Rolling statistics

what is it, implementations, types, performance

Rolling statistics

- Rolling window
- Rolling mean / moving average / SMA
  - Basic implementation
  - Online implementation
  - Basic vs on-line benchmark
  - EMA - not a rolling statistic
- Rolling minimum
- Rolling holistic aggregates
- Benchmarks, benchmarks, benchmarks

Rolling window: size 3

i = 1	
date	**price**
20/3	11
21/3	7
22/3	9
23/3	8
24/3	10
25/3	9

i = 2	
date	**price**
20/3	11
21/3	7
22/3	9
23/3	8
24/3	10
25/3	9

i = 3	
date	**price**
20/3	11
21/3	7
22/3	9
23/3	8
24/3	10
25/3	9

i = 4	
date	**price**
20/3	11
21/3	7
22/3	9
23/3	8
24/3	10
25/3	9

i = 5	
date	**price**
20/3	11
21/3	7
22/3	9
23/3	8
24/3	10
25/3	9

i = 6	
date	**price**
20/3	11
21/3	7
22/3	9
23/3	8
24/3	10
25/3	9

Rolling mean of window size 3

i = 1		
date	price	mean
20/3	11	NA
21/3	7	
22/3	9	
23/3	8	
24/3	10	
25/3	9	

i = 2		
date	price	mean
20/3	11	NA
21/3	7	NA
22/3	9	
23/3	8	
24/3	10	
25/3	9	

i = 3		
date	price	mean
20/3	11	NA
21/3	7	NA
22/3	9	9
23/3	8	
24/3	10	
25/3	9	

i = 4		
date	price	mean
20/3	11	NA
21/3	7	NA
22/3	9	9
23/3	8	8
24/3	10	
25/3	9	

i = 5		
date	price	mean
20/3	11	NA
21/3	7	NA
22/3	9	9
23/3	8	8
24/3	10	9
25/3	9	

i = 6		
date	price	mean
20/3	11	NA
21/3	7	NA
22/3	9	9
23/3	8	8
24/3	10	9
25/3	9	9

Any ideas for an efficient implementation?

Moving Average (SMA: simple moving average)

Most commonly used rolling statistic, often referred as SMA.

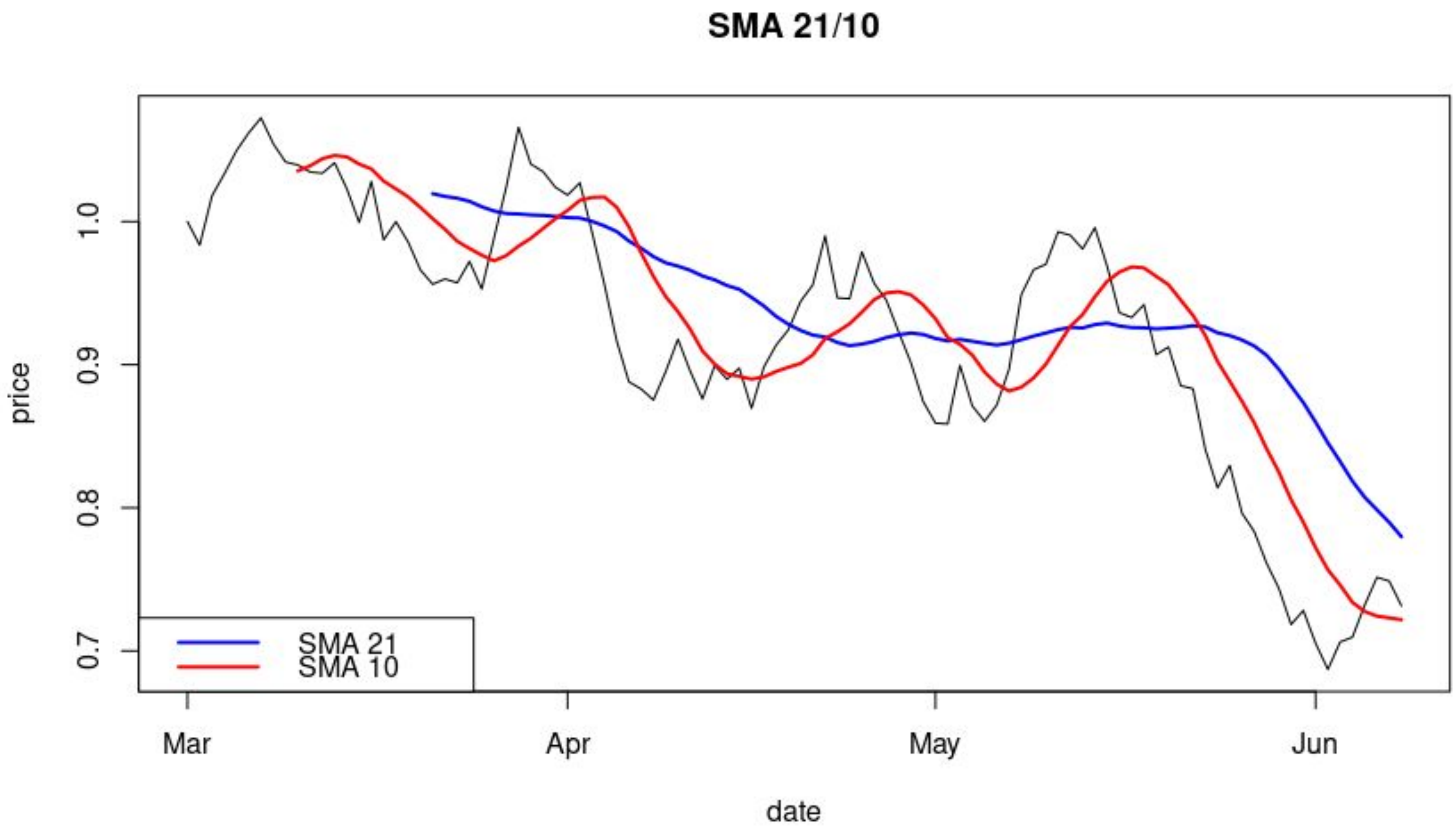
One can often see SMA 200 or SMA 50. The number defines how many periods are being included in a rolling window.

