PyCon JP 2024

FireDucksのすすめ

2024/09/27

大野 善之

自己紹介

大野 善之 / Yoshiyuki Ohno

(NEC セキュアシステムプラットフォーム研究所所属)

ソフトウェアを速くすることを仕事にしています

- アルゴリズム軽量化
- 並列計算・SIMDベクトル化

Python スキルは素人に毛が生えた程度

本発表について

日本発のデータフレームライブラリ FireDucks を紹介します

FireDucks は **pandas と API 互換で高速** なライブラリです
FireDucks を使うだけなら本発表を聞く必要はありません

使い方は pythonコマンドの引数に -m fireducks.pandas を指定するだけ

```
$ pip install fireducks
$ python3 -m fireducks.pandas your_pandas_program.py
```

本発表では FireDucks の速さ・互換性のための工夫や,
使いこなすための利用方法を紹介します.
本発表をとおして FireDucks を詳しく知り,
FireDucks を使いこなしていただければ幸いです

アジェンダ

本発表では，以下の内容をお話しします

- 導入 : DataFrame とは？ pandas とは？
- FireDucks のねらい，アーキテクチャ
- FireDucks を支える技術
  - 高速化の仕組み（自動最適化，マルチスレッド化）
  - pandas 互換性のための仕組み
- 他のデータフレームライブラリとの性能比較
- （実演とともに）FireDucks の導入方法，利用方法

データフレームとは？

- データ分析においてよく使用される形式のデータ構造
- 行と列の2次元の表形式のデータで各列が特定の変数や属性を表現

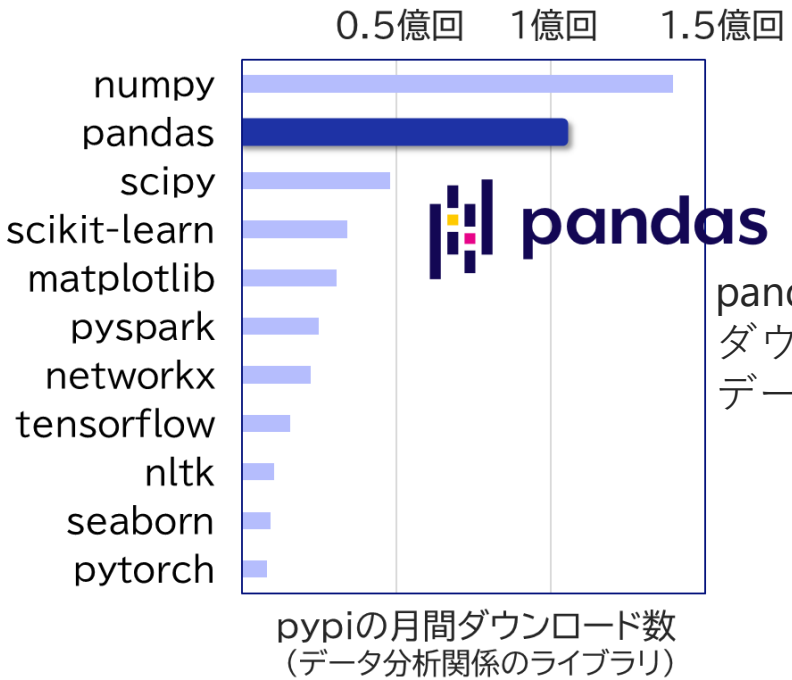
- Python の pandas を使用されることが多く，データの取得・変換・分析を行うことができ，データ分析の初期段階でのデータの取り込みや前処理に利用

データフレームの例：bitcoinの1分ごとの価格情報

	Timestamp	Open	High	Low	Close	Volume_(BTC)	Volume_(Currency)	Weighted_Price
0	1325317920	4.39	4.39	4.39	4.39	0.455581	2.000000	4.390000
1	1325317980	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	1325318040	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	1325318100	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4	1325318160	NaN	NaN	NaN	NaN	NaN	NaN	NaN
...	...	...	...	...	...	...	...	...
4857372	1617148560	58714.31	58714.31	58686.00	58686.00	1.384487	81259.372187	58692.753339
4857373	1617148620	58683.97	58693.43	58683.97	58685.81	7.294848	428158.146640	58693.226508
4857374	1617148680	58693.43	58723.84	58693.43	58723.84	1.705682	100117.070370	58696.198496
4857375	1617148740	58742.18	58770.38	58742.18	58760.59	0.720415	42332.958633	58761.866202
4857376	1617148800	58767.75	58778.18	58755.97	58778.18	2.712831	159417.751000	58764.349363

