# Working with Larger Datasets

Biostatistics 140.776

October 25, 2016

## R and Physical Memory

- R has historically not been good with large datasets
- Need to store working data in physical memory (as opposed to on disk)
- It's important to be aware of the limits of your computing environment with respect to available memory and how that may affect your ability to use R.
- In the event that your computer's physical memory is insufficient for some of your work, there have been some developments that allow R users to deal with objects out of physical memory and we will discuss them below.

## R and Physical Memory

- The first thing that is worth keeping in mind as you use R is how much physical memory your computer actually has.
- Typically, you can figure this out by looking at your operating system's settings.
- Example: This Macbook has 8 GB of RAM.
- The amount of RAM available to R will be quite a bit less than that, but it's a useful upper bound.
- You can't read an object into R that will be 16 GB of RAM.

## pryr Package

The `pryr` package provides a number of useful functions for interrogating the memory usage of your R session.

- The most basic is the `mem_used()` function, which tells you how much memory your current R session is using.

```
library(pryr)
mem_used()
25.5 MB
```

The primary use of this function is to make sure your memory usage in R isn't getting too big. If the output from `mem_used()` is in the neighborhood of 75%-80% of your total physical RAM, you might need to consider a few things.

## How Much Memory is being Used?

You can see the memory usage of objects in your workspace by calling the `object_size()` function.

```
data(airquality)
ls()  ## Show objects in workspace
[1] "airquality" "args"
object_size(airquality)
5.5 kB
```

- The `object_size()` function will print the number of bytes (or kilobytes, or megabytes) that a given object is using in your R session.

## How Much Memory is being Used?

You can view the change in memory usage by executing an R expression by using the `mem_change()` function.

```
mem_change(rm(airquality))
3.77 kB
```

Things won't always add up because of various kinds of memory management overhead.

## Back of the Envelope Calculations

- It is important to be cognizant of how much memory is being used up by all of the data objects residing in your workspace.
- One situation where it's particularly important to understand memory requirements is when you are reading in a new dataset into R.
- It's easy to make a back of the envelope calculation of how much memory will be required by a new dataset.

## Memory Requirements

It's difficult to generalize how much memory is used by data types in R. On most 64 bit systems today,

- integers are 32 bits (4 bytes)
- double-precision floating point numbers (numerics in R) are 64 bits (8 bytes).
- character data are usually 1 byte per character.

Because most data come in the form of numbers (integer or numeric) and letters, just knowing these three bits of information can be useful for doing many back of the envelope calculations.

## Memory Requirements

For example, an integer vector is roughly 4 bytes times the number of elements in the vector. We can see that for a zero-length vector, that still requires some memory to represent the data structure.

```
object_size(integer(0))
40 B
```

However, for longer vectors, the overhead stays roughly constant, and the size of the object is determined by the number of elements.

```
object_size(integer(1000))  ## 4 bytes per integer
4.04 kB
object_size(numeric(1000))  ## 8 bytes per numeric
8.04 kB
```

## Machine Details

```
str(.Machine)
List of 18
 $ double.eps              : num 2.22e-16
 $ double.neg.eps          : num 1.11e-16
 $ double.xmin             : num 2.23e-308
 $ double.xmax             : num 1.8e+308
 $ double.base             : int 2
 $ double.digits           : int 53
 $ double.rounding         : int 5
 $ double.guard            : int 0
 $ double.ulp.digits       : int -52
 $ double.neg.ulp.digits: int -53
 $ double.exponent         : int 11
 $ double.min.exp          : int -1022
 $ double.max.exp          : int 1024
 $ integer.max             : int 2147483647
 $ sizeof.long             : int 8
 $ sizeof.longlong         : int 8
```

## Machine Details

### Floating Point Numbers

- The floating point representation of a decimal number contains a set of bits representing the **exponent** and another set of bits representing the **significand** or the **mantissa**.
- The number of bits used for the exponent is 11, from `double.exponent`
- The number of bits for the significand is 53, from the `double.digits` element
- Each double precision floating point number requires 64 bits

### Integers

- We can see that the maximum integer indicated by the integer.max is 2147483647
- We can take the base 2 log of that number and see that it requires 31 bits to encode (plus 1 bit for sign)

## Back of the Envelope Calculations

- If you are reading in tabular data of integers and floating point numbers, you can roughly estimate the memory requirements for that table by multiplying the number of rows by the memory required for each of the columns.
- Do this **before** reading in a large tabular dataset!
- Example: I have a data frame with 1,500,000 rows and 120 columns, all of which are numeric data.
- How much memory is required to store this data frame?

## Back of the Envelope Calculations

On most modern computers double precision floating point numbers are stored using 64 bits of memory, or 8 bytes. Given that information, you can do the following calculation

| Rows | Bytes |
|---:|:---|
| 1,500,000 x 120 x 8 bytes/numeric | = 1,440,000,000 bytes |
| | = 1,440,000,000 / 2^20 bytes/MB |
| | = 1,373.29 MB |
| | = 1.34 GB |

So the dataset would require about 1.34 GB of RAM. Most computers these days have at least that much RAM.

