探索的ビッグデータ解析と再現可能研究 (WS-EBDA-RR-2022)

# ティックデータのフィルタリング: Daily TAQ データを例にして

小池祐太 (東京大学, CREST JST)

2022 年 8 月 28 日

東京大学大学院数理科学研究科, CREST JST

# 自己紹介

- 経歴
  - 2010 年: 東京工業大学 理学部数学科卒
  - 2012 年: 東京大学 大学院数理科学研究科修士課程修了 (指導教員: 吉田朋広教授)
  - 2014 年: 東京大学 大学院数理科学研究科博士課程中退
    (2015 年に同研究科で論文博士 (数理科学) を取得)
  - 統計数理研究所, 首都大学東京 (現: 東京都立大学) を経て, 2017 年 11 月より現職
- 研究テーマ: 確率過程に対する統計学, 金融高頻度データ解析, 高次元中心極限定理
- 今回の話題は, 2 番目の研究テーマに関連
  - プログラミングの素人が四苦八苦してどうにかティックデータを統計解析にかけられる状態にするまでの経験談を報告します

1. ティックデータとは
   - Daily TAQ データ
2. シェル上でのフィルタリング
   - 銘柄のフィルタリング
   - データクリーニング
   - Ver.2 の扱い
3. R 上でのデータ整形
4. まとめ

# ティックデータとは

# ティックデータとは

- 金融市場で取引される金融資産の価格や取引数, 注文数などの情報を, 約定や注文が発生するごとに記録したデータ
- Daily TAQ データ †
  - ニューヨーク証券取引所 (NYSE) が販売しているティックデータセットの 1 つ
  - 米国証券市場の主要取引所で取引されたすべての銘柄の最良気配と約定に関するデータを記録
  - データは決められたフォーマットのテキストファイルで提供される
  - ここでは, 提供されるデータのうちファイルサイズが最も大きい最良気配 (BBO) データのフィルタリングを中心に説明する

---

†詳細はウェブサイト `https://www.nyse.com/market-data/historical/daily-taq` を参照

# Daily TAQ データ

- 最良気配データ
  - 各銘柄ごとに, 最良気配に関する情報を注文発生ごとに記録したデータ
  - 最良気配とは, 市場に提示されている最高買値 (best bid) と最安売値 (best offer/best ask) のこと
  - 価格以外にも, タイムスタンプ, 提示されている数量, 注文が発生した取引所, . . . などの情報が記録されている

# Daily TAQ データ

- データのフォーマットはマニュアルに記載があり, 定期的に仕様変更がある (マニュアルはウェブサイトから入手可能)
- タイムスタンプのフォーマット変更に伴い, 過去に何度か大きな仕様変更があった
  (1) 2015 年 8 月 3 日: タイムスタンプの精度がミリ秒からマイクロ秒に変更されたことに伴う仕様変更
  (2) 2016 年 10 月 10 日: タイムスタンプの精度がマイクロ秒からナノ秒に変更されることに伴う仕様変更 ‡

‡取引所ごとに変更のタイミングが異なる. NASDAQ 上場銘柄は 2016 年 10 月 24 日, それ以外は 2017 年 9 月 18 日から変更

# Daily TAQ データ

- 便宜上, (1), (2) の仕様変更後のフォーマットをそれぞれ Ver.1, Ver.2 と呼ぶことにする
- ここでは講演者が実際に分析した Ver.1 のデータを主として扱うが, Ver.2 への対応についても軽く触れる
  - 後述するように, Ver.2 の方が扱いやすくなっている

# Daily TAQ データ

- Ver.1 のフォーマットでは, 取引日ごとに全銘柄について 26 項目の情報が記録されている
- 各項目は文字数が決められており, セパレータでは分割されていない
- ファイルサイズは日ごとにばらつきがあるが, たいていは 80GB から 150GB 程度
- 2015 年 8 月の場合, ファイルサイズの合計は約 2.3TB
- このままの状態で, 例えば R などの統計ソフトウェアに直接読み込むのは非効率 (不可能?) なので, まずはシェル上で最低限のフィルタリングを施してデータサイズを低減する [§]

[§]地道 (2018); 地道・阪 (2022) の用語では, この工程は「前処理」に対応する (はず)

