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

basic questions

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

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

basic questions

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

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

Join

データフレームライブラリ

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

basic questions

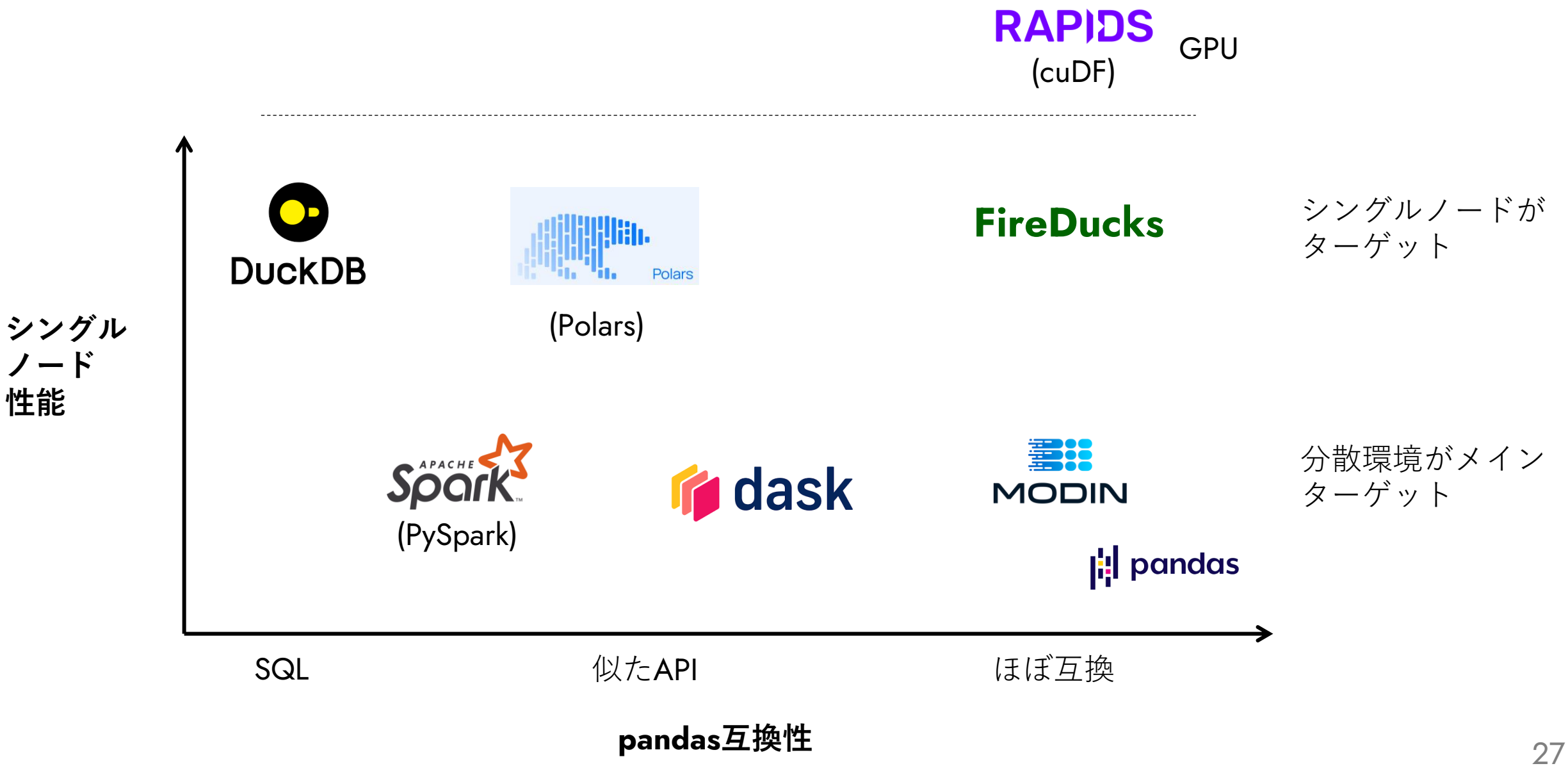
Input table: 1,000,000,000 rows x 9 columns (50 GB)

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

このベンチマークでの利用言語

今回は7ライブラリを掘り下げる

ライブラリ概観


RAPIDS
(cuDF)
GPU
DuckDB
Polars
(Polars)
FireDucks
シングルノードがターゲット
シングルノード性能
APACHE Spark
(PySpark)
dask
MODIN
pandas
分散環境がメインターゲット
SQL
似たAPI
ほぼ互換
pandas互換性