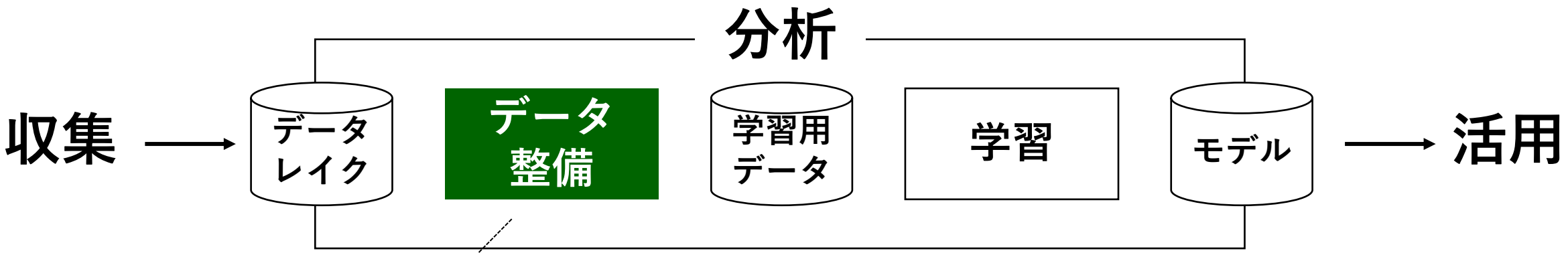
4857377 rows × 8 columns

https://www.kaggle.com/datasets/mczielinski/bitcoin-historical-data



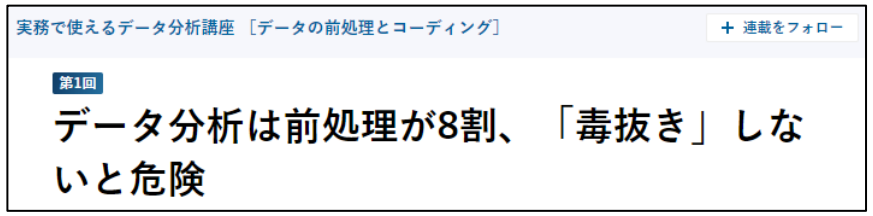
pandas は月間１億回以上ダウンロードされるデータ分析の標準ツール

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

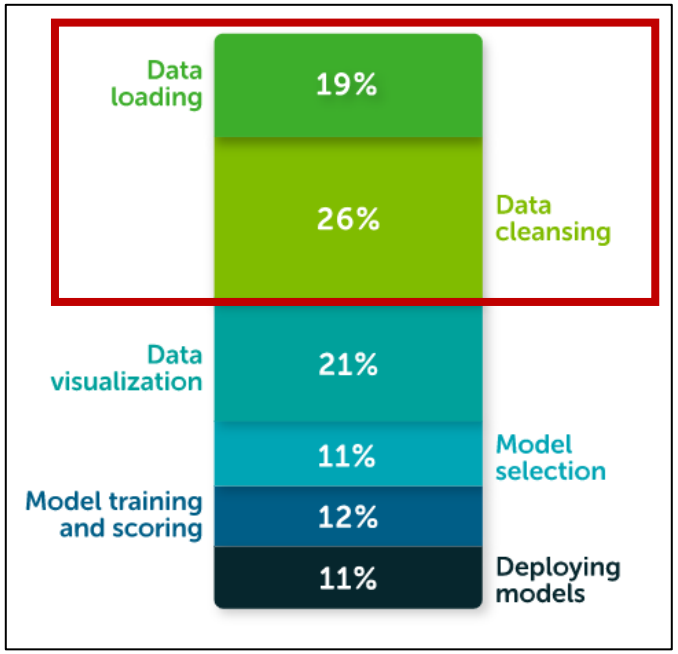

実務で使えるデータ分析講座［データの前処理とコーディング］ ＋ 連載をフォロー

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

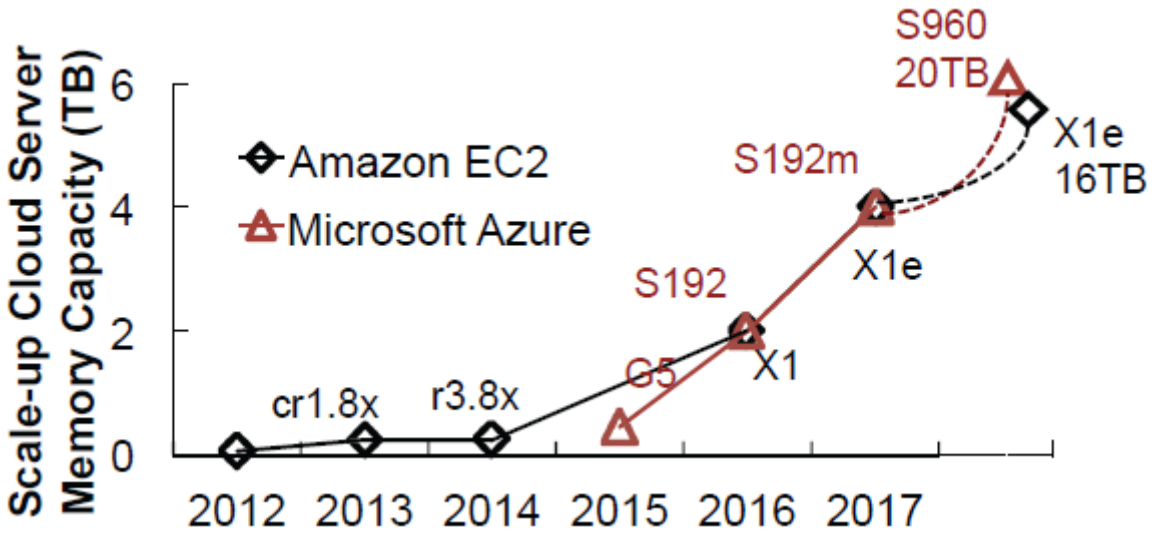


Anaconda The State of Data Science 2020

