

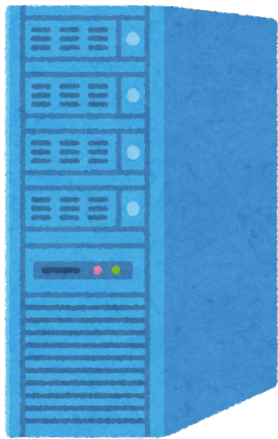
# 財務ビッグデータ解析のための データ形式及びツールに関する検討

宮本大輔

(東京大学情報理工学系研究科准教授)

# 概要

- 「あなたの共同研究者は巨大なCSVファイルを解析したい。あなたは、どのような方法を提案しますか」



Intel(R) Xeon(R) Silver 4314 CPU @ 2.40GHz x2  
物理計 32コア (論理64コア)  
メモリ 512GB / Swap 8GB



263,523,821行 x 88列  
116,250,751,535 bytes (116GBytes)

# pandas

- pandas (python data analysis library)
  - Pythonでデータ分析を行うためのライブラリ
  - 表形式のデータを効率的に操作できるデータフレーム構造を提供
  - データのクレンジング、変換、分析、可視化などの一連の処理が容易

```
% python3  
>>> import pandas as pd  
>>> df = pd.read_csv("data.csv")
```

# pandas / error #1

```
% python3
>>> import pandas as pd
>>> df = pd.read_csv("data.csv")
<stdin>:1: DtypeWarning: Columns
(4,5,7,8,9,10,11,17,19,20,21,22,78,80,81,83,84,85) have
mixed types. Specify dtype option on import or set
low_memory=False.
Killed
```

```
% python3
>>> import pandas as pd
>>> df = pd.read_csv("data.csv", low_memory=False)
Killed
```

# (FYI) Linux OOM Killer

- OOM Killer (Out of Memory Killer) とは
  - システムが本当にメモリ不足に陥った場合に、これを確認し、強制的に終了するプロセスを選択する OS の機能
  - CPU をさほど利用していない割に大量のメモリを使用しているプロセスが選択される傾向にある
    - badness が独自に計算される。計算式は以下の通り。
    - $\text{badness\_for\_task} = \text{total\_vm\_for\_task} / (\text{sqrt}(\text{cpu\_time\_in\_seconds}) * \text{sqrt}(\text{sqrt}(\text{cpu\_time\_in\_minutes})))$
- 特定のプロセスを OOM Killer の対象外とすることが可能
  - `/proc/[pid]/oom_adj` の値を変更する

# pandas / error #2

```
% python3
>>> import pandas as pd
>>> df = pd.read_csv("data.csv", low_memory=False)
```

```
% ps -auxw | grep python3
daisuke      71389  98.2  88.8  546091092
468870012 pts/0  Rl+  14:54  11:56 python3
% sudo echo -17 > /proc/71389/oom_adj
```

**client\_loop: send disconnect: Broken pipe**

おそらく高負荷を原因とし、ssh で接続していた端末が切断された

# pandas



データセット	データ読み込み	特定列の平均の計算
100行	0.006	0.000
1,000行	0.013	0.000
10,000行	0.096	0.000
100,000行	1.045	0.001
1,000,000行	10.973	0.005
10,000,000行	102.864	0.063
100,000,000行	1368.267	0.793
263,523,821行	(Killed)	-

(単位は秒, killedは途中終了、以降は計測不能)

D

numpy や pandas 以外に、データフレームを扱えるpythonのライブラリについておしえてください



Pythonには、numpyやpandas以外にも様々なデータフレームを扱うライブラリが存在します。以下にいくつか例を挙げます。

1. **Dask**: 大規模なデータセットを扱うためのライブラリで、pandasと互換性のあるAPIを持っています。Daskは大量のデータをメモリ外で処理する能力があり、並列処理や分散処理をサポートしています。したがって、大規模なデータセットに対する効率的な操作が可能です。
2. **Vaex**: 大規模なデータセットを効率的に扱うことができます。Vaexはデータをメモリに全て読み込む代わりに、一度に一部のデータだけを読み込むため、非常に大きなデータフレームも扱うことができます。また、Vaexは遅延評価を使用しており、必要になるまで実際の計算を遅らせることができます。
3. **Modin**: pandasのAPIと互換性を持ちながら、並列化と分散計算を実装することでpandasの計算速度を向上させます。Modinはpandasのコードをそのまま使用しつつ、バックエンドでの処理を最適化し、より高速なデータ操作を可能にします。
4. **Koalas**: Apache SparkのデータフレームAPIをpandas風のAPIで使えるようにするライブラリです。大規模データセットに対するデータ操作をSparkで行いたい場合、Koalasは非常に有用です。
5. **cuDF**: RAPIDS.aiプロジェクトの一部であり、GPUを利用してデータ操作を高速化します。cuDFはpandasと互換性のあるAPIを提供し、CPUに比べてGPUを用いることで大幅なパフォーマンス向上を可能にします。

