MIS 573:

Practical Big Data Analytics

Advanced Topics in R High Performance Computing

Yihuang K. Kang

"... you can't go from knowing nothing to becoming an expert without going through a period of great frustration and great suckiness..."

Hadley Wickham

Dive deeper into R

- We could use R to solve big data management and analysis problems with help of other software packages and frameworks, such as *Hadoop*, *MADlib*, *H2O*, and *Spark*. But, frankly, we do not know much about R's own limitations and capabilities when coping with Big Data.

- R is created by statisticians, tweaked by computer scientists, and contributed by many people in various data analysis communities. Many R packages are created by analysts and used to solve these analysts' own data problems, but not designed for efficiency purposes.

- So, in the case that we deal with massive datasets without having the access to a cluster of machines, we need to know how to take R to its limit! That is," *how to evaluate the performance of our R code?*", "*how to make our R code run faster?*", and "*how to process big datasets with limited main memory*?".

Why my R code is so slow?

- R has its own natural limitations as follows.
  - R is *interpreted language,* not complied language, which means your R code is re-interpreted every time you run it, even when it has not changed.
  - R is *single-threaded*, which means your R code runs sequentially with only one virtual CPU core.
  - R requires all *objects to be loaded into memory*, which means you couldn't load massive datasets when they are too big to fit in memory. Native R is not like some other software/languages that can read datasets on demand.
- Another simple but common reason… your R code sucks!

Profiling your R code

- So, then how to write "better" R code? It is all about how to write high-performance R code. Computer scientists have been measuring performance characteristics of programs in terms of two types of *complexities*: *time complexity*, which refers to the computing time required to run an R program, and *space complexity*, which refers to the main memory required to load and process the data.

- There are various measures of both complexities, such as execution time, memory utilization, CPU utilization, and disk I/O. As you may expect, R provides a bunch of packages that help us identify the biggest bottlenecks and give us idea of how the program and its parts perform.

Profiling your R code(cont.)

- The *execution time* refers to the total time to complete the execution of a program. It should be the most intuitive performance measure, as it directly impacts on how quickly we can get our analysis results.

```
# Measuring execution time for a block of expressions with system.time()
system.time({
  rnorm(1000000, 0,1)
  lm(mpg ~ ., mtcars)
})
  user  system elapsed
  0.074   0.002   0.077
```

- We can see there're 3 different time measures in seconds:
  - ✓ *User time* refers to the CPU time charged for the execution of user's instructions except those codes requesting system resources.
  - ✓ *System time*, on the other hand, refers to the time of user's instructions that requests for the access to the system resources, such as files in hard drive or memory spaces.
  - ✓ *Elapsed time* is literally the elapsed time taken to execute the given expressions. Note that it may longer than you expect, as your code/instruction may wait for resources occupied by other running processes of the OS.

Profiling your R code(cont.)

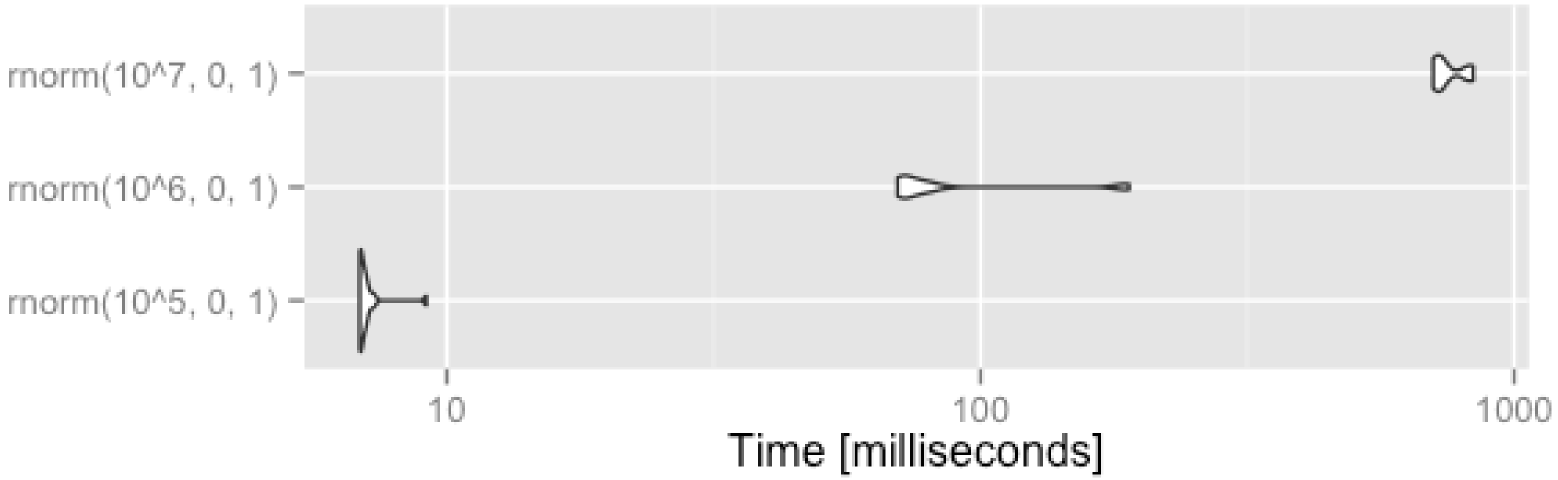
- You may have noticed a drawback of using *system.time*()—we do not know the execution time for each expression in the block! Package *rbenchmark* could help. Another important feature of it is that we can evaluate the code multiple time to get a sense of execution time distributions.

```
library(rbenchmark)
# Replicate & run each statement for 10 times.
bm10t = benchmark(
  rnorm(1000000, 0,1),
  lm(mpg ~ ., mtcars),
replications = rep(1,10))

# Get average elapsed time for each statement
by(bm10t$elapsed, INDICES = list("Elapsed time" = bm10t$test), mean)
Elapsed time: lm(mpg ~ ., mtcars)
[1] 0.0017
------------------------------------------------------------------------
Elapsed time: rnorm(1e+06, 0, 1)
[1] 0.0694

# Note that your output may be different from the above!
```

Profiling your R code(cont.)

- Or, using package *microbenchmark* makes our life easier. It also provides graphic functions used to compare executions times for different expressions.

```
library(microbenchmark) # Using microbenchmark
mb = microbenchmark(
  rnorm(10^5, 0,1), rnorm(10^6, 0,1), rnorm(10^7, 0,1),
  times = 20, unit = "s"); mb
autoplot(mb)
```



More about R Profiling

- Practically, our R program is often complex and interwoven with a sequence of selections, loops, function calls,…, etc. Using aforementioned benchmark tools may not be able to identify what makes our program slow. Consider the following example.

```r
# Different implementation of functions that compute sample mean
my_summary = function(d){
  # loop-based
  loop_mean = function(x){
    total = 0
    for(i in 1:length(x)) total = total + x[i]
    return(total / length(x))
  }
  # vectorized
  vector_mean = function(x) sum(x)/length(x)
  # built-in function
  builtin_mean = function(x) mean(x)
  # print result
  print(loop_mean(d)); print(vector_mean(d)); print(builtin_mean(d))
}
```

More about R Profiling(cont.)