SMA 200 - moving average of last 200 days/hours/etc. Takes last N observations and computes its mean.

```
set.seed(432)
n = 100
df = data.frame(
  date = as.Date("2021-03-01")+0:(n-1),
  price = cumprod(c(1, rnorm(n-1, 1, 0.025)))
)
```

SMA 21/10

```
head(df)
#         date     price
#1 2021-03-01 1.0000000
#2 2021-03-02 0.9835612
#3 2021-03-03 1.0179743
#4 2021-03-04 1.0332789
#5 2021-03-05 1.0496233
#6 2021-03-06 1.0620643

with(df, plot(date, price, type="l", main="SMA 21/10"))
with(df, lines(date, sma(price, 21), col="blue", lwd=2))
with(df, lines(date, sma(price, 10), col="red", lwd=2))
legend("bottomleft", c("SMA 21","SMA 10"),
  col=c("blue","red"), lty=1, lwd=2)
```


SMA 21/10
price
1.0
0.9
0.8
0.7
date
Mar
Apr
May
Jun
SMA 21
SMA 10

Basic moving average implementation

```
sma = function(x, n) {
  ans = rep(NA_real_, nx<-length(x))
  for (i in n:nx) {
    w = 0
    for (j in (n-1):0) {
      w = w + x[i-j]
    }
    ans[i] = w / n
  }
  ans
}
## in R we normally would use 'mean' to push inner loop to C
## here the aim is to demonstrate the algorithm
```

Online moving average implementation

```
fsma = function(x, n) {
  ans = vector("double", nx<-length(x))
  w = 0
  for (i in 1:(n-1)) {     ## i < n
    w = w + x[i]
    ans[i] = NA_real_
  }
  w = w + x[n]             ## i == n
  ans[n] = w / n
  for (i in (n+1):nx) {    ## i > n
    w = w - x[i-n]
    w = w + x[i]
    ans[i] = w / n
  }
  ans
}
```

Basic vs online implementation benchmark

```
x = rnorm(1e5)

system.time(a1 <- sma(x, 200))
#    user  system elapsed
#   1.062   0.006   1.067

system.time(a2 <- fsma(x, 200))
#    user  system elapsed
#   0.019   0.000   0.019

all.equal(a1, a2)
#[1] TRUE
```

EMA: exponential moving average

EMA reacts faster than SMA by giving more weight to the recent observations than the past ones. EMA it is not a rolling statistic despite often being classified as such. Classification comes from the use case rather than from the exact definition. EMA is a cumulative statistic as it requires complete history and cannot be computed on a rolling window.

```
library(TTR)
with(df, lines(date, EMA(price, 21), col="blue", lwd=2, lty=2))
with(df, lines(date, EMA(price, 10), col="red", lwd=2, lty=2))
legend("bottomleft", c("SMA 21","SMA 10","EMA 21","EMA 10"),
  col=c("blue","red","blue","red"), lty=c(1,1,2,2), lwd=2,
  bg="white")
```

Rolling minimum of window size 3

i = 1		
date	**price**	**min**
20/3	11	NA
21/3	7	
22/3	9	
23/3	8	
24/3	10	
25/3	9	

i = 2		
date	**price**	**min**
20/3	11	NA
21/3	7	NA
22/3	9	
23/3	8	
24/3	10	
25/3	9	

i = 3		
date	**price**	**min**
20/3	11	NA
21/3	7	NA
22/3	9	7
23/3	8	
24/3	10	
25/3	9	

i = 4		
date	**price**	**min**
20/3	11	NA
21/3	7	NA
22/3	9	7
23/3	8	7
24/3	10	
25/3	9	

i = 5		
date	**price**	**min**
20/3	11	NA
21/3	7	NA
22/3	9	7
23/3	8	7
24/3	10	8
25/3	9	

i = 6		
date	**price**	**min**
20/3	11	NA
21/3	7	NA
22/3	9	7
23/3	8	7
24/3	10	8
25/3	9	8

Any ideas for the online implementation?

