Los Angeles R Users' Group

# Taking R to the Limit, Part II: Working with Large Datasets

Ryan R. Rosario

August 17, 2010

## The Brutal Truth

We are here because we love R. Despite our enthusiasm, R has two major limitations, and some people may have a longer list.

1. Regardless of the number of cores on your CPU, R will only use 1 on a default build. (Part I)
2. R reads data into memory by default. Other packages (SAS particularly) and programming languages can read data from files on demand.
   - Easy to exhaust RAM by storing unnecessary data.
   - The OS and system architecture can only access $\frac{2^{32}}{1024^2} = 4\text{GB}$ of memory on a 32 bit system, but typically R will throw an exception at 2GB.
   - Not wise to use more memory than available. System will start swapping which leads to thrashing and slows your system to a crawl.

## The Brutal Truth

There are a couple of solutions:

1. Buy more RAM.
2. Use a database.
3. Build from source and use special build options for 64 bit. **Still not a solution!**
    1. R, even R64, does not have a `int64`[1] data type. Not possible to index data frames or matrices with huge number of rows or columns.
4. Sample, resample, or use some Monte Carlo method.
5. **Let clever developers solve these problems for you!**

We will discuss number 5 above.

[1] http://permalink.gmane.org/gmane.comp.lang.r.devel/17281

## The Agenda

This talk will survey several HPC packages in R with some demonstrations. We will focus on four areas of HPC:

1. **Explicit Parallelism:** the user controls the parallelization. (Part I)
2. **Implicit Parallelism:** the system abstracts it away. (Part I)
3. **Large Memory:** working with large amounts of data without choking R.
4. **Map/Reduce:** basically an abstraction for parallelism that divides *data* rather than *architecture*.

## Disclaimer

1. There is a **ton** of material here. I provide a lot of slides for your reference.
2. We may not be able to go over all of them in the time allowed so some slides will go by quickly.
3. Demonstrations will be time permitting.
4. All experiments were run on a Ubuntu 10.04 system with an Intel Core 2 6600 CPU (2.4GHz) with 2GB RAM. I am planning a sizable upgrade this summer.

## Big Data

"Big Data" is a catch phrase for any dataset or data application that does not fit into available RAM on one system.

R has ~~several~~ a few packages for big data support. We will talk about the following:

1. `bigmemory`
2. `ff`

We will also discuss some uses of parallelism to accomplish the same goal using Hadoop and MapReduce:

1. `HadoopStreaming`
2. `Rhipe`

## Some Basic Terminology

I will use the word **RAM** to refer to physical memory, for simplicity; chips that are installed on the system motherboard.

**Virtual memory**, or **swap** is disk space that is used to store objects that do not fit into RAM and are less frequently accessed. This is SLOW.

A **cluster** is a group of systems that communicate with each other to accomplish a computation.

## Large Datasets

R reads data into RAM all at once, if using the usual `read.table` function. Objects in R live in memory entirely. Keeping unnecessary data in RAM will cause R to choke eventually. Specifically,

1. on most systems it is not possible to use more than 2GB of memory.
2. the range of indexes that can be used is limited due to lack of a 64 bit integer data type in R and R64.
3. on 32 bit systems, the maximum amount of virtual memory space is limited to between 2 and 4GB.
4. relying on virtual memory will cause the system to grind to a halt – "thrashing."

## Large Datasets

There are two major solutions in R:

1. `bigmemory`: "It is ideal for problems involving the analysis in R of manageable subsets of the data, or when an analysis is conducted mostly in C++." It's part of the "`big`" family, some of which we will discuss.
2. `ff`: file-based access to datasets that cannot fit in memory.
3. can also use databases which provide fast read/write access for piecemeal analysis.

## The “big” Family

The big family consists of several packages for performing tasks on large datasets.