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

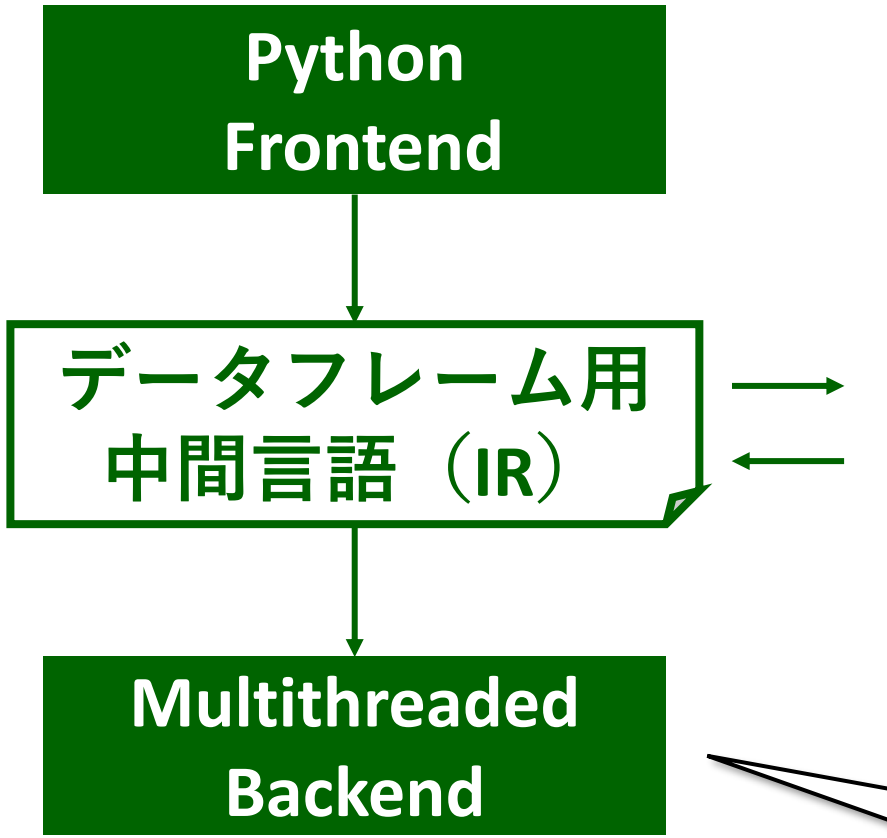
DataFrameライブラリの比較

	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

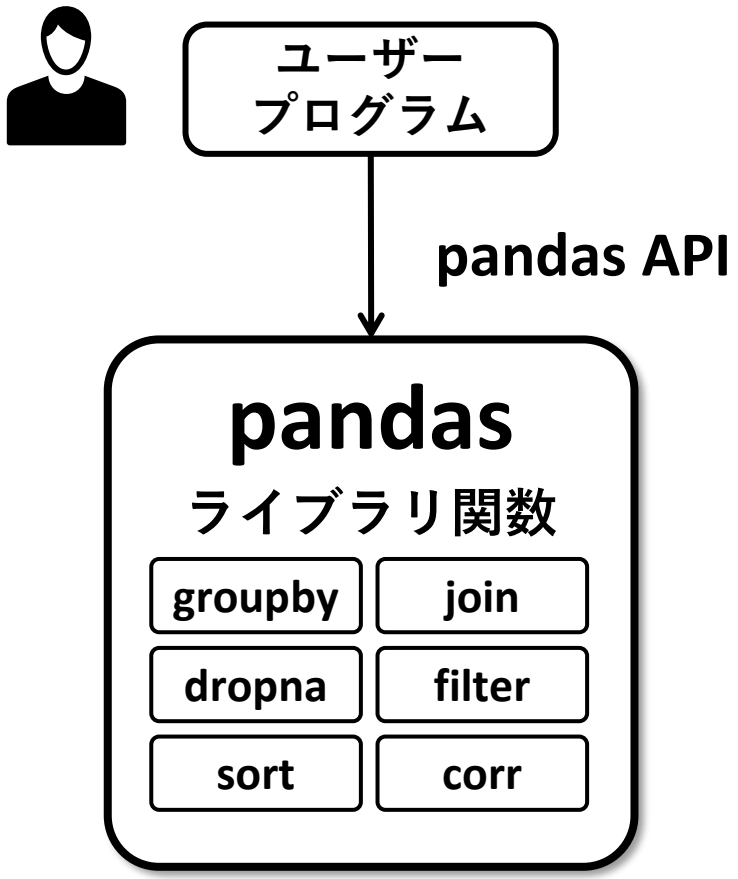
ユーザーAPIとは
独立したバックエンド
で実行

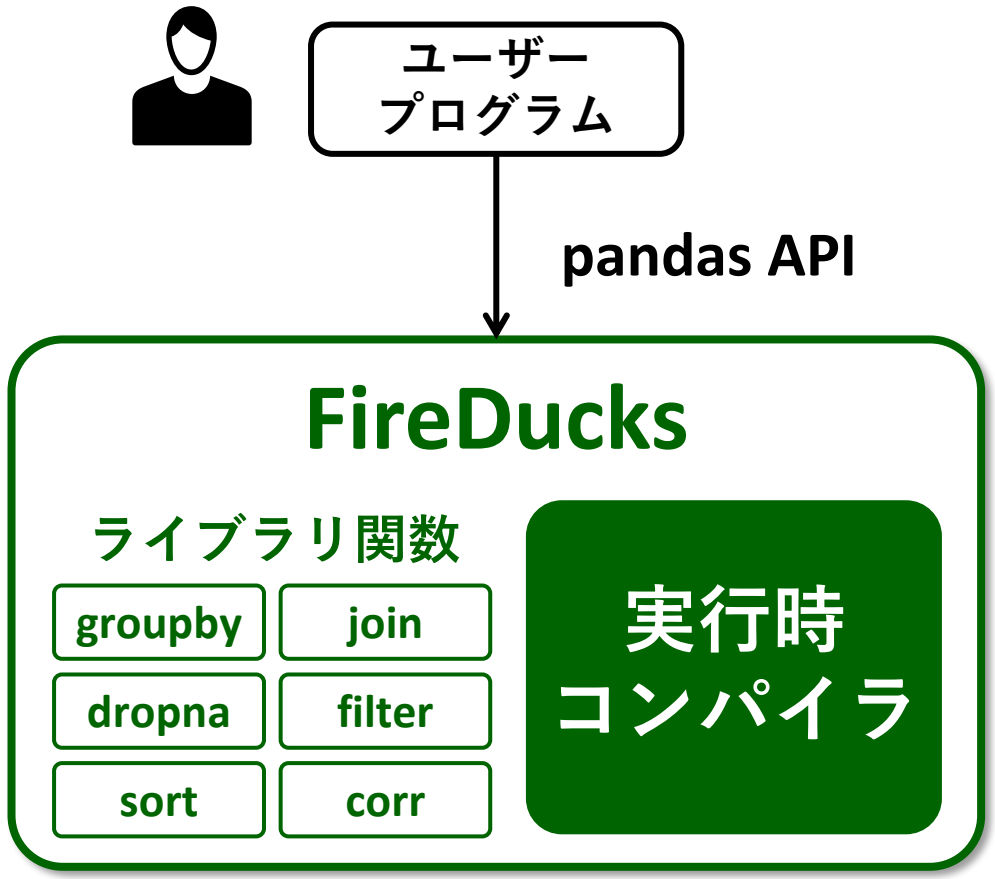
高速化2

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

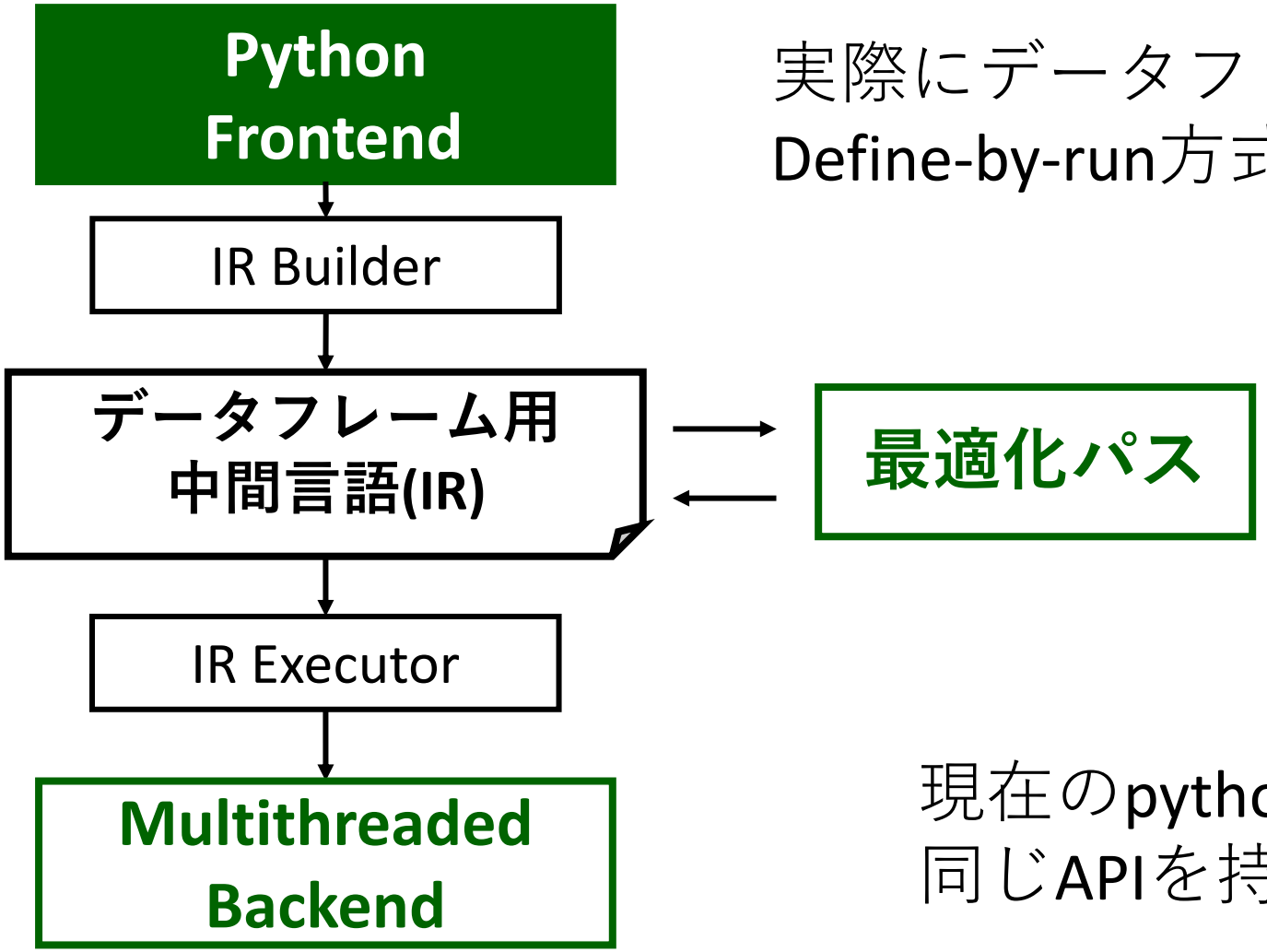
```
%t1 = read_csv('data.csv', %arg0)
%t2 = rolling_aggregate(%t1, 60, 60, 'mean')
%t3 = project(%t2, 'Close')
%t4 = slice(%t3, -1000, None, 1)
```