Rolling Holistic Aggregates

"All of the basic SQL aggregate functions like SUM and MAX can be computed by reading values one at a time and throwing them away. But there are some functions that potentially need to keep track of all the values before they can produce a result. These are called holistic aggregates, and they require more care when implementing."

https://duckdb.org/2021/11/12/moving-holistic.html

median, mode, quantile - all those cannot be so easily optimized

Rolling median

- **min-max heap:** "Optimal Median Smoothing" W. Härdle, W. Steiger 1994

  https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.45.993

- **sort-median:** "Median Filtering is Equivalent to Sorting" Jukka Suomela 2014

  https://arxiv.org/abs/1406.1717

What R ecosystem has to offer?

- zoo - ... for Regular and Irregular Time Series (Z's Ordered Observations)
- TTR - Technical Trading Rules
- roll - Rolling and Expanding Statistics
- rollRegres - Fast Rolling and Expanding Window Linear Regression
- RcppRoll - Efficient Rolling / Windowed Operations
- slider - Sliding Window Functions
- runner - Running Operations for Vectors
- duckdb - ... DuckDB Database Management System
- ...
- data.table - Extension of 'data.frame'

data.table rolling functions: `froll[mean|sum|...]`

- `fill = NA`
- `algo = c("fast","exact")`
- `align = c("right","left","center")`
- `na.rm = FALSE`
- `has.nf = NA`
- `adaptive = FALSE`
- `partial = FALSE`
- `give.names = FALSE`

`optimized funs: mean, sum, prod, min, max, median`
`TODO optimized funs: frollsd, frollvar`

`other funs:` **`frollapply`**

`frollapply`: "Fast" rolling user-defined function (UDF)

"Fast" - as fast as UDF, not compiled R code, can get... which is unfortunately not really that fast. Optimizations are:

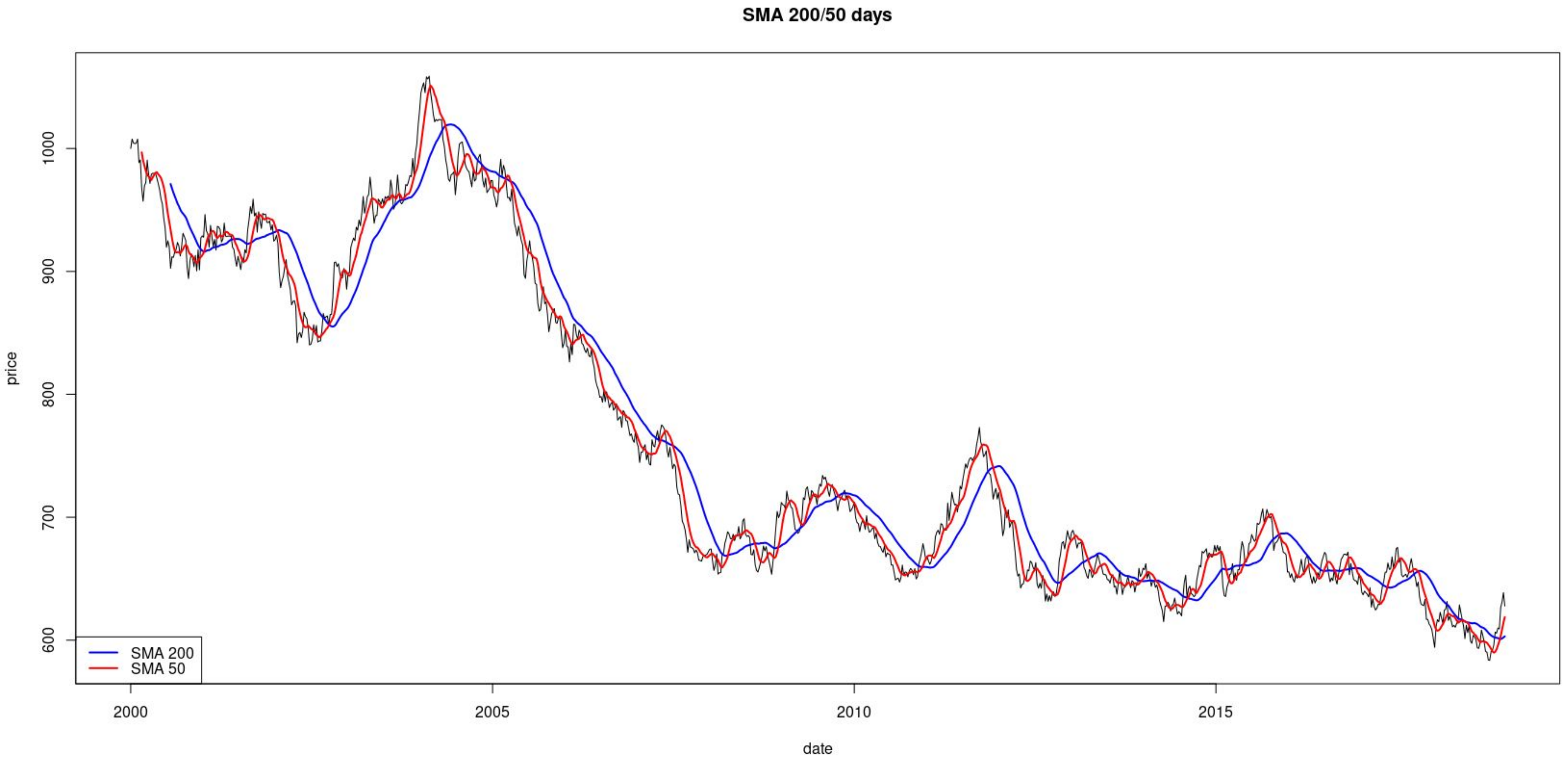
- parallel - computes iterations over multiple CPU threads (50% by default).
  <u>Using base R {parallel} package, openMP cannot parallelize R code!</u>

