FireDucks

Compiler Accelerated DataFrame Library with pandas API

2024/07/18

石坂一久

みんなのPython勉強会#106

はじめに

FireDucksを使うために，本発表を聞く必要はありません

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

あとはpandasだと思って使えば，数倍速いです．

※マシン環境やプログラムに依存します

本発表では，FireDucksがなぜ速いかをご紹介します．

本発表を聞いて，**FireDucksを使ってみようかな**と思って
頂ければ幸いです

自己紹介

石坂 一久

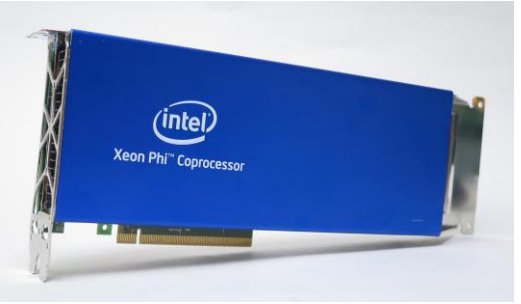
（NEC セキュアシステムプラットフォーム研究所所属）

＜これまでの関わってきた主な領域＞

自動並列化コンパイラ

並列処理・ベクトル処理

Intel Xeon Phi
（メニコア）



NEC SX-Aurora TSUBASA
（スパコン）

→ DataFrameコンパイラを作って，pandasを速くしよう！

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

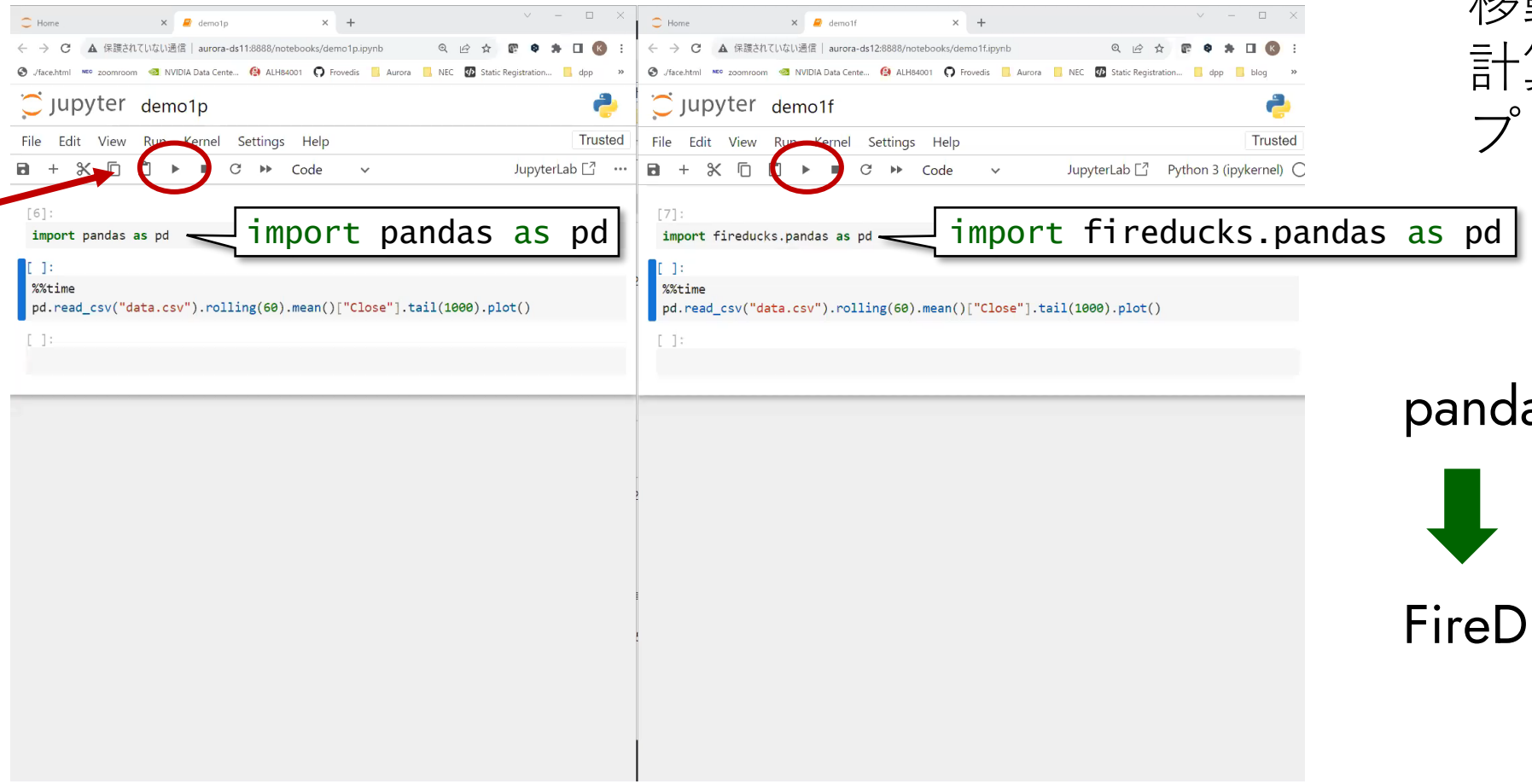
Demo

```
pd.read_csv(“data.csv”).rolling(60).mean()[“Close”].tail(1000).plot()
```