1. bigmemory is our focus.
2. biganalytics provides analysis routines on big.matrix such as GLM and bigkmeans.
3. synchronicity adds Boost mutex functionality to R.
4. bigtabulate adds table and split-like support for R matrices and big.matrix memory efficiently.
5. bigalgebra provides BLAS and LAPACK linear algebra routines for native R matrices and big.matrix.
6. bigvideo provides video camera streaming via OpenCV.

## `bigmemory`

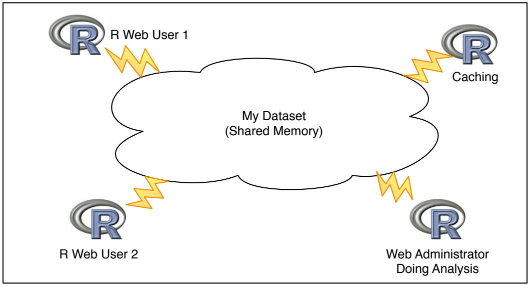
`bigmemory` implements several matrix objects.

1. `big.matrix` is an R object that simply points to a data structure in C++. Local to a single R process and is limited by available RAM.
2. `shared.big.matrix` is similar, but can be shared among multiple R processes (think *parallelism* on data!)
3. `filebacked.big.matrix` does not point to a data structure; rather, it points to a file on disk containing the matrix, and the file can be shared across a cluster!

**Pitfall!** Remember that matrices contain only *one* type of data. Additionally, the data types for elements are dictated by C++ **not** R: `double`, `integer`, `short`, `char`.

## bigmemory and Shared Memory

Shared Memory allows us to store data in RAM and share it among multiple processes. Suppose we want to store some data in shared memory so it can be read by **multiple instances of R**. This allows the user the ability to use multiple instances of R for performing different analytics simultaneously.



**RAM**

## Demonstration: The Logic

First, construct a `big.matrix` object. Let us suppose we want to create a large matrix of 0s and 1s.

We can construct a matrix of type `int` (4 bytes), `short` (2 bytes), `double` (8 bytes), or `char` (1 byte). Since all we need is 0 and 1, we use `char`. We also zero out the matrix.

```
> A <- big.matrix(m, n, type="char", init=0, shared=TRUE)
> A
An object of class big.matrix
Slot "address":
<pointer: 0x2d93490>
```

## Demonstration: The Logic

We have now created a **pointer** to a C++ matrix that is on disk. But, to share this matrix we need to share this descriptor.

Then, we can open a second R session, load the location of the matrix from disk, and bind the matrix to an R variable!

## Demonstration: The Code

**Session 1**

```
library(bigmemory)
library(biganalytics)
options(bigmemory.typecast.warning=FALSE)

A <- big.matrix(5000, 5000, type="char", init=0)
#Fill the matrix by randomly picking 20% of the positions
     for a 1.
x <- sample(1:5000,size=5000,replace=TRUE)
y <- sample(1:5000,size=5000,replace=TRUE)
for(i in 1:5000) {
    A[x[i],y[i]] <- 1
}
#Get the location in RAM of the pointer to A.
desc <- describe(A)
#Write it to disk.
dput(desc, file="/tmp/A.desc")
sums <- colsum(A, 1:20)
```

## Session 2

```
library(bigmemory)
library(biganalytics)

#Read the pointer from disk.
desc <- dget("/tmp/A.desc")
#Attach to the pointer in RAM.
A <- attach.big.matrix(desc)
#Check our results.
sums <- colsum(A, 1:20)
```

## Demonstration 1

**Demonstration here.**

## bigmemory: Importance of Data Type

Suppose we are not careful in the C++ data type we use for the `big.matrix`. Using `char`, the matrix requires about 24MB of RAM.

| Data Type | RAM |
|---|---:|
| `char` | 24 MB |
| `int` | 96 MB |
| `double` | 192 MB |
| `short` | 48 MB |

Of course, this assumes that you use dense matrices and there is no CPU optimization.

