

# R - Big Data Tricks

Data Wrangling in R

# Overview

There are several different tricks and approaches for handling “big” datasets in R. Most of these tricks either improve the importing of large datasets and/or speed up tasks applied to large datasets.

## R is memory intensive

R is pretty much limited by the amount of memory on your computer (or cluster), and potentially the number of cores on your computer (or cluster).

Getting around memory limits

## Using Disk

The package `ff` can “trick” R into using disk space as virtual memory instead of relying on physical memory

<https://cran.r-project.org/web/packages/ff/index.html>

## Using Disk

There are `read.csv` and `read.table` functions for leveraging these `ff` objects.

These functions are `read.csv.ffdf` and `read.table.ffdf` respectively.

```
suppressPackageStartupMessages(library(ff))  
mon_ff= read.csv.ffdf(file="../data/Monuments.csv")  
class(mon_ff)
```

```
[1] "ffdf"
```

# Using Disk

New columns must be converted to the `ff` class:

```
mon_ff$newCol = ff(1:nrow(mon_ff))
mon_ff[1:4,1:7]
```

	name	zipCode	neighborhood	councilDistrict
1	James Cardinal Gibbons	21201	Downtown	11
2	The Battle Monument	21202	Downtown	11
3	Negro Heroes of the U.S Monument	21202	Downtown	11
4	Star Bangled Banner	21202	Downtown	11

  

	policeDistrict	Location.1	newCol
1	CENTRAL	408 CHARLES ST\nBaltimore, MD\n	1
2	CENTRAL		2
3	CENTRAL		3
4	CENTRAL	100 HOLLIDAY ST\nBaltimore, MD\n	4

# Using Disk

More info, with examples, can be found here:

- <http://hsinay.blogspot.com/p/big-data-analysis-using-ff-and-12-nov.html>
- [http://ff.r-forge.r-project.org/bit&ff2.1-2\\_WU\\_Vienna2010.pdf](http://ff.r-forge.r-project.org/bit&ff2.1-2_WU_Vienna2010.pdf)
- <https://www.r-project.org/conferences/useR-2007/program/presentations/adler.pdf>



## Leveraging sparsity

If your data is very sparse, e.g. contains a lot of 0s, then the `sparseMatrix` command in the `Matrix` package might be helpful

<https://stat.ethz.ch/R-manual/R-devel/library/Matrix/html/sparseMatrix.html>

You need to provide which rows and columns have non-zero values (and the actual values themselves)

# Leveraging sparsity

We've had to do this for genomics projects:

- <http://genomebiology.biomedcentral.com/articles/10.1186/s13059-016-1118-6>  
with corresponding code:
- <https://github.com/nellore/runs/blob/master/sra/v2/countjunctions.R>

# Leveraging sparsity

Run length encoding (RLE) is a popular approach for handling sparse vectors...  
the `dataFrame()` function in the `IRanges` package creates a more flexible extension of the `data.frame` class

```
x= rep(c(0,1,0,1,0), times=c(10000,4,10000,6,10000))  
length(x)
```

```
[1] 30010
```

```
rle(x) # base r
```

```
Run Length Encoding  
  lengths: int [1:5] 10000 4 10000 6 10000  
  values  : num [1:5] 0 1 0 1 0
```

# Leveraging sparsity

```
suppressPackageStartupMessages(library(IRanges))  
df= data.frame(x=x)  
DF = DataFrame(df)  
Rle(x) # from IRanges, s4-methods
```

```
numeric-Rle of length 30010 with 5 runs  
  Lengths: 10000      4 10000      6 10000  
  Values :      0      1      0      1      0
```

```
DF$x = Rle(DF$x) # note the capital R  
head(DF)
```

DataFrame with 6 rows and 1 column

```
      x  
<Rle>  
1      0  
2      0  
3      0  
4      0  
5      0  
6      0
```

# Leveraging sparsity

How much space did we save in memory?

```
s1=object.size(df)  
s1
```

```
240808 bytes
```

```
s2=object.size(Df)  
s2
```

```
3032 bytes
```

```
as.numeric(round(s1/s2))
```

```
[1] 79
```

You can see this will add up for large datasets. Note this doesn't rely on sparsity per se, but rather many repeated fields that cluster together.