移動平均を計算するプログラム

pandas （違いはimport文だけ） **FireDucks**

実行開始ボタン

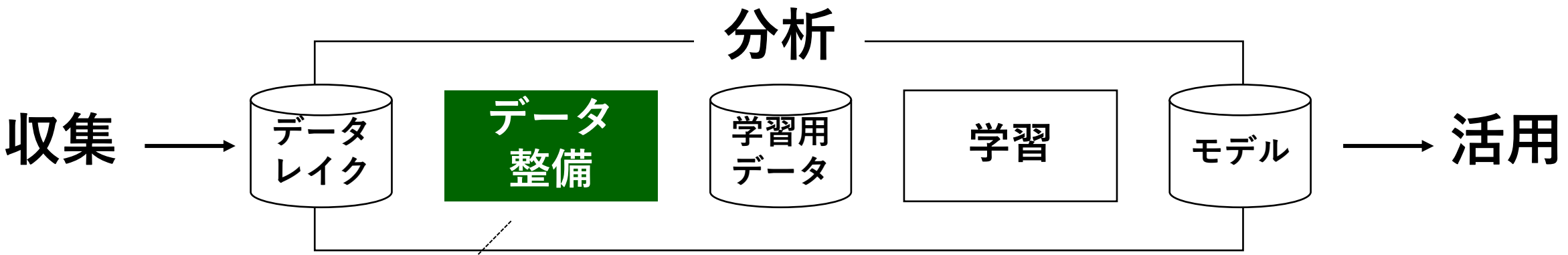


pandas: 4.06s

⬇ **約15倍**

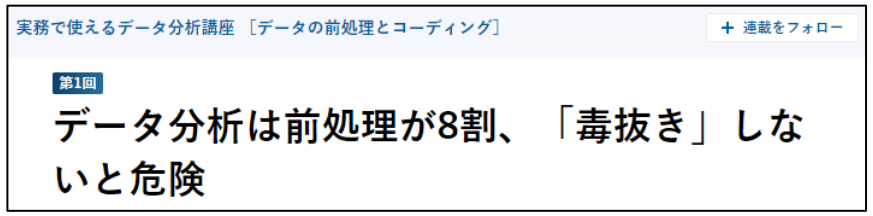
FireDucks: 275ms

pandas高速化のニーズ



pandasが良く使われるデータ整備がデータ分析のボトルネックに

- 探索的データ解析，学習用データの作成などの前処理
- 単純な整形だけでなく，複雑なアルゴリズムも登場

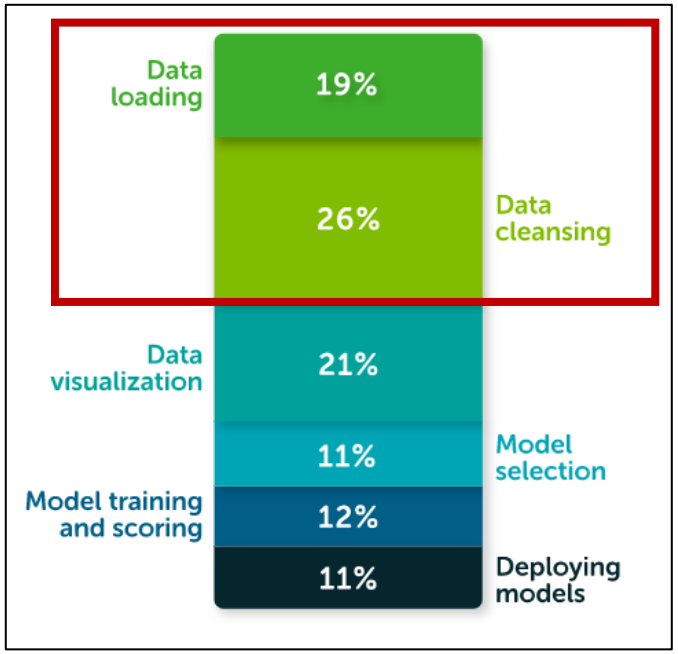

実務で使えるデータ分析講座［データの前処理とコーディング］ ＋ 連載をフォロー

第1回

データ分析は前処理が8割、「毒抜き」しないと危険

https://xtech.nikkei.com/atcl/learning/lecture/19/00110/00001/

データサイエンティストの時間の40%以上

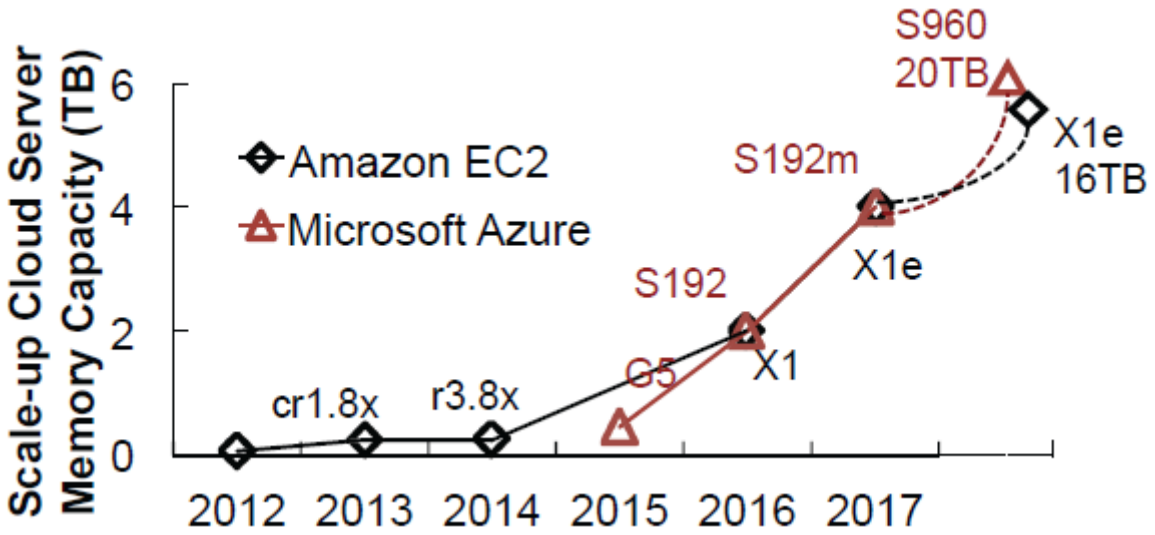


Anaconda The State of Data Science 2020

大量のデータを使えるようになったけど...

扱うデータ量や処理の複雑化に伴い速度課題が顕在化

クラウドサーバーのメモリ容量



コモディティサーバーでも
数百GBのメインメモリ

[5] Ogleari, M. et al.: String figure: A scalable and elastic memory network architecture, *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, pp. 647–660 (2019)

