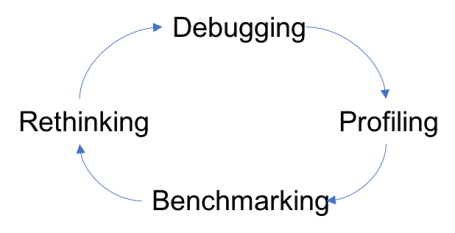
Optimising your R code is not always the priority. But when you run out of memory, or it just takes too long, you start to wonder if there are better ways to do things! In this blog post, I will show you my way of optimising my R code and the process behind it. As an example, I will use the code from my last blog post about gas prices in Germany.

There are different steps to optimise your code. From debugging and profiling over benchmarking to rethinking the whole method. You have to repeat these steps until you are satisfied with the result.



**A short recap of the initial code**

I had around 1700 .csv files, each containing the price changes of one day for all German gas stations from 2014 up till 2019. My plan was to analyse the price patterns, but first I had to prepare and load all the data. Normally, I would just load all the files, combine them into one data.table() , do some data preparation steps and then start my analysis. This was not possible here since the raw data was around 20 GB big and I wanted to rearrange them so that I would have half-hour prices. So, I settled for the solution of reading the files one after another, while preparing and aggregating each one at a time, and combining them at the end. There were eight steps during my loop:

# pseudo code

for (i.file in price\_files) {

# load data

# remove wrong prices (e.g. -0.001, 0 or 8.888)

# set TIME and DATE

# build a time grid including opening times

# add last day price and save next day price

# NA handling - last observation carried forward (locf)

# adding brand, motorway and post code info

# calculating mean and quantiles

}

Let’s see where I can optimise my steps!

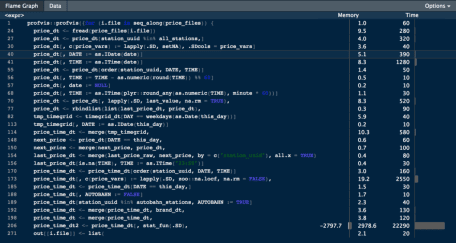
**Identifying the bottlenecks**

Since I want to optimise a loop, I can just concentrate on a few iterations instead of the whole loop. Also, I do not need to include every station for this test. Therefore, I subset my data to stations which are in the area around Frankfurt am Main.

stations\_dt <- stations\_dt[post\_code >= "60000" & post\_code < "66000",]

**Note**: Yes, you can filter strings like this – they are sorted alphabetically. Also, they are characters and not numbers since there are cases like “01896”.

With this small test data, I will continue my optimisation process. To get an overview of bottlenecks I use the package profvis. Since RStudio version 1.0, it is quite easy to use. I just select the lines of code with my for-loop and click *Profile > Profile Selected Line(s)* to get a report with the time and memory consumption in a new tab.



I filter only for those lines which have a non-zero value to have a nicer overview. My main task is to reduce time consumption, therefore I will focus on the longest time first. There is one major time consumer: stat\_fun(). It takes around 75% of the total 30000ms. So, what is this function of mine?

**a deeper look at stat\_fun()**

This function returns a data.table() with the mean, the 10% and 90% quantile and the number of observations. I use stat\_fun on each group defined by this\_by <- c("DATE", "TIME", "AUTOBAHN", "BRAND", "plz").

# get mean and quantiles 0.1 and 0.9

stat\_fun <- function(x, na.rm = TRUE) {

#x <- price\_time\_dt[,.SD, .SDcol = c("diesel", "e10")]

x\_quant1 <- x[, lapply(.SD, quantile, probs = c(0.1), na.rm = na.rm),

.SDcols = names(x)]

x\_quant9 <- x[, lapply(.SD, quantile, probs = c(0.9), na.rm = na.rm),

.SDcols = names(x)]

x\_mean <- x[, lapply(.SD, mean, na.rm = na.rm), .SDcols = names(x)]

x\_obs <- x[, lapply(.SD, function(y) sum(!is.na(y))), .SDcols = names(x)]

setnames(x\_quant1, paste0(names(x), "\_Q10"))

setnames(x\_quant9, paste0(names(x), "\_Q90"))

setnames(x\_mean, paste0(names(x), "\_MEAN"))

setnames(x\_obs, paste0(names(x), "\_OBS"))

out <- data.table(x\_quant1, x\_quant9, x\_mean, x\_obs)

return(out)

}

price\_time\_dt2 <- price\_time\_dt[, stat\_fun(.SD), .SDcols = price\_vars, by = this\_by]

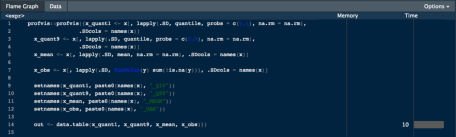
Why did I build it like this? My goal was to have one function, which returns all the values I need for my analysis later: the mean, the quantiles and the number of observations. Let’s check how we can optimise this!