# シェル上でのフィルタリング

# 銘柄のフィルタリング

- 通常は銘柄ごとに分析を行うので, まずは分析対象の銘柄に関するデータのみ抽出する
- この操作によってデータサイズを大幅に低減することもできるので, 後の作業のスピードアップにも役立つ
- Ver.1 のフォーマットでは, 銘柄情報は 14–29 列目 (16 文字分) に記載されており, 以下の形式を持つ:

$$\text{Root (6 文字)} + \text{Suffix (10 文字)}$$

  - 大雑把には, Root が銘柄のティッカーシンボル (を空白埋めして 6 文字にしたもの) に対応しているが, 諸事情でそうでないケースがある
  - 例えば複数の株式クラスがある場合がそうであり, Suffix はそのようなケースの区別に利用される

# 銘柄のフィルタリング

- 例: アップル社 (ティッカーシンボル: AAPL) の場合

$$\text{AAPL} + (\text{空白} \times 12)$$

- 「上の 16 文字が含まれている行を抽出する」という方針でフィルタリングを行う
- このためには `grep` コマンドか AWK が便利
  - AWK にはいくつか処理系があり, `awk -version` で確認できる
  - macOS のデフォルトでは `nawk` か `gawk` が処理系であることが多いよう
  - `mawk` という実装があり, とても速いため以下では主としてこれを使う

例: 2015 年 8 月 3 日の最良気配データ taqquote20150803 からアップル社のデータを抽出する (アップル社は特にデータ量が多いため, 「ワーストケース」と思える)

- grep を使う場合

```
sp=`printf "%12s"`
grep "AAPL${sp}" taqquote20150803 > aapl.txt
```

time コマンドで測った実行時間:

```
340.96s user 11.42s system 99% cpu 5:53.31 total
```

- mawk を使う場合

```
sp=`printf "%12s"`
mawk -v "symb=AAPL${sp}" '$0~symb' taqquote20150803 > aapl.txt
```

time コマンドで測った実行時間:

```
100.83s user 14.44s system 98% cpu 1:56.59 total
```

参考: mawk でなく gawk を使った場合

```
183.63s user 14.36s system 99% cpu 3:19.22 total
```

Perl でもできる:

```
perl -ne 'if(/AAPL {12}/){ print $_}' taqquote20150803 > aapl.txt
```

time コマンドで測った実行時間:

```
76.00s user 11.75s system 98% cpu 1:29.46 total
```

# 銘柄のフィルタリング

- 注意: Root だけで検索をかけてはいけない
  - 例 1: アジレント・テクノロジー社のティッカーシンボルは A で, Root も A $\rightarrow$ A だけで検索すると, 銘柄情報以外の項目にマッチするリスクがある
  - 例 2: アルファベット社は, 議決権あり・なしで A 株と C 株に分かれており, それぞれのティッカーシンボルは GOOGL, GOOG となっている
  - Daily TAQ データでは, GOOG の銘柄名は

    $$\text{GOOG} + (\text{空白} \times 12)$$

    であり, GOOGL の銘柄名は

    $$\text{GOOG} + (\text{空白} \times 2) + \text{L} + (\text{空白} \times 9)$$

    $\rightarrow$ GOOG ($+$ 空白 $\times$2) で検索をかけると, GOOGL も引っかかってしまう

# データクリーニング

- 次のステップとして, 「最低限の」データクリーニングを行う
- ティックデータはどの程度「クリーニング」すべきか？
  - 文献によって多少の差異があり, 完全には統一されていない
  - 分析目的や手法によっても変わってくる
- ここでは, 明らかに適用すべきというコンセンサスがあるものだけ適用し, それ以上のクリーニングは分析レベルで行うという方針をとる

# データクリーニング

- 具体的に実行すること
  (1) 取引所を 1 つだけに絞る
  (2) 気配値が 0 であるようなレコードを削除
  (3) Best Bid が Best Offer よりも小さいようなレコードを削除
  (4) (分析に必要な項目だけ残す)
- (1)–(3) は, Barndorff-Nielsen *et al.* (2009) で提案された P3, P2, Q2 にそれぞれ対応
- (2)–(3) は明らかにおかしいレコードを削除する操作
  - 実際にはそのようなレコードはわずかなので, この操作によるインパクトはほぼない