pandasはなぜ遅い？

ほとんどの処理はシングルスレッド実行

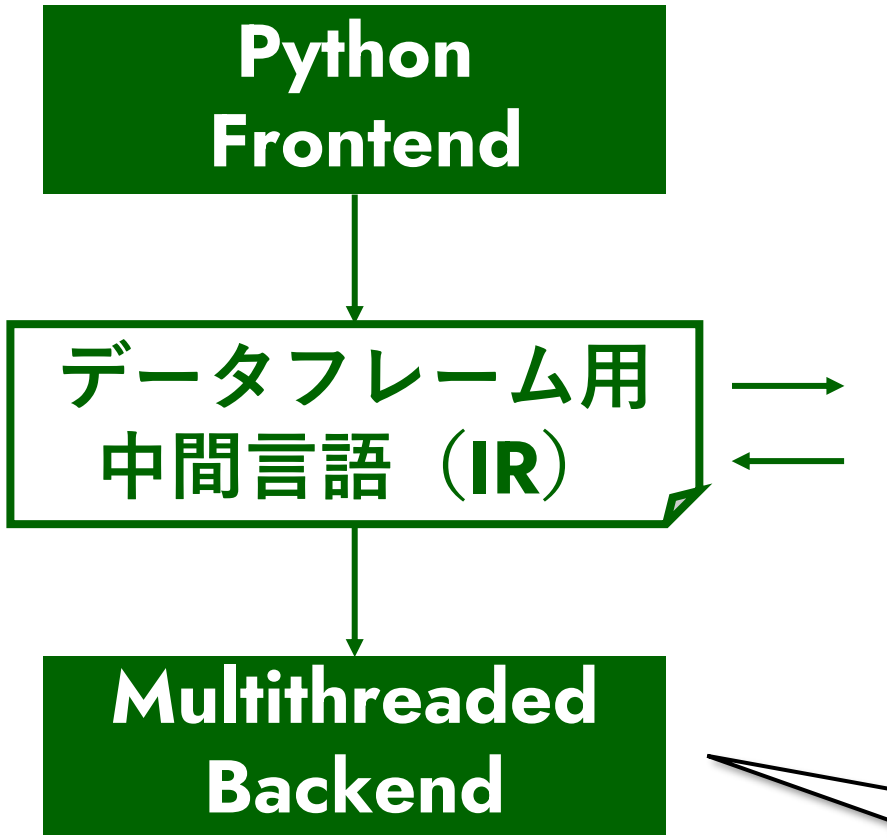
Eager実行
（SQLのクエリプランナーが行うような最適化がされない）

遅い書き方ができてしまう

FireDucksのアプローチ

中間言語（IR）を介することで，APIの変更なく，**最適化や実行を改善**

ユーザープログラムを
一度中間言語に変換



高速化1

pandasプログラムの
高速化ノウハウを自動化

最適化パス

中間言語上での自動最適化

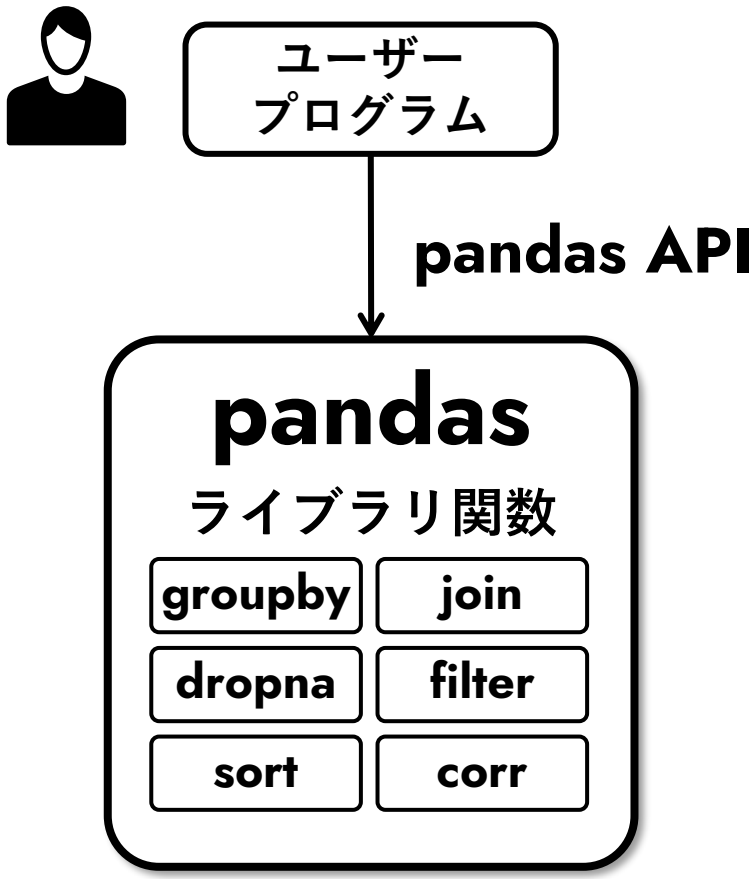
ユーザーAPIとは
独立したバックエンド
で実行

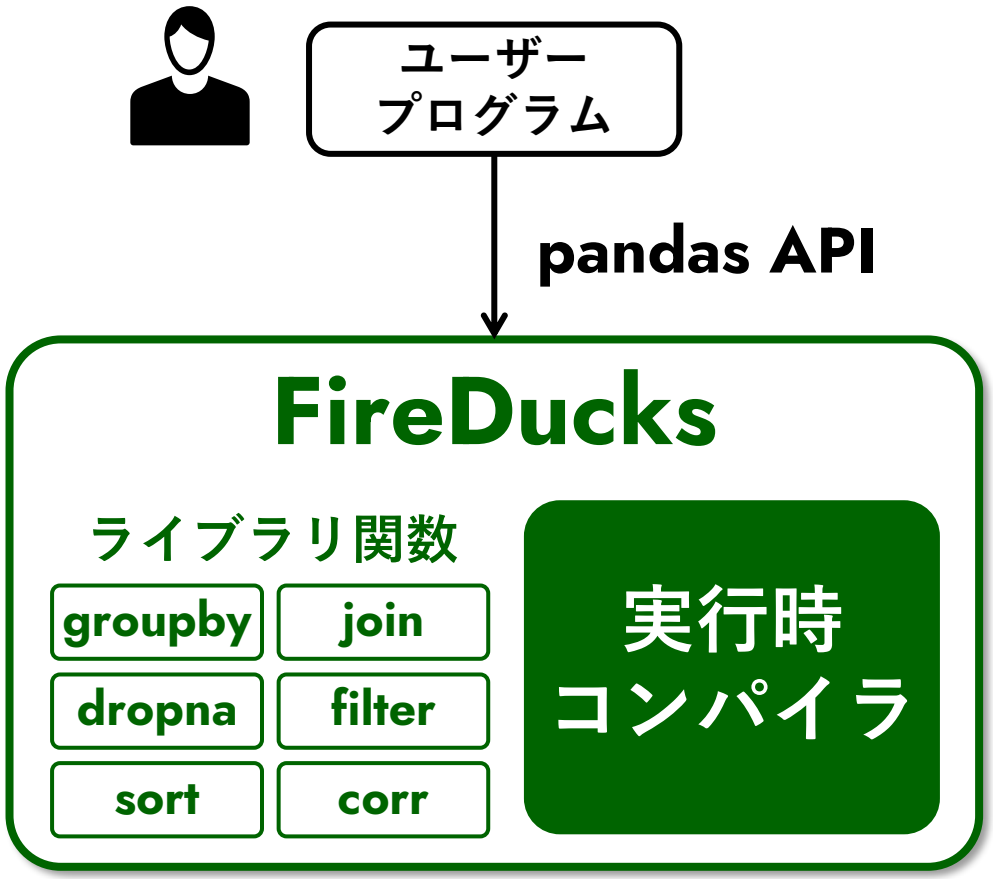
高速化2

マルチスレッド実行

実行時コンパイラで実現

ライブラリ中に埋め込まれた実行時コンパイラで，使い勝手やAPIを変えずに高速化





データフレーム用中間言語

データフレームの要素処理を命令としたドメイン特化型の中間言語（IR）

pythonプログラム

```
pd.read_csv(“data.csv”)
  .rolling(60).mean()
  [“Close”]
  .tail(1000)
  .plot()
```