※: 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()
```

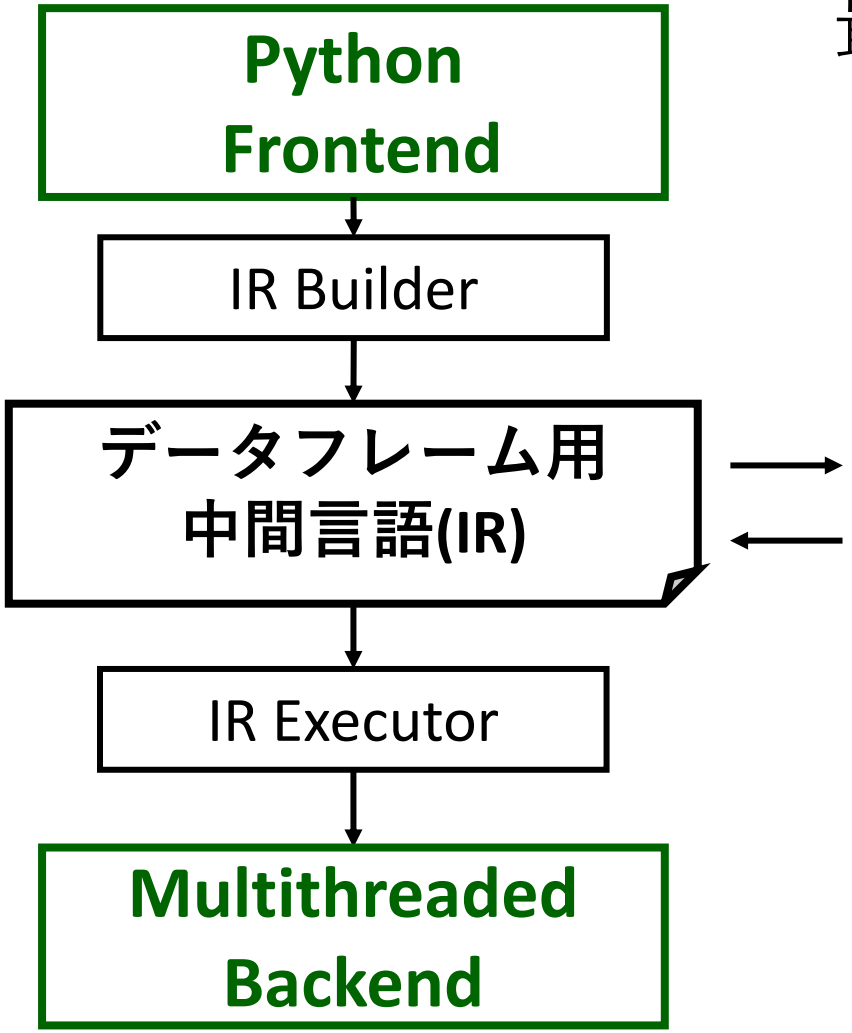
```
%t1 = read_csv('data.csv', %arg0)
%t2 = rolling_aggregate(%t1, 60, 60, 'mean')
%t3 = project(%t2, 'Close’)
%t4 = slice(%t3, -1000, None, 1)
```

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

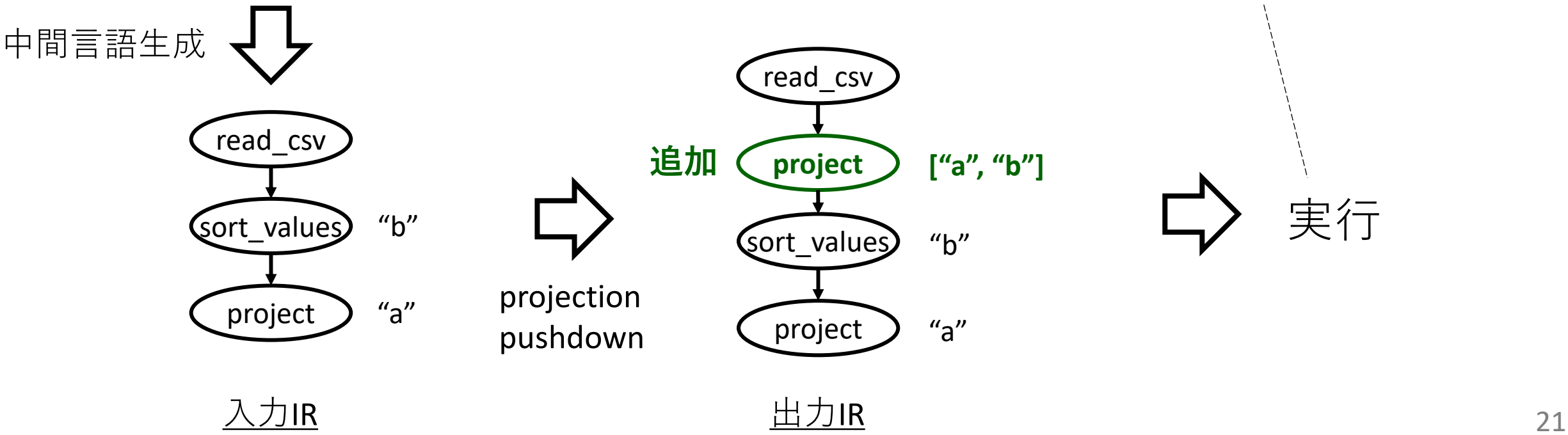
projection（列の抽出）を前出しすることで，中間データを削減

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null でない行だけ抽出 → a列がnullの行を削除

