実行時コンパイル技術による金融データ分析の高速化

2026年03月21日

大野善之、サハソウラブ、石坂一久 (NEC)

本日のアジェンダ

- 金融分野の分析手法
- 大規模金融データ分析における課題
  - pandas の課題とベストプラクティス
- 提案技術
- 実装
- 評価
- まとめ

金融分野の分析手法

金融データ分析とは

市場データや取引データ、財務情報などを分析し、投資判断の高度化、リスクの把握・管理、業務効率や収益性の向上を目的として行われる分析のことです。

金融の各分野におけるデータ分析の種類

リスク管理

- 取引相手に対する総エクスポージャーを測定
- デフォルト（債務不履行）の発生確率を推定

AML

- 不審な活動を検知
- マネーロンダリングのネットワーク（組織）を検知

顧客ポートフォリオ分析

- 類似した顧客をグループ化
- 長期的な収益性を測定
- 解約予測

市場データ分析

- 価格変動を特定
- 資産間の依存関係、ボラティリティ分析

ヘッジファンド

- 戦略分析のためにティックデータを集計
- ポートフォリオのパフォーマンス要因を分析

現在の課題

🚧 ビジネス上の課題

データ量が多い
市場データや取引ログが爆発的に増加

金融クエリの複雑化
複雑なジョイン、計算、統計モデルを必要

即時意思決定
不正検知・リスク監視の遅延は損失につなげる

コスト効率と拡張性
データ量増加に伴いインフラコストが上昇

🧩 ソフトウェア的な課題