FireDucks IR

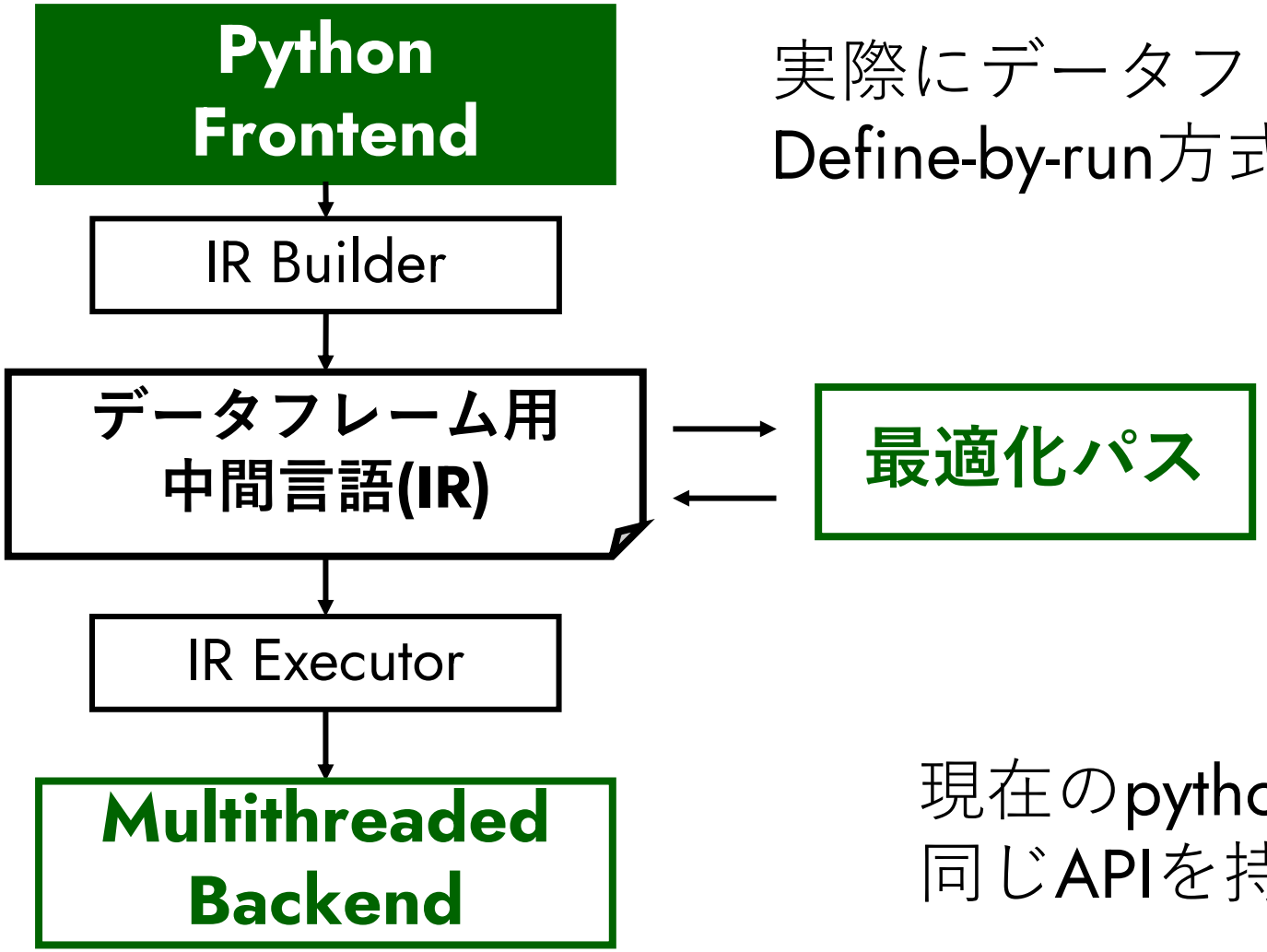
```
%t1 = read_csv('data.csv', %arg0)
%t2 = rolling_aggregate(%t1, 60, 60, 'mean')
%t3 = project(%t2, 'Close')
%t4 = slice(%t3, -1000, None, 1)
```

※: 最適化に適したSSA形式

1命令がデータフレーム操作の意味を持つので,
データフレーム特化の最適化を行いやすい

独自の中間言語を定義できるコンパイラフレームワークを利用
（LLVMのサブプロジェクト）

Python Frontend



FireDucksの構造

実際にデータフレーム処理をするのでなく，
Define-by-run方式で中間言語の生成を行う

現在のpython frontendはpandasと
同じAPIを持つ

Define-by-runによる中間言語生成

read_csvを実行

```
pd.read_csv(“data.csv”)
  .rolling(60).mean()
  [“Close”]
  .tail(1000)
  .plot()
```

FireDucks内部でread_csv opが生成される

```
%t1 = read_csv('data.csv', %arg0)
```

この時点では実際のcsvファイルの
読み込みは行われない

Define-by-runによる中間言語生成

read_csvを実行

```
pd.read_csv(“data.csv”)
  .rolling(60).mean()
  [“Close”]
  .tail(1000)
  .plot()
```

FireDucks内部でread_csv opが生成される

```
%t1 = read_csv('data.csv', %arg0)
```

この時点では実際のcsvファイルの
読み込みは行われない

FireDucksのread_csvの実装（簡略版）

```
def read_csv(filename):
    value = irbuilder.build_op(OP_read_csv, filename)
    return DataFrame(value)
```

Define-by-runによる中間言語生成

rolling.meanを実行

```
pd.read_csv(“data.csv”)
  .rolling(60).mean()
  [“Close”]
  .tail(1000)
  .plot()
```

rolling_aggregate opが生成される

```
%t1 = read_csv('data.csv', %arg0)
%t2 = rolling_aggregate(%t1, 60, 60, 'mean')
```

Define-by-runによる中間言語生成

__getitem__を実行
（列の取り出し）

```
pd.read_csv(“data.csv”)
  .rolling(60).mean()
  [“Close”]
  .tail(1000)
  .plot()
```

rolling_aggregate opが生成される

```
%t1 = read_csv('data.csv', %arg0)
%t2 = rolling_aggregate(%t1, 60, 60, 'mean')
%t3 = project(%t2, 'Close')
```

Define-by-runによる中間言語生成

tailを実行

```
pd.read_csv(“data.csv”)
  .rolling(60).mean()
  [“Close”]
  .tail(1000)
  .plot()
```

slice opが生成される

```
%t1 = read_csv('data.csv', %arg0)
%t2 = rolling_aggregate(%t1, 60, 60, 'mean')
%t3 = project(%t2, 'Close’)
%t4 = slice(%t3, -1000, None, 1)
```

中間言語の実行開始

特定のAPIが実行されると，中間言語の実行を開始（まとめて遅延実行）

plotを実行

```
pd.read_csv(“data.csv”)
  .rolling(60).mean()
  [“Close”]
  .tail(1000)
  .plot()
```

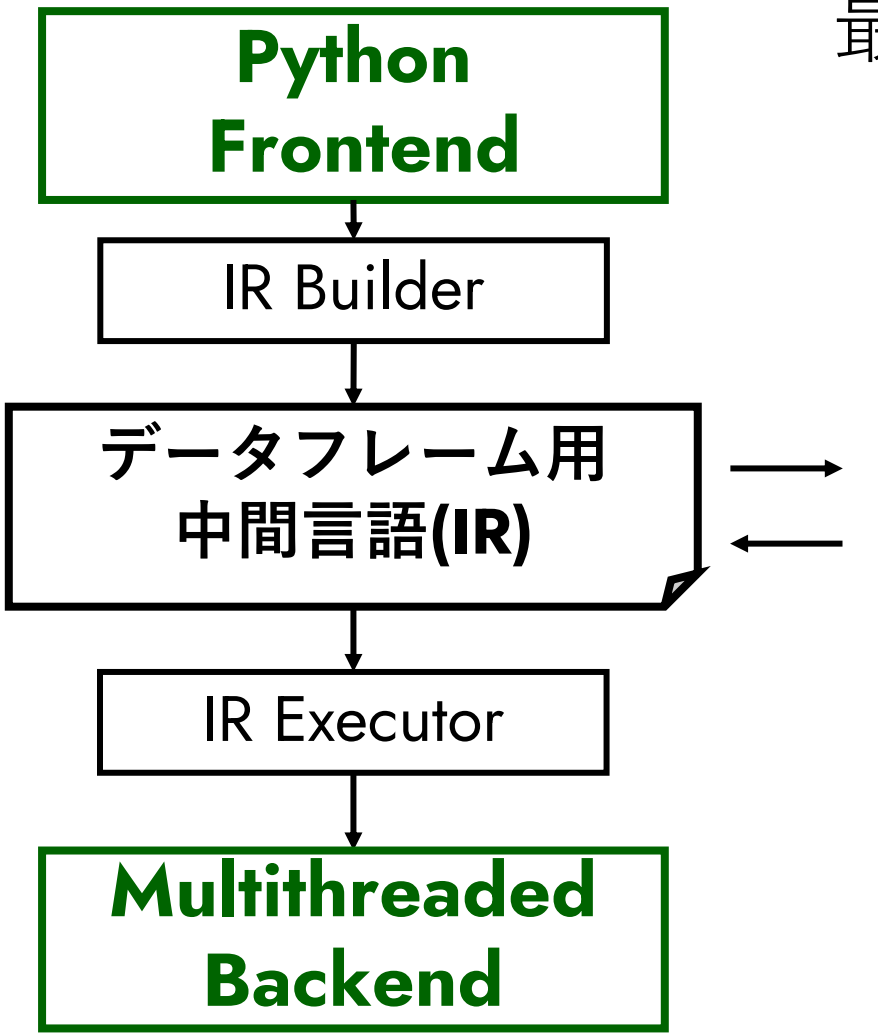
```
%t1 = read_csv('data.csv', %arg0)
%t2 = rolling_aggregate(%t1, 60, 60, 'mean')
%t3 = project(%t2, 'Close’)
%t4 = slice(%t3, -1000, None, 1)
```