これらのライブラリはそれぞれ異なるニーズに対応しており、使用するデータセットのサイズや必要な計算速度、利用可能なリソース（メモリ、CPU、GPUなど）に応じて選択することが推奨されます。



# Dask

- Dask

- 大規模なデータセットを扱うためのライブラリ
- pandas, numpy, Scikit-learn等のPythonの標準的なデータサイエンス用途に用いられるライブラリと互換性がある
- 遅延評価を行う

```
% python3
```

```
>>> import dask.dataframe as dd
```

```
>>> df = dd.read_csv("data.csv", low_memory=False)
```

```
>>>
```

計算結果が実際に必要とされるまでその計算を遅らせること（遅延評価とも）  
ファイルの読み込みは瞬時に終わったように見えるが、読み込みは現時点では行われていない

# Dask

- Dask
  - 平均の計算
    - 列 (sales) の平均を計算
  - データタイプの指定

```
result = df[column].mean().compute()
```

```
dtypes={'EBIT': 'float64',  
        'EBITDA': 'float64',  
        'PL_after_tax': 'float64',  
        'PL_before_tax': 'float64',  
        'PL_extr': 'float64',  
        'SIC_code2': 'float64',  
        ...
```

```
% python3  
>>> import dask.dataframe as dd  
>>> df = dd.read_csv("data.csv", low_memory=False, dtype=dtypes)  
>>> result = df[column].mean().compute()  
>>> print(result)
```

# Dask



データセット	データ読み込み	特定列の平均の計算
100行	0.009	0.204
1,000行	0.011	0.022
10,000行	0.009	0.107
100,000行	0.014	0.838
1,000,000行	0.027	10.173
10,000,000行	0.037	124.270
100,000,000行	0.081	1148.639
263,523,821行	0.092	2731.113

データセット	データ読み込み	特定列の平均の計算
pandas / 100,000,000行	1368.267	0.793

# Dask (並列計算)

- Dask

- Dask は単一のマシンのマルチ処理・クラスタ上の分散処理の機能を持つ

```
% python3
>>> import dask.dataframe as dd
>>> from dask.distributed import Client
>>> client = Client()
>>> df = dd.read_csv("data.csv", low_memory=False, dtype=dtypes)
>>> result = df[column].mean().compute()
```

# Dask (並列計算)

データセット	データ読み込み	特定列の平均の計算
100行	0.010	0.103
1,000行	0.010	0.066
10,000行	0.010	0.166
100,000行	0.018	1.250
1,000,000行	0.028	1.857
10,000,000行	0.038	16.578
100,000,000行	0.035	135.473
263,523,821行	0.178	322.303

データセット	データ読み込み	特定列の平均の計算
pandas / 100,000,000行	1368.267	0.793
Dask (単一) / 100,000,000行	0.081	1148.639

# Vaex

- Vaex
  - 同じくデータサイエンス用のライブラリ
  - 遅延実行を行う

```
% python3
>>> import vaex
>>> df = vaex.read_csv("data.csv", low_memory=False, dtype=dtypes)
>>> result = df[column].mean()
>>> print(result)
```

# Vaex



Vaex が CSV ファイルを読み込む際には、内部的に HDF5 / Apache Arrow 形式に変換される (この処理は遅延実行されない)

データセット	データ読み込み	特定列の平均の計算
100行	0.010	0.023
1,000行	0.020	0.002
10,000行	0.122	0.006
100,000行	1.200	0.024
1,000,000行	12.666	0.046
10,000,000行	121.208	0.025
100,000,000行	1835.651	1.715
263,523,821行	(Killed)	-

データセット	データ読み込み	特定列の平均の計算
Dask (並列) / 100,000,000行	0.035	135.473

# Modin

- Modin
  - 同様に pandas と互換性のあるライブラリ
  - 内部的に Dask / Ray 等を利用でき、Workerを起動して処理

```
% python3
>>> import modin.pandas as pd
>>> df = pd.read_csv("data.csv", low_memory=False, dtype=dtypes)
>>> result = df[column].mean()
>>> print(result)
```



# Modin



エラーで終了

- distributed.worker.memory – WARNING –  
Unmanaged memory use is high.