➡ 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

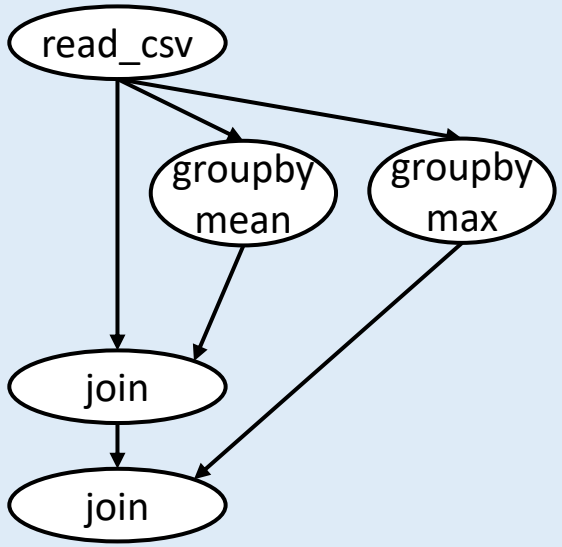
追加

time	val
10:20	100
10:30	120

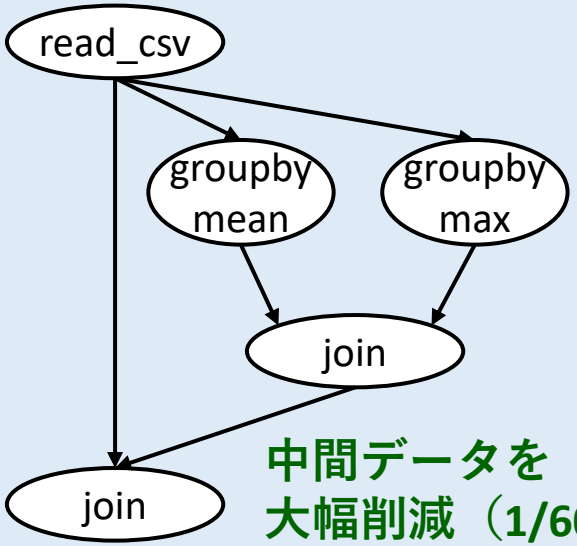
⇨

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

元の計算フロー

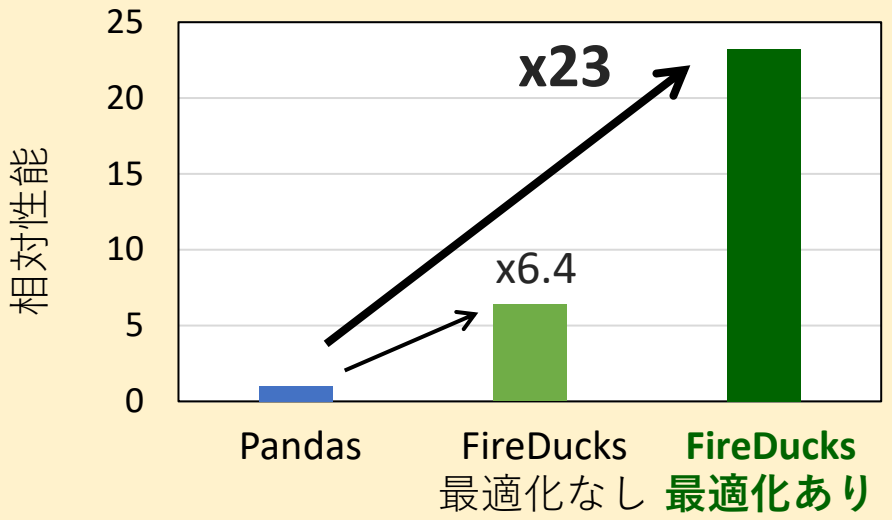


最適化後のフロー

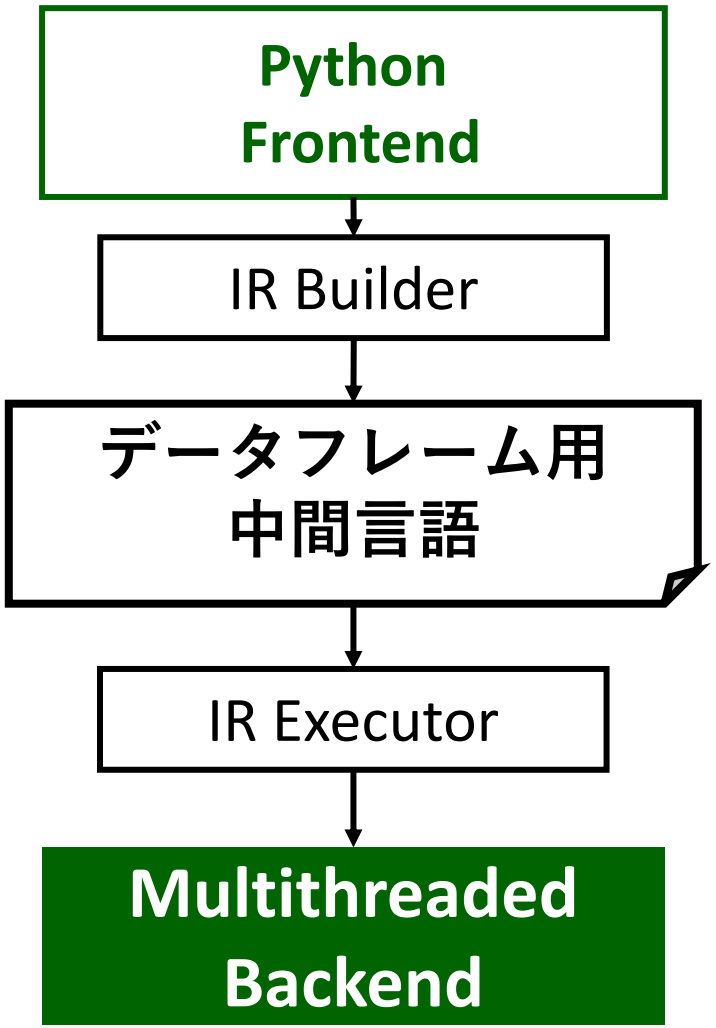


中間データを大幅削減（1/60）

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()
```