plotはいくつかある評価ポイントの一つ（他には`__repr__`など）

```
print(df)
```

__repr__はprint内で利用される

最適化パスでの自動最適化



FireDucksの構造

最適化パスがIRをより良いIRに変換

- IR変換として，各種のデータフレーム高速化テクニックを実装

現在も鋭意拡充中

1) projection pushdowns最適化

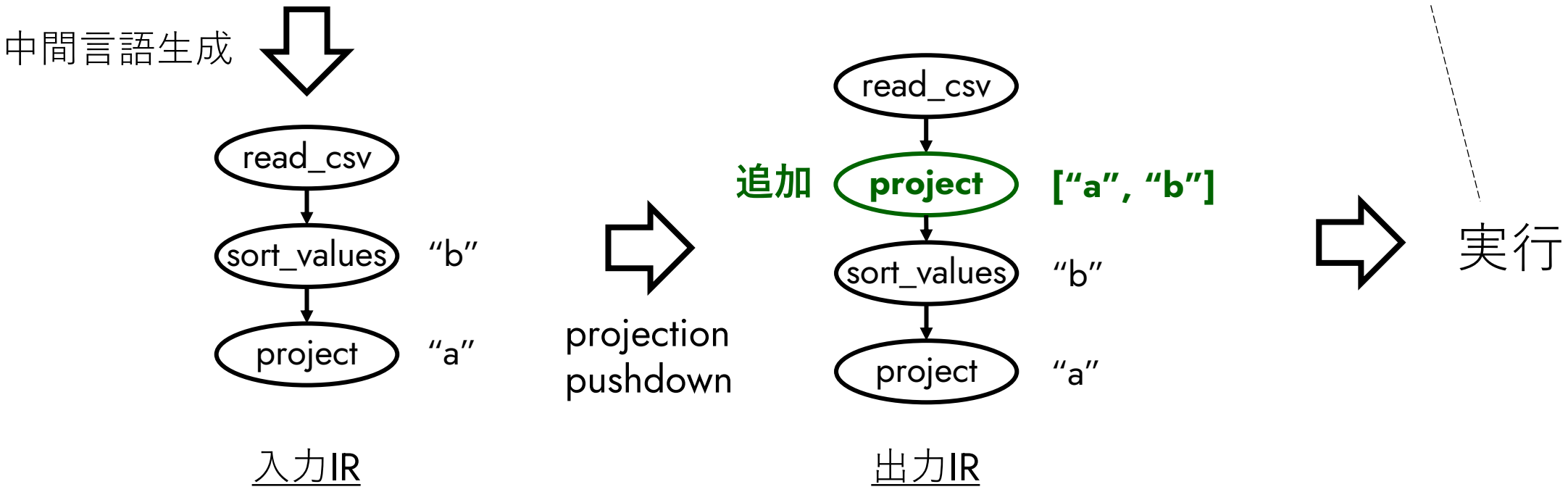
projection（列の抽出）を前出しすることで，中間データを削減

ユーザーが書いたプログラム

```
df = pd.read_csv("sample.csv")
sorted = df.sort_values("b")
result = sorted[["a"]]
```

実際に実行される処理

```
df = pd.read_csv("sample.csv")
df2 = df[[“a”, “b”]]
sorted = df2.sort_values("b")
result = sorted[["a"]]
```



2) パターン最適化

特定の命令の組み合わせをより良い組み合わせに変換する

```
df[~df["a"].isnull()]      # a列がnulのl行を削除
```

➡ df.**dropna**("a")

```
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("a", **sort=False**).sum().sort_values("b")

sort=Falseを追加

3) 集約特徴量計算向けの最適化

集約特徴量計算の例