- avoid repeated allocations.
  Allocates single window (per CPU thread) and re-uses it across all iterations by only copy data into it, rather than allocate on every iteration.
  <u>Copy is cheap, allocation is expensive!</u>

Benchmark: demo

```
library(data.table)
set.seed(440)
n = 1e7
dt = data.table(
  date = as.POSIXct("2000-01-01 00:00:00")+(0:(n-1))*60,
  price = cumprod(c(1e3, rnorm(n-1, 1, 0.0001)))
)
w = c(200,50)*24*60
system.time(ma <- frollmean(dt$price, w))
#   user  system elapsed
#  0.064   0.064   0.066
dt[, c("sma200","sma50") := ma]
dt[seq(1, n, by=1e4), {
  plot(date, price, type="l", main="SMA 200/50 days")
  lines(date, sma200, col="blue", lwd=2)
  lines(date, sma50, col="red", lwd=2)
}]
legend("bottomleft", c("SMA 200","SMA 50"),
  col=c("blue","red"), lty=1, lwd=2)
```

demo SMA200/50 days on 10M obs



Rolling median: base R vs. data.table

```
rollmedian = function(x, n) {
  ans = rep(NA_real_, nx<-length(x))
  if (n<=nx)
    for (i in n:nx)
      ans[i] = median(x[(i-n+1L):(i)])
  ans
}
library(data.table) ## uses 4 CPU threads

set.seed(108)
x = rnorm(1e5)
```

```
n = 100

system.time(rollmedian(x, n))
#   user  system elapsed
#  4.389   0.000   4.389
system.time(frollapply(x, n, median, simplify=unlist))
#   user  system elapsed
#  5.603   0.163   1.465
system.time(frollmedian(x, n))
#   user  system elapsed
#  0.016   0.000   0.009
```

```
n = 1000

system.time(rollmedian(x, n))
#    user  system elapsed
#   7.011   0.028   7.040
system.time(frollapply(x, n, median, simplify=unlist))
#    user  system elapsed
#   8.720   0.173   2.240
system.time(frollmedian(x, n))
#    user  system elapsed
#   0.015   0.004   0.009
```

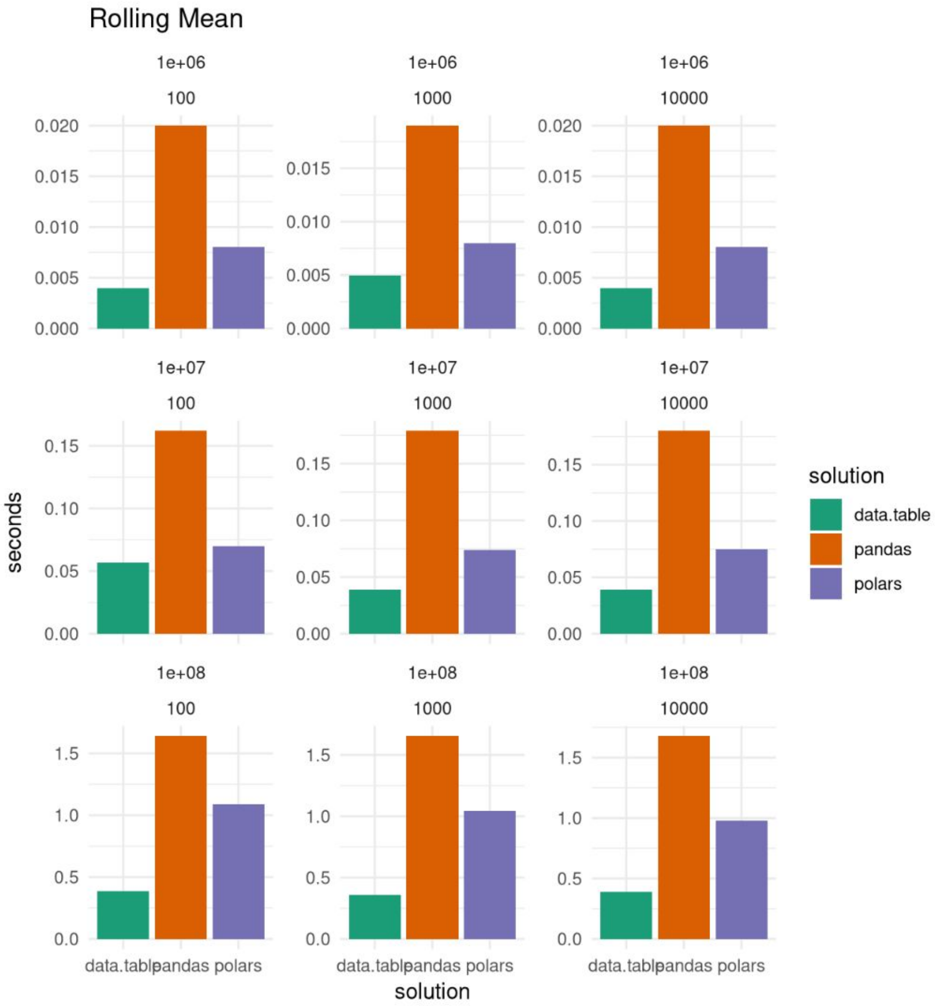
```
n = 10000

system.time(rollmedian(x, n))
#   user  system elapsed
# 34.190   0.012  34.206
system.time(frollapply(x, n, median, simplify=unlist))
#   user  system elapsed
# 40.500   0.456  10.288
system.time(frollmedian(x, n))
#   user  system elapsed
#  0.023   0.016   0.016
```