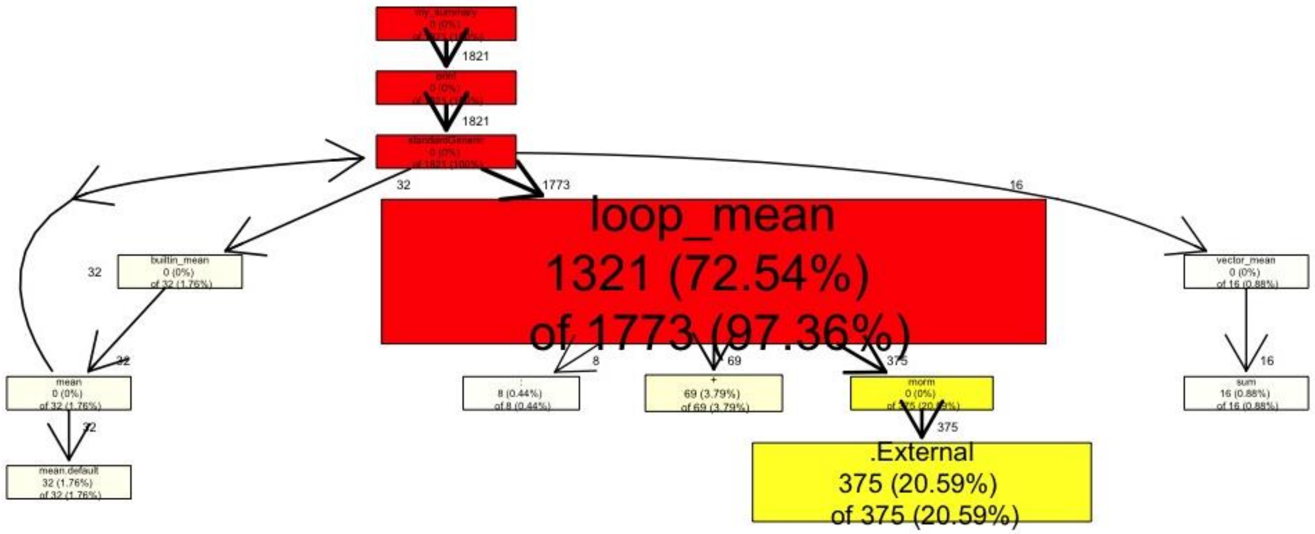
- There are 3 different implementations of "*mean*" function in previous page. We expect the one that uses *for* loop would take longer. Function *Rprof*() could help measure the execution time and memory consumption of each function/statement in *my_summary*(). Also note 2nd *Rprof*() is used to stop R profiling.

```
# Rprof() is used to measure execution time and memory usages
Rprof("profile.txt", memory.profiling = T)
my_summary(rnorm(10^8,0,1))
Rprof(NULL); summaryRprof("profile.txt", memory = "both")
$by.self
               self.time self.pct total.time total.pct mem.total
"loop_mean"        27.22    72.09      36.74     97.30   19933.6
".External"         7.76    20.55       7.76     20.55     762.9
...
$by.total
                  total.time total.pct mem.total self.time self.pct
"my_summary"           37.76    100.00   19952.6      0.00     0.00
"print"                37.76    100.00   19952.6      0.00     0.00
"standardGeneric"      37.76    100.00   19952.6      0.00     0.00
"loop_mean"            36.74     97.30   19933.6     27.22    72.09
...
# "by.self" doesn't include the elapsed time of other functions called
by our my_summary().
```

More about R Profiling(cont.)

- Visual representations of R profiling result is also available.

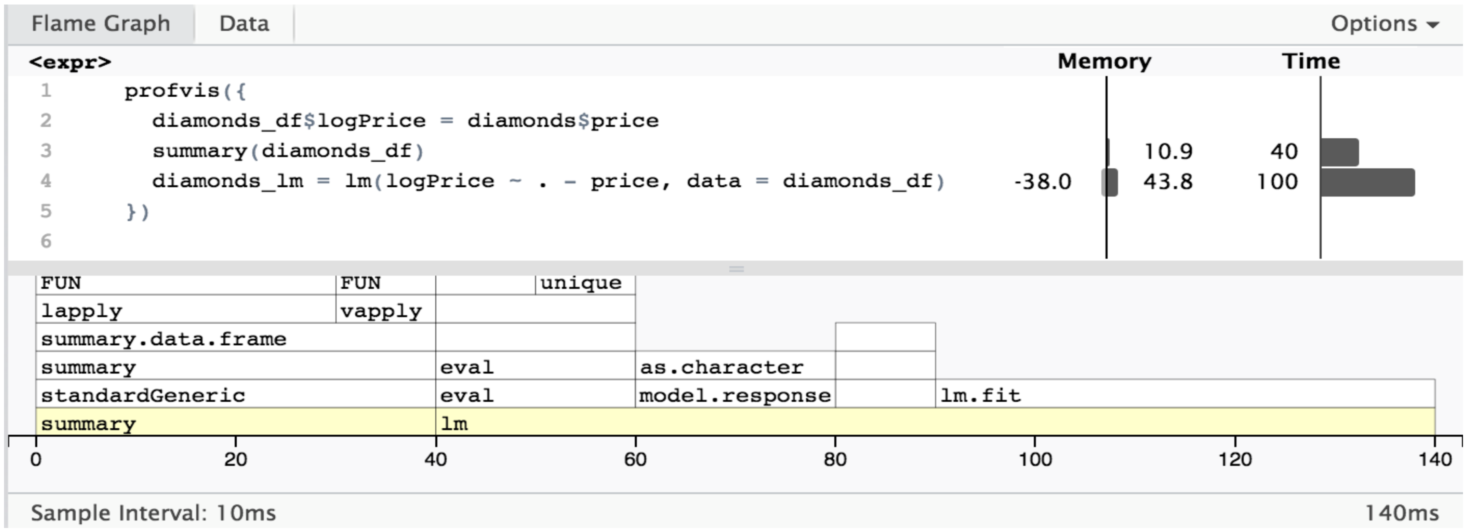
```
# Using package proftools
install.packages("proftools", type = "source")
source("http://bioconductor.org/biocLite.R")
biocLite(c("graph", "Rgraphviz"))  # packages on Bioconductor.org
Map(require, c("proftools", "graph", "Rgraphviz"), character.only=T)
Rprof("profile_no_mem.txt"); my_summary(rnorm(10^8,0,1))
Rprof(NULL); summaryRprof("profile_no_mem.txt")
p = readProfileData("profile_no_mem.txt")
plotProfileCallGraph(p, style=google.style, score="total")
```



More about R Profiling(cont.)

- Another package *profvis* developed by RStudio.com provides a better interactive R code profiling interface. You may also just click the RStudio menu "Profile" -> "Start Profiling" to start profiling.

```r
library(profvis) # Using package profvis
diamonds_df = as.data.frame(diamonds)
profvis({
  diamonds_df$logPrice = diamonds$price
  summary(diamonds_df)
  diamonds_lm = lm(logPrice ~ . - price, data = diamonds_df)
})
```



```
<expr>                                                          Memory          Time
1   profvis({
2     diamonds_df$logPrice = diamonds$price
3     summary(diamonds_df)                                             10.9     40
4     diamonds_lm = lm(logPrice ~ . - price, data = diamonds_df)  -38.0  43.8    100
5   })
6
```

```
FUN                FUN          unique
lapply             vapply
summary.data.frame
summary            eval         as.character
standardGeneric    eval         model.response          lm.fit
summary            lm
0        20        40        60        80        100        120        140
Sample Interval: 10ms                                                140ms
```

Try It!