従来の処理速度ではリアルタイム分析が困難

🧠 人間的な課題

「遅い」書き方ができてしまう

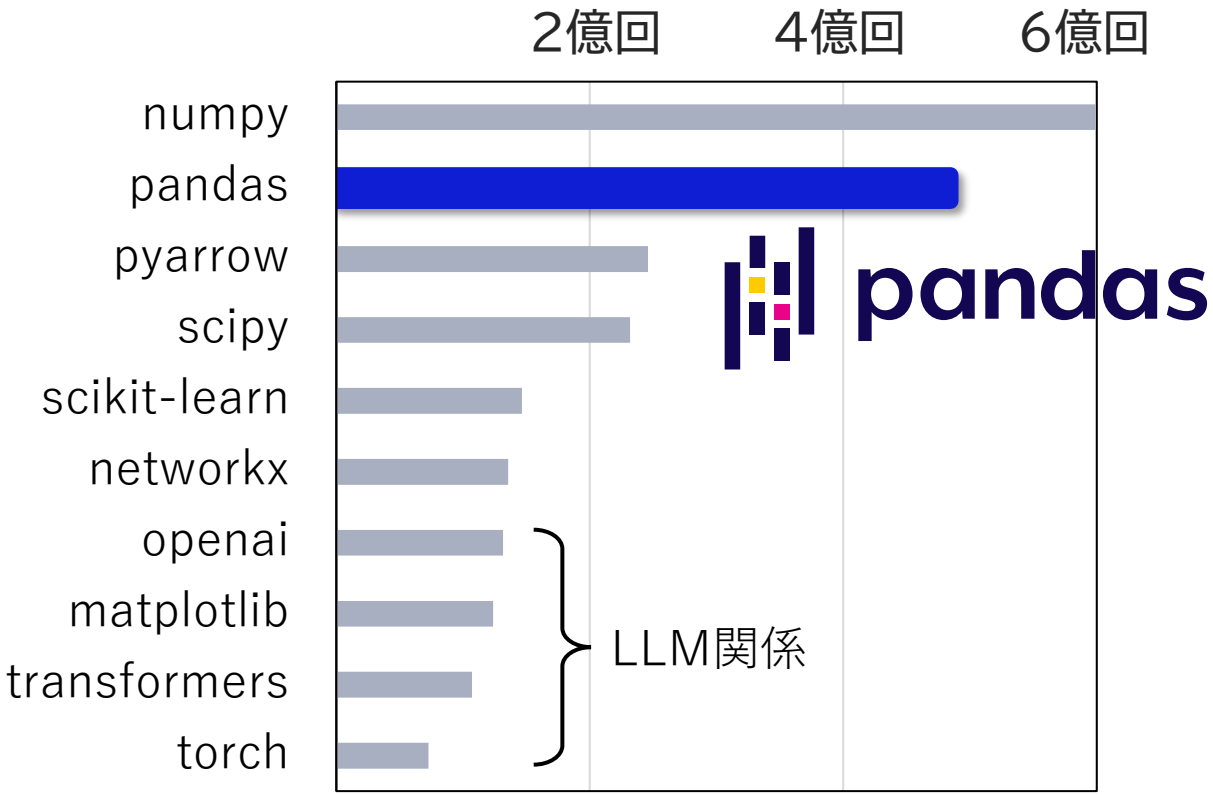
研究目的

① 金融データ分析でよく使われているpandasのプログラムを全く変更せずに高速化する

② 金融の様々な領域の分析におけるその高速化の有効性確認する

pandas: データサイエンティストの必須ツール

- 月間4億回以上ダウンロードされるPythonデータ分析の標準的なライブラリ
- データ分析/機械学習分野の多くのライブラリと併用され，中心的役割を担う



pypiの月間ダウンロード数
（データ分析関係のライブラリ 2025年11月）

✅ **長所**
- 使いやすく直感的
- 時系列データに強い
- Pythonツールと統合しやすい

❌ **短所：**
- 単一スレッドで処理が遅い
- 最適化機能がついてない
- メモリ使用量が大きい
- 大規模データに弱い

🔥 **「書き方」によって性能は劣化する！**

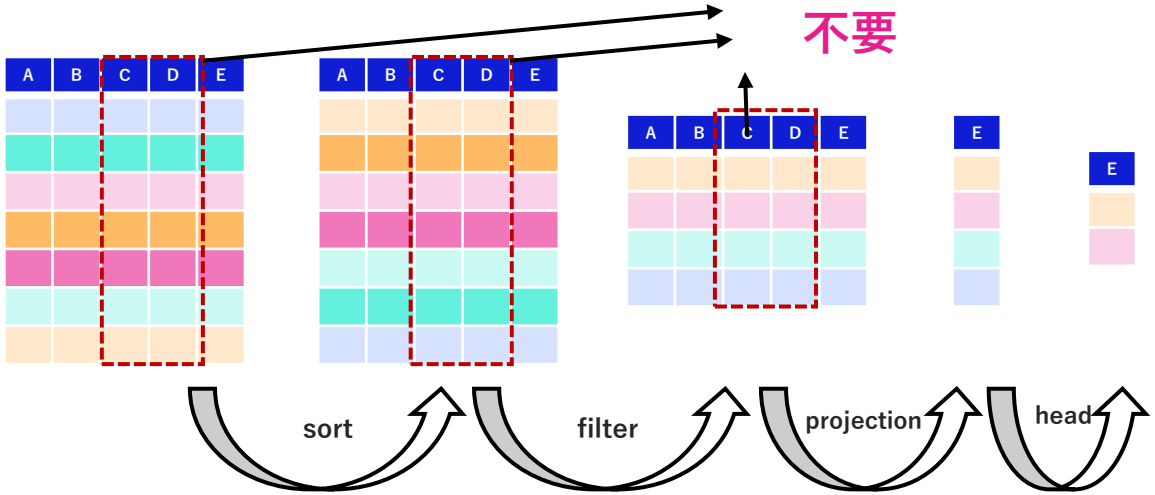
大規模金融データ分析におけるベストプラクティス

大規模データ処理においては、実行順序が重要

SAMPLE QUERY

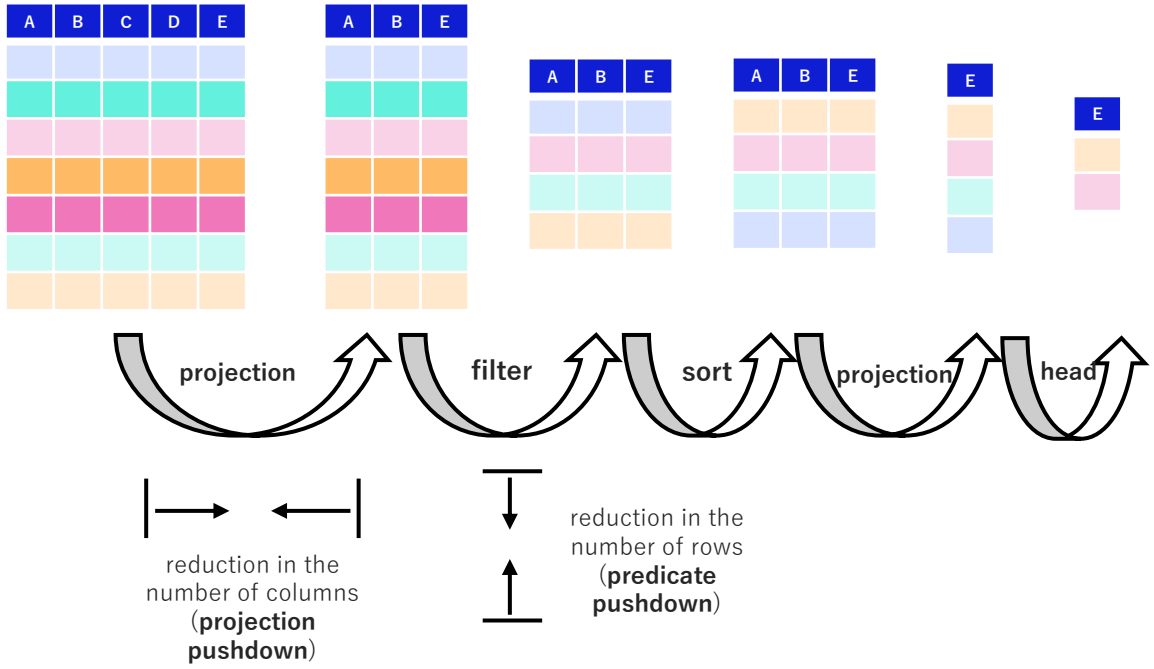
```
df.sort_values(“A”)
  .query(“B > 1”)[“E”]
  .head(2)
```