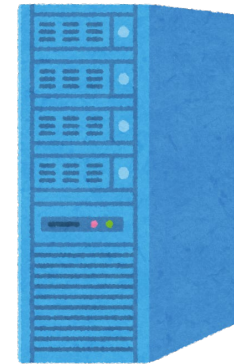
データセット	データ読み込み	特定列の平均の計算
100行	6.422	1.294
1,000行	5.361	1.367
10,000行	5.539	1.773
100,000行	5.594	1.642
1,000,000行	6.368	1.561
10,000,000行	16.848	1.406
100,000,000行	116.860	1.425
263,523,821行	(Error)	-

データセット	データ読み込み	特定列の平均の計算
Dask (並列) / 100,000,000行	0.035	135.473

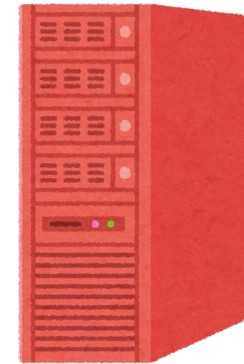
# cuDF

- cuDF

- NVIDIAの提供するRAPIDS群に含まれる GPU を用いたデータフレームを制御するライブラリ
- pandas と親和性がある



Xeon Silver 4314 CPU  
論理64 / メモリ 512GB



Xeon Silver 4214 CPU  
論理48 / メモリ 384GB  
Tesla V100-PCIE-16GB

```
% python3
>>> import dask_cudf
>>> df = dask_cudf.read_csv("data.csv", low_memory=False, dtype=dtypes)
>>> result = df[column].mean()
>>> print(result)
```

# cuDF



エラーで終了  
cudaErrorMemoryAllocation out of memory

データセット	データ読み込み	特定列の平均の計算
100行	1.094	0.436
1,000行	0.040	0.035
10,000行	0.043	0.051
100,000行	0.051	0.083
1,000,000行	0.061	0.542
10,000,000行	0.745	(Error)
100,000,000行	-	-
263,523,821行	-	-

データセット	データ読み込み	特定列の平均の計算
Dask (並列) / 1,000,000行	0.027	10.173



# R 標準の read.csv ()

- R の標準的なライブラリで読み込む例
  - 実際には Sys.time() を使って計測

```
filenames <- c("data.csv")

for (file in filenames) {
  data <- read.csv(file, header=TRUE, stringsAsFactors=FALSE)
  print(paste(file, "のsalesの平均 (欠損値除外):", mean(data$sales, na.rm = TRUE)))
}
```

# R 標準の read.csv()



データセット	データ読み込み	特定列の平均の計算
100行	0.006	0.000
1,000行	0.020	5.555
10,000行	0.207	0.000
100,000行	4.460	0.000
1,000,000行	34.461	0.000
10,000,000行	272.509	0.110
100,000,000行	2130.101	1.176
263,523,821行	(Killed)	-

データセット	データ読み込み	特定列の平均の計算
pandas / 100,000,000行	1368.267	0.793
Dask (単一) / 100,000,000行	0.081	1148.639



# readr + dplyr

- readr

- R のデータサイエンス用のパッケージ集 tidyverse で提供されているライブラリで CSV の読み込みなどに用いられる
- 標準の R 関数よりも高速にデータを読み込む

- dplyr

- 同じく tidyverse の提供するライブラリで、データフレームの操作を行う際に用いられる

```
library(readr)
library(dplyr)
data <- readr::read_csv(file)
avg_sales <- data %>%
  filter(!is.na(sales)) %>%
  summarize(average = mean(sales))
```

# readr + dplyr



データセット	データ読み込み	特定列の平均の計算
100行	0.288	0.015
1,000行	0.139	0.004
10,000行	0.171	0.006
100,000行	0.851	0.047
1,000,000行	8.509	0.712
10,000,000行	87.186	7.089
100,000,000行	995.52	61.111
263,523,821行	2588.786	104.997

データセット	データ読み込み	特定列の平均の計算
pandas / 100,000,000行	1368.267	0.793
Dask (単一) / 100,000,000行	0.081	1148.639



# data.table

- data.table
  - data.frame の高速な処理を行うパッケージ
  - 簡単な並列分散機能を保持しており、内部的に並列化される
  - data.frame を data.table に置換する操作も可能

```
library(data.table)
data <- fread(file)
avg <- data[, mean(sales, na.rm = TRUE)]
```



# data.table



データセット	データ読み込み	特定列の平均の計算
100行	0.004	0.000
1,000行	0.007	0.000
10,000行	0.031	0.000
100,000行	0.496	0.000
1,000,000行	6.626	0.001
10,000,000行	66.044	0.006
100,000,000行	718.05	0.051
263,523,821行	1683.199	0.187

データセット	データ読み込み	特定列の平均の計算
Dask (単一) / 100,000,000行	0.081	1148.639
readr + dplyr / 100,000,000行	995.52	61.111