## bigmemory: Other Operations

`big.matrix` requires its own optimized versions of the R matrix functions provided in `biganalytics`:

```
colmean(x, cols, na.rm)
colmin(x, cols, na.rm)
colmax(x, cols, na.rm)
colvar(x, cols, na.rm)
colsd(x, cols, na.rm)
colsum(x, cols, na.rm)
colprod(x, cols, na.rm)
colna(x, cols)
```

where `x` is a `big.matrix`, `cols` is a vector of column indices, `na.rm` is `TRUE` if R should remove missing values first.

## The `big` Example: The Data

Let's do something useful with some big data. Consider airline on-time performance from 1987 to 2008. This is the format of the data:

- 11GB comma-separated values file.
- 120 million rows, 29 columns
- Factors coded as integers.

We will estimate the **age** of an aircraft at each departure.

## Step 1: Read in the Data

```r
library(bigmemory)
library(biganalytics)
x <- read.big.matrix("airline.csv", type="
    integer", header=TRUE,
      backingfile="airline.bin",
      descriptorfile="airline.desc",
      extraCols="Age")
```

Initially takes about 28 minutes to run, **the first time only.** Subsequent accesses are very fast.

## Step 1: Read in the Data

`read.big.matrix` inherits from `read.table` so your favorite parameters are available for use.
It also adds a few more:

1. `type`, the C++ type to use for the matrix.
2. `separated`, separate the columns into individual files if `TRUE`.
3. `extracols` **explicitly adds columns to the matrix that you may use later.**
4. `backingfile`, `backingpath` and `descriptorfile` control where important data about the matrix is stored on disk.

## Step 2a: Some Initial Tasks

Next, estimate the birthmonth of the plane using the first departure of that plane.

```
birthmonth <- function(y) {
      minYear <- min(y[,'Year'], na.rm=TRUE)
      these <- which(y[,'Year']==minYear)
      minMonth <- min(y[these,'Month'], na.rm=TRUE)
      return(12*minYear + minMonth - 1)
}
```

## Step 2b v1: Calculate Each Plane's Birthmonth the Dumb Way

We could use a loop or possible an `apply` variant...

```r
aircrafts <- unique(x[,'TailNum'])
acStart <- rep(0, length(aircrafts))
for (i in aircrafts) {
 acStart[i]<-birthmonth( x[mwhich(x, 'TailNum', i, 'eq'),
                   c('Year', 'Month'), drop=FALSE])
}
```

...this takes about 9 hours...

## Step 2b v2: Calculate Each Plane's Birthmonth the `big` Way

First separate/divide the data by tail number (aircraft ID).

```
library(bigtabulate)
acindices <- bigsplit(x, 'TailNum')
```

Each entry *i* of `acindices` contains a vector of indices corresponding to `TailFin` *i*.

`bigsplit` runs about twice as fast as `split` (6s) and requires about 2/3 peak RAM.

## Step 2c v1: Compute an Estimate of Birthmonth using sapply

Now that we have used bigsplit to quickly split up the big.matrix, we can use the native sapply:

```
acStart <- sapply(acindices, function(i)
    birthmonth(x[i, c('Year','Month'), drop=
    FALSE]))
```

which took a mere 8 seconds!

## Step 2c v2: Compute an Estimate of Birthmonth using `foreach`

We can also use this opportunity to revisit `foreach`:

```
library(doMC)
registerDoMC(cores=2)
acStart <- foreach(i=acindices, .combine=c)
      %dopar% {
   return(birthmonth(x[i, c('Year', 'Month'),
      drop=FALSE]))
}
```

Both cores share access to the same instance of the data *smoothly*. `Year` and `Month` are cached in RAM. This example took 0.663s.

## Step 3: Finally, Compute an Estimate of Age

```
x[,'Age'] <- x[,'Year']*as.integer(12) + x[,'Month'] -
    as.integer(acStart[x[,'TailNum']])
```

Here, arithmetic is conducted using R vectors that are extracted from the `big.matrix`. Careful use of `as.integer` helps reduce memory overhead.

This data management task requires 8 minutes.

