Big Data in R

Importing data into R: 1.75GB file

Table 1: Comparison of importing data into R

Packages	Functions	Time Taken (second)	Remark/Note
base	`read.csv`	> 2,394	My machine (8GB of memory) ran out of memory before the data could be loaded in.
Laf	`laf_open_csv`	49.92	4.82GB memory used by R. Cannot read all csv files
sqldf	`read.csv.sql`	172.97	4.23GB of memory used by R. Note sqldf does not treat quotes as unique, so if an entry in a column is "a , b" (including quotes), then sqldf will treat it as two separate items. ["a] and [b"].
bigmemory	`read.big.matrix`	147.62	27MB used by R. Cannot handle factors (yet). Factor columns will be represented as a column of NA's. You will need to read in the particular factor columns using read.csv, and then code them as integers.

Base: read.csv

```
rm(list=ls())
system.time({large = read.csv("large.csv")})
#Ran out of memory at 39.9 minutes - my machine has 8GB of memory - R used as much as it
could.
```

Package: LaF

```
rm(list=ls())
library(LaF) #LaF is a package for accessing large files that are too big to fit into
memory.
system.time({
  large.laf = laf_open_csv(filename = "large.csv",
column_types=c("integer","factor","numeric","numeric"), column_names = c("X", "dim",
"fact1", "fact2"), skip = 1) #Need to specify column names and types.
  large = large.laf[,]
}) #49.92 seconds, 4.82GB memory used.
```

Package: sqldf

```
rm(list=ls())
library(sqldf)
system.time({large = read.csv.sql("large.csv")})  #172.97 seconds, 4.23GB of memory used
by R
```

Package: bigmemory

```
library(bigmemory)
system.time({
  line1 = read.csv("large.csv", header = TRUE, nrows = 1)
  col.names = colnames(line1)
  read.big.matrix("large.csv", header = TRUE, backingfile = "large.backing",
descriptorfile = "large.backing.desc", col.names = col.names)
```

```
}) # 146.42 seconds, 27mb memory used.
large = attach.big.matrix("large.backing.desc")
```

Exporting CSV files from R:

Table 2: Memory and time taken to export data from R to a CSV file. Two separate CSV files were used as test data, with file sizes of 23MB and 44MB respectively. The files were read into R using read.csv. The base memory usage after the files were loaded in was 101MB and 309MB respectively. The objective was to stack the data on itself and export it to a csv file.

Package	Functions	Time (seconds)		Peak Memory Usage		Comments
		23MB file	44MB file	23MB file	44MB file	
Base R	rbind & write.csv	39.34	NA	441 MB	> 1270 MB	This method could not be completed for the 44MB file, as R ran out of memory.
	write.table	35.74	181.41	224 MB	735 MB	
	rbind & writeLines	29.81	182.54	354 MB	1172 MB	Categorical data is not treated properly.
bigmemory	as.big.matrix & write.big.matrix	92.55	181.66	406 MB, (210 while writing csv)	753 MB (174 while writing csv)	Using "integer" type matricies. Converts categorical data into numerical factor levels
ff	as.ffdf & write.table.ffdf	44.30	185.47	171 MB	398 MB	
RH2 and sqldf	rbind & sqldf	More than 1200 seconds	Not tested	368 MB, only 126 MB during sqldf	Not tested	Need to use H2 method.

Use the ff package, as it is much more scalable than base R in terms of memory usage, meaning it can handle large amounts of data. However it still takes the same amount of time as base R. Unfortunately none of the packages explored appeared to perform any better than base R in terms of speed. In the future I may explore accessing databases through the package RODBC.