First, I look into the function with profvis again. Since there is not much to see but that it takes 7900ms for one iteration I profile the lines in the stat\_fun. Therefore I set the inputs and then select the lines in the function for profiling.

x <- price\_time\_dt[,.SD, .SDcol = c("diesel", "e10")]

na.rm <- TRUE

The result is quite surprising, there is not much time used even though I used the whole data instead of one group.



How many times does this function get called? Well, for each combination of the by key. For the first file, this is around 1500 times. I will now check if mean() and quantile() are taking a lot of time, if they are used by each group.

x\_mean <- price\_time\_dt[, lapply(.SD, mean, na.rm = TRUE),

.SDcols = price\_vars,

by = this\_by]

x\_quant1 <- price\_time\_dt[, lapply(.SD, quantile,

probs = 0.1,

na.rm = TRUE),

.SDcols = price\_vars,

by = this\_by]

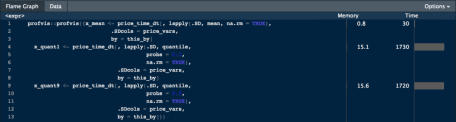
x\_quant9 <- price\_time\_dt[, lapply(.SD, quantile,

probs = 0.9,

na.rm = TRUE),

.SDcols = price\_vars,

by = this\_by]

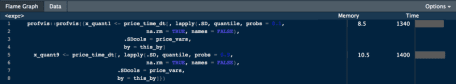


The mean() function is fast and not a problem, but quantile() takes quite long. I could combine the calculation for the 10% and 90% quantile into one call, but the resulting table I would have to reshape into my desired output – if I cannot find any other tweaks, I will come back to this approach.

While reading the help pages for the quantile() function, I stumble upon the argument names:

names: logical; if true, the result has a names attribute. Set to FALSE for speedup with many probs.

I am not using many probs, but this still improves the time by around 700ms.



This is not enough, so I will have to test the other approach!

**implemanting stat\_fun() as part of the main code**

The new approach first creates a data.table() for the mean, the quantiles and the number of observations and then merges them by the groups together. With this approach, I can combine the two quantile calculations, but have to include the merging part.

# calculating the mean

x\_mean <- price\_time\_dt[, lapply(.SD, mean, na.rm = TRUE),

.SDcols = price\_vars,

by = this\_by]

# calculating the quantiles

x\_quant <- price\_time\_dt[, lapply(.SD, quantile, probs = c(0.1, 0.9),

na.rm = TRUE, names = FALSE),

.SDcols = price\_vars,

by = this\_by]

x\_quant[, QUANT := c("Q10", "Q90")]

eq\_quant <- paste0(paste0(this\_by, collapse = "+"), "~QUANT")

x\_quant <- dcast(data = x\_quant, formula = eq\_quant, value.var = price\_vars)

setkeyv(x\_quant, NULL)

# number of observations

x\_obs <- price\_time\_dt[, lapply(.SD, function(y) sum(!is.na(y))),

.SDcols = price\_vars,

by = this\_by]

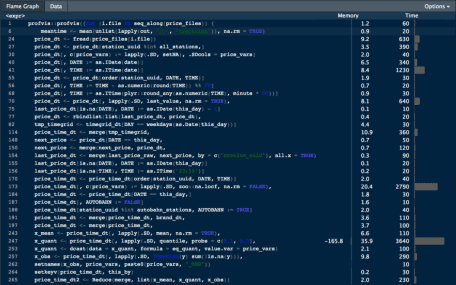
setnames(x\_mean, price\_vars, paste0(price\_vars, "\_MEAN"))

setnames(x\_obs, price\_vars, paste0(price\_vars, "\_OBS"))

setkeyv(price\_time\_dt, this\_by)

price\_time\_dt2 <- Reduce(merge, list(x\_mean, x\_quant, x\_obs))

As it turns out, merging is fast as well – I would not have thought so! Instead of nearly 30s it only needs 12s now! With the biggest issue out of the way, there seems to be some more potential in other lines as well. Next up are the time formatting and the missing values handling with the last observation carried forward (locf).