```
df_encoded, new_cols = xfeat.aggregation(
    df,
    group_key = 'hour',
    group_values = df.drop('hour', axis=1).columns,
    agg_methods = ['mean', 'max'],
)
```

一時間単位の平均と最大値を
元のテーブルに新しい列として追加する

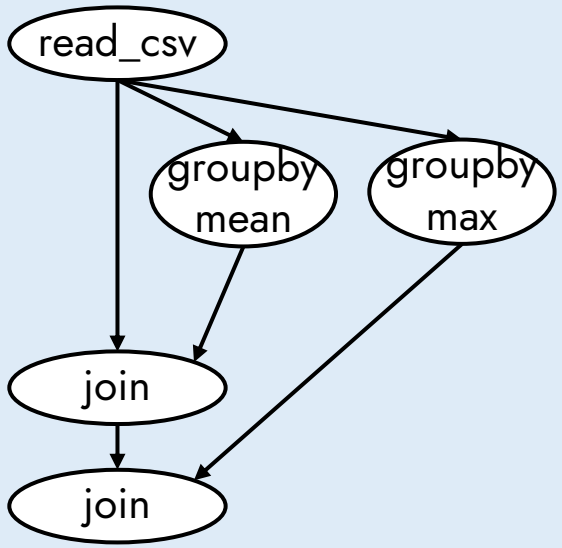
追加

time	val
10:20	100
10:30	120

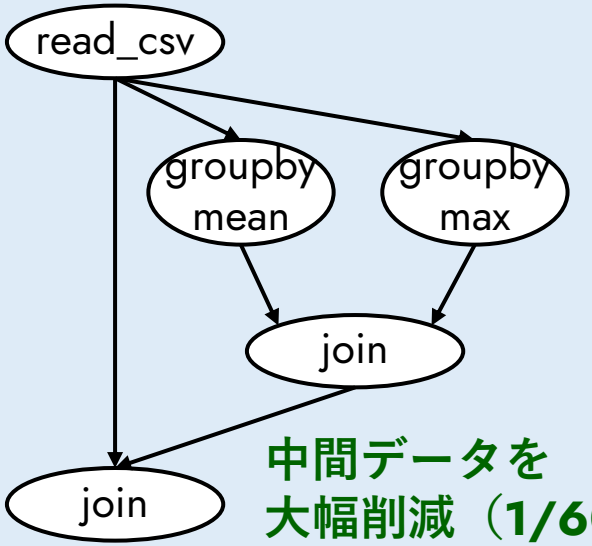
⇨

time	val	val_mean	val_max
10:20	100	110	120
10:30	120	110	120

元の計算フロー

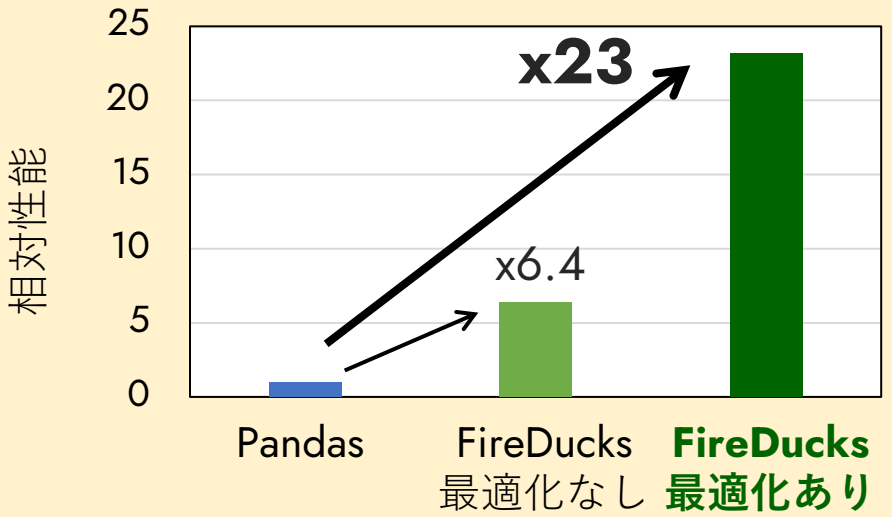


最適化後のフロー

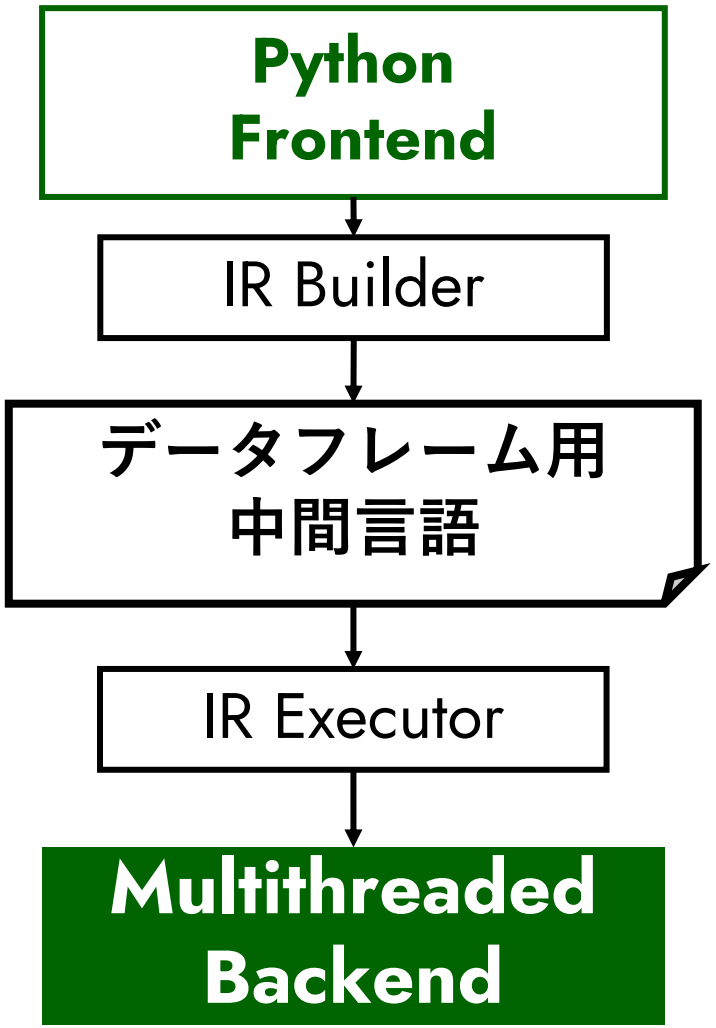


中間データを
大幅削減（1/60）

性能例（31列 x 2集約関数）



Backend