※ 並び順：黄 → 赤 → 緑 → 青
※ B=1：濃い色調、B=2：淡い色調



OPTIMIZED QUERY

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



パターン：AMLにおける同一ペア間での高額かつ迅速な取引

sender_id	receiver_id	amount	timestamp
622	1141	2925.67	2025-01-01 16:04:48
622	1141	69.04	2025-01-01 17:12:49
622	1141	3091.09	2025-01-01 19:13:23
622	1141	10442.10	2025-01-01 23:51:18
622	1141	7164.37	2025-01-01 23:52:10
622	1141	5562.49	2025-01-01 23:53:58


```
suspicious = (
        trans_data.merge(
            trans_data.rename(columns=lambda x: x + "_t2"),
            left_on=[“sender_id”, “receiver_id”],
            right_on=["sender_id_t2", "receiver_id_t2"],
        )
        .pipe(
            lambda x: x[
                abs(x["timestamp"] - x["timestamp_t2"]) <= pd.Timedelta(minutes=2)
            ]
        ).pipe(lambda x: x[x["amount"] > 5000])
         .groupby(["sender_id", "receiver_id"])
         .agg(txn_count=("txn_id", "count"), total_amount=("amount_t2", "sum"))
         .query("txn_count >= 5 and total_amount > 100000")
)
```

← **自己結合**：同一アカウントペア内での取引

← **超短期取引**：極めて短い時間枠によるフィルタリング

← **高額取引の絞り込み**：高リスクな資金移動

Predicate pushdown: 手動最適化

```
suspicious = (
        trans_data.merge(
            trans_data.rename(columns=lambda x: x + "_t2"),
            left_on=[“sender_id”, “receiver_id”], right_on=["sender_id_t2", "receiver_id_t2"],
        )
        .pipe(
            lambda x: x[abs(x["timestamp"] - x["timestamp_t2"]) <= pd.Timedelta(minutes=2)]
        ).pipe(lambda x: x[x["amount"] > 5000])
         .groupby(["sender_id", "receiver_id"]).agg(txn_count=("txn_id", "count"), total_amount=("amount_t2", "sum"))
         .query("txn_count >= 5 and total_amount > 100000")
)
```

merge -> filter

実行時間:
２分 １４秒

実行時間:
２５秒

5.4倍

```
trans_data = trans_data[trans_data[“amount”] > 5000]
suspicious = (
        trans_data.merge(
            trans_data.rename(columns=lambda x: x + "_t2"),
            left_on=[“sender_id”, “receiver_id”], right_on=["sender_id_t2", "receiver_id_t2"],
        )
        .pipe(
            lambda x: x[abs(x["timestamp"] - x["timestamp_t2"]) <= pd.Timedelta(minutes=2)]
        ).groupby(["sender_id", "receiver_id"]).agg(txn_count=("txn_id", "count"), total_amount=("amount_t2", "sum"))
         .query("txn_count >= 5 and total_amount > 100000")
)
```