**last observations slowly carried forward**

price\_time\_dt <- price\_time\_dt[order(station\_uuid, DATE, TIME)]

price\_time\_dt[, c(price\_vars) := lapply(.SD, zoo::na.locf, na.rm = FALSE),

.SDcols = price\_vars,

by = station\_uuid]

Sometimes it is hard to reinvent the wheel – so l search the vast knowledge of the internet .They describe exactly my problem: a faster locf-method with grouping.

price\_time\_dt <- price\_time\_dt[order(station\_uuid, DATE, TIME)]

id\_change <- price\_time\_dt[, c(TRUE, station\_uuid[-1] != station\_uuid[-.N])]

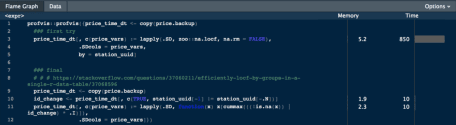
price\_time\_dt[, c(price\_vars) :=

lapply(.SD, function(x) x[cummax(((!is.na(x)) | id\_change) \* .I)]),

.SDcols = price\_vars]

To understand this method I made this table with a toy example. The last column is used as the index, which replaces missing values with the last observation. Because this method is vectorised and works without a by group, it is much faster.

| **.I** | **id** | **id\_change** | **x** | **!is.na(x)** | **(!is.na(x)|id\_change)\*.I** | **cummax(…)** |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | TRUE | 4 | TRUE | 1 \* 1 = 1 | 1 |
| 2 | 1 | FALSE | NA | FALSE | 0 \* 2 = 0 | 1 |
| 3 | 1 | FALSE | NA | FALSE | 0 \* 3 = 0 | 1 |
| 4 | 2 | TRUE | 9 | TRUE | 1 \* 4 = 4 | 4 |
| 5 | 2 | FALSE | NA | FALSE | 0 \* 5 = 0 | 4 |
| 6 | 2 | FALSE | 5 | TRUE | 1 \* 6 = 6 | 6 |
| 7 | 2 | FALSE | NA | FALSE | 0 \* 7 = 0 | 6 |
| 8 | 3 | TRUE | NA | FALSE | 1 \* 8 = 8 | 8 |
| 9 | 3 | FALSE | 17 | TRUE | 1 \* 9 = 9 | 9 |



**even formatting time is relative**

The raw data contains a column with the timestamp of a price change. There are multiple options to start to transform this string into a date and time variable:

* *base* R with as.POSIXct()
* lubridate\* with ymd\_hms()
* *data.table* with as.IDate() and as.ITime()

There are two things I want from the time transformation. First, for my aggregation over daily hours analysis, I want the date and the time in separate columns. Second, I want to round to a given minute (e.g. to half hours or hours).

Since I am a *data.table* person and I think fewer packages in a project are a good thing, my first choices for these tasks are as.IDate() and as.ITime(). But – the go-to package when it comes to time and dates – *lubridate* with the round\_date() function to round to a specific minute, for example, might be a good candidate as well. Time to do some benchmarking!

**my benchmark setup**

To test the different functions I will use one of the files, reduced to the date, DAY and TIME columns:

date DAY TIME

1: 2014-06-08 09:50:01 2014-06-08 09:50:01

2: 2014-06-08 09:50:01 2014-06-08 09:50:01

3: 2014-06-08 09:50:01 2014-06-08 09:50:01

4: 2014-06-08 09:50:01 2014-06-08 09:50:01

5: 2014-06-08 09:50:01 2014-06-08 09:50:01

**splitting the format into date and time**

microbenchmark(

# lubridate

"lubridate::ymd()" = DT[, DAY\_ymd := ymd(DAY)],

"lubridate::hms()" = DT[, TIME\_hms := as.numeric(hms(TIME))],

# data.table

"data.table::as.IDate()" = DT[, DATE\_IDate := as.IDate(date)],

"data.table::as.ITime()" = DT[, DATE\_ITime := as.ITime(date)],

# settings

times = 100L, unit = "ms")