Base R

```
# rbind and write
  output = rbind(myDF, myDF)        # bind df's together
  system.time({write.csv(output, "test.csv")})     # write output to csv

 # write.table
  write.table(myDF, "test.csv", append = FALSE, sep = ",", row.names = FALSE)
# write myDF to csv file
```

```
  write.table(myDF, "test.csv", append = TRUE, sep = ",", row.names = FALSE, col.names =
FALSE)  # Write myDF to the same csv file.

#rbind and writeLines:  Doesn't quote categorical data
  output = rbind(myDF, myDF)
# Write to csv file manually creating the lines.
writeLines(c( do.call(function(...) paste(..., sep = ","), as.list(colnames(output))) ,
do.call(function(...) paste(..., sep = ","), as.list(output[,]))), file("test.csv"))
```

Package: bigmemory

```
# Type = "integer
library(bigmemory)
n = nrow(myDF)
output.big = big.matrix(nrow = 2*n, ncol = ncol(myDF), type = "integer")  #Create empty
big.matrix
#Coercing data.frame to matrix via factor level numberings
myDF.big = as.big.matrix(myDF, type = "integer")
# Fill the matrix
output.big[1:n,] = myDF.big[,]
output.big[(n+1):(2*n),] = myDF.big[,]
#Write to csv file
write.big.matrix(output.big, "test.csv", sep = ",", col.names = T)
```

Package: ff

```
## Package: ff: no faster but uses less memory than write.table.
library(ff)
myDF.ffdf = as.ffdf(myDF) # Convert to ff dataframe
# Write first part of table
write.table.ffdf(myDF.ffdf, "test.csv", sep = ",", row.names = FALSE, col.names = T)
# Append the existing file
write.table.ffdf(myDF.ffdf, "test.csv", sep = ",", row.names = FALSE, col.names = F,
append = T)
```

Package: sqldf

```
# Load RH2 library first to use H2 method.
library(RH2)
library(sqldf)
#Only H2 mode can export to CSV
#Worked for small files, but took to long for large files.
x = sqldf("CALL CSVWRITE('test.csv', 'SELECT * FROM myDF')")
```

Calculating aggregates:

The test set of data was given to me by Kristy: 55mb, 137473 rows, 170 columns. The goal is to sum the time series columns by two categorical variables, "Switch", and "Technology". The first four columns are categorical data, with the remaining 166 columns consisting of integers.

Table 3: Comparison of calculating aggregates - Summing across two different factors

Packages	Functions	Time Taken (second)	Remark/Note
base	`aggregate`	100.74	This is the basic method in R

	group.sums	0.78	This is my own function, and is available in "Glenn's Smart Functions.r".
data.table	data.table	2.06	Need to first convert the data frame to a data table (included in time).
	Accessing a data table object	0.75	Time taken if the data table is already built.
plyr	ddply with sapply	7.67	
	ddply with colSums	10.84	
sqldf	sqldf	12.62	Time taken using a temporary database (default)
		14.23	Time taken using a permanent database given the table does not already exist in the database.
		4.15	Time taken using a permanent database given the table has already been created in a previous sqldf command.

Importing the data

```
names = colnames(read.csv("demand.series_ER.csv", nrows=1))
library(LaF)
system.time({
  data.laf = laf_open_csv(filename = "demand.series_ER.csv", column_types =
c(rep("factor",4),rep("integer",length(names)-4)), column_names = names, skip = 1)
  data = data.laf[,]
})   #12.51 seconds
```

Base R:

```
system.time({x0 = aggregate(data[,-(1:4)],by=list(data$Switch,data$Technology),FUN=sum)})
#100.74 seconds, baseR, sorts by Technology first

system.time({x = group.sums(data, c("Switch", "Technology"), -(1:4))}) #0.78 seconds
```

Package: data.table

```
library(data.table)
system.time({
  data.dt = data.table(data[,-c(1,2)])    # Build the data table from the original data,
including only the columns to be summed over and the grouping factors, 0.75 seconds
  x = data.dt[, lapply(.SD,sum), by=c("Switch","Technology")] # Calculate sums, 1.31
seconds
})  #2.06 seconds in total, sorts by Technology first.
```

Package: plyr

```
library(plyr)
system.time({x = ddply(data, c("Switch","Technology"), function(df) sapply(df[,-
(1:4)],sum))}) # 7.67 seconds, sorts by Switch first
system.time({x = ddply(data, c("Switch","Technology"), function(df) colSums(df[,-
(1:4)]))}) # 10.84 seconds
```