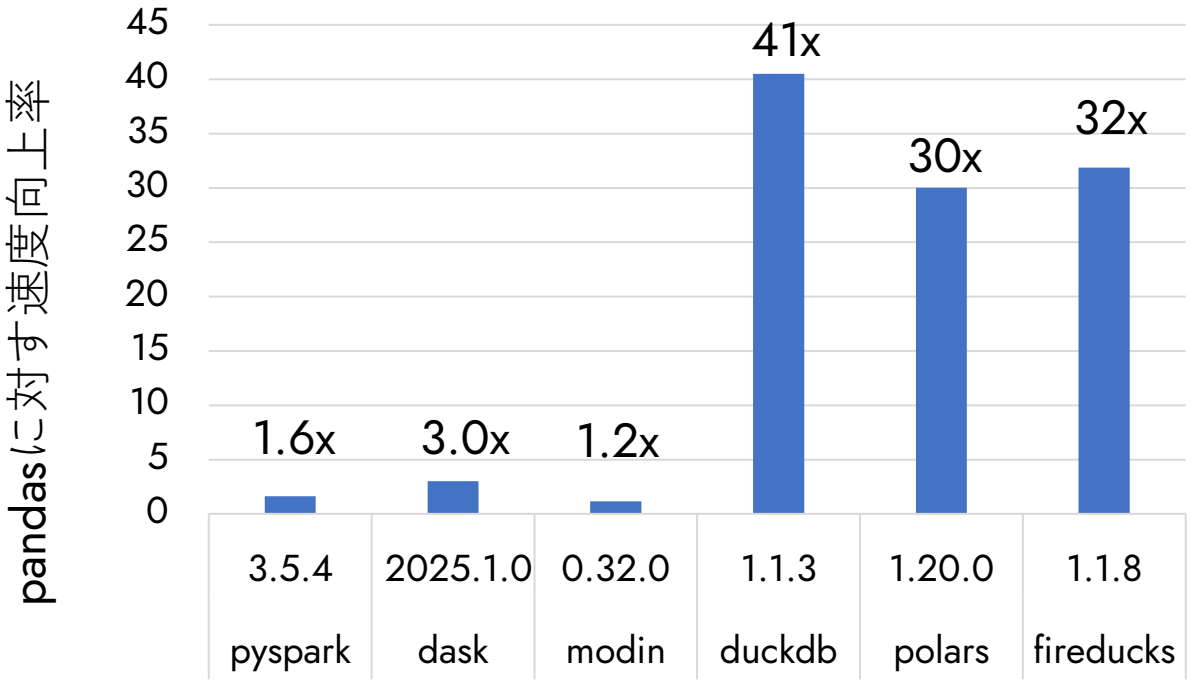
性能ベンチマーク

TPC-Hベンチマーク（22クエリ）

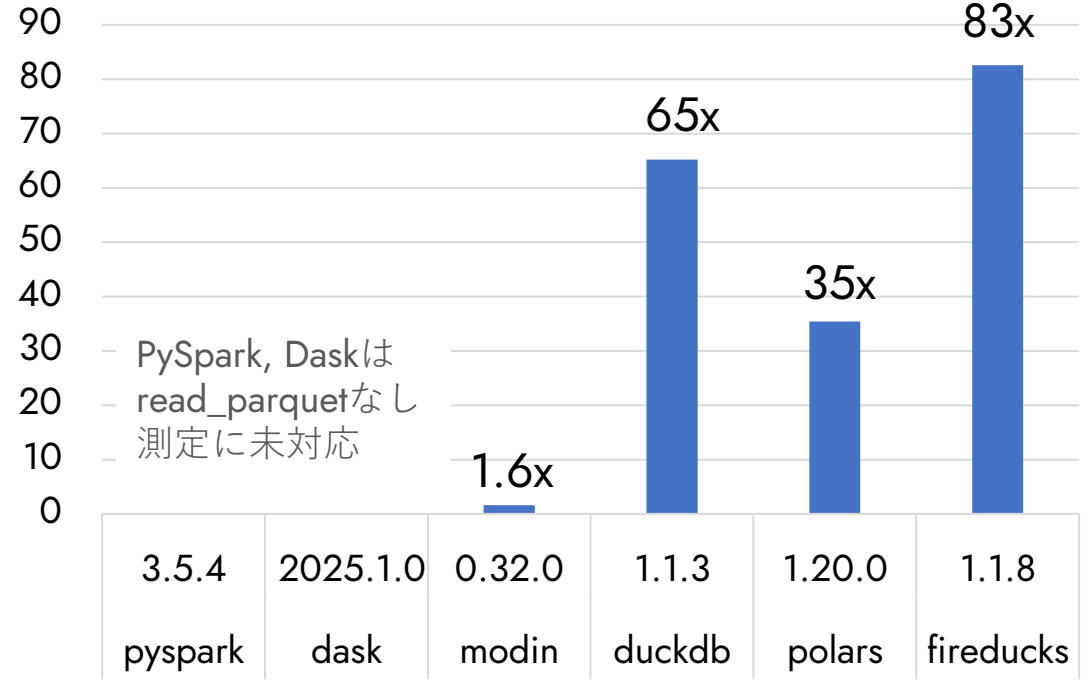
実装: https://github.com/pola-rs/polars-benchmark