Unit: milliseconds

expr min lq mean median uq max neval

lubridate::ymd() 7.210735 10.18305 12.62154 11.96969 14.31220 27.62012 100

lubridate::hms() 9.611753 10.26598 11.08295 10.60343 11.38001 16.48317 100

data.table::as.IDate 74.908261 75.91852 78.98697 77.86806 80.62390 91.97302 100

data.table::as.ITime 298.047305 306.15576 314.78720 309.89135 315.41061 466.35973 100

Well, *lubridate* is the fastest! But, the output of hms() cannot simply be saved into a data.table column because it is a nested object of the class Period from *lubridate*. Therefore, I will use this little workaround:

price\_dt[, c("DATE", "TIME") := tstrsplit(date, " ")]

price\_dt[, DATE := lubridate::ymd(DATE)]

price\_dt[, TIME := as.ITime(as.numeric(lubridate::hms(TIME)))]

With this, I can continue to use the as.ITime() function and its format. Also, using as.ITime() on a numeric is much faster than on the original date.

**rounding to the minutes**

For the second task, I will compare lubridate::round\_date() and ply::round\_any()

minute <- 30

microbenchmark(

# rounding lubridate

"lubridate::round\_date()" =

DT[, ROUND\_TIME\_lb := lubridate::round\_date(DATE\_ymd\_hms, paste0(minute, " mins"))],

# rounding plyr

"plyr::round\_any()" =

DT[, ROUND\_TIME\_pl := as.ITime(plyr::round\_any(as.numeric(DATE\_ITime), minute \* 60))],

# settings

times = 100L, unit = "ms")

Unit: milliseconds

expr min lq mean median uq max neval

lubridate::round\_date() 24.075546 25.582259 27.27308 26.944734 28.120435 39.62532 100

plyr::round\_any() 1.351571 1.533044 1.97845 1.727221 2.249484 4.37735 100

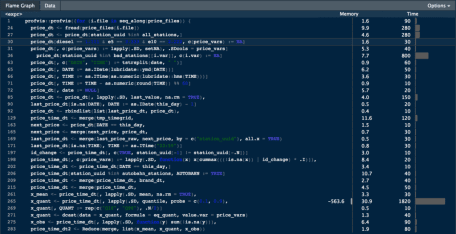
On the one hand, round\_date() has an easy to understand input format, but on the other hand round\_any() is much faster since it works on numeric values.

With this setup, I only use lubridate and plyr in one part of my code and for the rest data.table – That makes my inner self.data.table happy and it still is quite fast.

With all the changes I have made, I am curious how much the speed has improved.

**fixing the bottlenecks**

That is quite an improvement! The code now only takes 4330ms. There are probably even more enhancements possible, but I think that is enough of a speed boost.



The for-loop in R, can be very slow in its raw un-optimised form, especially when dealing with larger data sets. There are a number of ways you can make your logics run fast, but you will be really surprised how fast you can actually go.  
This posts shows a number of approaches including simple tweaks to logic design, parallel processing and

Rcpp

, increasing the speed by orders of several magnitudes, so you can comfortably process data as large as 100 Million rows and more.

Lets try to improve the speed of a logic that involves a for-loop and a condition checking statement (if-else) to create a column that gets appended to the input data frame (df). The code below creates that initial input data frame.

# Create the data frame

col1 <- runif (12^5, 0, 2)

col2 <- rnorm (12^5, 0, 2)

col3 <- rpois (12^5, 3)

col4 <- rchisq (12^5, 2)

df <- data.frame (col1, col2, col3, col4)

**The logic we are about to optimise:**  
For every row on this data frame (df), check if the sum of all values is greater than 4. If it is, a new 5th variable gets the value “greater\_than\_4”, else, it gets “lesser\_than\_4”.

# Original R code: Before vectorization and pre-allocation

system.time({

for (i in 1:nrow(df)) { # for every row

if ((df[i, 'col1'] + df[i, 'col2'] + df[i, 'col3'] + df[i, 'col4']) > 4) { # check if > 4

df[i, 5] <- "greater\_than\_4" # assign 5th column

} else {

df[i, 5] <- "lesser\_than\_4" # assign 5th column

}

}

})

All the computations below, for processing times, were done on a MAC OS X with 2.6 Ghz processor and 8GB RAM.