filter -> merge

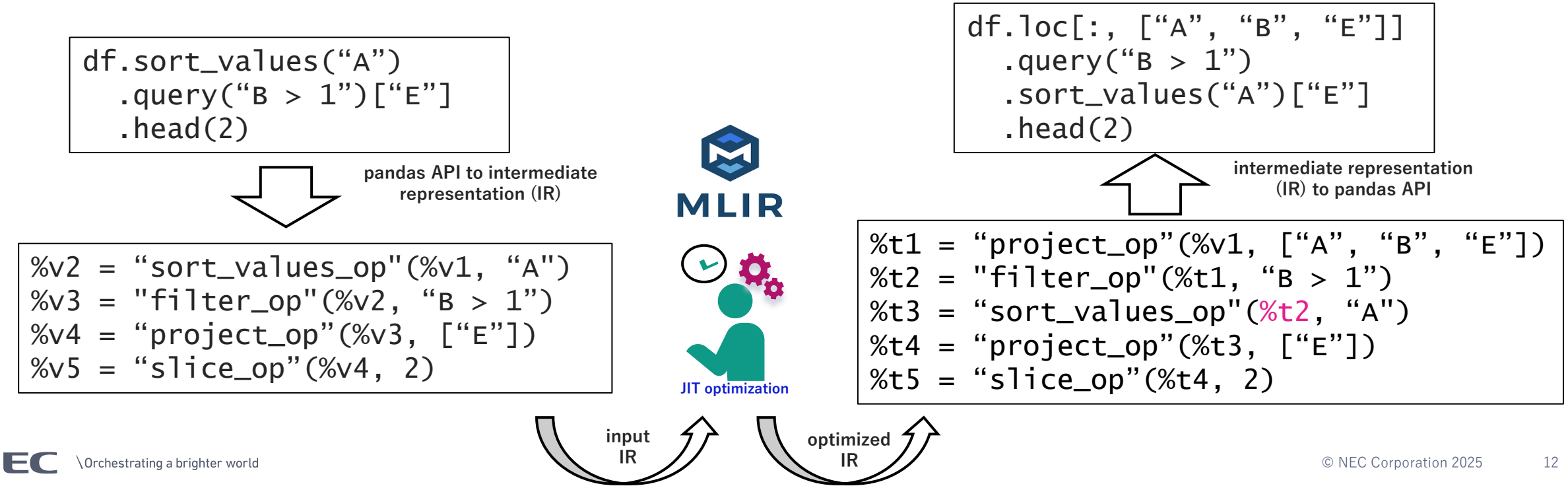
提案技術

アイデア＃1

- **ユーザプログラム全く変更せずに、このような最適化を自動化することは可能？**

👉 **以下のような提案通り実装検討した：**

1) LLVM・MLIRを用いて、「define-by-run」のメカニズムを実装し、各pandas APIから専用の中間言語（IR）を生成する。
2) 生成されたIRに対しては、ドメイン固有の様々な最適化（projection pushdown、predicate pushdown等）を適用する。
3) 最適化されたIRは、最適化されたpandasフローに逆変換する。



アイデア＃2

- **データフローの最適化だけでpandasの問題を解決可能？**
  - pandasは、シングルコア計算とメモリ効率の悪さが原因で低速です。
  - **解決するために、以下のような提案通り実装検討した:**
    1) DataFrame関連のメソッドを並列化する独自のライブラリをバックエンドとして開発する。
    2) 最適化されたIRを（pandas APIではなく）バックエンドライブラリのAPIに変換する。

```
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)
```

input IR

optimized IR

```
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);
```

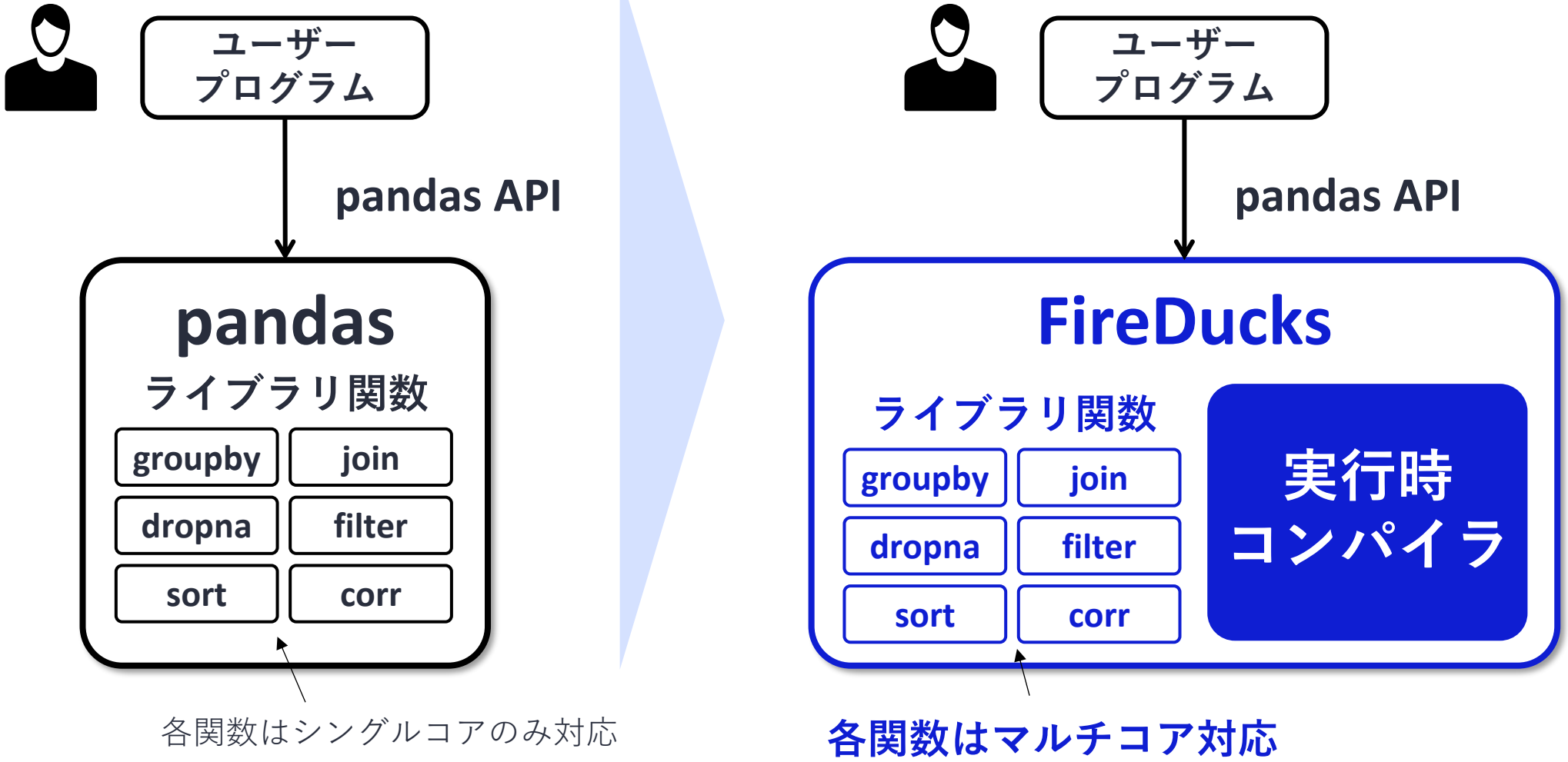
intermediate representation (IR) to parallelized backend API

```
%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)
```