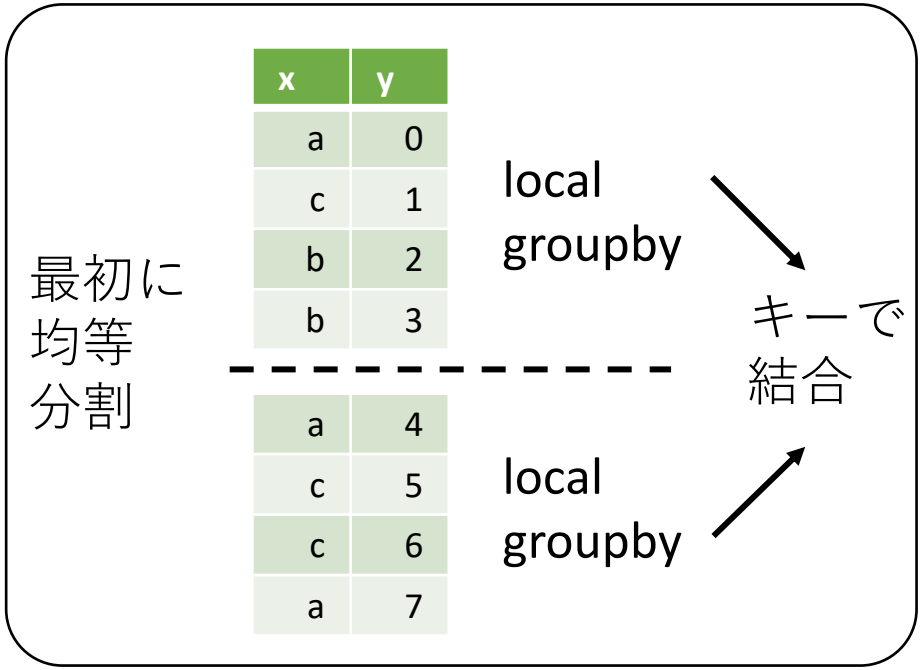
グループ数推定

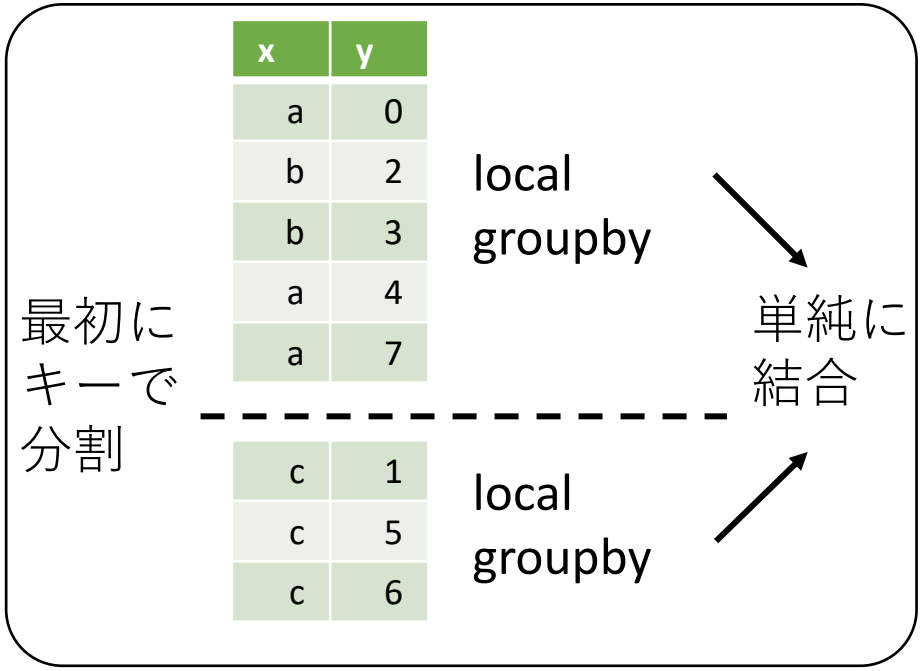
グループ数が少ないときに向いた
並列groupbyアルゴリズム

グループ数が多いときに向いた
並列groupbyアルゴリズム

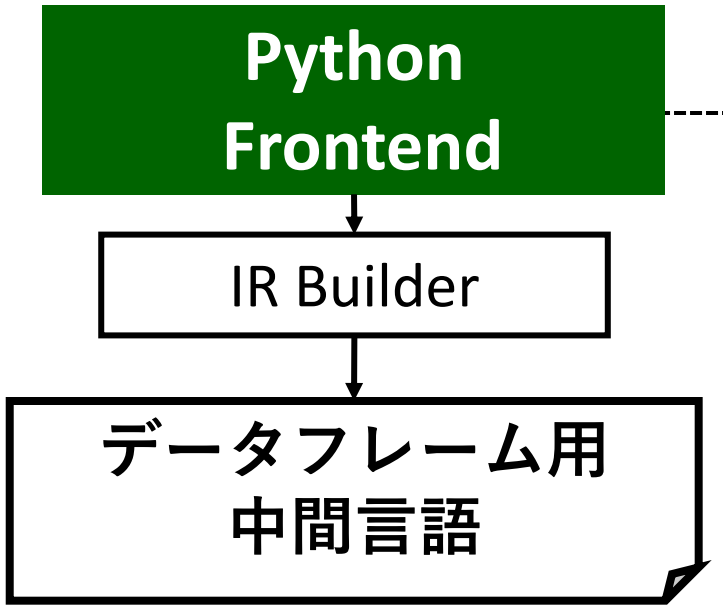
元データ

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

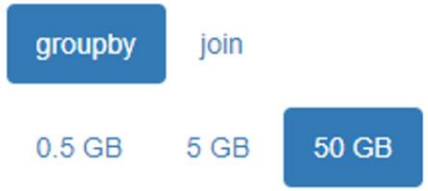
互換性は上がるけど，性能は上がらない（FireDucksで速くならないときはほぼこれ）

```
$ FIREDUCKS_FLAGS="-Wfallback" python -mfireducks.pandas 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）

FireDucksの性能 (要素処理 groupby, join)

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



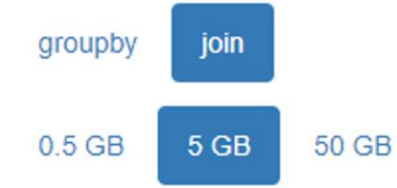
basic questions

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