## Reading in Large Datasets

Even if you have the theoretical amount of memory required, you need to be aware of

- what other programs might be running on your computer, using up RAM
- what other R objects might already be taking up RAM in your workspace

Reading in a large dataset for which you do not have enough RAM is one easy way to freeze up your computer (or at least your R session)!

## Internal Memory Management in R

- R has a garbage collection system that recycles unused memory and gives it back to R.
- Garbage collection happens automatically without the need for user intervention.
- R will periodically cycle through all of the objects that have been created and see if there are still any references to the object somewhere in the session.
- If there are no references, the object is garbage-collected and the memory returned.
- Under normal usage, the garbage collection is not noticeable, but occasionally, when working with very large R objects, you may notice a “hiccup” in your R session when R triggers a garbage collection to reclaim unused memory.
- (There’s not really anything you can do about this except not panic when it happens.)

## Internal Memory Management in R

The gc() function can be used to explicitly trigger a garbage collection in R.

- Calling gc() explicitly is never actually needed, but it does produce some output that is worth understanding.

```
gc()
         used (Mb) gc trigger (Mb) max used (Mb)
Ncells 376500 20.2     750400 40.1   543730 29.1
Vcells 579319  4.5    1308461 10.0   831673  6.4
```

- The used column gives you the amount of memory currently being used by R (the distinction between Ncells and Vcells is not important)
- The gc trigger column gives you the amount of memory that can be used before a garbage collection is triggered.
- Generally, you will see this number go up as you allocate more objects and use more memory.

## Dealing with Large Datasets

In-memory strategies

- Basically a collection of functions/packages that handle input/output more efficiently than R's built in functions
- The `readr` package (and `read_csv()`) is one example
- The `data.table` package is another example
- Functions in `dplyr` and `data.table` are often much faster for manipulating large datasets
- Still need to hold entire dataset in memory
- Gentleman's Law: Turn big data into small data as quickly as possible

## Dealing with Large Datasets

Out of memory strategies

- File-based or remote repositories
- Relational databases and SQL (RSQlite, RMySQL)
- Memory mapping (`ff` package)
- Other file formats for data (feather, hdf5)

## Databases

- There are some options to explore and model the dataset without ever loading it into R, while still using R commands and working from the R console or an R script.
- These options can make working with large datasets more efficient, because they let other software handle the heavy lifting of sifting through the data and avoid loading large datasets into RAM
- Database management systems (DBMS) are optimized to more efficiently store and better search through large sets of data
- DBMS allows for transactions

## DBI Package

- The DBI package is a generalized interface for connecting R code with a database management system
- It provides a top-level interface to a number of different database management systems
- There is system-specific code applied by a lower-level, more specific R package.
- There are numerous R packages that allow you to connect your R session to many major database implementations
- Examples: MySQL, PostgresQL, RSQLite, Oracle
- DBI-compliant R packages have not been written for every database management system, so there are some databases for which DBI commands will not work.
- If a package requires other outside softare, *you must have that installed to use these packages*.

## Databases

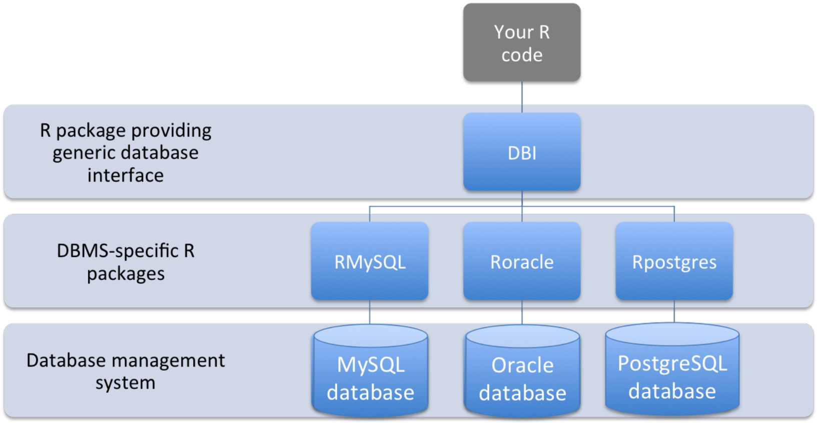

Figure 1:How R Interfaces with DBMS

## Databases: The DBI Package

| Task | DBI Function |
|---|---|
| Create a new driver object for a database | `dbDriver` |
| Connect to database instance | `dbConnect` |
| Find available tables in a database | `dbListTables` |
| Find available fields within a table | `dbListFields` |
| Query a connected database instance | `dbSendQuery` |
| Pull a data frame into R from a query result | `dbFetch` |
| Jointly query and pull data from a database | `dbGetQuery` |
| Close result set from a query | `dbClearResult` |
| Write a new table in a database instance | `dbWriteTable` |
| Remove a table from a database instance | `dbRemoveTable` |
| Disconnect from a database instance | `dbDisconnect` |