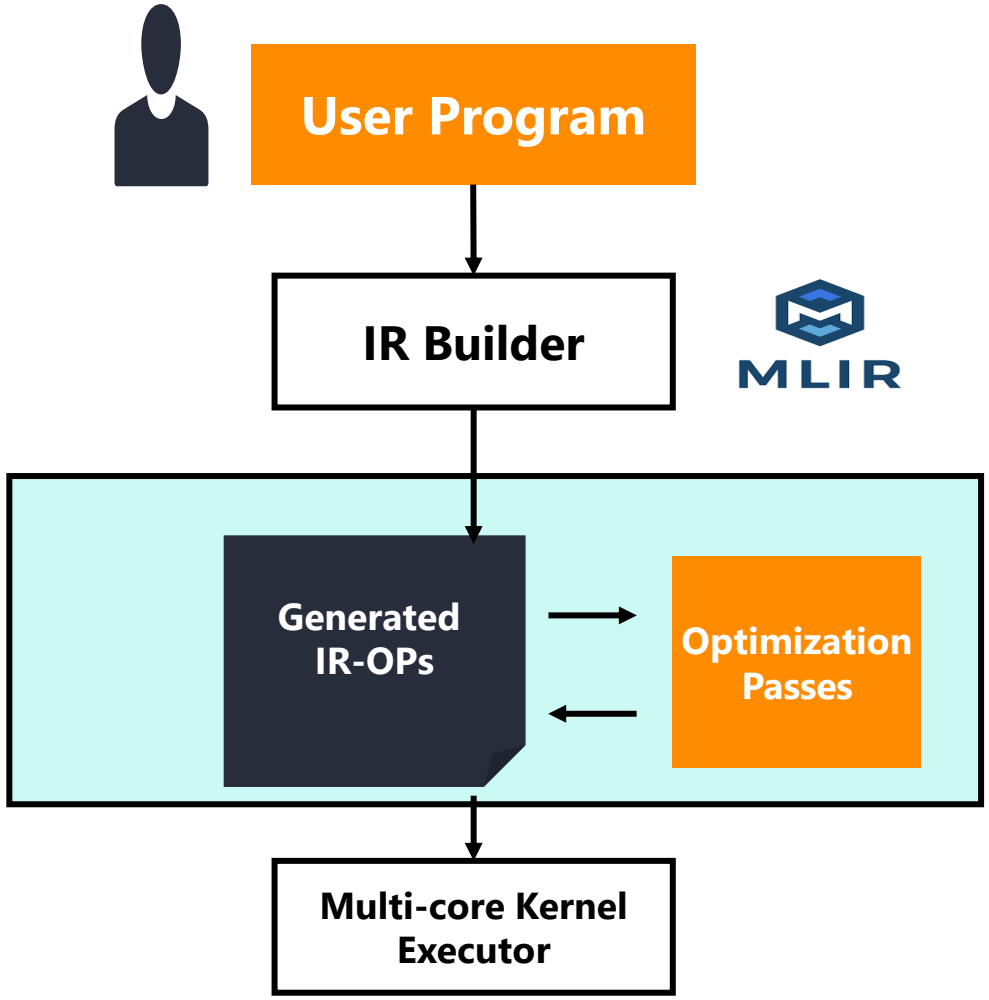
実装

FireDucks: 実行時コンパイラとマルチコアでpandasを高速化

ライブラリ中に実行時コンパイラを導入することで，APIを変えずに自動高速化



FireDucksはどう動くの？




```
sorted = df.sort_values("b")
result = sorted["a"]
```

↓

```
%v2 = "fireducks.sort_values"(%v1,"b")
%v3 = "fireducks.project"(%v2,["a"])
```

↓ **print (result)**

```
%v11 = "fireducks.project"(%v1,["a","b"])
%v2 = "fireducks.sort_values"(%v11,"b")
%v3 = "fireducks.project"(%v2,["a"])
```