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	0.7.13	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

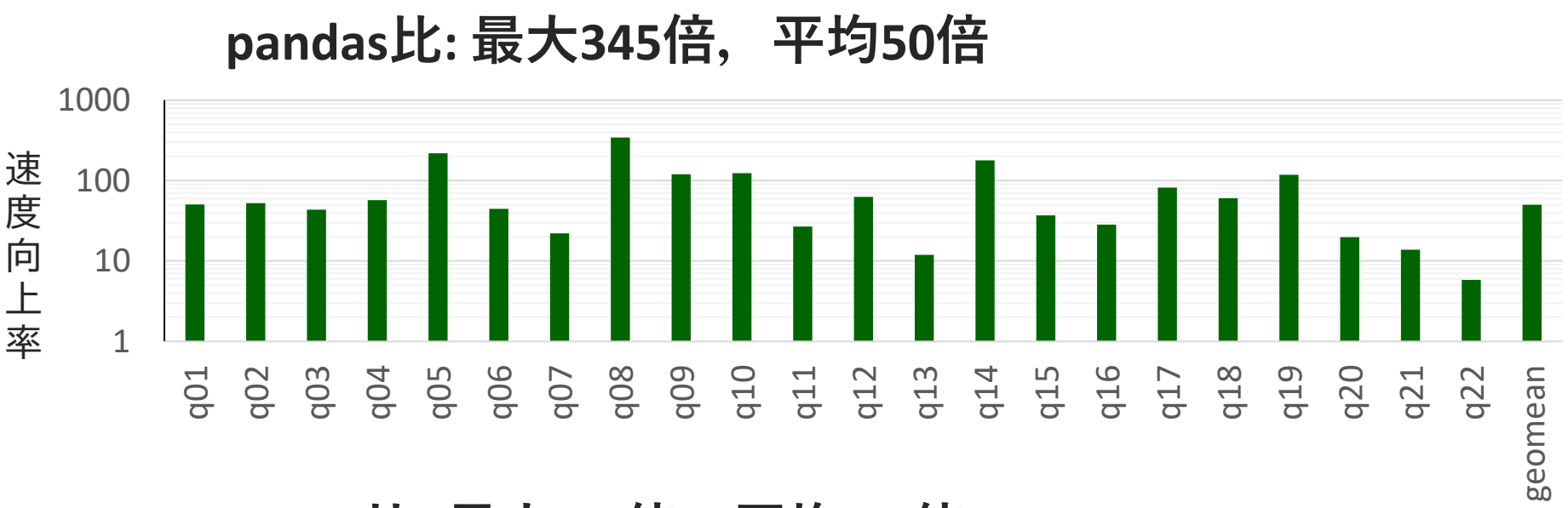
Input table: 100,000,000 rows x 7 columns (5 GB)

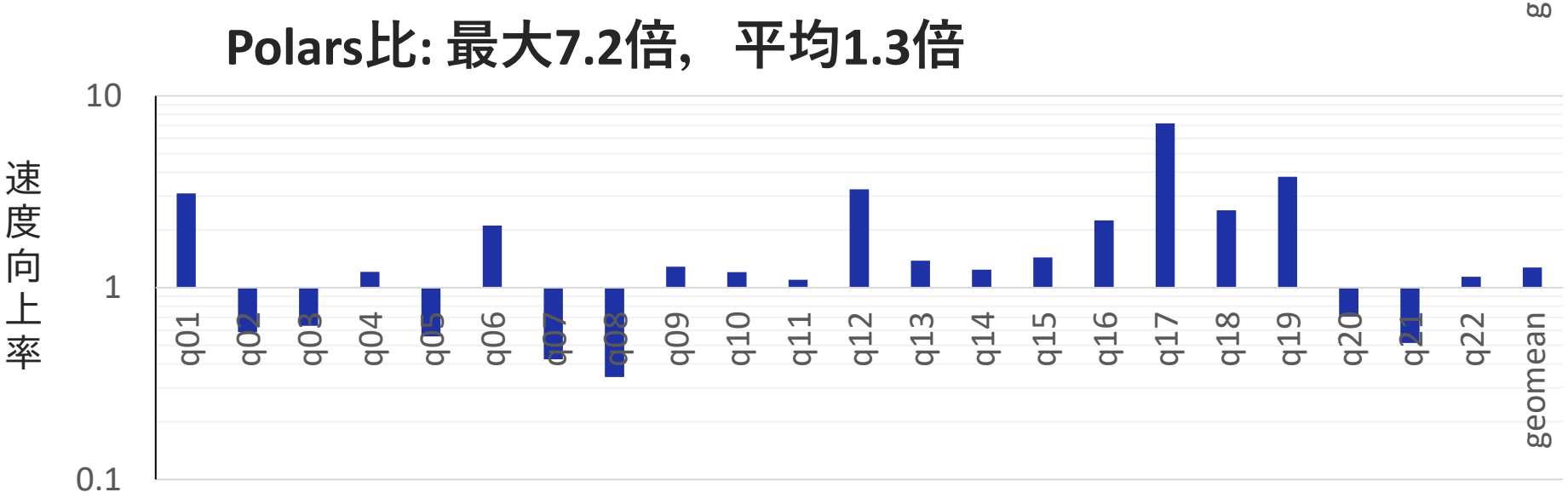
rank-1

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	0.7.13	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

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

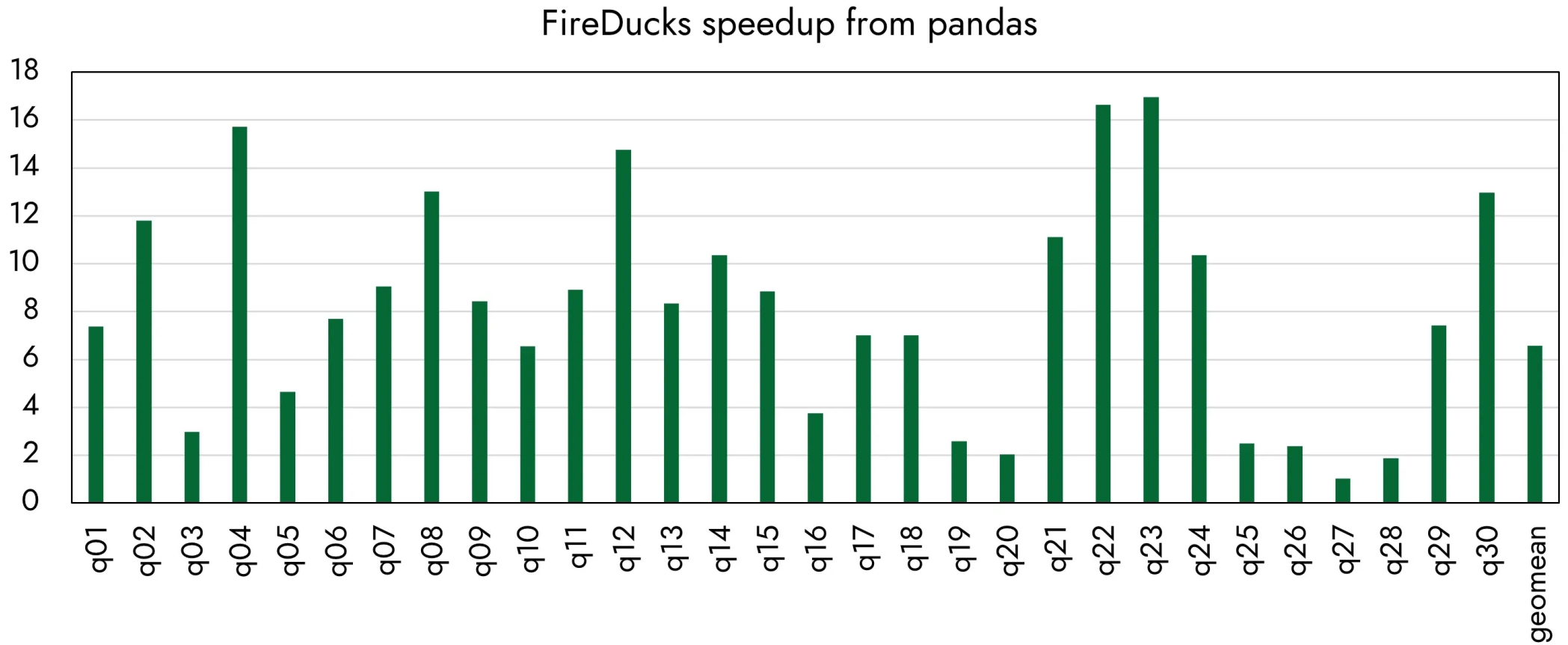
pandas 2.2.2
polars 1.6.0
FireDucks 1.0.3

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

インストール

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）

利用方法

方法１：Import Hook で pandas プログラムから自動変換

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 as pd
```