- ❑ Load the dataset *ggplot2::diamonds* and perform a series of principle component analysis processes on it numeric variables. Try to profile your R code and see which part(s)/expression(s) takes longer execution time and consumes bigger memory space.

Tips to make better 'n faster R code

- You don't have to be an R expert to write high-performance R code. Here we have some simple tips that just require a little effort to make your code faster & clearer:
  - Being *vectorized*. Those array programming languages like R and MATLAB can directly manipulate vectors and matrices without bothering with looping around scalar values.
  - Being *functional*. R was born "functional", which means it support *first-class functions* that allow you to do any manipulations on functions (e.g. store them in a list and pass them to other functions), just like what you are doing with vectors.
  - Being *primitive*. Try to use as simple & native data structures (e.g. those data structures implemented in C/C++) as possible. If we only need a numeric vector to process our data, why should we use a data frame?

Vectorization

- Vectorization is about "thinking in vectors". Many array programming texts and literature suggest that we should always replace loops with vectorized forms if possible. The same applies to R. Unlike many other languages such as C++ and Java, the very basic data type of R is actually a *vector*. Most of our daily data manipulation tasks, especially iterations, could be replaced by a series of vectorized operations. Consider the following example that uses loops, just like we normally do in C++ or Java.

```
# Get lagged(1) differences of a vector.
set.seed(1)
rand_1to10 = sample(1:10, 1000, T)
lagged_rand = numeric(999)
library(microbenchmark)
# Loop
microbenchmark(
  for(i in 1:999){
    for(j in 2:1000) lagged_rand[i] = rand_1to10[j] - rand_1to10[i]},
  times = 10, unit = "s")
Unit: seconds
expr      min        lq        mean     median        uq       max     neval
 for... 1.081075  1.100092  1.125811  1.107559   1.121421  1.291934    10
```

Vectorization (cont.)

- How about we do it in vectorized forms?

```
# Vectorized
microbenchmark(
  lagged_rand = rand_1to10[2:1000] -  rand_1to10[1:999],
  times = 10, unit = "s"
)
Unit: seconds
        expr        min         lq         mean       median         uq         max neval
 lagged_rand 1.2057e-05 1.2394e-05 1.34987e-05 1.27385e-05 1.2914e-05 2.0756e-05    10

# How about using built-in function, diff()
microbenchmark(
  lagged_rand = diff(rand_1to10, 1),
  times = 10, unit = "s"
)
Unit: seconds
        expr        min         lq         mean       median         uq         max neval
 lagged_rand 5.1793e-05 5.2203e-05 5.88249e-05 5.29495e-05 5.3107e-05 0.00010965    10
```

- We can see that vectorized forms of the code is much more concise and fast!

Vectorization (cont.)

- You may explicitly "vectorize" your functions that deal with scalar values or other unique R objects.

```r
# Function to generate key-value pairs(a letter and its size) using loop
# The object.size() is not a vectorized function
genKV = function(keys, values){
  kv_pairs = list(); kv_len = length(keys)
  for(i in 1:kv_len) kv_pairs[keys[i]] = object.size(values[i])
  return(kv_pairs)
}
genKV(letters, 1:26) # Create the function that works on scalar value
genKV_v = function(k, v){ kv = list(); kv[k] = object.size(v); return(kv) }
genKV_v(letters, 1:26) # It doesn't work as we expected
# Say, we'd like to vectorize the function genKV_v()
genKV_v2 = Vectorize( genKV_v, vectorize.args = c("k", "v"), USE.NAMES = T)
genKV_v2(letters, 1:26)
Map(genKV_v, k = letters, v = 1:26) # Or we can just use Map()
```

- And, yes, the vectorization of common iterations on scalar values may significantly improve the performance of your code. However, it doesn't mean all vectorizations are necessary. Such *Vectorize*() should be used to simply your code, not intended to replace loops!

Try It!

- ❑ Please download the top 10,000 English words as below:

```
download.file("https://raw.githubusercontent.com/first20hours/google-
10000-english/master/google-10000-english.txt","english_10000.txt")

eng_10k_file = read.table("english_10000.txt")
```

Compute the pairwise *Levenshtein distances*. Try to vectorize and profile your R code to make it run faster!

R Functional Programming

- R supports three important features of *functional programming* (FP) languages—*anonymous function*, *closures*, and *lists of functions*. These FP features are often twined together with the vectorizations and distributed computing to form powerful, concise, and blazingly fast R programming styles!

- To begin the FP styles of R programming, we should always start with very simple functions, combine them together into a more complex nested function blocks, test them from the bottom to the top, then apply them with confidence. The idea behind the FP styles is "*Don't Repeat Yourself*" (DRY), i.e. "*Every piece of knowledge must have a single, unambiguous, authoritative representation within a system*".

Anonymous Function

- Let's begin our journey to the R's FP styles. We understand everything, even a function, in R is actually an object. An object is usually bound to a name. However, R allows you to create & use a function without a name—an *anonymous function*.

```
# Using anonymous functions
(function(x) x^3)(2) # 2^3
(function(x, y, z) x+y+z)(1,2,3) # 1 + 2 + 3
# More complicated example. DO NOT do it.
(function(oper) if(oper == "square")
   function(x) x^2 else function(x) x^3)("square")(2)
[1] 4

# Give the function a name
pow = function(oper)
   if(oper == "square") function(x) x^2 else function(x) x^3
pow("")(2)
[1] 8

# Anonymous function also has formals() & body()
formals(function(x, y) x^2 + y^2)
body(function(x, y) x^2 + y^2)
```

Anonymous Function (cont.)

- The anonymous function is usually used with *functionals*—functions that take functions as inputs and outputs. The most common functionals in R are the "apply-family" functions.

```
# Using anonymous function with functionals
lapply(movies, function(x) any(is.na(x)) ) # any NA in any columns?

# My own lapply(), a functional, with 1 anonymous function
my_lapply = function(x, FUN) for(i in colnames(x))
  {l <- list(); l[[i]] <- FUN(x[i]); print( l )}

# Equivalent to lapply() but just to print the results
my_lapply(ggplot2movies::movies, function(x) any(is.na(x)) )

# My nested functional with 2 anonymous functions
my_nested_fun = function(x, FUN1, FUN2) FUN1(FUN2(x))

# A example of using of the nested functional:
# create a matrix with the same size of the input data
my_nested_fun(mtcars, function(x) matrix(nrow=x[1], ncol=x[2]),
function(x) dim(x) )
```

Closures

- The anonymous functions are small functions created for temporary use. *Closures* are also similar functions but are created by parent (environment) functions—a concept of *function factory*.

```
# A function to create functions to get a value near n percentile
closeToNPercentile = function(n){
  function(v) v[which.min(abs(v - quantile(v, probs = n/100)))]
}
# Creating functions that get a value close to 10% or 25% percentile
closeTo10Percentile = closeToNPercentile(10)
closeTo25Percentile = closeToNPercentile(25)
closeTo10Percentile(mtcars$mpg)
[1] 14.3
closeTo25Percentile(mtcars$mpg)
[1] 15.5
```

- Another important feature of Closures is that, as they are generated by parent functions, they are certainly *enclosed* with the objects of parent functions, i.e. closures have access to the objects of their parent functions!