# データクリーニング

- (1) はなぜ必要か？
  - 米国証券市場では, 各取引所での約定・気配情報が Securities Information Processor (SIP) に集約されたのち, 市場全体での最良売買気配 (NBBO) が (一般の) 投資家に提供される
  - Daily TAQ データの「Time」の項目に記載されているタイムスタンプは, 各取引所で生じた約定・気配情報が SIP で処理された時刻に対応
  - 従って, SIP から地理的に離れた取引所の場合, 実際に注文が発生してから SIP へと情報を送信するのにかかるタイムラグがタイムスタンプに上乗せされている
  - タイムラグの大きさはマイクロ秒単位だが, ティックデータではこれが問題となる
  - 従って, 異なる取引所でのレコードを混ぜてしまうと, 観測の順序が逆転するリスクが発生してしまう


引用元:

https://www.nytimes.com/2013/05/14/technology/north-jersey-data-center-industry-blurs-utility-real-estate-boundaries.html

# データクリーニング

- (1) はなぜ必要か？（続き）
  - 実は, Daily TAQ データには,「Participant Timestamp」という項目があり, 取引所から SIP へと情報が送信された時刻が記録されている
  - しかし, 異なる取引所間では「時計」にわずかなズレがある可能性があり, マイクロ秒の世界ではこのズレが問題となりうる (clock synchronization の問題)
  - 各取引所は協定世界時 (UTC) からの時計のズレを 0.1ms 以内に抑えることが法的に要求されているが, 微小なズレは存在しうる ¶

¶実際のズレは平均 36 マイクロ秒程度らしい. (Hasbrouck, 2021, page 12) 参照

# データクリーニング: セパレータの挿入

- (1)–(4) はどう実行する? → AWK を使えばよい (Perl でもできる?)
- AWK を使うには, 各項目がセパレータで分割されていると便利なので, セパレータを挿入して CSV ファイルを作成することにする (データサイズを増やすので, 本当はあまりよくないかもしれない)
  - 実は, Ver.2 では項目ごとにセパレータが挿入される仕様変更が行われたので, この操作は不要
- セパレータを挿入するには, sed か Perl を使うのが便利
- どちらが速いかは作業量次第のよう: 私の手元の環境では,
  - 作業量少 → sed が Perl より速かった
  - 作業量多 → Perl が sed より速かった

# データクリーニング: セパレータの挿入

- 例: 先ほど作成した `aapl.txt` にセパレータを挿入して CSV ファイルに変換
- 文字列の置換は, `sed`, Perl ともに以下の構文で実行できる

```
s/置換元文字列/置換後文字列
```

- 文字列には正規表現が利用できる (`sed` と Perl でわずかに書き方に違いがある)

# データクリーニング: セパレータの挿入

- 例えば, 前から 12 列目にコンマ , を挿入するには,
  - sed の場合

    ```
    sed -e "s/^\(.\{12\}\)/\1,/" aapl.txt > aapl.csv
    ```

    time コマンドで測った実行時間:

    ```
    1.70s user 0.40s system 98% cpu 2.133 total
    ```

  - Perl の場合

    ```
    perl -pe "s/^(.{12})/\1,/" aapl.txt > aapl.csv
    ```

    time コマンドで測った実行時間:

    ```
    3.66s user 0.29s system 98% cpu 4.002 total
    ```

- 実行したいコードが複数ある場合, ファイルに記載しておいて実行するのが便利

# データクリーニング: セパレータの挿入

sed の場合:

```
s/^\(.\{12\}\)/\1,/
s/^\(.\{14\}\)/\1,/
s/^\(.\{31\}\)/\1,/
```

と記載されたファイル taqquote.sed を用意しておいて,

```
sed -f taqquote.sed aapl.txt > aapl.csv
```

を実行すれば, 元ファイルの 12, 13, 29 列目にコンマが挿入される

# データクリーニング: セパレータの挿入

Perl の場合:

```
while(<>){
    s/^(.{12})/\1,/;
    s/^(.{14})/\1,/;
    s/^(.{31})/\1,/;
    print ;
}
```