Benchmark: mini rolling statistics benchmark

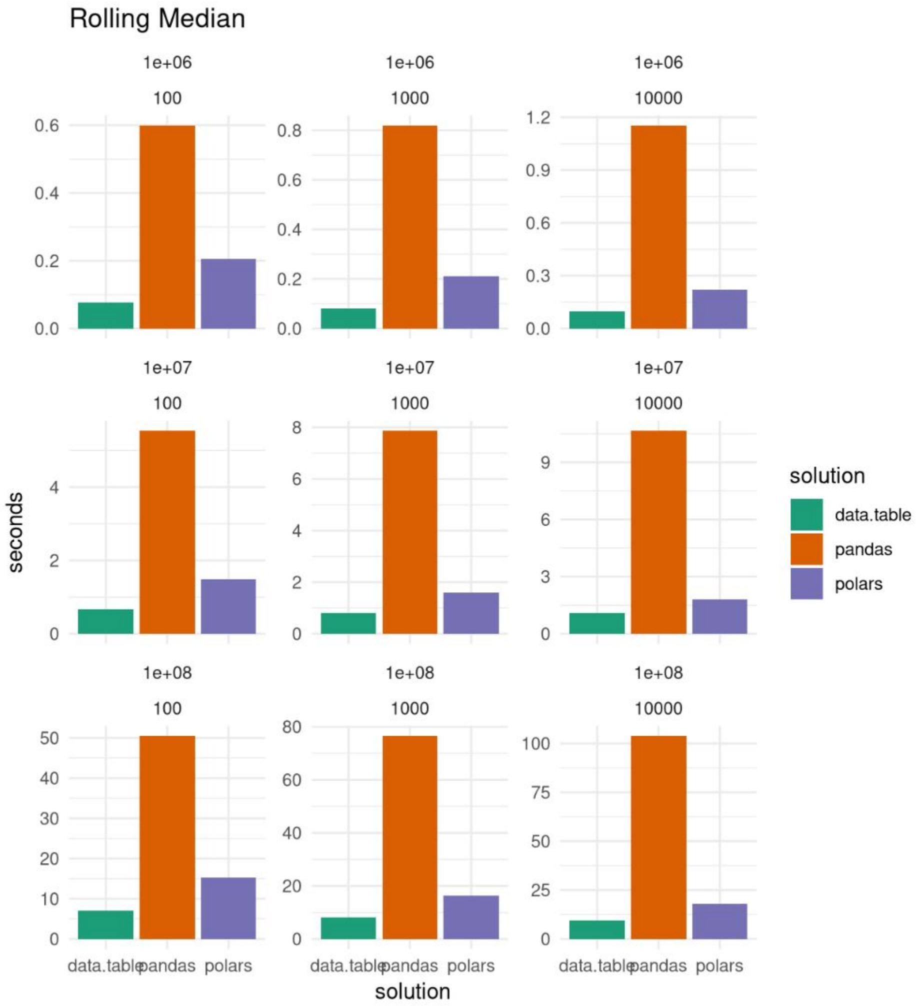
"rollbench": https://github.com/jangorecki/rollbench

Compares python **pandas**, R **data.table** and R **polars** by:

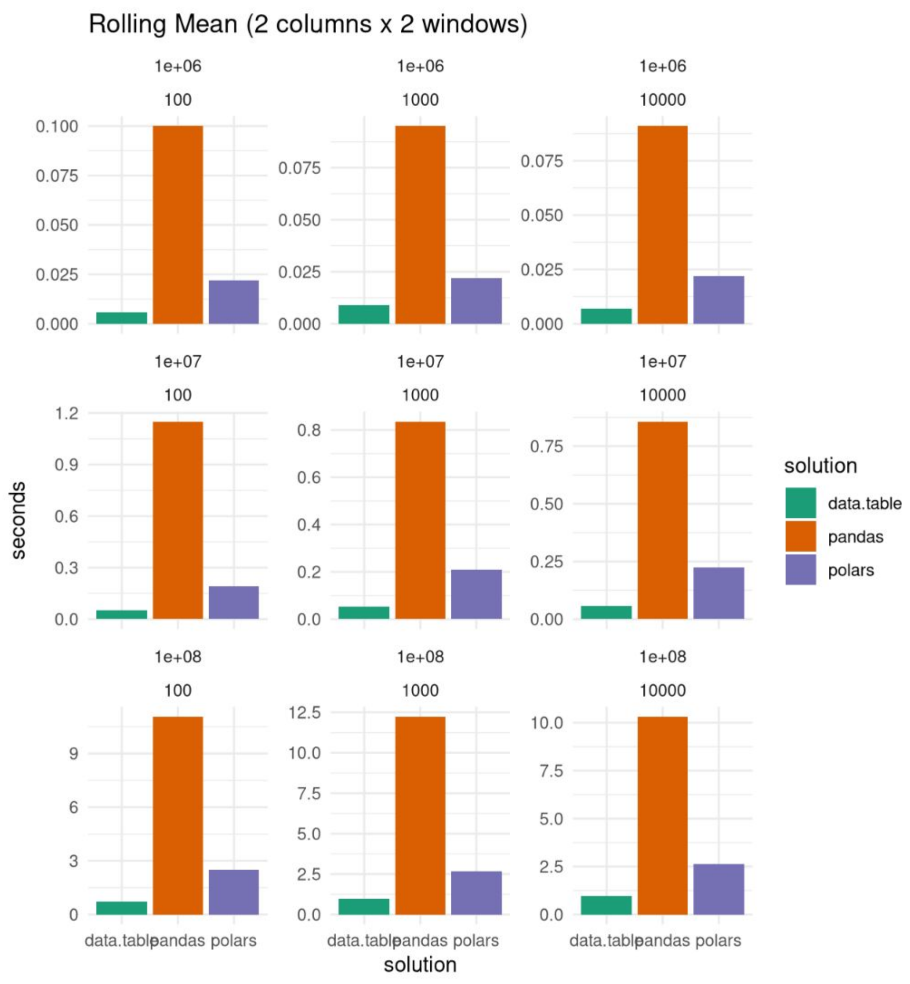
- input size: 1e6, 1e7, 1e8
- rolling window size: 1e2, 1e3, 1e4
- rolling functions: mean and median
- batching: single computation and quadruple (2 columns x 2 windows) computation

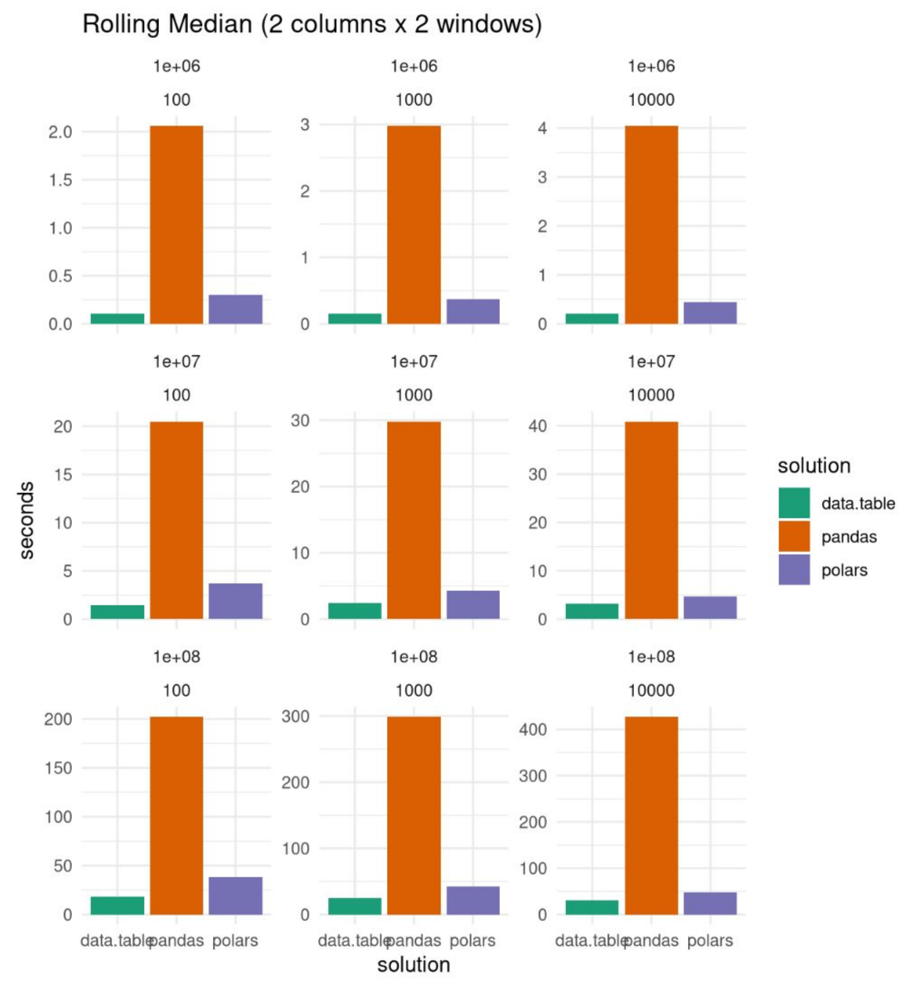


Rolling Mean
1e+06
100
1000
10000
1e+07
1e+08
seconds
solution
data.table
pandas
polars



Rolling Median
1e+06
100
1000
10000
1e+07
1e+08
seconds
solution
data.table
pandas
polars