**Vectorise and pre-allocate data structures**

Always initialize your data structures and output variable to required length and data type before taking it to loop for computations. Try not to incrementally increase the size of your data inside the loop. Lets compare how vectorisation improves speed on a range of data sizes from 1000 to 100,000 rows.

# after vectorization and pre-allocation

output <- character (nrow(df)) # initialize output vector

system.time({

for (i in 1:nrow(df)) {

if ((df[i, 'col1'] + df[i, 'col2'] + df[i, 'col3'] + df[i, 'col4']) > 4) {

output[i] <- "greater\_than\_4"

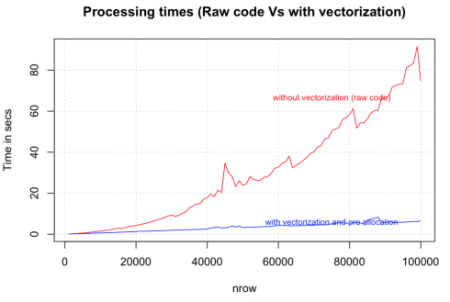
} else {

output[i] <- "lesser\_than\_4"

}

}

df$output})

Raw Code Vs With vectorisation:  
[](https://i2.wp.com/datascienceplus.com/wp-content/uploads/2016/01/raw_vs_with_vectorization.png)

**Take statements that check for conditions (if statements) outside the loop**

Taking the condition checking outside the loop the speed is compared against the previous version that had vectorisation alone. The tests were done on dataset size range from 100,000 to 1,000,000 rows. The gain in speed is again dramatic.

# after vectorization and pre-allocation, taking the condition checking outside the loop.

output <- character (nrow(df))

condition <- (df$col1 + df$col2 + df$col3 + df$col4) > 4 # condition check outside the loop

system.time({

for (i in 1:nrow(df)) {

if (condition[i]) {

output[i] <- "greater\_than\_4"

} else {

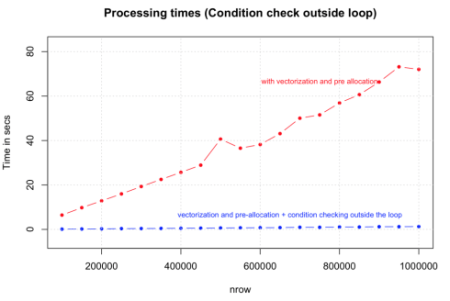
output[i] <- "lesser\_than\_4"

}

}

df$output <- output

})

Condition Checking outside loops:  
[](https://i1.wp.com/datascienceplus.com/wp-content/uploads/2016/01/condition_checking_outside_loops.png)

**Run the loop only for True conditions**

Another optimisation we can do here is to run the loop only for condition cases that are ‘True’, by initialising (pre-allocating) the default value of output vector to that of ‘False’ state. The speed improvement here largely depends on the proportion of ‘True’ cases in your data.

The tests compared the performance of this against the previous case (2) on data size ranging from 1,000,000 to 10,000,000 rows. Note that we have increase a ‘0’ here. As expected there is a consistent and considerable improvement.

output <- character(nrow(df))

condition <- (df$col1 + df$col2 + df$col3 + df$col4) > 4

system.time({

for (i in (1:nrow(df))[condition]) { # run loop only for true conditions

if (condition[i]) {

output[i] <- "greater\_than\_4"

} else {

output[i] <- "lesser\_than\_4"

}

}

df$output })

Running Loop Only On True Conditions:  
[](https://i1.wp.com/datascienceplus.com/wp-content/uploads/2016/01/running_loop_only_true_conditions.png)

**Use ifelse() whenever possible**

You can make this logic much simpler and faster by using the

ifelse()

statement. The syntax is similar to the

if

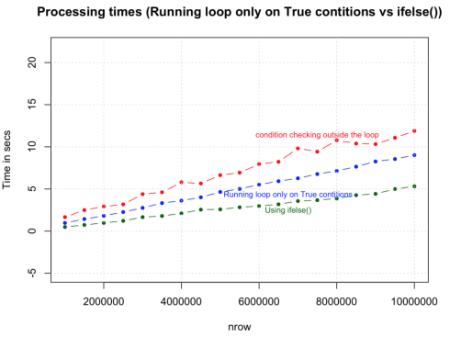
function in MS Excel, but the speed increase is phenomenal, especially considering that there is no vector pre-allocation here and the condition is checked in every case. Looks like this is going to be a highly preferred option to speed up simple loops.

system.time({

output <- ifelse ((df$col1 + df$col2 + df$col3 + df$col4) > 4, "greater\_than\_4", "lesser\_than\_4")

df$output <- output

})

True conditions only vs ifelse:  
[](https://i2.wp.com/datascienceplus.com/wp-content/uploads/2016/01/true_conditions_only_vs_ifelse.png)

**Using which()**

By using

which()

command to select the rows, we are able to achieve one-third the speed of

Rcpp

.