FireDucksの構造

中間言語中の各命令を実行するカーネルの集合

CPU用のマルチスレッドバックエンド（C++）

- データ構造にApache Arrowを利用
- Arrowが提供するカーネルに加えて，
  並列化・最適化を強化したカーネルを追加

Backendでの最適化例: groupby

```
df.groupby("x").sum()
```

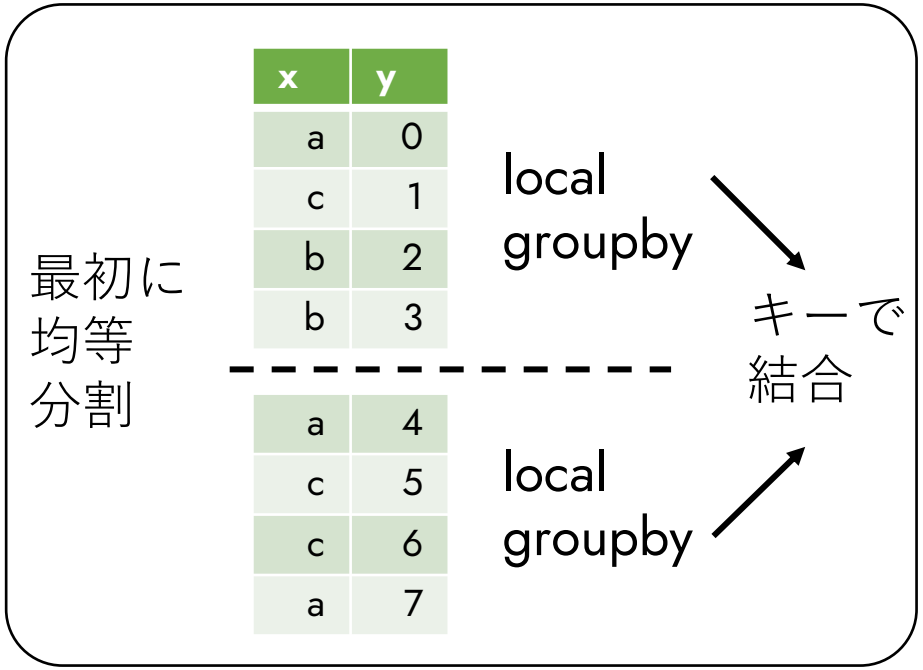
グループ数推定

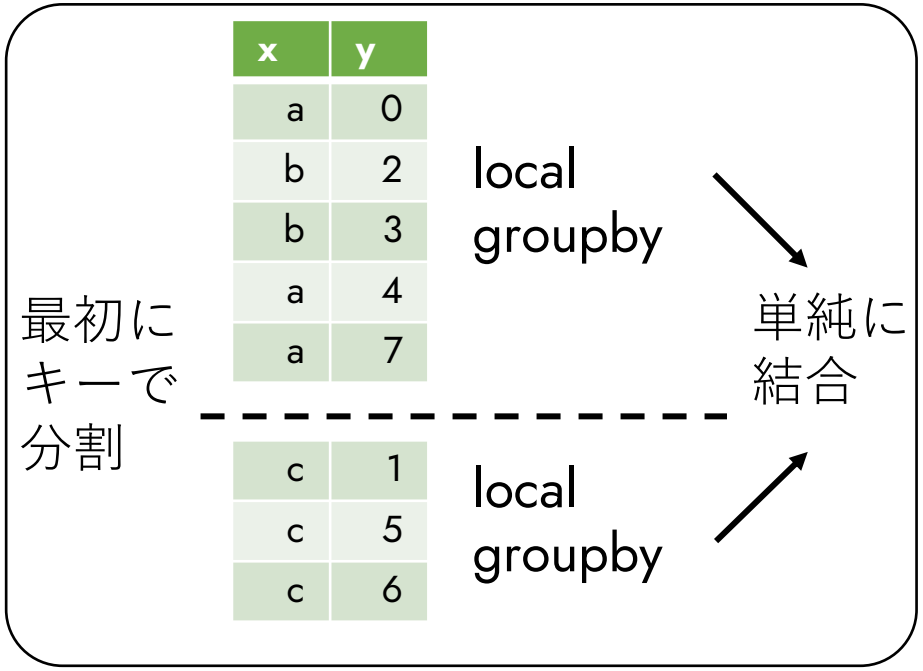
グループ数が少ないときに向いた
並列groupbyアルゴリズム

グループ数が多いときに向いた
並列groupbyアルゴリズム

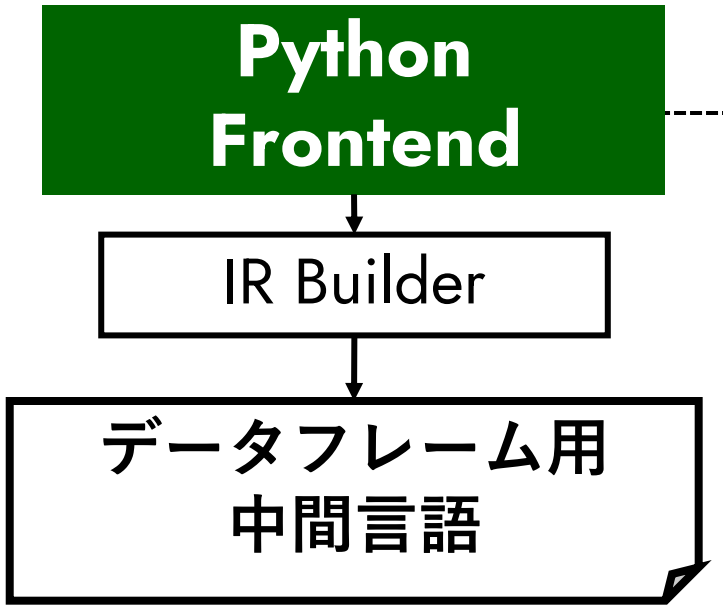
元データ

x	y
a	0
c	1
b	2
b	3
a	4
c	5
c	6
a	7





互換性向上の仕組み: Fallback



pandas

Fallback

中間言語を作らずに，Frontendからpandasを直接呼び出す

互換性は上がるけど，性能は上がらない（FireDucksで速くならないときはほぼこれ）