INTEL(R) XEON(R) GOLD 6526Y
32 cores, 512GB
Ubuntu 24.04

pandasに対する速度向上率（22クエリの平均値※）



read_parquet込みの場合



read_parquetを除いた場合

※ Daskは実装されてる7クエリの平均，PySparkは動作した17クエリの平均

pandas互換性

FireDucks, Modin, cuDFの高いpandas互換性

Import文の変更のみで，プログラム本体の変更は不要

```
# import pandas as pd
import modin.pandas as pd
```

FireDucks, cuDFはimportの自動変更も可能

```
$ python3 -m fireducks.pandas program.py
```

```
%load_ext fireducks.pandas
import pandas
```

Dask, PySpark

- 分散環境をメインターゲットとしたライブラリ
- pandasと似たAPIも提供しており，同じ様な書き方も可能
- 性能を出すには独自の書き方や最適化が必要

pandas

```
q_final = (
    customer[customer["c_mktsegment"] == var1]
    .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["o_orderdate"] < var2])
    .pipe(lambda df: df[df["l_shipdate"] > var2])
    .assign(revenue=lambda df: df["l_extendedprice"] * (1 - df["l_discount"
    .groupby(["l_orderkey", "o_orderdate", "o_shippriority"], as_index=Fals
    .agg({"revenue": "sum"})[
        [
            "l_orderkey",
            "revenue",
            "o_orderdate",
            "o_shippriority",
        ]
    ]
    .sort_values(["revenue", "o_orderdate"], ascending=[False, True])
    .head(10)
)
return q_final
```

Dask

```
q_final = (
    customer[customer["c_mktsegment"] == var1]
    .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["o_orderdate"] < var2])
    .pipe(lambda df: df[df["l_shipdate"] > var2])
    .assign(revenue=lambda df: df["l_extendedprice"] * (1 - df["l_discount"
    .groupby(["l_orderkey", "o_orderdate", "o_shippriority"])
    .agg({"revenue": "sum"})
    .reset_index()[
        [
            "l_orderkey",
            "revenue",
            "o_orderdate",
            "o_shippriority",
        ]
    ]
    .sort_values(["revenue", "o_orderdate"], ascending=[False, True])
    .head(10)
)
return q_final
```

Ex: TPC-H Q3

groubyにas_indexオプションがないので
reset_indexが必要

Polars

- Rustで実装されたマルチスレッド化されたエンジン，最適化機能
- 独自の洗練されたPython API

pandas

```
q_final = (
    customer[customer["c_mktsegment"] == var1]
    .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["o_orderdate"] < var2])
    .pipe(lambda df: df[df["l_shipdate"] > var2])
    .assign(revenue=lambda df: df["l_extendedprice"] * (1 - df["l_discount"
    .groupby(["l_orderkey", "o_orderdate", "o_shippriority"], as_index=Fals
    .agg({"revenue": "sum"})[
        [
            "l_orderkey",
            "revenue",
            "o_orderdate",
            "o_shippriority",
        ]
    ]
    .sort_values(["revenue", "o_orderdate"], ascending=[False, True])
    .head(10)
)
return q_final
```