と記載されたファイル `taqquote.pl` を用意しておいて,

```
perl taqquote.pl aapl.txt > aapl.csv
```

を実行すれば, 元ファイルの 12, 13, 29 列目にコンマが挿入される

# データクリーニング: セパレータの挿入

- 実際には 26 項目あるので, それらの間すべてにセパレータを挿入する
  - `sed` の場合の実行時間:

    ```
    145.64s user 1.05s system 99% cpu 2:28.07 total
    ```

  - Perl の場合の実行時間:

    ```
    87.35s user 0.98s system 99% cpu 1:28.66 total
    ```

# データクリーニング: AWK によるフィルタリング

- AWK では, セパレータで区切られた項目はフィールドと呼ばれる
- `i` 番目のフィールドは`$i` で参照できる
- 例: NASDAQ 取引所のデータのみに絞り込む

```
mawk -F , '$2~"[QT]"' aapl.csv > out.csv
```

  - `-F ,`: セパレータにコンマを指定
  - `$2~"[QT]"`: ２番目のフィールドが Q か T の行を抽出 (2 番目の列が取引所の項目で, Q と T が NASDAQ を表すため)
- 前と同様に, 実行したいコードが複数ある場合, ファイルに記載しておいて実行するのが便利

# データクリーニング: AWK によるフィルタリング

```
BEGIN {FS=","; OFS=","}
$2 ~ EXC && # select exchange
$4 !~ /00000000000/ && # delete zero bid prices
$6 !~ /00000000000/ && # delete zero ask prices
$4 < $6 # delete non-positive spreads
```

と記載されたファイル `cleaning.awk` を用意しておいて,

```
mawk -f cleaning.awk EXC="[QT]" aapl.csv > out.csv
```

を実行すれば, 「NASDAQ 取引所のデータのみに絞り込んで, (2) と (3) の処理を行う」という操作が行われる (絞り込む取引所は変数 `EXC` としてコードに組み込んでいる)

# データクリーニング: AWK によるフィルタリング

- (4) の操作は以下のコマンドで実行できる:

```
mawk 'BEGIN{FS=",";OFS=","}{print $1,$2,$4,$5,$6,$7,$8,
    $10,$11,$24}' out.csv > out2.csv
```

  (1, 2, 4–8, 10, 11, 24 番目のフィールドのみ残す)

- また, タイムスタンプが HHMMSSxxxxxx という形式で, このままだと R で処理する際に不便なので, HH, MM, SS, xxxxxx をコンマでセパレートする
- タイムスタンプは通常の「Time」と「Participant Timestamp」の 2 つの項目があり, 上の操作の結果, それぞれファイルの最初と最後の項目に対応することに注意すると, これは sed か Perl で実行できる

sed の場合,

```
s/^\(.\{2\}\)/\1,/
s/^\(.\{5\}\)/\1,/
s/^\(.\{8\}\)/\1,/
s/\(.\{6\}\)$/,\1/
s/\(.\{9\}\)$/,\1/
s/\(.\{12\}\)$/,\1/
```

と記載されたファイル time.sed を用意しておいて,

```
sed -f time.sed out2.csv > out3.csv
```

を実行すればよい

Perl の場合,

```
while(<>){
    s/^(.{2})/\1,/;
    s/^(.{5})/\1,/;
    s/^(.{8})/\1,/;
    s/(.{6})$/,\1/;
    s/(.{9})$/,\1/;
    s/(.{12})$/,\1/;
    print ;
}
```

と記載されたファイル `time.pl` を用意しておいて,

```
perl time.pl out2.csv > out3.csv
```

を実行すればよい

以上の操作は, パイプで繋いで一気にやってしまった方がファイルをたくさん作らず済むのでよい

- sed の場合

```
mawk -f cleaning.awk EXC="[QT]" aapl.csv | mawk 'BEGIN{FS=",";
    OFS=","}{print $1,$2,$4,$5,$6,$7,$8,$10,$11,$24}' | sed -f
     time.sed > aapl-clean.csv
```

time コマンドで測った実行時間:

```
11.62s user 0.09s system 99% cpu 11.775 total
```

- Perl の場合

```
mawk -f cleaning.awk EXC="[QT]" aapl.csv | mawk 'BEGIN{FS=",";
    OFS=","}{print $1,$2,$4,$5,$6,$7,$8,$10,$11,$24}' | perl
    time.pl > aapl-clean.csv
```