Rolling Mean (2 columns x 2 windows)
1e+06
100
1000
10000
1e+07
1e+08
seconds
solution
data.table
pandas
polars


Rolling Median (2 columns x 2 windows)
1e+06
100
1000
10000
1e+07
1e+08
seconds
solution
data.table
pandas
polars

Benchmark: db-benchmark

New address: https://duckdblabs.github.io/db-benchmark

Solutions used: pandas, dplyr, data.table, spark, duckdb

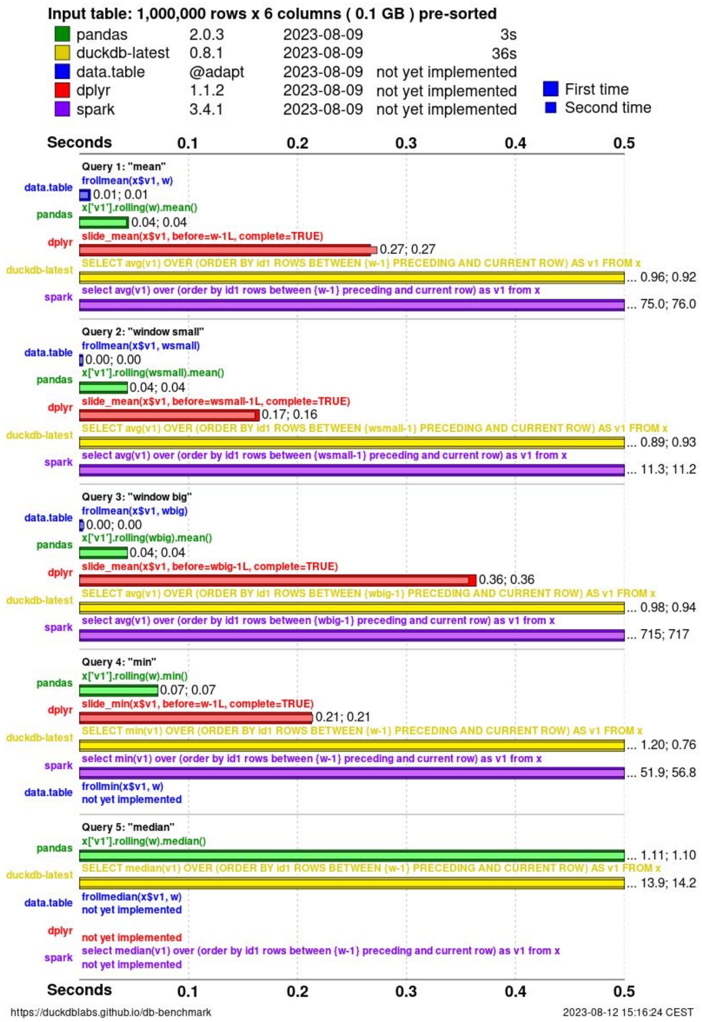
basic questions:

- rolling mean (SMA) of different window widths
- rolling min
- rolling median

advanced questions:

- multiple variables and multiple window widths at once
- weighted mean
- time aware rolling mean (unevenly spaced time series)
- rolling regression

https://github.com/duckdblabs/db-benchmark/pull/9


Input table: 1,000,000 rows x 6 columns (0.1 GB) pre-sorted
pandas 2.0.3 2023-08-09 3s
duckdb-latest 0.8.1 2023-08-09 36s
data.table @adapt 2023-08-09 not yet implemented
dplyr 1.1.2 2023-08-09 not yet implemented
spark 3.4.1 2023-08-09 not yet implemented
First time
Second time
Seconds 0.1 0.2 0.3 0.4 0.5
Query 1: "mean"
data.table frollmean(x$v1, w) 0.01; 0.01
pandas x['v1'].rolling(w).mean() 0.04; 0.04
dplyr slide_mean(x$v1, before=w-1L, complete=TRUE) 0.27; 0.27
duckdb-latest SELECT avg(v1) OVER (ORDER BY id1 ROWS BETWEEN (w-1) PRECEDING AND CURRENT ROW) AS v1 FROM x ... 0.96; 0.92
spark select avg(v1) over (order by id1 rows between {w-1} preceding and current row) as v1 from x ... 75.0; 76.0
Query 2: "window small"
data.table frollmean(x$v1, wsmall) 0.00; 0.00
pandas x['v1'].rolling(wsmall).mean() 0.04; 0.04
dplyr slide_mean(x$v1, before=wsmall-1L, complete=TRUE) 0.17; 0.16
duckdb-latest SELECT avg(v1) OVER (ORDER BY id1 ROWS BETWEEN (wsmall-1) PRECEDING AND CURRENT ROW) AS v1 FROM x ... 0.89; 0.93
spark select avg(v1) over (order by id1 rows between {wsmall-1} preceding and current row) as v1 from x ... 11.3; 11.2
Query 3: "window big"
data.table frollmean(x$v1, wbig) 0.00; 0.00
pandas x['v1'].rolling(wbig).mean() 0.04; 0.04
dplyr slide_mean(x$v1, before=wbig-1L, complete=TRUE) 0.36; 0.36
duckdb-latest SELECT avg(v1) OVER (ORDER BY id1 ROWS BETWEEN (wbig-1) PRECEDING AND CURRENT ROW) AS v1 FROM x ... 0.98; 0.94
spark select avg(v1) over (order by id1 rows between {wbig-1} preceding and current row) as v1 from x ... 715; 717
Query 4: "min"
pandas x['v1'].rolling(w).min() 0.07; 0.07
dplyr slide_min(x$v1, before=w-1L, complete=TRUE) 0.21; 0.21
duckdb-latest SELECT min(v1) OVER (ORDER BY id1 ROWS BETWEEN (w-1) PRECEDING AND CURRENT ROW) AS v1 FROM x ... 1.20; 0.76
spark select min(v1) over (order by id1 rows between {w-1} preceding and current row) as v1 from x ... 51.9; 56.8
data.table frollmin(x$v1, w) not yet implemented
Query 5: "median"
pandas x['v1'].rolling(w).median() ... 1.11; 1.10
duckdb-latest SELECT median(v1) OVER (ORDER BY id1 ROWS BETWEEN (w-1) PRECEDING AND CURRENT ROW) AS v1 FROM x ... 13.9; 14.2
data.table frollmedian(x$v1, w) not yet implemented
dplyr not yet implemented
spark select median(v1) over (order by id1 rows between {w-1} preceding and current row) as v1 from x not yet implemented
Seconds 0.1 0.2 0.3 0.4 0.5
https://duckdblabs.github.io/db-benchmark
2023-08-12 15:16:24 CEST