Polars

```
q_final = (
    customer.filter(pl.col("c_mktsegment") == var1)
    .join(orders, left_on="c_custkey", right_on="o_custkey")
    .join(lineitem, left_on="o_orderkey", right_on="l_orderkey")
    .filter(pl.col("o_orderdate") < var2)
    .filter(pl.col("l_shipdate") > var2)
    .with_columns(
        (pl.col("l_extendedprice") * (1 - pl.col("l_discount"))).alias("revenue
    )
    .group_by("o_orderkey", "o_orderdate", "o_shippriority")
    .agg(pl.sum("revenue"))
    .select(
        pl.col("o_orderkey").alias("l_orderkey"),
        "revenue",
        "o_orderdate",
        "o_shippriority",
    )
    .sort(by=["revenue", "o_orderdate"], descending=[True, False])
    .head(10)
)
```

Ex: TPC-H Q3

DuckDB

OLAP Database

Python Interfaceもあるが,
SQLを書く

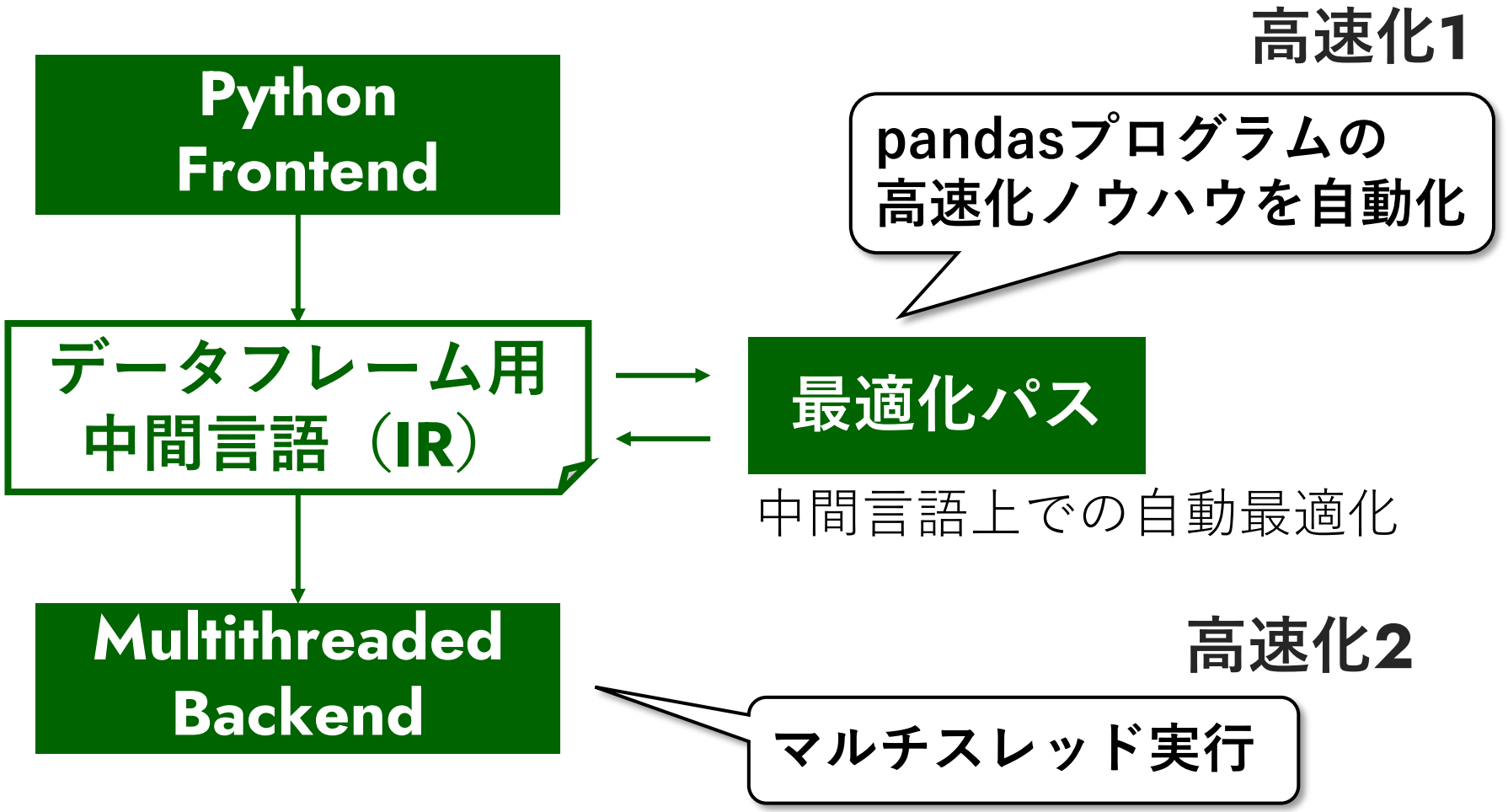
Ex: TPC-H Q3

```
query_str = f"""
select
    l_orderkey,
    sum(l_extendedprice * (1 - l_discount)) as revenue,
    o_orderdate,
    o_shippriority
from
    {customer_ds},
    {orders_ds},
    {line_item_ds}
where
    c_mktsegment = 'BUILDING'
    and c_custkey = o_custkey
    and l_orderkey = o_orderkey
    and o_orderdate < '1995-03-15'
    and l_shipdate > '1995-03-15'
group by
    l_orderkey,
    o_orderdate,
    o_shippriority
order by
    revenue desc,
    o_orderdate
limit 10
"""

q_final = duckdb.sql(query_str)
```