## Before we Continue: The Point

The point of this demonstration:

1. show how, by using file descriptors and file-backed matrices, working with big data is easy.
2. show how `bigmemory` can be integrated with parallelism packages relatively easily.
3. show how other `big` packages assist with the purpose of the `bigmemory` package to provide the user the familiar R interface after performing some initial maintenance.

## The big Example 2: Linear Models

Now we will try to predict arrival delay (`ArrDelay`) using `Age` and `Year`.

$$\widehat{\text{ArrDelay}} = \hat{\beta}_0 + \hat{\beta}_1 \text{Age} + \hat{\beta}_2 \text{Year}$$

`biganalytics` provides a wrapper to the `biglm` package.

## The big Example 2: Linear Models

Fitting a linear model looks relatively familiar...

```
library(biganalytics)
blm <- biglm.big.matrix(ArrDelay ~ Age + Year,
     data=x)
```

1. Without bigmemory, this process would take about 10GB of RAM.
2. Expected time to completion would be $\infty$ since most systems do not have that much RAM.
3. With bigmemory processing took 4.5 minutes with a few hundred MB of memory overhead.

## The big Example 2: Linear Models

Just like with base lm, we can get information about the fitted model.

```
> blm
Large data regression model: biglm(formula = formula, data = data, ...)
Sample size =  6452
> summary(blm)
Large data regression model: biglm(formula = formula, data = data, ...)
Sample size =  6452
                 Coef      (95%        CI)        SE      p
(Intercept) 6580.4311 1916.9282 11243.9340 2331.7515 0.0048
Age            0.2687    0.0804     0.4569    0.0941 0.0043
Year          -3.2753   -5.6005    -0.9500    1.1626 0.0048
```

## bigmemory: Other Features

- can write a big.matrix to ASCII file using write.big.matrix.
- can bin data for counting, creating histograms or visualizations using binit.
- create a copy of the content using deepcopy.
- can create a hash into a big.matrix using hash.mat
- search a matrix using mwhich including powerful comparison operators using C++, not R.
- separated columns: columns of a matrix are separated in RAM, rather than contiguous.

## `bigmemory`: Important Considerations

- passing a `big.matrix` to a function is *call by reference* **not** *call by value*!
- **Nerd alert!** `bigmemory` provides transparent read/write locking to `big.matrix`, so race conditions are minimized.

For more information, see `http://www.bigmemory.org`, or the documentation.

## `bigmemory`: Conclusion

**Advantages**

1. can store a matrix in memory, restart R, and gain access to the matrix *without reloading data*. **Great for big data.**
2. can share the matrix among multiple R instances or sessions.
3. access is **fast** because RAM is fast. C++ also helps!

**Disadvantages**

1. no communication among instances of R; can use files instead.
2. limited by available RAM, unless using `filebacked.big.matrix`.
3. matrix disappears on reboot, unless using `filebacked.big.matrix`.
4. filesize limitations on 32 bit systems for filebacked matrices.

## `ff`: "fast access files"

`ff` is another solution that is based on using files.

- provides data structures that are stored on disk.
- they act as if they are in memory; only necessary/active parts of the data from disk are mapped into main memory.
- supports R standard atomic types: `double`, `logical`, `raw`, `integer`,
- as well as non-standard atomic types `boolean` (1 bit), `quad` (2 bit unsigned), `nibble` (4 bit unsigned), `byte` (1 byte signed with `NA`), `ubyte` (1 byte unsigned), `short` and `ushort` (1 byte signed w/`NA` and unsigned resp.), and `single` (4 byte float with `NA`).
- *and* non-atomic types such as `factor`, `ordered`, `POSIXct`, `Date`, etc.

## ff

- C support for vectors, matrices and arrays. (FAST)
- provides an analog for `data.frame` called `ffdf`, also with import/export functionality.
- `ff` objects can be stored and reopened across R sessions.
- `ff` files can be shared by multiple `ff` R objects in the same, or different R sessions.
- tons of optimizations provide little noticeable overhead.
- Virtual functions allow matrix operations without touching a single byte on disk.
- **Disk I/O is SLOW**. `ff` optimizes by using binary files.
- closely integrated with package `bit` which can manipulate and process, well, bits.