方法２：プログラム中の import 文を明示的に書き換えても OK

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

Tips：遅延実行モデル

FireDucks は遅延実行モデルで
結果が必要になったときにまとめて処理が実行される

(例)

```
df = pd.read_csv("data.csv")
df = df.sort_values("a")
df.to_csv("sorted.csv")
```

to_csv のタイミングで,
read / sort / write がまとめて行われる

主な評価タイミング

- 結果の書き出し to_csv, to_parquet, print, __repr__ など
- 明示的に _evaluate() で指示したとき
- fallback が発生したとき

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

```
df._evaluate()
t0 = time.time()
df.sort_values("a")._evaluate()
t1 = time.time()
print(t1 - t0)
```

_evaluate() で明示的な実行の指示

Tips : FireDucks では高速化できない書き方

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()
```

ループで書くと 高速化できないだけでなく，
極端に遅くなる可能性あり

適切な pandas API で記述することで
FireDucks で高速化可能

FireDucks で速くならないときには

まずは fallback を疑う
(FireDucksで速くならない原因の多くはfallback ）

Fallback の有無は環境変数をつけて実行することで確認可能

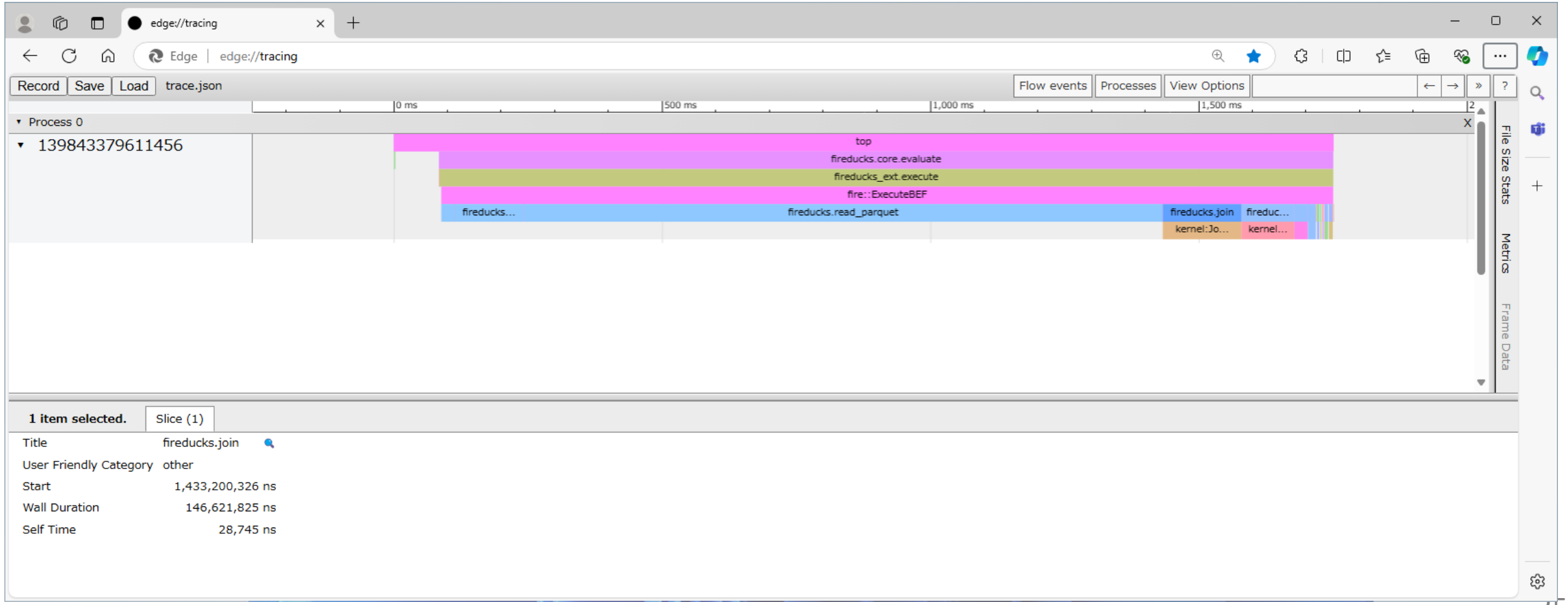
```
$ FIREDUCKS_FLAGS="-Wfallback" python -mfireducks.pandas demo.py
demo.py:4: FallbackWarning: Series.plot 0.201566 sec ...
```

同じAPI でもパラメータによって， fallback したり，
FireDucks で動いたりすることがあるため注意が必要

※ github の issue や slack 等でご報告していただけると助かります

Trace を見てみる

Trace ファイルを出力して，処理ごとの実行時間をグラフィカルに確認することができます



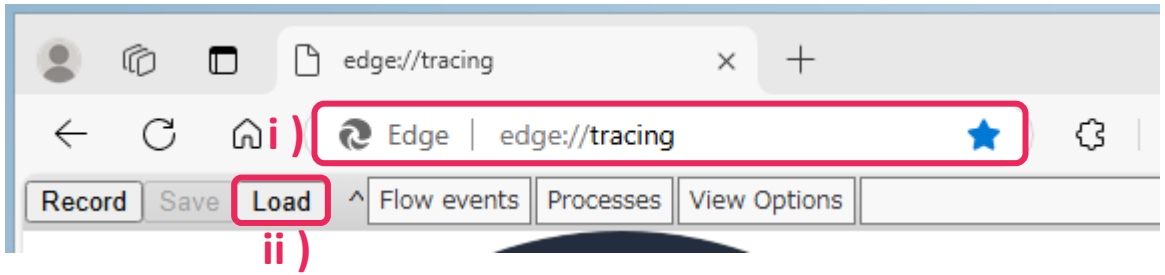
Trace を見てみる

① 環境変数をつけて実行 → trace ファイルを出力

```
$ FIREDUCKS_FLAGS="--trace=3" python -mfireducks.pandas demo.py
$ ls ./
demo.py         trace.json
```

② edge もしくは chrome の trace-viewer で確認

(※ trace-viewer は edge/chrome の標準機能で追加の install は不要です)



i. アドレスバーに edge://tracing (もしくは chrome::tracing) と入力
ii. Load ボタンをクリックして trace.json ファイルを選択して開く

実演

時間が許す限り実演します

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

FireDucks スポンサーブースのご案内

スポンサーブースでは FireDucks の開発チームメンバーが常駐しています

FireDucks に関してもっと詳しい話を聞きたい方
利用に関してレクチャを受けたい方
使ってみたけど速くならなかった方 etc...

開発メンバーが応対しますので，ぜひスポンサーブースへお越しください

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