# Thanks to Gabe Becker

system.time({

want = which(rowSums(df) > 4)

output = rep("less than 4", times = nrow(df))

output[want] = "greater than 4"

})

# nrow = 3 Million rows (approx)

user system elapsed

0.396 0.074 0.481

**Use apply family of functions instead of for-loops**

Using apply() function to compute the same logic and comparing it against the vectorised for-loop. The results again is faster in order of magnitudes but slower than

ifelse()

and the version where condition checking was done outside the loop. This can be very useful, but you will need to be a bit crafty when handling complex logic.

# apply family

system.time({

myfunc <- function(x) {

if ((x['col1'] + x['col2'] + x['col3'] + x['col4']) > 4) {

"greater\_than\_4"

} else {

"lesser\_than\_4"

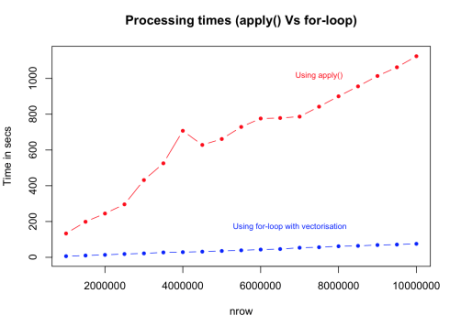
}

}

output <- apply(df[, c(1:4)], 1, FUN=myfunc) # apply 'myfunc' on every row

df$output <- output

})

apply function Vs For loop in R:  
[](https://i2.wp.com/datascienceplus.com/wp-content/uploads/2016/01/apply_Vs_For_loop.png)

**Use byte code compilation for functions cmpfun() from compiler package, rather than the actual function itself**

This may not be the best example to illustrate the effectiveness of byte code compilation, as the time taken is marginally higher than the regular form. However, for more complex functions, byte-code compilation is known to perform faster. So you should definitely give it a shot.

# byte code compilation

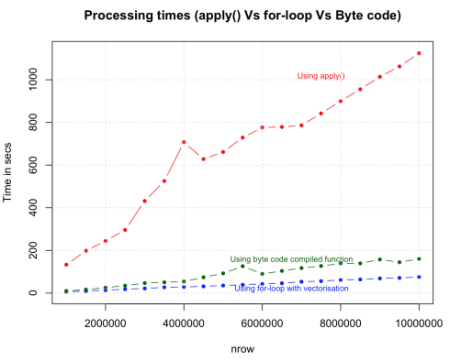
library(compiler)

myFuncCmp <- cmpfun(myfunc)

system.time({

output <- apply(df[, c (1:4)], 1, FUN=myFuncCmp)

})

apply vs for-loop vs byte code compiled functions:  
[](https://i1.wp.com/datascienceplus.com/wp-content/uploads/2016/01/apply-vs-for-loop-vs-byte-code-compiled-functions.png)

**Use Rcpp**

Lets turn this up a notch. So far we have gained speed and capacity by various strategies and found the most optimal one using the

ifelse()

statement. What if we add one more zero? Below we execute the same logic but with

Rcpp

, and with a data size is increased to 100 Million rows. We will compare the speed of

Rcpp

to the

ifelse()

method.

library(Rcpp)

sourceCpp("MyFunc.cpp")

system.time (output <- myFunc(df)) # see Rcpp function below

Below is the same logic executed in C++ code using Rcpp package. Save the code below as “MyFunc.cpp” in your R session’s working directory (else you just have to sourceCpp from the full filepath). Note: the

// [[Rcpp::export]]

comment is mandatory and has to be placed just before the function that you want to execute from R.