FireDucks

実行時コンパイラ技術を使った高速データフレームライブラリ



LLVM COMPILER INFRASTRUCTURE

MLIR

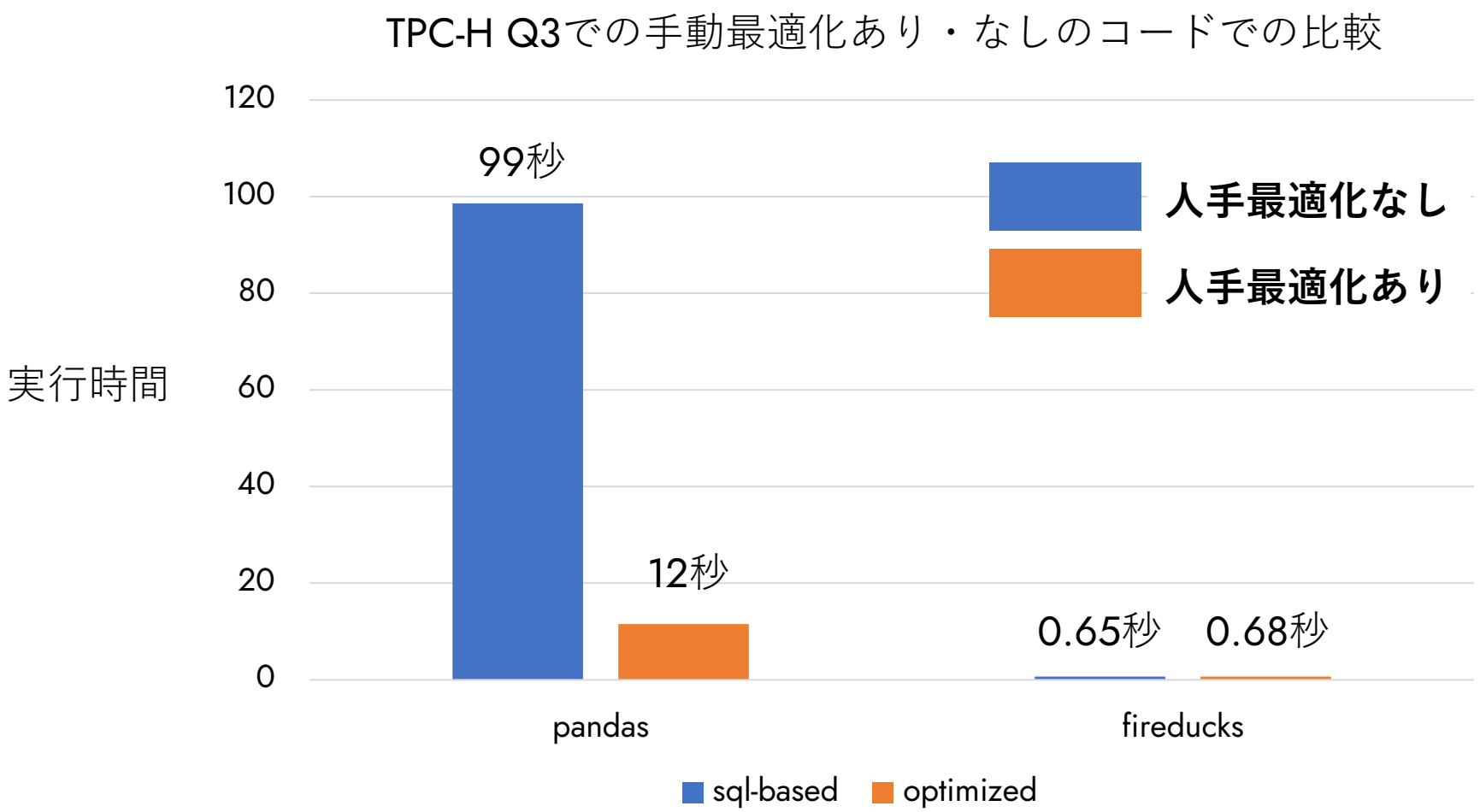
コンパイラフレームワーク

FireDucksの最適化

最適化	FireDucks	備考
不要データの削除 （メソッドチェーン）	○	自動でもできるがメソッドチェーンがお薦め
共通部分式の削除	○	
デッドコード削除	○	
Pushdown最適化	○	
パターン最適化	○	現在15パターン
Applyの自動変換	×	研究中
csv→parquet変換	×	