↓

```
tmp = df[["a","b"]]
sorted = tmp.sort_values("b")
result = sorted["a"]
```

※IR: Intermediate Representation（中間言語）

自動最適化のその他の例: パターン最適化

特定のAPIの組み合わせを，より高速な書き方に変換します

```
df[~df["a"].isnull()]  # 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")
                                  # sort=Falseを追加
```

➡

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

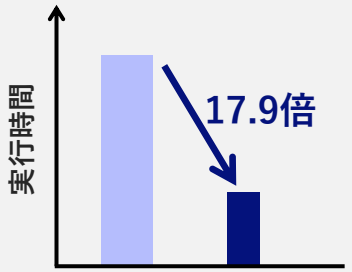
評価

FireDucksによる様々な金融データ分析での速度向上

pandas FireDucks

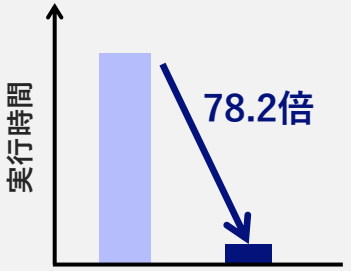
- FireDucks は、金融分析全般において、pandas と比較して圧倒的な高速性と効率性を発揮する.
- **最終出力には全く影響がない.**

リスク管理



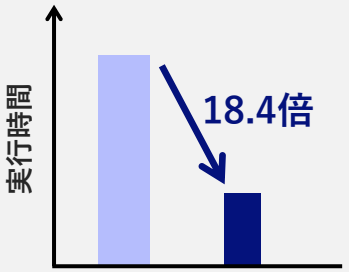
日次信用エクスポージャーの上限超過を検知する

AML



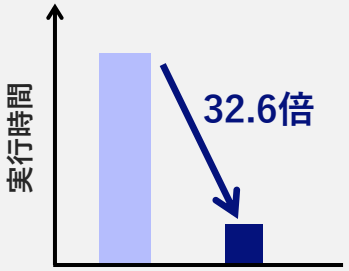
不審な取引の検出する

顧客ポートフォリオ分析



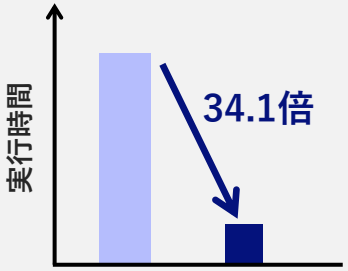
顧客セグメント別利益計算

損益計算



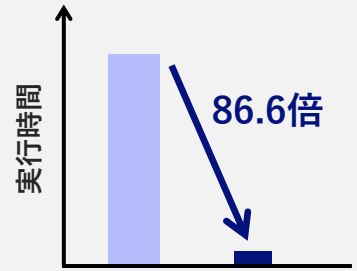
損益計算のための売買取引マッチング

市場データ分析



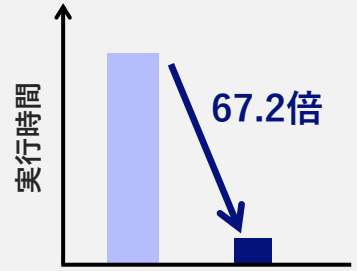
リターンとボラティリティの計算

高頻度取引分析



高頻度ティックデータのバー生成

不正分析



頻繁に返金関連取引の識別を行う顧客

Intel(R) Xeon(R) Gold 6526Y x2（合計32物理コア),
Memory: 512GB

競合ベンチーマーク

様々なライブラリ主要な処理（groupby, join）を比較するベンチーマークで最速

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	0.7.13	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)

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

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

Intel(R) Xeon(R) Platinum 8375C (128cores) 256GB

まとめ

FireDucksは，pandas互換の高速データフレームライブラリ

- コード変更なしで金融データ分析において高い有効性を確認
- 金融データの分析にとどまらず、単一ノード上での大規模データ処理を扱う領域においても実現可能
- データ分析効率の向上，クラウドコスト削減，サーバー資源活用など大きな価値を提供
- **今後は最適化パスの拡充と実運用での検証を継続**