Input table: 100,000,000 rows x 6 columns (10 GB) pre-sorted
pandas 2.0.3 2023-08-09 420s
data.table @adapt 2023-08-09 not yet implemented
dplyr 1.1.2 2023-08-09 not yet implemented
spark 3.4.1 2023-08-09 timeout
duckdb-latest 0.8.1 2023-08-09 timeout
First time
Second time
Seconds 20 40 60 80 100
Query 1: "mean"
data.table frollmean(x$v1, w) 1.45; 1.46
pandas x['v1'].rolling(w).mean() 5.42; 5.42
dplyr slide_mean(x$v1, before=w-1L, complete=TRUE) 51.8; 51.5
duckdb-latest SELECT avg(v1) OVER (ORDER BY id1 ROWS BETWEEN (w-1) PRECEDING AND CURRENT ROW) AS v1 FROM x ... 88.2; 116
spark select avg(v1) over (order by id1 rows between {w-1} preceding and current row) as v1 from x timeout
Query 2: "window small"
data.table frollmean(x$v1, wsmall) 1.45; 1.46
pandas x['v1'].rolling(wsmall).mean() 5.41; 5.41
dplyr slide_mean(x$v1, before=wsmall-1L, complete=TRUE) 41.5; 41.1
duckdb-latest SELECT avg(v1) OVER (ORDER BY id1 ROWS BETWEEN (wsmall-1) PRECEDING AND CURRENT ROW) AS v1 FROM x ... 111; 110
spark select avg(v1) over (order by id1 rows between {wsmall-1} preceding and current row) as v1 from x timeout
Query 3: "window big"
data.table frollmean(x$v1, wbig) 1.46; 1.79
pandas x['v1'].rolling(wbig).mean() 4.60; 4.60
dplyr slide_mean(x$v1, before=wbig-1L, complete=TRUE) 59.6; 59.6
duckdb-latest SELECT avg(v1) OVER (ORDER BY id1 ROWS BETWEEN (wbig-1) PRECEDING AND CURRENT ROW) AS v1 FROM x ... 84.5; 108
spark select avg(v1) over (order by id1 rows between {wbig-1} preceding and current row) as v1 from x timeout
Query 4: "min"
pandas x['v1'].rolling(w).min() 6.43; 6.43
dplyr slide_min(x$v1, before=w-1L, complete=TRUE) 28.3; 28.1
duckdb-latest SELECT min(v1) OVER (ORDER BY id1 ROWS BETWEEN (w-1) PRECEDING AND CURRENT ROW) AS v1 FROM x ... 96.8; 143
data.table frollmin(x$v1, w) not yet implemented
spark select min(v1) over (order by id1 rows between {w-1} preceding and current row) as v1 from x timeout
Query 5: "median"
pandas x['v1'].rolling(w).median() ... 188; 187
data.table frollmedian(x$v1, w) not yet implemented
dplyr not yet implemented
spark select median(v1) over (order by id1 rows between {w-1} preceding and current row) as v1 from x not yet implemented
duckdb-latest SELECT median(v1) OVER (ORDER BY id1 ROWS BETWEEN (w-1) PRECEDING AND CURRENT ROW) AS v1 FROM x timeout
Seconds 20 40 60 80 100
https://duckdblabs.github.io/db-benchmark
2023-08-12 15:16:25 CEST

Special thanks 🙏

- hosting the meeting: EdinbR user group http://edinbr.org
- organizing: Mike Spencer
- reviving data.table after 3 years of inactivity: Toby Dylan Hocking
- sponsoring travel: NSF POSE grant: https://tinyurl.com/DT-travel-grant
- presence: You

Questions?

github.com/jangorecki
fosstodon.org/@jangorecki
jangorecki @ protonmail.ch