## Databases: The `DBI` Package

| Database Management System | R packages |
|---|---|
| Oracle | `ROracle` |
| MySQL | `RMySQL` |
| Microsoft SQL Server | `RSQLServer` |
| PostgreSQL | `RPostgres` |
| SQLite | `RSQLite` |

## `RSQLite` Package

- The `RSQLite` package is convenient because it is simple and straight forward to install (from CRAN)
- Databases are represented as single files on the hard drive
- SQL commands can be sent to the database via the `DBI` functions
- Good for medium to large sized problems that do not need network communication

## RSQLite Package

```
library(RSQLite)
Loading required package: DBI
Loading required package: methods
drv <- dbDriver("SQLite")
db <- dbConnect(drv, "airquality")
dbListTables(db)
[1] "ozone"
```

## RSQLite Package

Extract some data

```
ozone <- dbReadTable(db, "ozone")
str(ozone)
'data.frame':   153 obs. of  6 variables:
 $ Ozone  : int  41 36 12 18 NA 28 23 19 8 NA ...
 $ Solar.R: int  190 118 149 313 NA NA 299 99 19 194 ...
 $ Wind   : num  7.4 8 12.6 11.5 14.3 14.9 8.6 13.8 20.1 8
 $ Temp   : int  67 72 74 62 56 66 65 59 61 69 ...
 $ Month  : int  5 5 5 5 5 5 5 5 5 5 ...
 $ Day    : int  1 2 3 4 5 6 7 8 9 10 ...
```

## RSQLite Package

Send query, Fetch data

```
result <- dbSendQuery(db,"SELECT Ozone, Wind, Month
                      FROM ozone WHERE Month = 5")
ozone.sub1 <- dbFetch(result, n = 10)
str(ozone.sub1)
'data.frame':   10 obs. of  3 variables:
 $ Ozone: int  41 36 12 18 NA 28 23 19 8 NA
 $ Wind : num  7.4 8 12.6 11.5 14.3 14.9 8.6 13.8 20.1 8.6
 $ Month: int  5 5 5 5 5 5 5 5 5 5
dbHasCompleted(result)
[1] FALSE
```

## RSQLite Package

Continue fetching

```
ozone.sub2 <- dbFetch(result, n = 10)
str(ozone.sub2)
'data.frame':	10 obs. of  3 variables:
 $ Ozone: int  7 16 11 14 18 14 34 6 30 11
 $ Wind : num  6.9 9.7 9.2 10.9 13.2 11.5 12 18.4 11.5 9.7
 $ Month: int  5 5 5 5 5 5 5 5 5 5
dbHasCompleted(result)
[1] FALSE
dbClearResult(result)
[1] TRUE
```

## RSQLite Package

Select some rows and fetch results together

```
ozone.sub <- dbGetQuery(db, "SELECT Ozone, Wind, Month
                        FROM ozone WHERE Month = 5")
str(ozone.sub)
'data.frame':   31 obs. of  3 variables:
 $ Ozone: int  41 36 12 18 NA 28 23 19 8 NA ...
 $ Wind : num  7.4 8 12.6 11.5 14.3 14.9 8.6 13.8 20.1 8.6
 $ Month: int  5 5 5 5 5 5 5 5 5 5 ...
```

Only do this if you know in advance that the result will fit in memory!

## RSQLite Package

Write a new table to the database

```
values <- c("Ozone level in ppb",
            "Solar radiation in lang",
            "Wind speed in mph",
            "Temperature in degrees F",
            "Month 1-12",
            "Day of month 1-31")
meta <- data.frame(name = names(ozone),
                   value = values)
dbWriteTable(db, "metadata", meta)
[1] TRUE
dbListTables(db)
[1] "metadata" "ozone"
```

## RSQLite Package

Show the metadata table

```
dbReadTable(db, "metadata")
     name                      value
1   Ozone         Ozone level in ppb
2 Solar.R  Solar radiation in lang
3    Wind          Wind speed in mph
4    Temp Temperature in degrees F
5   Month                 Month 1-12
6     Day          Day of month 1-31
```

## RSQLite Package

Remove a table

```
dbRemoveTable(db, "metadata")
[1] TRUE
dbListTables(db)
[1] "ozone"
dbDisconnect(db)
[1] TRUE
```

## Summary

- Memory usage is something you need to keep in the back of your mind
- Know how much memory your computer has
- Make rough calculations of data size before reading into R
- Make big data small data as quickly as possible
- Use external databases for truly large datasets to push computations closer to the data
- Databases not necessarily faster, but may be only feasible option.