time コマンドで測った実行時間:

```
3.07s user 0.05s system 88% cpu 3.533 total
```

# Ver.2 の扱い

- Ver.2 では, 最良気配データは 1 つのファイルに全銘柄が記録されているのではなく, 銘柄名の頭文字ごとにさらに 26 分割されている (おそらくファイルサイズがあまりに大きくなったため)
- 例えば, 2017 年 1 月 3 日のアップル社の最良気配データは `SPLITS_US_ALL_BBO_A_20170103` というファイルに記録されている
- ファイルごとのサイズは当然小さくなっているので (上のケースでは 3.91GB), 銘柄のフィルタリングの計算負荷は減る
- また, 項目数は 23 個となり, 各項目は `|` で区切られるている

# Ver.2 の扱い

- 銘柄情報は 3 番目の項目に記載されており, 以下の形式を持つ:

$$\text{Root (最大 6 文字)} + \text{空白} + \text{Suffix (10 文字)}$$

- Ver.1 と異なり, 空白埋めは行われない
- 例
  - アップル社: AAPL
  - アルファベット社 C 株: GOOG
  - アルファベット社 A 株: GOOG L
- 従って, 「項目 3 が特定の文字列と完全一致する」という方針でフィルタリングを行う必要がある $\rightarrow$ AWK を使えばよい

# Ver.2 の扱い

例: 2017 年 1 月 3 日の最良気配データからアップル社のデータを抽出する

```
mawk -F "|" '$3 == "AAPL"' SPLITS_US_ALL_BBO_A_20170103 >
    aapl2.txt
```

`time` コマンドで測った実行時間:

```
21.32s user 0.78s system 99% cpu 22.240 total
```

# Ver.2 の扱い

データクリーニングのステップでは, すでにセパレータが挿入されているので, AWK による処理だけ行えばよい:

```
BEGIN {FS="|"; OFS="|"}
$2 ~ EXC && # select exchange
$4 != 0 && # delete zero bid prices
$6 != 0 && # delete zero ask prices
$4 < $6 # delete non-positive spreads
```

と記載されたファイル `cleaning2.awk` を用意する

(1)–(4) の実行 (NASDAQ 取引所のデータのみ):

```
mawk -f cleaning2.awk EXC="[QT]" aapl2.txt | mawk 'BEGIN{FS
    ="|";OFS=","}{print $1,$2,$4,$5,$6,$7,$8,$13,$20}' | perl
     time.pl > aapl-clean2.csv
```

`time` コマンドで測った実行時間:

```
1.90s user 0.02s system 98% cpu 1.962 total
```

# R 上でのデータ整形

# R 上でのデータ整形

- 銘柄にもよるが, ここまでの操作でデータサイズは数十から数百 MB 程度になり, R でも「工夫すれば」ストレスなく扱える
  → `data.frame` ではなく `data.table` として扱う
- `data.table` とは？
  - パッケージ `data.table` で定義されている S3 クラス
  - `data.frame` を継承している
  - 特に, `data.frame` に適用できる関数はそのまま適用できる (例えばパッケージ `dplyr` の関数群など)
- 最大の特徴: 基本的に, `data.table` の変更の際に新たにオブジェクトを生成しない (shallow copy しかしない)‖
  → 巨大なオブジェクトの生成は時間がかかるので, 非常に有用