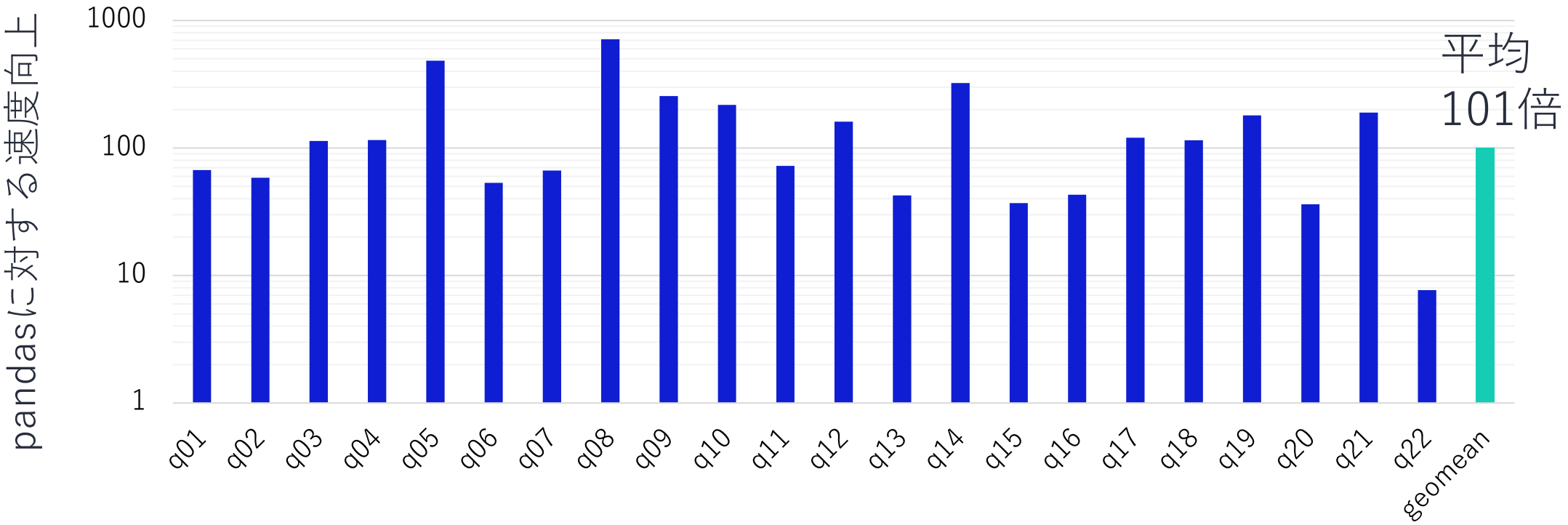
ご清聴ありがとうございました。

補足

FireDucksの性能（pandasに対する性能向上）

コード変更なくpandasに対して平均で101倍の性能向上

（TPC-Hベンチーマークに含まれる22種類のデータ分析処理の平均）

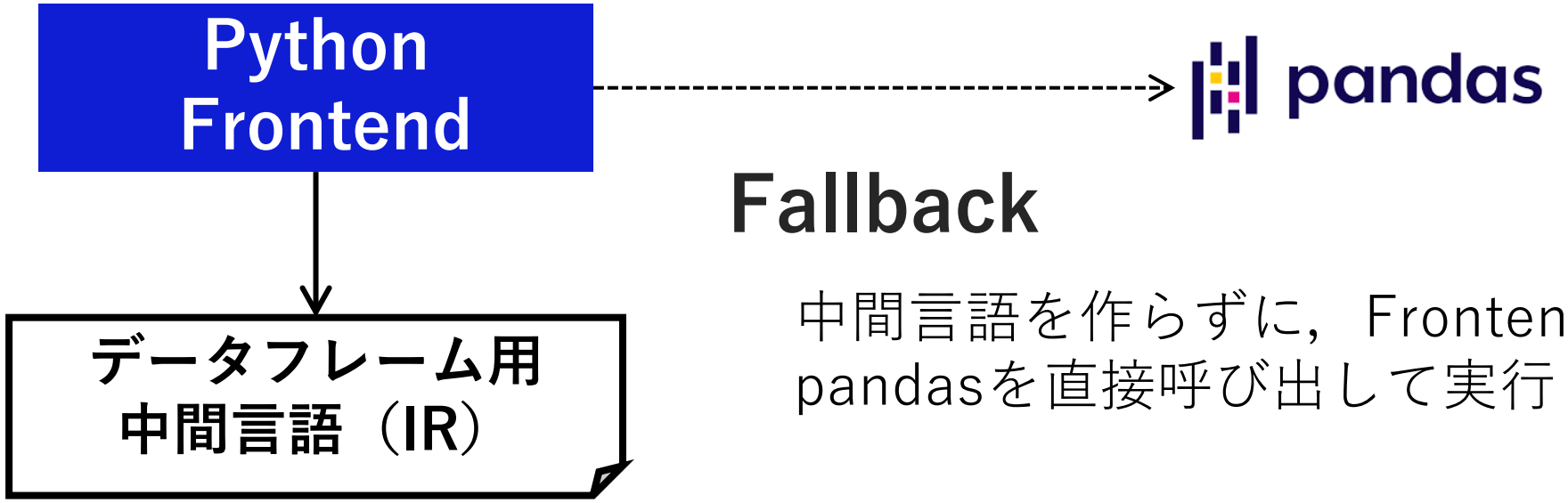


TPC-H:　データベースの評価によく使われるベンチマークの一つで，
売上集計やサプライチェーン管理関係の処理が含まれる

INTEL(R) XEON(R) GOLD 6526Y
32 cores, 512GB, Ubuntu 24.04,
pandas-2.3.3, io_type=skip

pandas互換性向上の仕組み: Fallback

FireDucksが未対応の機能はフロントエンドでpandasを呼び出すことで互換性向上



中間言語を作らずに，Frontendから
pandasを直接呼び出して実行

Fallbackの仕組みにより，

- 速度が必要な処理はFireDucks
- 互換性が必要な処理はpandas

と自動で切り替え，使い勝手を向上

FireDucksの利用方法

FireDucksの利用は非常に簡単.
コード変更不要なのでいつでもpandasに戻せる.