# Leveraging Sparsity

We've also used these file encoding for genomics projects:

- <https://academic.oup.com/nar/article/45/2/e9/2953306> with package:
- <https://www.bioconductor.org/packages/release/bioc/html/derfinder.html>

# Using Databases

Jeff covered this yesterday. If you can store the really large dataset in a SQL database, then it's easy to create your own SQL queries to read in only a subset of the dataset.

Speeding up computing



# Parallel libraries

Almost all R functions use only a single core on your computer but you can leverage more cores. You can usually speed up code by using multiple cores on your computer.

- [doParallel](#)
- [Parallel Basics](#)
- [mclapply](#)
- [HPC](#)

## Parallel libraries

You can parallelize `lapply()` (or `sapply()`) statements with the `parallel::mclapply()` function.

You can parallel `for` loops with the `foreach()` with `%dopar%` syntax. [`foreach` with `%do%` are analogous to `for` loops]

# mclapply

Let's make some data

```
set.seed(34)
x = rnorm(10000)
datList = replicate(10000, sample(1:10000), simplify = FALSE)
class(datList)
```

```
[1] "list"
```

```
lapply(datList[1:5], head)
```

```
[[1]]
[1] 1700 2051 4768 5656 6004 3441
```

```
[[2]]
[1] 6783 3124 588 3511 2937 4465
```

```
[[3]]
[1] 6169 1489 5476 4587 4004 6012
```

```
[[4]]
[1] 1181 9154 1687 2279 1136 8789
```

```
[[5]]
[1] 8583 4311 4138 2445 1281 5442
```

```
head(x)
```

# mclapply

Let's try to run `mclapply()` and compare to the output of `lapply()`

```
system.time(lapply(datList, cor, x))
```

```
   user  system elapsed  
1.730   0.491   2.376
```

```
system.time(mclapply(datList, cor, x))
```

```
   user  system elapsed  
0.006   0.360   1.785
```

???

# mclapply

This only works on Linux/Mac, and not Windows : (

```
> system.time(mclapply(datList,function(y) cor(x,y),mc.cores=4))  
Error in mclapply(datList, sort, mc.cores = 4) :  
  'mc.cores' > 1 is not supported on Windows
```

# doParallel

The doParallel library works better on Windows:

```
suppressPackageStartupMessages(library(doParallel))  
system.time(foreach(i=1:10000) %do% cor(datList[[i]],x))
```

```
   user  system elapsed  
4.721    0.586    5.538
```

```
system.time(foreach(i=1:10000) %dopar% cor(datList[[i]],x))
```

```
   user  system elapsed  
1.709    0.497    3.898
```

Oops! You need to initiate a parallel backend first!

# doParallel

Implementing parallel backend:

```
registerDoParallel(cores=2)  
system.time(foreach(i=1:10000) %dopar% cor(datList[[i]], x))
```

```
   user  system elapsed  
1.844    0.547    4.267
```

This task was not particularly amenable to parallelization, but you get the idea of the code.

This is a good quick-start guide: <http://blog.aicry.com/r-parallel-computing-in-5-minutes/>

# Vectorization

Some tasks can be computed quickly just using vector or matrix-based calculations instead of iterating:

```
datMat = simplify2array(datList)
class(datMat)
```

```
[1] "matrix" "array"
```

```
dim(datMat)
```

```
[1] 10000 10000
```

```
system.time(cor(x, datMat))
```

user	system	elapsed
1.144	0.253	1.447



# Rcpp

The `Rcpp` package allows you to write C++ code and run it within R.

<http://gallery.rcpp.org/> <http://dirk.eddelbuettel.com/code/rcpp/Rcpp-introduction.pdf> <https://cran.r-project.org/web/packages/Rcpp/index.html>

[Note: I am not very good at C++, and it usually takes me a lot less time to just let code run longer than figure out how to do something in C]

# GPU computing

There are packages that can rely on “graphics processing units” aka video cards that can greatly speed up linear computations

<http://www.r-tutor.com/gpu-computing> <http://www.r-tutor.com/gpu-computing/clustering/distance-matrix>

[Note: I have never actually used this, and it usually takes me a lot less time to just let code run longer than figure out how to do something using the GPU]