Package: sqldf

```
library(sqldf)

names = colnames(data)
system.time({
  mystring = numeric(0)
  for (i in 5:ncol(data)) mystring = paste(mystring,", sum(", names[i], ")")
  mystring = paste("Select Switch, Technology", mystring, "from data group by Switch,
Technology")
  x = sqldf(mystring)
}) # 15.57 seconds

system.time({
```

```
  mystring = numeric(0)
  for (i in 5:ncol(data)) mystring = paste(mystring,", sum(", names[i], ")")
  mystring = paste("Select Switch, Technology", mystring, "from main.data group by Switch,
Technology")
  x = sqldf(c("create index ix on data(Switch, Technology)",mystring))
}) # 12.62 seconds
```

With the sqldf package above, the data is being copied to a new database, then the SQL operations are being applied to the database, then the SQL database is destroyed. If we want to use the sqldf package multiple times, it is best to create a permanent database for efficiency.

```
system.time({
    mystring = numeric(0)
  for (i in 5:ncol(data)) mystring = paste(mystring,", sum(", names[i], ")")
  mystring = paste("Select Switch, Technology", mystring, "from main.data group by Switch,
Technology")
  sqldf(dbname = "mydb") #create a file named "mydb" to use as a permanent database
   x = sqldf(c("create index ix on data(Switch, Technology)",mystring), dbname = "mydb")
#Performs the operation, copying the data in mydb
}) # 14.17 seconds

#Performing the same operation again
system.time({x = sqldf(mystring, dbname = "mydb")}) # 4.17 seconds
```

The above results are very similar for calculating means instead of sums, and they are also similar when only considering one grouping factor.

For example, calculating the means, using "Switch" as a grouping factor.

Table 4: Comparison of calculating aggregates - Calculating the mean across a single factor

Packages	Functions	Time Taken (second)	Remark/Note
base	`aggregate`	107.72	This is the basic method in R
	`group.means`	1.14	This is my own function, and is available in "Glenn's Smart Functions.r".
	`group.fun`	13.40	Another of my functions. It can only consider a single grouping factor but the user can specify the aggregate to calculate.
	`tapply embedded in apply`	31.11	This method can only consider a single grouping factor.
data.table	`data.table`	1.47	Need to first convert the data frame to a data table (included in time).
	`Accessing a data table object`	0.42	Time taken if the data table is already built.
plyr	`ddply with sapply`	10.28	
	`ddply with colMeans`	10.36	
sqldf	`sqldf`	12.52	Time taken using a temporary database (default)
		14.17	Time taken using a permanent database given the table does not already exist in the database.
		4.34	Time taken using a permanent database given the table has already been created in a previous sqldf command.
LaF	`process.blocks`	?	Need to investigate further – functions are not straight forward to write for LaF since it consider data

			block by block.

Base R:

```
system.time({x = aggregate(data[,-(1:4)],by=list(data$Switch),FUN=mean)}) #107.72 seconds
system.time({x = apply(data[,-c(1:4)] ,2, function (x) tapply(x, data$Switch,mean))})
#31.11 seconds
system.time({x = group.fun(data[,-(1:4)],data$Switch,mean) }) #13.40 seconds, my own
function available in the attachment. Can handle any function, but only one grouping
factor
system.time({x = group.means(data, c("Switch" ), -(1:4))}) #1.14 seconds, my own function
available in the attachment
```

Package: data.table

```
library(data.table)
system.time({
  data.dt = data.table(data[,-c(1,2,4)], key = "Switch")    #1.05 seconds
  x = data.dt[, lapply(.SD,mean), by=c("Switch")]           #0.42 seconds
})
```

Package: plyr

```
library(plyr)
system.time({x = ddply(data, .(Switch), function(df) sapply(df[,-(1:4)],mean))}) #10.28
seconds
system.time({x = ddply(data, .(Switch), function(df) colMeans(df[,-(1:4)]))})
#10.36 seconds
#1.47 seconds total
```