// Source for MyFunc.cpp

#include

using namespace Rcpp;

// [[Rcpp::export]]

CharacterVector myFunc(DataFrame x) {

NumericVector col1 = as(x["col1"]);

NumericVector col2 = as(x["col2"]);

NumericVector col3 = as(x["col3"]);

NumericVector col4 = as(x["col4"]);

int n = col1.size();

CharacterVector out(n);

for (int i=0; i 4){

out[i] = "greater\_than\_4";

} else {

out[i] = "lesser\_than\_4";

}

}

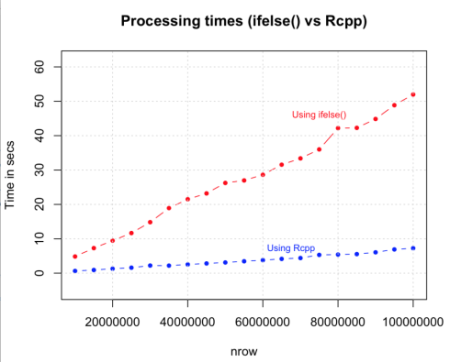
return out;

}

Rcpp

speed performance against

ifelse

:  
[](https://i1.wp.com/datascienceplus.com/wp-content/uploads/2016/01/Rcpp-speed-performance-against-ifelse.png)

**Use parallel processing if you have a multicore machine**

Parallel processing:

# parallel processing

library(foreach)

library(doSNOW)

cl <- makeCluster(4, type="SOCK") # for 4 cores machine

registerDoSNOW (cl)

condition <- (df$col1 + df$col2 + df$col3 + df$col4) > 4

# parallelization with vectorization

system.time({

output <- foreach(i = 1:nrow(df), .combine=c) %dopar% {

if (condition[i]) {

return("greater\_than\_4")

} else {

return("lesser\_than\_4")

}

}

})

df$output <- output

**Remove variables and flush memory as early as possible**

Remove objects

rm()

that are no longer needed, as early as possible in code, especially before going in to lengthy loop operations. Sometimes, flushing

gc()

at the end of each iteration with in the loops can help.

**Use data structures that consume lesser memory**

Data.table()

is an excellent example, as it reduces the memory overload which helps to speed up operations like merging data.

dt <- data.table(df) # create the data.table

system.time({

for (i in 1:nrow (dt)) {

if ((dt[i, col1] + dt[i, col2] + dt[i, col3] + dt[i, col4]) > 4) {

dt[i, col5:="greater\_than\_4"] # assign the output as 5th column

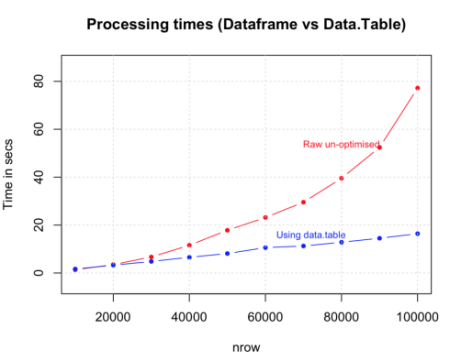
} else {

dt[i, col5:="lesser\_than\_4"] # assign the output as 5th column

}

}

})

Dataframe Vs Data.Table:  
[](https://i0.wp.com/datascienceplus.com/wp-content/uploads/2016/01/Dataframe-Vs-Data.Table_.png)

**Speed Summary**

**Method:** Speed, nrow(df)/time\_taken = n rows per second  
**Raw:** 1X, 120000/140.15 = 856.2255 rows per second (normalised to 1)  
**Vectorised:** 738X, 120000/0.19 = 631578.9 rows per second  
**True Conditions only:** 1002X, 120000/0.14 = 857142.9 rows per second  
**ifelse:** 1752X, 1200000/0.78 = 1500000 rows per second  
**which:** 8806X, 2985984/0.396 = 7540364 rows per second  
**Rcpp:** 13476X, 1200000/0.09 = 11538462 rows per second

The numbers above are approximate and are based in arbitrary runs. The results are not calculated for

data.table()

, byte code compilation and parallelisation methods as they will vary on a case to case basis, depending upon how you apply it.

**To sum it all up**

There are a lot of ways how you can improve and optimise your code. Some changes might be quite settled whilst others demand a change in your way of thinking about a problem. There are also a lot of packages out there which are optimised in doing a specific job (e.g. lubridate). So the question often is not „Can this be improved?“ but more like „Which package is better at this job?“.

To be honest, it has taken me quite a while to finish this post. Whilst finding the bottlenecks and fixing them, I also found myself changing parts of the data preparation and adding new data error handling parts.