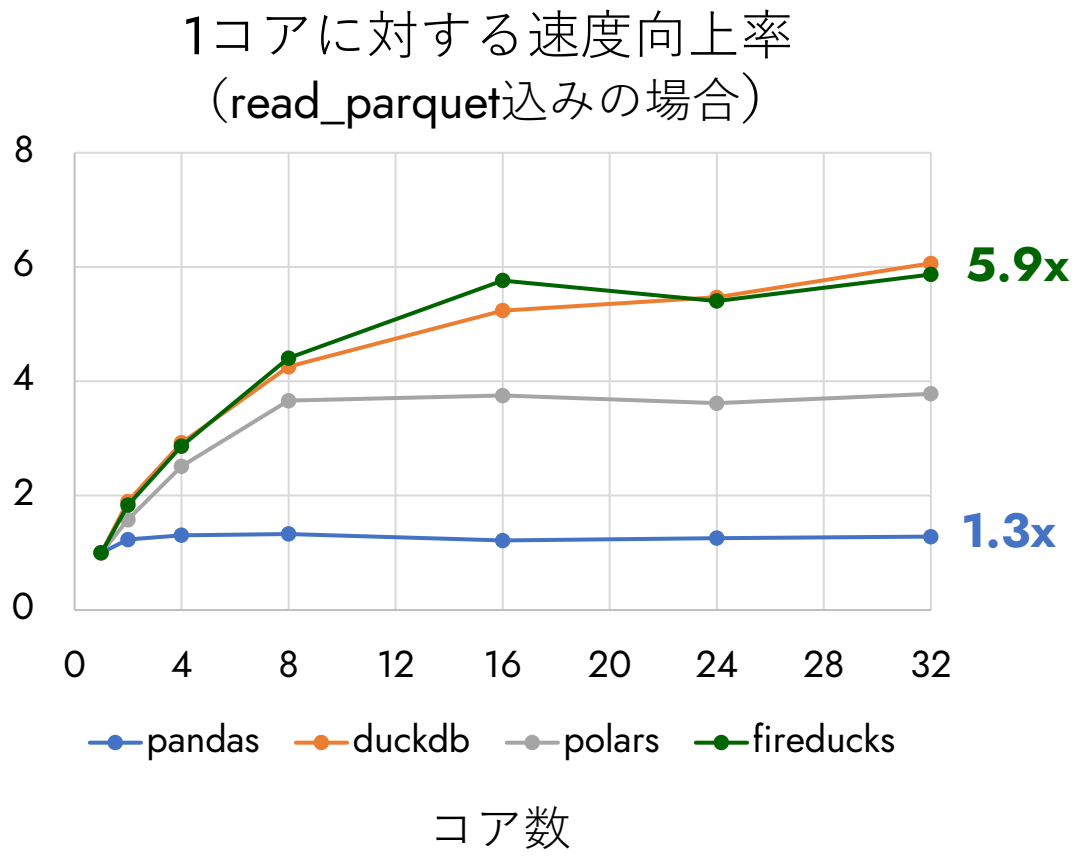
FireDucksの最適化

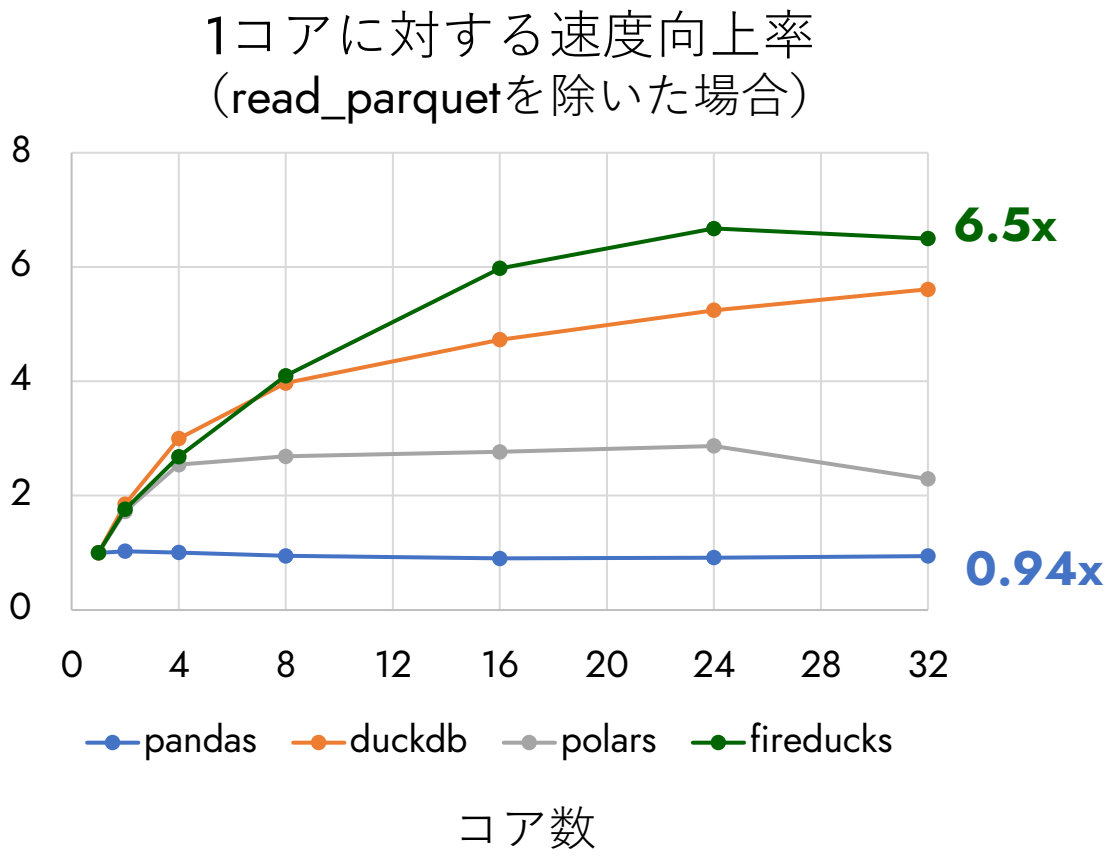
ライブラリの最適化機能により人手による負担が大幅削減



スケーラビリティ

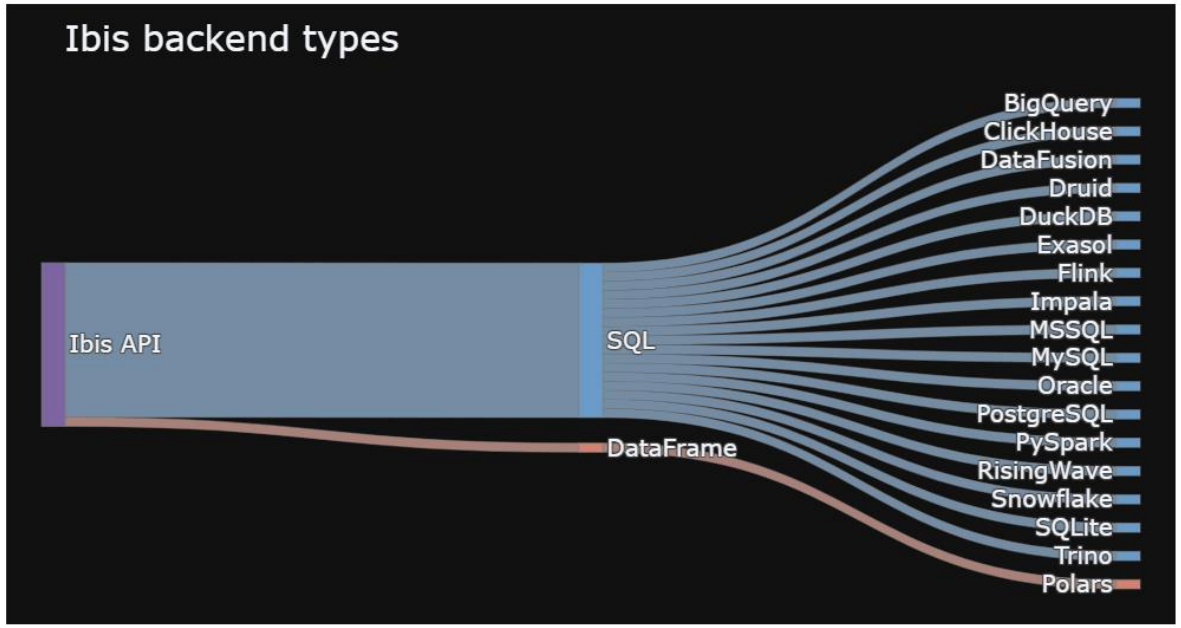
マルチスレッドに対応したライブラリであれば，良いマシンの恩恵あり