Modifying data prior to importing

## Slicing and dicing large data

Say you have a large tab-delimited matrix that has dimensions: 100,000,000 (rows) x 500 (columns) called `foo.txt`

The easiest tricks are for Mac or Linux computers since you can utilize very fast linux file processing on text files prior to importing data into R

## (Linux) Piping

R can read in the output of Linux commands using the `pipe()` function.

```
dat = read.delim(pipe([some command]))
```

The most useful two linux functions are `awk` (for selecting rows) and `cut` (for selecting columns)

# awk

Here are a collection of useful “one liners” using `awk`

<http://www.pement.org/awk/awk1line.txt>

Note, I almost always have to google how to use `awk` to remember the exact syntax for specific tasks

# awk

Let's say we want to select all rows where the second column has the values "chr21"

The awk command looks like:

```
awk -F"\t" '$2 == "chr21" { print $0 }' foo.txt
```

Note in linux you would probably redirect this to a file, but R can handle this "on the fly"

# awk

```
> awkCall = "awk -F\"\\t\\\" '$2 == \"chr21\\\" { print $0 }' foo.txt"
> read.delim(pipe(awkCall),header=FALSE)
  V1      V2      V3      V4      V5      V6      V7      V8
1 row6 chr21 0.06470508 0.2104643 0.2630865 0.6748644 0.5838493 0.59531516
2 row7 chr21 0.69930753 0.9533248 0.7929459 0.7084059 0.5305247 0.37651206
3 row8 chr21 0.31752770 0.6364067 0.7774075 0.6607039 0.5074746 0.43583281
4 row9 chr21 0.18445757 0.1926704 0.5522359 0.4889265 0.2885484 0.36124031
5 row10 chr21 0.96333352 0.9455556 0.2251288 0.7133460 0.2673244 0.01864215
  V9      V10     V11
1 0.5832685 0.8670779 0.08756482
2 0.9471309 0.6032699 0.37931679
3 0.2296118 0.4037352 0.37954743
4 0.3595676 0.6895200 0.09194656
5 0.1601413 0.4742028 0.07500326
```



# awk

Maybe we don't want all of the columns

```
> awkCall12 = "awk -F\"\\t\" '$2 == \\\"chr21\\\" { print $2\"\\t\"$3\"\\t\"$4 }' foo.txt"
> read.delim(pipe(awkCall12), header=FALSE)
      V1      V2      V3
1 chr21 0.06470508 0.2104643
2 chr21 0.69930753 0.9533248
3 chr21 0.31752770 0.6364067
4 chr21 0.18445757 0.1926704
5 chr21 0.96333352 0.9455556
```

## awk

Note you have to “escape” all of the quotation marks that are within the actual awk call so that they are not treated by quotation marks in R

# cut

cut selects certain columns

Let's take columns 2, 3, and 4 for all rows:

```
> cutCall = "cut -f2,3,4 foo.txt"
> read.delim(pipe(cutCall), header=FALSE)
      V1      V2      V3
1  chr1 0.55127556 0.06161779
2  chr1 0.97378813 0.84200893
3  chr1 0.67479999 0.54580091
4  chr1 0.90762683 0.62246401
5  chr1 0.27251173 0.09567016
6 chr21 0.06470508 0.21046433
7 chr21 0.69930753 0.95332484
8 chr21 0.31752770 0.63640675
9 chr21 0.18445757 0.19267041
10 chr21 0.96333352 0.94555563
```

# cut

The `-d` option specifies that string that separates columns

# Combining awk + cut

You can also use the linux pipe "|" within the R `pipe()` call:

```
> bothCall = paste("awk -F\"\\t\" '$2 == \"chr21\" { print $0 }' foo.txt | cut -f2,3,4")
> read.delim(pipe(bothCall), header=FALSE)
      V1      V2      V3
1 chr21 0.06470508 0.2104643
2 chr21 0.69930753 0.9533248
3 chr21 0.31752770 0.6364067
4 chr21 0.18445757 0.1926704
5 chr21 0.96333352 0.9455556
```

## Windows users

Doing all of this in Windows takes a lot more setup, and there are a more limited set of Linux commands: <http://stackoverflow.com/questions/23963106/read-columns-of-a-csv-file-using-shell-or-pipe-inside-r-windows>