Closures(cont.)

- *"An object is data with functions, A closure is a function with data"*. Such feature of "access to parental objects" allows us to create static functions that could update & keep information (*state*) every time the functions have been called—*Mutable State* of FP, which creates a powerful & flexible style of R function calls.

```
# An closeToNpercentle() factory but with a variable to count
# "how many times such functions are called by users".
closeToNPercentile_ct = function(n){
  counter = 0
  function(v){
    counter <<- counter + 1
    print(paste("This function has been called", counter, "time(s)."))
    v[which.min(abs(v - quantile(v, probs = n/100)))]
  }
}
closeTo10Percentile = closeToNPercentile_ct(10)
closeTo25Percentile = closeToNPercentile_ct(25)
closeTo10Percentile(mtcars$mpg)
[1] "This function has been called 1 time(s)."
[1] 14.3
closeTo10Percentile(mtcars$mpg)
closeTo25Percentile(mtcars$mpg)
```

Closures(cont.)

- Almost every function in R is a closure, even those functions without explicitly-defined parent function. Yes, these functions' "parent" is R *global environment*. Also, we have been using "<<-" operator to get access to parental objects. It could surely use in daily data management tasks (but please use with caution!)

```
# Let's say we'd like to standardize r1 to r10 in "movies"
library(ggplot2movies); data(movies)
col2replace = paste("r", 1:10, sep="") # r1 to r10

# Using bulit-in function It's a bit slow.
movies[, col2replace] = scale(movies[, col2replace])

# Using loop with a few iterations is faster.
for(i in col2replace) movies[, i] = scale(movies[, i])

# Using anonymous closure with access to parental dataset
Map(function(x) {movies[,x] <<- scale(movies[,x]);},
 list(col2replace))
```

Lists of Functions

- We have been discussing how to create different kinds of functions, but how to store them? As functions in R are also serializable objects, they can be stored in an R list and serialized as raw binary values.

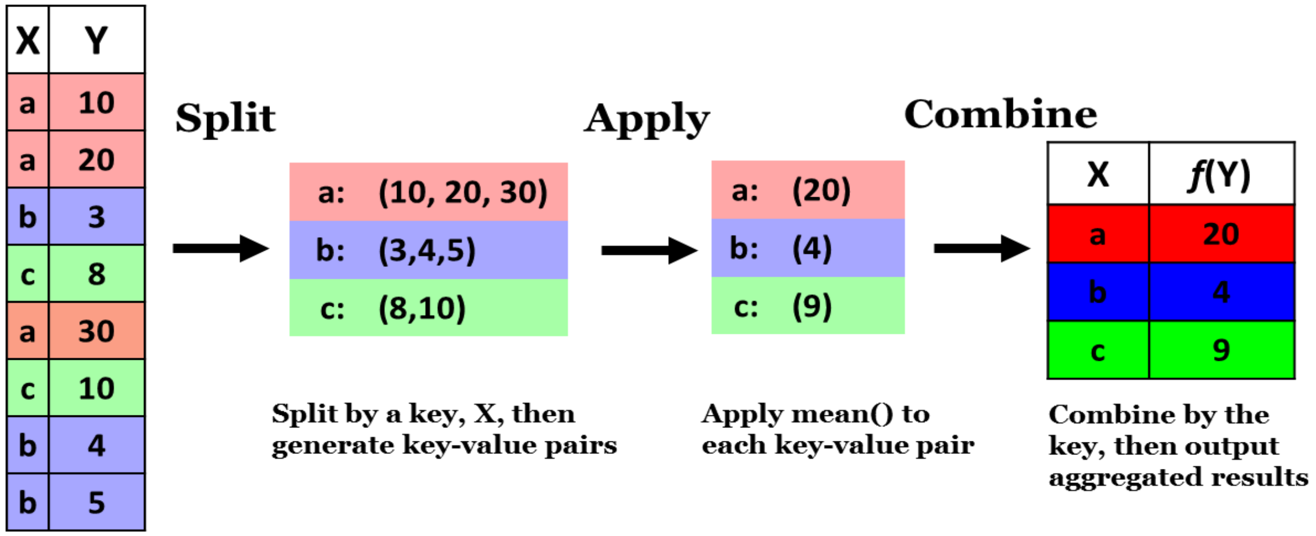
```r
# Any function can be stored in an R list
sum_fun = list( min, max, mean, sd, median)
# Serialized functions are just raw binary values stored in a vector
s_sum_fun = serialize(sum_fun, NULL)
# Serialized object can be stored in a hard drive or transmitted
# over the Internet, then be "unserialized" at the received ends.
unserialize(s_sum_fun)

# Lists of functions in practice. A user-defined summary()
my.summary = function(df){
  sum_fun = list(min, max, mean, sd, median)
  Map(function(f){ sapply(df, function(v) f(v, na.rm = T) )},
    sum_fun)
}
my.summary(mtcars)
```

"To become significantly more reliable, code must become more transparent. In particular, nested conditions and loops must be viewed with great suspicion. Complicated control flows confuse programmers. Messy code often hides bugs."

Bjarne Stroustrup

More about Functionals

- As mentioned previously, a *functional* is a function that take function(s) as input and output. It is often considered a complement of closure and a replacement of loop. We have created a specialization of functionals previously, *my_lapply*() that imitates *lapply*() of "*apply*"-family functions, which implement the strategy of *split-apply-combine* for data analysis.


X
Y
a 10
a 20
b 3
c 8
a 30
c 10
b 4
b 5
Split
a: (10, 20, 30)
b: (3,4,5)
c: (8,10)
Split by a key, X, then generate key-value pairs
Apply
a: (20)
b: (4)
c: (9)
Apply mean() to each key-value pair
Combine
X
f(Y)
a 20
b 4
c 9
Combine by the key, then output aggregated results

- Such "apply" functions create a container for output, apply *f*() to each component, and fill the container with results—a loop-like pattern that reduces bugs in our codes, replaces common loops, and conveys higher-level goal of data analysis.

Functionals vs. Loop

- Loops in R are usually considered slow and meaningless as they are just used to loop over elements of a vector. In most cases, you may replace loops with functionals to make your code more efficient.

```
# A trivial example to demonstrate loop patterns using
# functionals instead of loop.
m = matrix(rep(0,10^6), ncol = 10^3)
microbenchmark(
  for(i in 1:10^3) for(j in 1:10^3) m[i,j] = i + j,
  times = 5, unit = "s")
microbenchmark(
  Map(function(i){
    Map(function(i, j){
      m[i, j] <<- i + j
    }, i, list(1:10^3))
  }, list(1:10^3) ),times = 5, unit = "s")
# Add "print(length(j))" before "m[i,j] ..." and see what happened!
```

- Using nested *Map*() function is faster simply because it vectorizes the inner *Map*() and thus its output is also "vectorized". It suggests we cannot always replace loops in the cases that the output is a scalar object!