```
$ FIREDUCKS_FLAGS="-Wfallback" python -mfireducks.imhook demo.py
demo.py:4: FallbackWarning: Series.plot 0.201566 sec ...
```

※ Warningを出すことは可能（ご報告して下さい）

Tips

applyやループを利用しない

（A列が2より大きい行のB列の合計）

ループ

```
s = 0
for i in range(len(df)):
    if df["A"][i] > 2:
        s += df["B"][i]
```

apply

```
s = 0
def func(row):
    if row[“a”] > 2:
        s += row[“B”]

df.apply(func)
```

```
s = df[df["A"] > 2]["B"].sum()
```

時間計測の注意

```
t0 = time.time()
df.sort_values(“a”)
t1 = time.time()
print(t1 – t0)
```

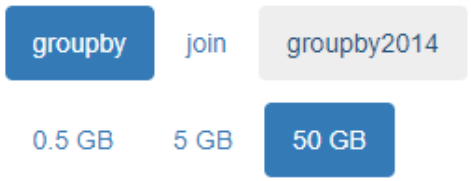
正しい時間計測ができない（遅延実行）

```
df._evaluate()
t0 = time.time()
df.sort_values(“a”)._evaluate()
t1 = time.time()
print(t1 – t0)
```

明示的な実行の指示

FireDucksの性能 (要素処理 groupby, join)

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



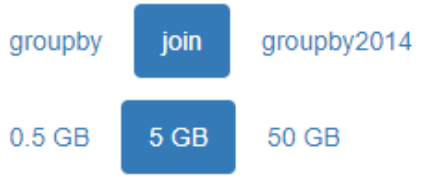
basic questions

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

1位

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.jl	0.7.13	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



basic questions

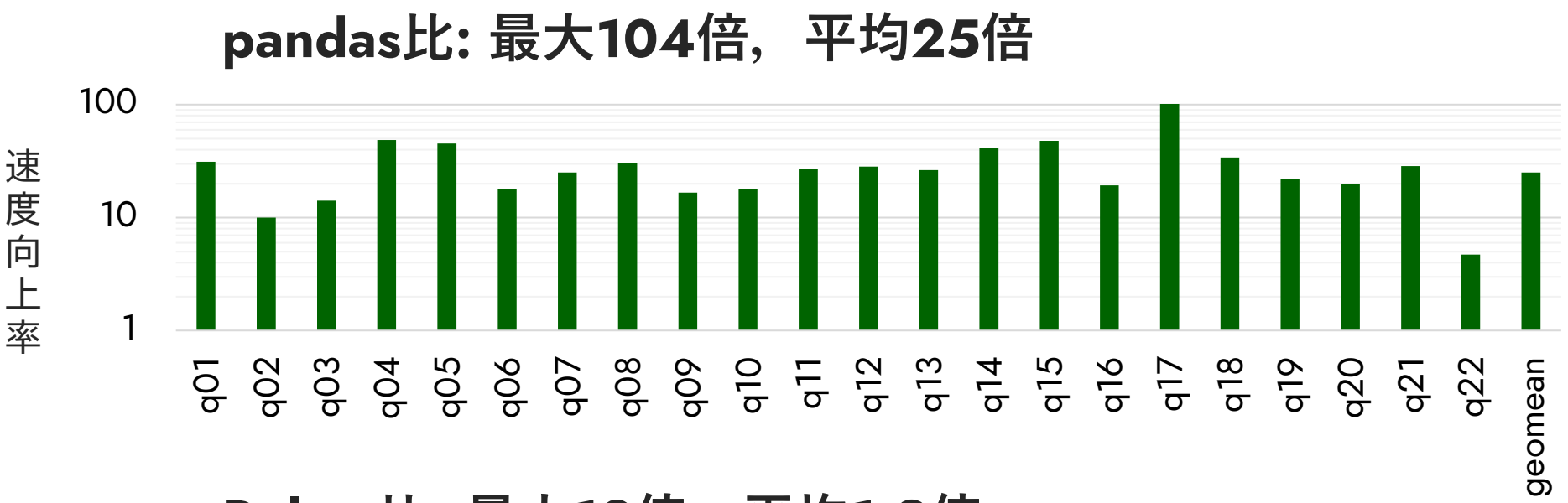
Input table: 100,000,000 rows x 7 columns (5 GB)

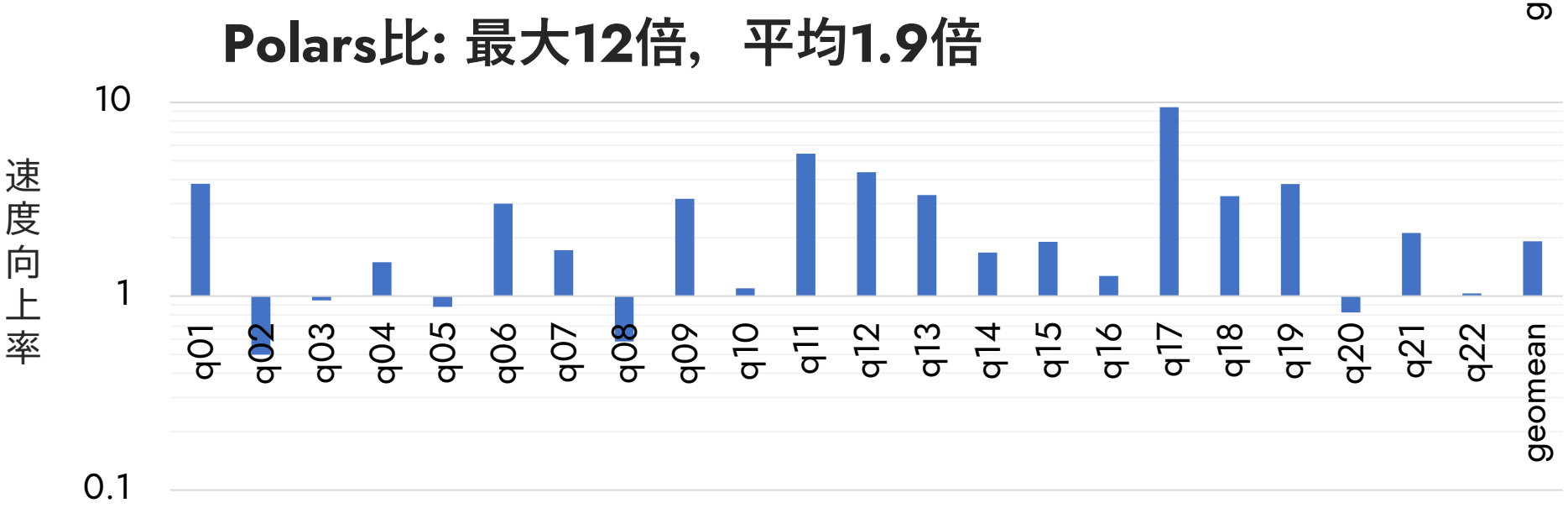
3位

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.jl	0.7.13	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

FireDucksの性能 (TPC-Hベンチマーク Scale Factor=10)





評価環境

インテル® Xeon® Gold 5317 プロセッサー
(12コア x 2ソケット)

メモリ: 256GB
OS: Linux

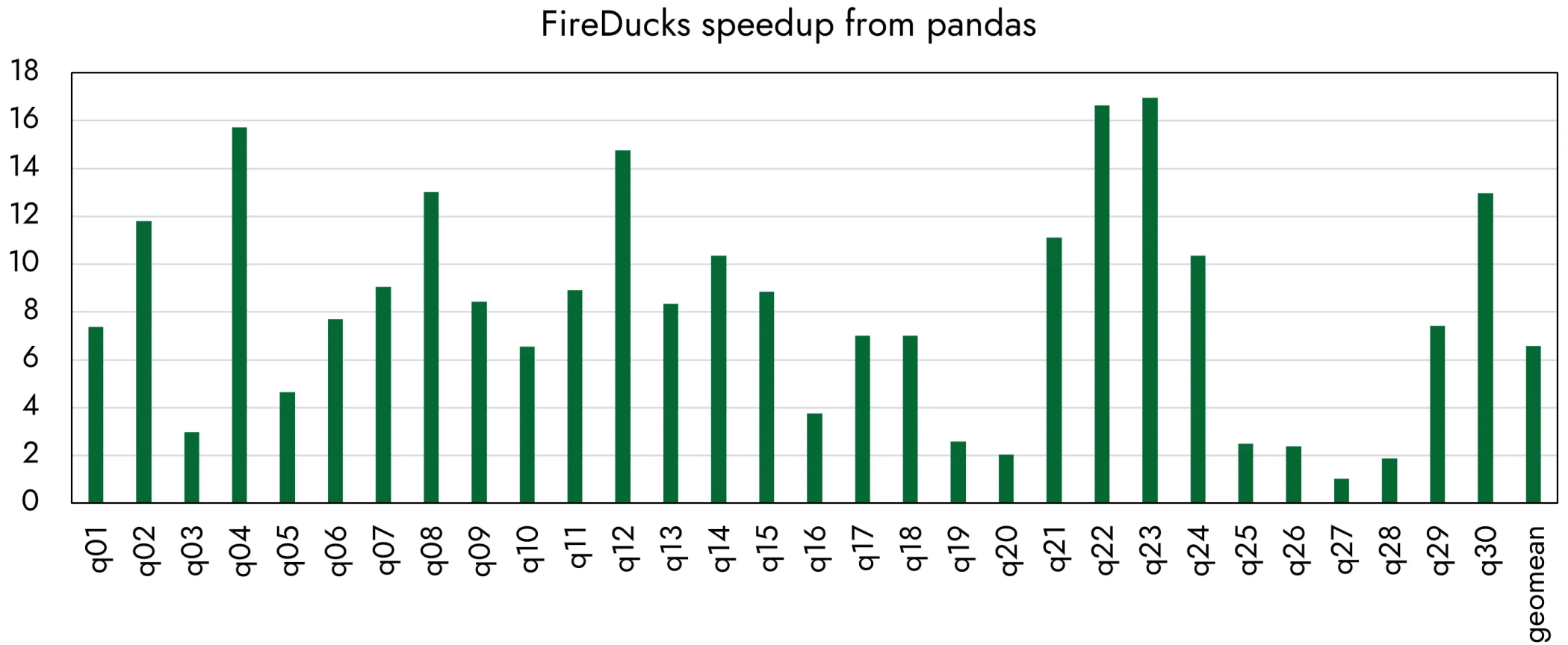
pandas 2.2.0
polars 0.20.7
FireDucks 0.10.1

ベンチマークコード

https://github.com/fireducks-dev/polars-tpch/tree/fireducks

FireDucksの性能（TPCx-BBベンチマーク）

ETL(抽出、変換、ロード)および機械学習のワークフロー



- pandas-2.1.4
- fireducks-0.9.3

- CPU: Intel(R) Xeon(R) Gold 5317 CPU @ 3.00GHz x 2sockets （合計48HWスレッド）
- メインメモリ: 256GB

DataFrameライブラリの比較

	pandas 互換性	シングルノード 性能	マルチノード 性能
FireDucks	◎	◎	×
Polars	×	○	×
Modin	○	△	○
Dask/Vaex	△	△	○
Pandas	○	×	×

FireDucksの利用

pipコマンドでインストール可能（BSDライセンス）

```
$ pip install fireducks
```

※ 現在はLinuxのみサポート（WSL可）

import文の書き換えだけでなく，pandasからの自動変換も可能

pythonコマンドの引数で指定

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

jupyter notebookではマジックコマンド

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

Resource

Webサイト

https://fireducks-dev.github.io/ja/
（ユーザーガイド，ベンチマークなど）

github (issue report)

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

slack (Q&A，雑談)

twitter/X (リリース情報)

https://x.com/fireducksdev

おわりに

FireDucksは，pandasのdrop-in replacementで使える高速データフレームライブラリです

実行時コンパイラ技術の活用により，pandasの弱点であるマルチスレッド実行，自動最適化を行います

ぜひご活用下さい