# disk.frame



- disk.frame

- メモリ上の data.frame とは異なり、データをディスク上においたまままで操作するため、メモリを超えるデータを効率的に処理できる
- 巨大なファイルを個別のファイルとしてディスクに置き、並列的に処理できる
- 遅延評価を行う
- tidyverse の文法と互換性

```
library(disk.frame)
library(dplyr)
setup_disk.frame()
df <- csv_to_disk.frame(file)
avg_sales <- df %>%
  filter(!is.na(sales)) %>%
  summarise(avg_sales = mean(sales)) %>%
  collect()
```

# disk.frame



データセット	データ読み込み	特定列の平均の計算
100行	0.047	70.071
1,000行	0.036	55.982
10,000行	0.067	45.260
100,000行	0.924	45.146
1,000,000行	7.816	46.215
10,000,000行	80.148	77.139
100,000,000行	797.667	697.77
263,523,821行	2113.075	1621.114

データセット	データ読み込み	特定列の平均の計算
readr + dplyr / 100,000,000行	995.52	61.111
data.table / 100,000,000行	718.05	0.051

# CSV ではない形式なら？

- Arrow (Feather)
  - Apache Arrow プロジェクトが開発する  
列志向のデータフレームのフォーマット
- Parquet
  - Apache Arrow とは独立に開発された  
列志向のデータフレームのフォーマット
- 各種データベース
  - PostgreSQL (今年のワークショップ)



263,523,821行 x 88列  
116,250,751,535 bytes (116GBytes)

# Apache Arrow

- Arrow (Feather) ファイルの作成
  - R で作成した
- Arrow (Feather) ファイルの読み込み
  - 必要最低限の列だけを読み込みたい

(Python)

```
import pyarrow.feather as feather
import pandas as pd
df = feather.read_feather("data.feather", columns=['sales'])
avg_sales = df['sales'].mean()
```

(R)

```
library(arrow)
df <- read_feather("data-all.feather ", col_select = c("sales"))
average_sales <- mean(df$sales, na.rm = TRUE)
```

```
library(data.table)
library(arrow)
```

```
csv_file <- "data.csv"
data <- fread(csv_file)
```

```
feather_file <- "data.feather"
write_feather(data, feather_file)
```

44,562,020,514 (44.5GB)

# Apache Parquet

- Parquet ファイルの作成
- Parquet ファイルの読み込み

(python)

```
>>> import pyarrow.parquet as parquet
>>> import pandas as pd
>>> df = feather.read_feather("data.parquet", columns=['sales'])
>>> avg_sales = df['sales'].mean()
```

(R)

```
library(arrow)
df <- read_parquet("data-all.parquet", col_select = c("sales"))
average_sales <- mean(df$sales, na.rm = TRUE)
```

```
library(data.table)
library(arrow)
```

```
csv_file <- "data.csv"
data <- fread(csv_file)
```

```
parquet_file <- "data.parquet"
write_parquet(data, parquet)
```

10,484,097,359 (10.5GB)

ほぼArrowと変わらない操作で変換できる

# Arrow / Parquet



データセット (263,523,821行)	特定列のデータ読み込み	特定列の平均の計算
PyArrow / Arrow	5.684	1.565
PyArrow / Parquet	6.088	1.527
R / Arrow	6.346	0.527
R / Parquet	5.279	0.877

データセット (263,523,821行)	全てのデータ読み込み	特定列の平均の計算
Python / Dask (単一)	0.092	2731.113
Python / Dask (並列)	0.178	322.303
R / readr + dplyr	2588.786	104.997
R / data.table	1683.199	0.187
R / disk.frame	2113.075	1621.114

# (備考)

- そもそも CSV ファイルから列を指定して読み込めるのでは?
- Dask (python3) の場合
  - `read_csv("data.csv", dtype=dtypes, low_memory=False, usecols=['sales'])`
  - 読み込み 0.153秒 / 平均の計算 71.101秒
- data.table の場合
  - `fread("data.csv", select = "sales")`
  - 読み込み 22.372秒 / 平均の計算 1.455秒



# まとめ

- 「あなたの共同研究者は巨大なCSVファイルを解析したい。  
あなたは、どのような方法を提案しますか」
- 「Linux で普通に処理しようとする と OS がプロセスを  
停止しますね」
- 「Python3 なら Dask、R なら data.table でどうですか。  
メモリが足りないなら R の disk.frame もありますよ」
- 「解析手法にマッチするなら特定の列だけ読み込む方法もありますよ。  
その場合、CSV ではなく Arrow や Parquet 形式にすると、  
もっと早くなりますよ」