Map, Reduce, and Filter

- Almost all functional programming languages like R have these 3 functionals used to alter, collapse, and subset complex data structures. We have been using *Map*() a lot, and thus we here will be focusing on the roles of *Reduce*() and *Filter*() in our daily data management.

- *Reduce*() reduces a set of elements into a single element by calling a function of two arguments to combine two elements recursively.

```
# With "accumulate = T" to return result of each step
Reduce( function(v1, v2) v1 + v2, 1:5, accumulate = T)
[1]  1  3  6 10 15

# Folding from "right to left"
Reduce( "+", 1:5, accumulate = T, right = T)
[1] 15 14 12  9  5

# Recursive set operations
setOfNum = list(1, 4, 6, c(1,3,5,7,9), c(3, 5))
Reduce(union, setOfNum)
```

Map, Reduce, and Filter(cont.)

- Using *Reduce*() is also an elegant way to convert two-input functions into functions that take any number of inputs.

```
# Reduce() in action
L1 = list(1:10); L2 = letters[1:10]; L3 = list(1:5)
Reduce(function(x,y) cbind(data.frame(x), data.frame(y)),
   list(L1, L2, L3))

library(randomForest)
# Let's say we have 3 different random forests trained by
# different nodes/machines.
rf10 = randomForest(mpg ~ . , data = mtcars, ntree = 10)
rf20 = randomForest(mpg ~ . , data = mtcars, ntree = 20)
rf30 = randomForest(mpg ~ . , data = mtcars, ntree = 30)
# Combine these forests to form a bigger one
rf60 = Reduce(randomForest::combine, list(rf10, rf20, rf30)); rf60
# We can surely use Map() to generate a list of forests
# used here to get the result.
Reduce(function(f1, f2) combine(f1,f2),
       Map(function(nt) randomForest(mpg ~ . , data = mtcars,
             ntree = nt), list(10, 20, 30)))
```

Predicate Functionals

- Functionals that take an input function that ONLY returns TRUE or FALSE (e.g. is.* functions in R) are called *predicate functionals*. *Filter*()/*Position*() are such common functions that literally "filter out" / "find the position of" values in a vector with such input function (also called *predicate*).

```
list_w_na = list(x1 = c(0,1,3), x2 = c("a", NA, "0"),
   x3 = c(4,5,NA))

# Keep x1 values > 0
Filter(function(x) x > 0, list_w_na$x1)

# Keep NOT (x1 values > 0) with Negate()
Filter(Negate(function(x) x > 0), list_w_na$x1)

# Keep all values > 0
Map(function(v) Filter(function(x) x > 0, v), list_w_na)

# Find the positions of "NA"s
Map(function(v) Position(is.na, v), list_w_na)
```

Functionals in parallel

- You may have seen that the concept of functionals have been widely-used in various real-world applications, especially in parallel and distributed computing. In addition to those famous big data analytics framework(e.g. H2O and Apache Spark), R does provide some useful built-in features that allow you to run your functions in parallel.

```r
library(parallel); library(randomForest) # Package "parallel"
# Use all CPU cores may freeze your machine!
cl = makeCluster(detectCores() - 1)
# Export randomForest to all R instances
clusterExport(cl, "randomForest")
# Train a forest with different formulas
microbenchmark(
  Map(function(f) randomForest(f, data = diamonds, ntree = 10),
      list(log(price) ~ carat, log(price) ~ y , log(price) ~ cut,
        log(price) ~ clarity)), times = 10, unit = "s")
microbenchmark(
  parLapply(cl, list(log(price) ~ carat, log(price) ~ y ,
        log(price) ~ cut, log(price) ~ clarity),
        function(f) randomForest(f, data = diamonds, ntree = 10)
         ),times = 10, unit = "s"
)
stopCluster(cl) # Stop all worker instances
```

"… premature optimization is the root of all evil…"

Donald E. Knuth

Being Primitive

- R provides various built-in data types that work differently in terms of memory & computation consumption. Being a bit parsimonious & simple is always a good practice when we deal with big datasets. Here is a simple ranking of these data types:

Data Frame > **List** > **Matrix** > **Vector**

- Unfortunately, most of R built-in data types, such as data frame, are not designed for processing massive datasets since R was born. Various data structures with packages (e.g. in-memory columnar data tables and memory-mapped files), especially those implemented in compiled languages like C/C++, have been created to facilitate big data managements. Some of them (e.g. *tibble* and *data.table*) are even compatible with built-in data types! It is recommended to use them instead of creating your own data types.

Data wrangling with *dplyr*

- Package *dplyr,* which is considered a new version of *plyr* developed by Wickham and Francois, proposes an advanced data frame-like data structure, introduces a grammar of data manipulations, focuses more on inputs/outputs of tabular data, and promotes *Tidy Data*.

- A tidy data is simply a dataset ready to be analyzed (e.g. model fitting), where "*each variable is saved in its own column, and each observation is saved in its own row*." As you may expect, the concepts of the tidy data is closely related to Codd’s relational algebra & database. A tidy dataset can here be considered a relation that meets 3rd Normal Form.

- The key object of *dplyr* is a *tibble*, a efficient representation of a tabular (data frame-like) data structure implemented in C++. The *dplyr* supports a set of data management verb functions to "pipe" tibbles and the same interface to work with data no matter where it's stored, whether in a data.frame, a data.table, a table in relational databases, or even a Apache Spark DataFrame!

Defining your *tibble*

- Before managing data using *dplyr*, we first need to define our data "tibble". As discussed previously, the tibble can be from any data sources, even a table in a remote relational database.

```
library(dplyr)
# Say, our data is simple an R data frame. "mtcars" again?
# Convert data frame into "tibble". "mtcars" has become a "tbl"
mtcars_tb = as_tibble(mtcars); class(mtcars_tb)
glimpse(mtcars_tb, width = 80 ) # Get a "glimpse" of a tibble
print(mtcars_tb, n = 30) # 30 rows to show

# Using contains() and between(). Only works on local tibbles
mtcars_tb[dplyr::between(mtcars_tb$mpg, 20, 25),]
mtcars_tb[dplyr::contains("toyota", ignore.case = T,
   vars = rownames(mtcars_tb)),]
slice(mtcars_tb, 3:5) # 3-5th rows

# Just like the way we create R data frame
aTibble = tibble(x = letters[1:5], y = lubridate::today(),
   z = runif(5,0,1))
# More efficient versions of rbind() and cbind()
bind_rows(aTibble, aTibble)
bind_cols(mtcars_tb, tibble(rowNum = 1:nrow(mtcars)))
```

Defining your *tibble*(cont.)

```
library(RSQLite) # If our dataset is in an SQLite database
# Create a temporary in-memory SQLite database with a table for "movies"
con = dbConnect(RSQLite::SQLite(), dbname = ":memory:")
# Or, create a SQLite embedded database file in hard drive
# con = dbConnect(RSQLite::SQLite(), dbname = "./myDB.Sqlite")
dbWriteTable(con, "movies", ggplot2movies::movies )
dbListTables(con) # list tables in the database
# Get result of a query as a data frame (a SQL pass-through query)
movies_df = dbGetQuery(con, "select * from movies")

# Say we'd like to directly manipulate tables in the database
sqliteDB = dbplyr::src_dbi(con, auto_disconnect = F)
# movies & movies_8 are simply pointers to the tables in the database!
movies_tb = tbl(sqliteDB, "movies")
movies_tb_8 = movies_tb %>% filter(rating > 8) %>%
   select(title, year, rating)
# tbl_lazy" denotes the "lazy evaluation" of the data manipulation
class(movies_tb_8)
# Both tibbles are simply "pointers" to the tables in the database
# The size of them are relatively smaller than original data frame
pryr::object_size(ggplot2movies::movies)
pryr::object_size(movies_tb)
# We can surely check out the query plan
dplyr::explain(movies_tb_8)
```