Package: sqldf

```
library(sqldf)

system.time({
  mystring = numeric(0)
  for (i in 5:ncol(data)) mystring = paste(mystring,", avg(", names[i], ")")
  mystring = paste("Select Switch", mystring, "from main.data group by Switch")
  x = sqldf(c("create index ix on data(Switch)",mystring))
}) # 12.52 seconds

system.time({
    mystring = numeric(0)
  for (i in 5:ncol(data)) mystring = paste(mystring,", avg(", names[i], ")")
  mystring = paste("Select Switch", mystring, "from main.data group by Switch")
  sqldf(dbname = "mydb") #create a file named "mydb" to use as a permanent database
   x = sqldf(c("create index ix on data(Switch)",mystring), dbname = "mydb") #Performs the
operation, copying the data in mydb
}) # 14.17 seconds

#Performing the same operation again
system.time({x = sqldf(mystring, dbname = "mydb")}) # 4.34 seconds
```

Package: LaF

Note: The LaF package has potential to do the above without loading the data into memory, but functions are not as straight forward to write. I will explore this further at some point.

Finding subsets:

The following calculations were done a 1.75GB file (same as I used for importing data), using my own machine with 8GB of memory. The goal is to find all observations which have dim (categorical variable) == "b" and fact1 (numerical variable) > 0.

Table 5: Comparison finding subsets of data

Packages	Functions	Time Taken (second)	Remark/Note
base	`Subset data.frame object`	3.04	This is the basic method in R
data.table	`data.table`	3.24	Need to first convert the data frame to a data table and set the key (included in time).
	`Subset data.table object`	0.00	Time taken if the data table is already built and keyed.
sqldf	`sqldf`	NA	sqldf could not be used to specify an value for the categorical variable because it was stored with quotes.
LaF	`Subset LaF object`	50 (approx)	Subset of data obtained without loading in data

Base R

```
system.time({x = large[large$dim == '"b"' & large$fact1 > 0,]}) #3.04 seconds
```

Package: data.table

```
library(data.table)
system.time({
  dt = data.table(large) #1.09 seconds
  setkey(dt,dim)         #2.15 seconds, these steps can be combined: dt =
data.table(large, key = "dim")
  y = dt[J('"b"')]       #0 seconds, can subset on multiple factors, but need to set
multiple keys
  x = y[y$fact1 > 0,]    #0 seconds
})                       #3.24 seconds total.
```

Clearly the above is good if the data is already in a data.table format, or multiple subsets need to be obtained, since the data table only has to be created and keyed once.

Package: sqldf - had problems using sqldf because of how the data was stored.

```
system.time({x = sqldf("select * from large where dim = '"b"'")} #too
many quote marks, as data was stored as "b", not b.
```

Package: LaF

```
system.time({x = huge.laf[huge.laf$fact1[] > 0 & huge.laf$dim[] == "b",]}
# Subsets without loading in data, hence it takes as long as loading in the data (50
# seconds), because accessing a hard drive is slow compared to accessing RAM.
```

Further Comments on sqldf:

The benefit of using sqldf, is that data can be permanently stored in an SQL database file. R can be restarted, and all one has to do is tell R where the database file is, using `sqldf(dbname = "mydb")`. Since the database is already created, sqldf can efficiently calculate summary statistics and find subsets of the data. Unfortunately, more complicated analysis cannot be done in this framework, but if the database already exists, then loading in the data should take half the time it would take using a temporary database. Additionally, names in R that have a "." in them (Such as column names like Jan.2012) will automatically be changed such that the period becomes an underscore. This is because the period symbol is an operator in SQL. Finally, sqldf does not treat quotes as being unique. For example, consider the following line from a csv file, which has three columns:

“a , b ”, “A”, 0.5,
The sqldf package will treat it as if there were four columns | “a | b” | “A” | 0.5 | which is not ideal. So beware when using sqldf.

Maximum object sizes in R

On all builds of **R** prior to version 3.0.0 (32 and 64 bit), the maximum length (number of elements) of a vector is 2^31 - 1 ~ 2*10^9. Since the R stores matrices as vectors and dataframes as lists, the maximum number of cells allowed in these objects is also 2^31 – 1. Therefore the product of the number of rows and columns of a matrix or dataframe cannot exceed 2^31 – 1. The table below outlines the maximum matrix and dataframe dimensions. The bigmemory package can overcome these limitations.

Number of rows	**Maximum number of columns**
10,000	214, 748
100,000	21,474
1 million	2,147
10 million	214
100 million	21
1 billion	2