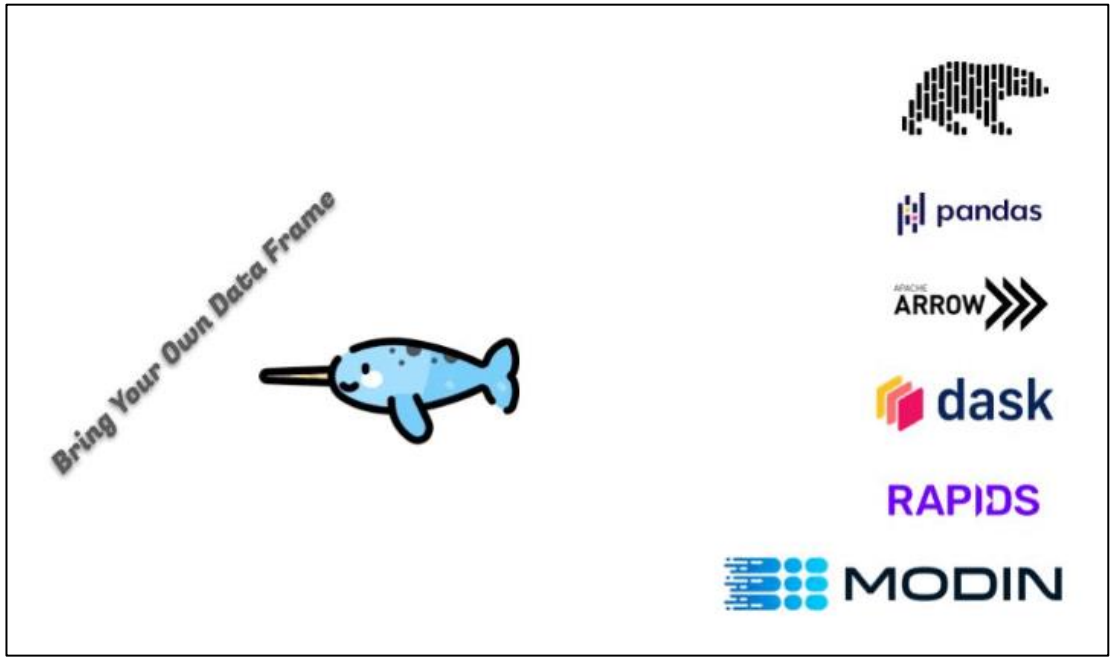


統合ライブラリ（Ibis, Narwhals）

Ibis



Narwhals



共通のAPIを，様々なライブラリにつなぐ

まとめ: pandasプログラムが遅いとき

速度課題のあるpandasプログラムがあるなら，
まずは互換性が高く，シングルノード対象のFireDucksを試す

pandas APIにこだわらないなら，PolarsやDuckDBもターゲット

FireDucks, Polars, DuckDBなどに乗り換えたら，良いマシンも！

データが巨大であれば，DaskやPySparkが選択肢

FireDucksとPolarsの比較

TPC-Hベンチマーク（22クエリ）

INTEL(R) XEON(R) GOLD 6526Y
32 cores, 512GB
Ubuntu 24.04