‖講演者の理解が正しければ

# R 上でのデータ整形

ファイルの読み込みにはパッケージ `data.table` の関数 `fread()` を用いる

```
(x <- fread("aapl-clean.csv"))
```

```
        V1 V2 V3     V4 V5      V6 V7      V8 V9 V10 V11 V12 V13 V14 V15    V16
     1:  4  0  1 904759  T   10000  1 1325000  4   R   T   T   4   0   1 904521
     2:  4  0 24 215904  T   10000  1 1325000  4   R   T   T   4   0  24 215731
     3:  4  1 37  81958  T 1213200  2 1325000  4   R   T   T   4   1  37  81745
     4:  4  2 21 331963  T 1213200  2 1214500  1   R   T   T   4   2  21 331791
     5:  4  2 21 944054  T 1213200  2 1214500  2   R   T   T   4   2  21 943827
    ---
746691: 20  0  0  86463  T 1173800  5 1178500 32   R   T   T  20   0   0  86180
746692: 20  0  0  97073  T 1171200  2 1178500 32   R   T   T  20   0   0  96920
746693: 20  0  0  97146  T 1170000  1 1178500 32   R   T   T  20   0   0  96992
746694: 20  0  0  97655  T 1170000  1 1178800  4   R   T   T  20   0   0  97376
746695: 20  0  0  98610  T 1170000  1 1207700 10   R   T   T  20   0   0  98427
```

## 速度比較

- `fread()`

```
system.time(x <- fread("aapl-clean.csv"))
```

```
ユーザ　　システム　　経過
     0.186       0.022       0.366
```

- `read.csv()`

```
system.time(x <- read.csv("aapl-clean.csv", header = FALSE))
```

```
 ユーザ　　システム　　経過
     2.186       0.042       2.230
```

- `read_csv()` (パッケージ `readr`)

```
system.time(x <- read_csv("aapl-clean.csv", col_names = FALSE))
```

```
 ユーザ　　システム　　経過
     1.392       0.027       0.803
```

# R 上でのデータ整形

- ここからの整形は分析目的によって異なる **
- ここでは以下の操作を行なって, 対数仲値と対数マイクロプライスの時系列を作成する
  (1) 0 時からの経過秒でタイムスタンプを作成する (SIP のタイムスタンプを使う)
  (2) 仲値とマイクロプライスを計算する
  (3) データを取引時間内に絞る
  (4) タイムスタンプが重複しているデータは最後のものだけ残す

---

**地道 (2018); 地道・阪 (2022) の用語では, この工程は「データラングリング」に対応する (はず)

# R 上でのデータ整形

- (3) と (4) はそれぞれ Barndorff-Nielsen *et al.* (2009) の P1 と Q1 に対応
  - (4) では「メディアンをとる」「数量で重みつけた平均をとる」などいくつか流儀がある
  - タイムスタンプの精度がマイクロ秒のデータでは重複は稀であり, (4) の操作はほぼインパクトがない (かつては大きな問題であった)

# R 上でのデータ整形

(1) と (2) の処理はただのベクトル計算

```
## タイムインデックスの作成
Time <- round(3600 * x$V1 + 60 * x$V2 + x$V3 + x$V4 * 1e-6,
              digits = 6)

## ビッド・アスク価格と枚数の取得
## 整数の桁溢れを回避するためにnumericに変換しておく
b <- as.numeric(x$V6) # ベストビッド
vb <- as.numeric(x$V7) # ベストビッドの枚数
a <- as.numeric(x$V8) # ベストアスク
va <- as.numeric(x$V9) # ベストアスクでの枚数

## 仲値とマイクロプライスの計算
mid <- (a + b)/2 # 仲値
micro <- (vb * a + va * b)/(va + vb) # マイクロプライス
```

# R 上でのデータ整形

(3) と (4) の処理が肝要

```
library(magrittr) # パイプ用

y <- data.table(time = Time, logmid = log(mid),
                    logmicro = log(micro)) %>%
    .[34200 <= time & time <= 57600] %>%
    .[ , tail(.SD, n= 1), by = time]

# 上の処理の説明
# %>%はパイプ演算子．連続した処理に使用
# 1つ目の処理: data.tableオブジェクトの作成
# 2つ目の処理: 取引時刻(9時半から16時まで)のデータを抽出
# 3つ目の処理: timeが同一のデータの最後尾のデータを抽出
# .SDの意味はhelp("special-symbols")参照
# (help(data.table)のExamplesを見た方が早いかも)
```

# R 上でのデータ整形

実行時間

```
system.time(
y <- data.table(time = Time, logmid = log(mid),
                logmicro = log(micro)) %>%
    .[34200 <= time & time <= 57600] %>%
    .[ , tail(.SD, n = 1), by = time]
)
```

```
ユーザ    システム        経過
     0.044       0.007       0.050
```