Piping your *tibble*

```
# "movies_8" actually does not exist.
# Let's execute and store result in remote database server
sqliteDB # or run dbListTables(con)
compute(movies_tb_8, name = "movie_8_table"); sqliteDB
# We can surely download/copy the table from the database
movies_8_df = collect(movies_tb_8)
# How about the other way around? Copy a data frame to remote database.
copy_to(dest = sqliteDB, df = mtcars,
   name = "mtcars_table", temporary = F)
dbListTables(con)
```

- The *dplyr*'s own pipe operator "*%>%*" allows us to pipe data tibble output from one function to the input of another function. Instead of using nested functions with multiple parentheses, processing data flow with pipe operator improves readability of our code.
- In RStudio, we can use keyboard shortcut: **Ctrl** + **Shift** + **M** (Windows & Linux), **Cmd** + **Shift** + **M** (Mac) to insert "%>%". Also notice that "%>%" passes tibble on left-hand side as first argument (or ".") of function on right-hand side. And x %>% f(y) is the same as *f(x, y)*, *y %>% f(x, ., z)* is the same as *f(x, y, z)*.

Piping your *tibble*(cont.)

- To provide a unified interface able to work on various data sources, *dplyr* and related packages actually convert our data piping flow (a chain of verb functions) into a series of SQL-like operations. Let's first consider *select*() along with select-helper functions(enter "*?select_helpers*" for more information), which is able to do more than just column selection by their names.

```
# Select by names. Equivalent to select(movies_tb, title, length, rating)
movies_tb %>% select(title, length, rating)
# title with r1-r5
movies_tb_r1to5 = movies_tb %>%
   select(title, num_range(prefix="r", range = 1:5))
explain(movies_tb_r1to5)

# all variables except r1 to r10
movies_tb %>% select(-(r1:r10))
# title with any variable names that contains numbers
movies_tb %>% select(title, matches("[[:digit:]]"))
# select and rename
movies_tb %>% select(title, rating_1to10 = rating )
```

Piping your *tibble*(cont.)

- Another common data operations is to filter datasets. *filter*() should suit your need. However, if you use *dplyr* to work with database servers, be aware that any functions not supported by the database engines CANNOT be used in filter condition statements.

```
# A local tibble. "movies" is already a tibble.
movies_tb_local = as_tibble(ggplot2movies::movies)
# The good, the bad, and the ugly(very long movie title?)
movies_tb_local %>% select(title, length, rating ) %>%
  filter(rating > 9 | rating < 2 | nchar(title) > 80) %>%
  arrange(desc(nchar(title)))
# Give me "Start Trek" series!
movies_tb_local %>% select(title, length, rating) %>%
  filter(between(length, 60, 150),
    base::grepl("star trek", title, ignore.case = T))
# Note that "grepl()" doesn't support by SQLite. Below will show an ERROR !
movies_tb %>% select(title, length) %>% filter(grepl("star trek", title,
ignore.case = T))
...
Error in rsqlite_send_query(conn@ptr, statement) :
...
   Named arguments ignored for SQL GREPL
```

Piping your *tibble*(cont.)

- As the successor of *plyr*, *dplyr* provides a set of SQL-like aggregation functions that allows you to pipe datasets with split-apply-combine patterns.

```
# Split by "longShort" (Grouped by: longShort)
movies_longShort = movies_tb %>%
   group_by(longShort = ifelse(length > 130, "long", "short"))
# Apply n() and Combine. Type ?summarise for more aggregation functions
movies_longShort = movies_longShort %>% summarize(ct = n())

# Movies rating counter
movies_rating_ct = movies_tb %>% select(rating) %>%
   group_by(round_rating = round(rating, 0)) %>%
   dplyr::summarise( numOfObs = n()) %>%
   arrange(desc(round_rating)) %>% head(n = 5)
# Equivalent to the SQL explained below
dplyr::explain(movies_rating_ct)

# Add new columns with mutate(). Only keep new columns with transmute()
movies_tb %>% select(title, length, rating) %>%
  mutate(lenOfTitle = nchar(title), round_rating = round(rating, 0))
# Sampling with sample_n() and sample_frac(). May only work on local R tibbles
movies_tb_local %>% select(title, length, rating)
   %>% sample_frac(size = 0.7, replace = F)
```

Reshaping your tibbles with *tidyr*

- Another package often used with *dplyr* is *tidyr*, a successor of *reshape2*. As expected, it provides functions that convert data from wide/long to long/wide formats. One additional feature is function *unite*() and *separate*(), which unite columns into one and separates one column into multiple columns respectively. Also notice that functions in *tidyr* ONLY work on tibbles in local R environment.

```
library(tidyr) # Reshaping your tibbles with tidyr
reshape_tb = tibble(ID=c(1,1,1,2,2,3), time=c("t1", "t2", "t3", "t1", "t3",
   "t1"), var1=c(2,4,2,1,3,2), var2=c(3,1,3,2,4,3))
# from "wide" to "long" format, and from "long" to "wide" format
gathered_tb = reshape_tb %>% gather(., key = "var", value = "val", fill= 3:4) %>%
   arrange(., ID, time)
gathered_tb %>% spread(., key = "var", value = "val")
# Copy "diamonds" to SQLite database
diamonds_tb_local = ggplot2::diamonds %>% mutate(logPrice = log(price)) %>%
   select(-price)
diamonds_tb_remote = copy_to(sqliteDB, diamonds_tb_local,
                             "diamonds", overwrite = T, temporary = F)
# Unite columns into one using unite() and unite_()
diamonds_tb_local %>% tidyr::unite(col = xyz, x, y, z, sep = ",")
# Separate columns with separate()
hitters_tb_local = as_tibble(ISLR::Hitters) %>% mutate(hitter_name = rownames(.))
hitters_tb_local %>% select(hitter_name, Years) %>%
  separate(col=hitter_name,into = c("first_name", "last_name"), sep = " ")
```

Tips to use *dplyr* and *tibble*

- With help of package *dplyr*, we can work with massive datasets no matter whether they are data frames stored in memory or tables in remote database servers. We can also manipulate big data even if it is too big to fit in one single machine. Another advantage is that complex data processing flows can be expressed as sequences of easy-to-understand data piping steps. However, there are still things to consider when using *dplyr*:
  - Most R functions cannot be used in piping verb functions unless our tibble is in local R environment.
  - Data piping patterns do not work on the cases when we have multiple inputs or outputs.
  - Working with remote database means that we put data process loading on the remote database server/cluster. And therefore the performance surely depends on the server.