- **CPU: x86_64プロセッサ**
- **python: 3.9 ~ 3.13**
- **OS: Linux, Mac**

利用方法1: pandasの自動置き換え（ソースコード変更不要）

pythonコマンドオプションで指定

```
$ 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
```

黄色字部分を追加するだけで，コード中のpandasをFireDucksに自動変換

利用方法2: プログラムのimport文を書き換え（一行だけ変更）

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

import文を書き換え，pdの実体をFireDucksに変更

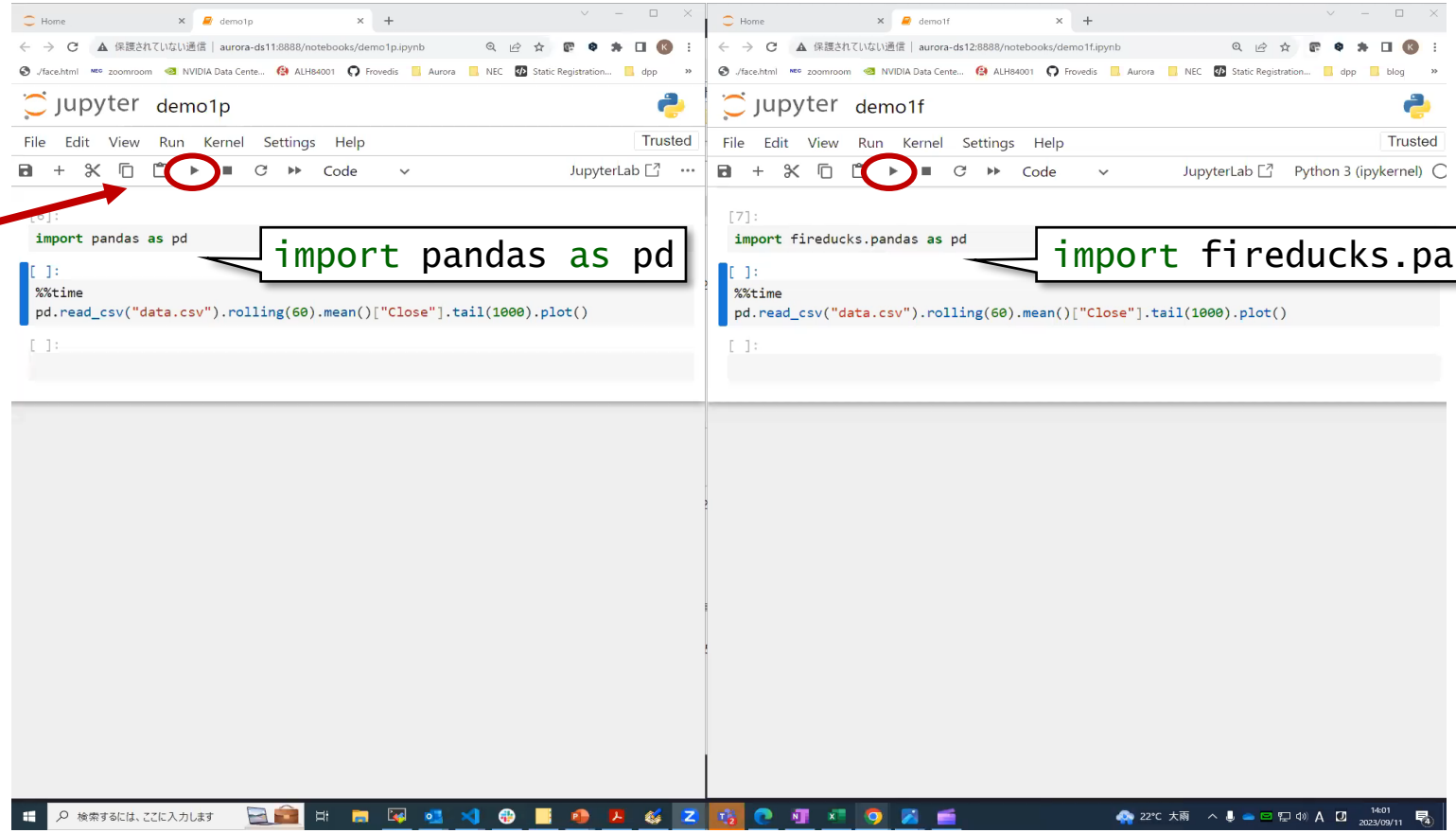
Let’s Have a Quick Demo!

```
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



data.csv:
[Bitcoin Historical Data]

pandas: 4.06s

↓ **~15x**

FireDucks: 275ms