# R 上でのデータ整形

パッケージ dplyr を使っても同じ操作ができるが, 実行時間が大幅に遅くなる:

```
library(dplyr)
system.time(
  z <- data.table(time = Time, logmid = log(mid),
                  logmicro = log(micro)) %>%
    filter(34200 <= time & time <= 57600) %>%
    group_by(time) %>%
    summarise(logmid = tail(logmid, n = 1) , logmicro = tail(
    logmicro, n = 1))
)
```

```
ユーザ    システム        経過
     6.267      0.110      6.392
```

# R 上でのデータ整形

- 書き方の問題でなければ, これは dplyr が処理時に新たなオブジェクトを生成している (deep copy している) ことが原因と思われる
- この問題はパッケージ dtplyr をロードすることで解消できる

```
library(dtplyr)
system.time(
  z <- data.table(time = Time, logmid = log(mid),
                  logmicro = log(micro)) %>%
    filter(34200 <= time & time <= 57600) %>%
    group_by(time) %>%
    summarise(logmid = tail(logmid, n = 1) , logmicro = tail(
    logmicro, n = 1))
)
```

```
ユーザ     システム         経過
     0.010       0.002       0.014
```

# R 上でのデータ整形

- 注意: 上のコードで最も時間を食っているのは `data.table` を作成する箇所だと思われる
- これを回避するには, 最初に読み込んだ `x` を直接編集していけばよい (コードが長くなるのでやっていないが)

# まとめ

# まとめ

R で大規模データを扱うには, `data.table` と `dtplyr` を使うとよい

今日話せなかった課題

- Daily TAQ データを扱う上での最大の困難は, データのダウンロード
- データの提供方法は過去に何度も変遷してきた

$$\text{FTP} \to \text{MFT} \to \text{MFT/AWS}$$

- いまは AWS (Amazon Web Services) に苦戦しています

# コンピュータ環境

ハードウェアのローカル環境

- Macine: iMac Pro (2017)
  OS: macOS Big Sur (11.6)
  CPU: 2.3 GHz 18 コア Intel Xeron W
  メモリ: 64GB
- Macine: MacBook Air (M1, 2020)
  OS: macOS Big Sur (11.6)
  CPU: Apple M1 チップ, 8 コア
  メモリ: 16GB

# コンピュータ環境: R の sessionInfo() の出力

```
R version 4.2.0 (2022-04-22)
Platform: aarch64-apple-darwin20 (64-bit)
Running under: macOS Big Sur 11.6.7

Matrix products: default
LAPACK: /Library/Frameworks/R.framework/Versions/4.2-arm64/Resources/lib/
libRlapack.dylib

locale:
[1] ja_JP.UTF-8/ja_JP.UTF-8/ja_JP.UTF-8/C/ja_JP.UTF-8/ja_JP.UTF-8

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
[1] dtplyr_1.2.2      dplyr_1.0.9       magrittr_2.0.3    data.table_1.14.2

loaded via a namespace (and not attached):
 [1] fansi_1.0.3      utf8_1.2.2       crayon_1.5.0     R6_2.5.1
 [5] lifecycle_1.0.1  pillar_1.7.0     rlang_1.0.2      cli_3.2.0
 [9] rstudioapi_0.13  vctrs_0.4.1      generics_0.1.2   ellipsis_0.3.2
[13] tools_4.2.0      glue_1.6.2       purrr_0.3.4      compiler_4.2.0
[17] pkgconfig_2.0.3  tidyselect_1.1.2 tibble_3.1.8
```

# References


[1] Barndorff-Nielsen, Ole E., Peter Reinhard Hansen, Asger Lunde, & Neil Shephard, 'Realized kernels in practice: trades and quotes.' *Econom. J.*, **12**, pp. C1–C32, *2009*.

[2] Hasbrouck, Joel, 'Price Discovery in High Resolution.' *Journal of Financial Econometrics*, **19** (3), pp. 395–430, *2021*.

[3] 地道 正行,「探索的財務ビッグデータ解析：前処理, データラ ングリング, 再現可能性」,『商学論究』, **66**, 1–31 頁, 2018.

[4] 地道 正行・阪　智香,「探索的財務ビッグデータ解析と再現可能研究：非上場企業のデータラングリング」,『商学論究』, **69**, 83–120 頁, 2022.