Using data.table

- The R *data.table* package provides an in-memory columnar structure called *data.table*, which is compatible with data.frame but is much faster when the data is big. You can even replace data.frame with data.table for most of your data management!

- The syntax of using data.table is just like using data.frame, but its more flexible (and fun!). It comes with a general syntax, as:

  DT[where, select | update | do, by]

  It can be interpreted as “take data.table **DT**, subset/update rows or calculate using **select/update/do** grouped by "**by**".

- Remember to check out data.table Github for more data.table syntax!

Using data.table(cont.)

- A data.table object is usually smaller than data.frame, and its manipulation is relatively faster.

```r
# Load dataset "movies"
library(ggplot2movies); library(data.table)
movies_df = data.frame(movies, stringsAsFactors = F)
movies_dt = data.table(movies, stringsAsFactors = F)
# Dataset size are approximately the same in
# new version of data.table
Map(function(d) pryr::object_size(d),
    list("DF" = movies_df, "DT" = movies_dt))

# A simple test on filtering data
microbenchmark(
  nrow(movies_df[movies_df$rating > 6,]),
  times = 10, unit = "s"
)
microbenchmark(
  nrow(movies_dt[rating > 6]),
  times = 10, unit = "s"
)
```

Using data.table(cont.)

- The data.table borrows many features from existing packages to provide elegant and concise syntax.

```
# Save dataset "diamonds" as data.table
diamonds_dt = data.table(diamonds)
# Frequencies of cut by clarity. Here ".N" is used to count # of rows
diamonds_dt[, .N, .(cut, clarity) ]

# Find thoses big diamonds > 6mm in length, width, and depth
# setorder() is used for super fast row reordering of a data.table
diamonds_6mm = setorder(diamonds_dt[ (x > 6 & y > 6 & z > 6),]
   , carat); diamonds_6mm

# Equivalent to "select avg(price) as avg_price,
# count(*) as ct from diamonds_dt where carat > 0.5 group by cut"
diamonds_dt[carat > 0.5, .(avg_price = mean(price),
   ct = length(price), N=.N ), .(cut) ]

# data.table also supports melt() and dcast()
melted_genre = setorder(data.table::melt(movies_dt, id = 1,
   measure=list(18:24) ), title)
```

Using data.table(cont.)

- A few of typical SQL syntax are also supported.

```
# All existing data.tables in the R global environment
data.table::tables()
# Show me "Star Trek"
movies_dt[title %like% "Star Trek", .(title, rating)]
# Rating between 6 and 7
movies_dt[rating %between% list(8,9), .(title, rating)]
```

- Also, when you do data import tasks, check out super fast plain text data file reader/writer, *fwrite*() and *fread*()!

```
# Fast plain text data file reader and writer
fwrite(movies_dt, "movies_dt.csv", sep=",")
movies_dt_copy = fread("movies_dt.csv", sep=",",
   stringsAsFactors = F,fill = T, verbose = T)
# "Real" data set operations. Note that 13L & 15L are integers.
setTbl1 = data.table( "X1" = 1:5, "X2" = 11:15)
setTbl2 = data.table( "X1" = 4:5, "X2" = c(13L,15L))
fsetdiff(setTbl1, setTbl2)
fintersect(setTbl1, setTbl2)
funion(setTbl1, setTbl2)
```

Joining data.table

- Yes. It's time to join data.tables. The data.table provides its own *merge*() but the syntax is almost identical to *base::merge*(). One big difference is that we can create an index (set a key) on a data.table to further improve its performance.

```
# Joining data.table
tbl1 = data.table( "ID1" = sample(1:10^5), "X" = runif(10^5,0,1) )
tbl2 = data.table( "ID2" = sample(1:10^5), "Y" = runif(10^5,0,1) )
microbenchmark( merge(tbl1, tbl2, by.x = "ID1", by.y = "ID2"),
  times = 20, unit = "s" )
setkey(tbl1, ID1); setkey(tbl2, ID2)
microbenchmark( merge(tbl1, tbl2, by.x = "ID1", by.y = "ID2"),
  times = 20, unit = "s")
setkey(movies_dt, title)
# Access records by a "key"
movies_dt["Star Trek: Nemesis"]
```

- Function *setkey*() sorts a data.table in ascending order by the key column(s). The table is then marked as "sorted" by referenced key values and is therefore very memory efficient.

Updating data.table

- Another powerful feature of data.table is that it allows for adding and updating on multiple columns with the following syntax:

  c("LHS1", "LHS2", …, "LHSn") := list(RHS1, RHS2, …, RHSn)

  where left hand side (LHS) is a simple character vector that contains variable name(s). Right hand side (RHS), however, is a list of new or updated key-value pairs of data.

```
# Update with rounded values
diamonds_dt[,
   c("x", "y", "z") := list(round(x), round(y), round(z))]

# Update column names
setnames(diamonds_dt, c("x", "y", "z"),
   c("length", "width", "depth") )

# Delete columns by using NULL
diamonds_dt[, c("x", "y", "z") := NULL];
```

Advanced data.table operations

- We can "chain" data.table operations to avoid creating intermediate objects and unnecessary assignments.

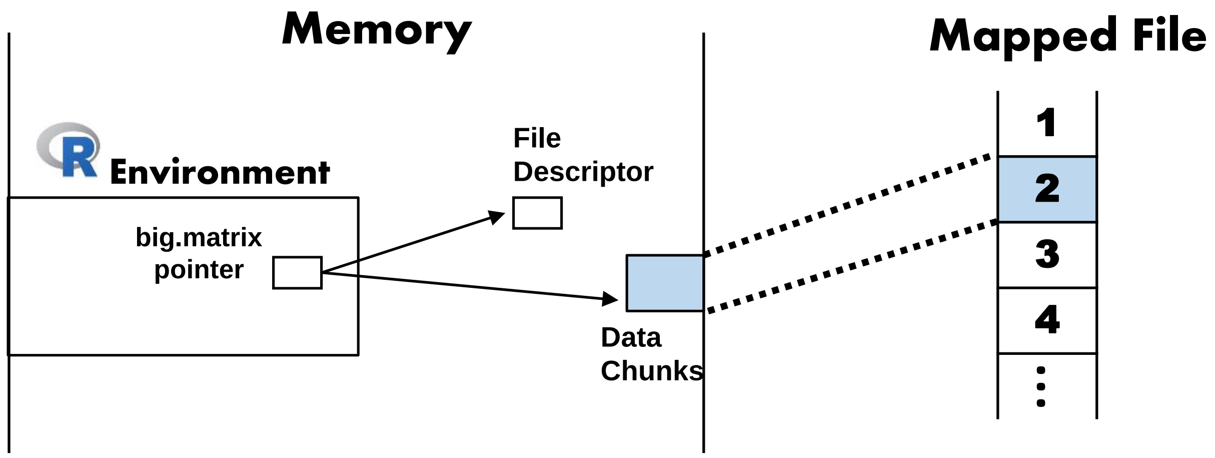
```
# Chainning data.table. Equivalent to "select cut, count(*)
# as N from diamonds_dt group by cut order by N desc"
diamonds_dt[, .N , .(cut)][order(-N)]

# More complicated example
diamonds_dt[, .N , .(cut, color)][order(-color, cut)]
   [color %in% c("G", "H") ]
```

- Using ".SD" keyword to get grouped temporary values of all columns except variables specified in "by".

```
# Print intermediate tables group by "color"
diamonds_dt[, print(.SD) , .(color)]
# Count the number of rows
diamonds_dt[, nrow(.SD) , .(color)]
diamonds_dt[, .N , .(color)]
# Get max value of each columns grouped by "color"
diamonds_dt[, Map(max,.SD) , .(color)]
```

Using memory-mapped files

- The data.table is surely fast as it puts everything in the memory. How about if our datasets are "bigger-than-memory"? RAM is relatively cheaper today but chances are we need to deal with massive datasets (say 200GB?) without a cluster of computers or database servers.
- R embedded database computing packages may help. Another simple solution, which is also implemented by many proprietary software, is to save the datasets as *memory-mapped files* that store data chunks in virtual memory as files in hard drives and load data in need as if they are in primary memory.



Using memory-mapped files(cont.)

- In R, we can manipulate such files by using *"big"*-family packages.

bigmemory	Matrix-like data structure called *big.matrix* that allows us to manage massive dataset in matrix form either in main memory or as memory-mapped files. It supports double (8 bytes, by default), integer (4 bytes), short (2 bytes), and char (1 byte).
biganalytics	A companion package of *bigmemory* that provides summary statistics and generalized linear model functions that work on *big.matrix* objects.
bigtabulate	Tabulate and split-apply-combine functions
bigalgebra	BLAS routines for native R matrices and the *big.matrix* objects.

Using *big.matrix*

- The *big.matrix* objects can be stored as files in hard drive. Any operation on these objects will trigger the data chunk loading & unloading between the main memory and files. Good news is that all these data manipulations and computations are done behind the scene by functions in package *bigmemory*.

```
# Using "big"-family functions to create memory-mapped file
library(bigmemory)
bigIntegerMatrix = as.big.matrix( matrix(sample(1:5, 10^8,
   replace = T ), nrow = 10^4), type = "integer",
   backingfile = "bigIntegerMatrix.bm",
   descriptorfile = "bigIntegerMatrix.bm.desc")

# It's actually just a pointer to a file in hard drive.
bigIntegerMatrix; object.size(bigIntegerMatrix)

# The size of the actual files? 4 Bytes * 10^4 ~= 400,000,000 Bytes
list.files(pattern = "bigIntegerMatrix.bm");
file.info("bigIntegerMatrix.bm")
```

Using *big.matrix*(cont.)

- Because the matrix is simply a memory-mapped file in our hard drive, we can always keep the file and get it back into our R environment later.

```
# Remove all objects in R environment
rm(list = ls())
# Get the matrix back by loading the file descriptor first.
bigIntegerMatrix_desc = dget("bigIntegerMatrix.bm.desc")

# We have it back by "attaching" it again
bigIntegerMatrix = attach.big.matrix(bigIntegerMatrix_desc)
# Get the info. of the object
describe(bigIntegerMatrix)

# Get summary statistics of the matrix
library(biganalytics)
summary(bigIntegerMatrix[, 1:100])
colna(bigIntegerMatrix) # Numbers of NAs

# Remove the files if no longer needed
file.remove("bigIntegerMatrix.bm", "bigIntegerMatrix.bm.desc")
```

Using *big.matrix*(cont.)

- You may have notice a drawback of using *big.matrix* is that all the data must be in the same data type (e.g. double), which is the nature of all matrix-like data structures in R. Also, passing a *big.matrix* object to a function is "call-by-*reference*", not "call-by-*value*". After all, such memory-mapped object is actually a pointer to a binary raw file in an C++ data type.

- So, it might be a little tricky when using *big.matrix* in real-world applications. For example, a common R dataset used to create models may consist of data values of different types(e.g. factor and character). If we try to convert the dataset into *big.matrix* object, all the variable data types would be coerced into the same one ("double" by default).

```
# Let's say "diamonds" dataset is big
diamonds_bm = as.big.matrix(as.data.frame(ggplot2::diamonds))
Warning message:
In as.big.matrix(as.data.frame(ggplot2::diamonds)) :
  Coercing data.frame to matrix via factor level numberings.
```

Using *big.matrix*(cont.)

- This is a problem when we are fitting linear models with factor/categorical variables, which means we need to manually convert categorical variables into multiple indicator (dummy) variables. Thanks to powerful & comprehensive R communities, we can always find packages that do the jobs for us!

```
# Creating a big.matrix object ready for modeling
library(dummies) # using package dummies
# Remove orders
invisible( Map(function(var) diamonds_df[,var] <<-
   factor(diamonds_df[,var], ordered = F),
       c("cut", "color", "clarity")) )
diamonds_numeric = dummy.data.frame(diamonds_df, sep = "_")

# In case if some level names are not syntactically valid
colnames(diamonds_numeric) = make.names(colnames(diamonds_numeric))
# The variables in "diamonds" is now all numeric
str(diamonds_numeric)
# Creating big.matrix object
diamonds_numeric_bm = as.big.matrix(diamonds_numeric,
   type = "double")
```

Using *big.matrix*(cont.)

- Then, we now have the dataset in *big.matrix* and ready for modeling. The package *biganalytics* provides functions to fit generalized linear models. Also notice that the package *dummies* creates indicator variables for each level of a factor variable, which means, when we create models, we need to skip a level (indicator) as the reference level/group to avoid perfect multicollinearity problem!

```
# Fitting linear model. "Fair" as the reference level
diamonds_bigglm = bigglm(
  log(price) ~ carat + depth + table + x + y + z +
     cut_Good + cut_Very.Good + cut_Premium + cut_Ideal,
     data = diamonds_numeric_bm)
# We skipped "cut_Fair"
summary(diamonds_bigglm)
# It is eqivalent to...
diamonds_df$cut = factor(diamonds_df$cut, ordered = F)
summary(lm(log(price) ~ carat + depth + table +
   x + y + z + cut, diamonds_df) )
# Notice that we removed the "order" of variable "cut".
# Did we lost some information?
```

Before using *bigmemory*…

- No package/framework is perfect. There are surely pros & cons we should be aware:

- **Advantage**
  - ✓It's super-fast when processing real big data.
  - ✓No more maximum memory limitation.
  - ✓Dealing "big data" without big machine or cluster
  - ✓Passing a *big.matrix* to a function is "call by reference" (disadvantage?).

- **Disadvantage**
  - ✗Memory-mapped *big.matrix* consumes more disk space.
  - ✗Relatively slow when dealing with small data (of course…).
  - ✗Need to generate indicators for categorical variables.
  - ✗Until the end of 2015, it only supports generalized linear models.

Final comments

- In this unit, we have dived a little deeper into R's core and discuss some possible solutions to improve the performance of our R code when dealing with "big data". Three simple tips to tweak our code is to be *vectorized*, *functional*, and *primitive*.

- Vectorization is about identifying problems that can be vectorized, not about replacing all loops in our code. On the other hand, the idea behind R functional programming is to remove repetition, ambiguity, and redundancy in our code by "thinking in functionals", which is mainly to consider functions as primary objects that can be treated as input and output.

- Also, we talked about using simple data structures and even those implemented in compiled languages. However, no solution is perfect. There are always some tradeoffs we should consider before using such primitive data structures.