## ff: The Logic

In bigmemory R keeps a pointer to a C++ matrix. The matrix is stored in RAM or on disk. In ff R keeps *metadata* about the object, and the object is stored in a flat binary file.

ff is somewhat difficult to jump into because there are so few examples and only one tutorial. There is a lot of information about the technical side of the package.

In any case, the goal is to get rid of the following error message:

```
> x <- rep(0, 2^31 - 1)
Error: cannot allocate vector of length
    2147483647
```

## `ff`

The package `ff` performs the following functionality to the user:

- creating and/or opening flat files using `ffopen`. Using the parameters `length` or `dim` will create a new file.
- I/O operations using the common brackets `[ ]` notation.
- Other functions for `ff` objects such as the usual `dim` and `length` as well as some other useful functions such as `sample`.

## ff Example: Introduction

Let's start with an introductory demonstration. To create a one-dimensional flat file,

```r
library(ff)
#creating the file
my.obj <- ff(vmode="double", length=10)
#modifying the file.
my.obj[1:10] <- iris$Sepal.Width[1:10]
```

**Let's take a look...**

## ff Example: Introduction

We can also create a multi-dimensional flat file.

```
#creating a multidimensional file
multi.obj <- ff(vmode="double", dim=c(10, 3))
multi.obj[1:10, 1] <- iris[1:10,1]
```

**Let's take a look...**

## ff Example: Introduction

We can also create an ff data frame made up of ff atomics.

```
Girth <- ff(trees$Girth)
Height <- ff(trees$Height)
Volume <- ff(trees$Volume)
#Create data frame with some added parameters.
fftrees <- ffdf(Girth=Girth,Height=Height,Volume=Volume)
```

**Let's take a look...**

## `ff` Parameters

`ff`, `ffm` and to some degree `ffdf` support some other options you may need. R is usually pretty good at picking good defaults for you. Some of them are below:

| Parameter | Description |
|---|---|
| `initdata` | Value to use to initialize the object for construction. |
| `length` | Optional length of vector. Used for construction. |
| `vmode` | Virtual storage mode (makes big difference in memory overhead). |
| `filename` | Give a name for the file created for the object. |
| `overwrite` | If TRUE, allows overwriting of file objects. |

If no `filename` is given, a new file is created. If a filename is given and the file exists, the object will be loaded into R.

## ff Saving Data

We can use the ffsave function to save a series of objects to file. The specified objects are saved using save and are given the extension .RData. The ff files related to the objects are saved and zipped using an external zip application, and given the extension .ffData. ffsave has some useful parameters. See ?ffsave for more information:

```
ffsave(..., list = character(0L), file = stop("'file' must be specified",
     envir = parent.frame(), rootpath = NULL, add = FALSE,
     move = FALSE, compress = !move, compression_level = 6,
     precheck=TRUE)}
```

- ... the objects to be saved.
- the file parameters specifies the root name (no extension) for the file. It is best to use absolute paths.
- add=TRUE to add these objects to an existing archive.
- compress=TRUE to save and compress.
- safe=TRUE to write a temporary file first for verification, then move to a persistent location.

## `ff`: Other File I/O Operations

Before discussing `ffload` in a few slides, there are some other operations worth mentioning.

- `ffsave.image` allows the user to save the entire workspace to an `ffarchive`.
- `ffinfo`, when passed the path for an `ffarchive` (without extension) displays information about the archive.
- `ffdrop` allows the user to delete an `ffarchive` (no extension).

## ff Example

As we saw earlier, we can use biglm to fit a linear model to big data.

```
library(biganalytics)
model <- biglm(log(Volume)~log(Girth)+
    log(Height),data=fftrees)
```

**Demonstration here.**

## ff: Conclusion

**Advantages**

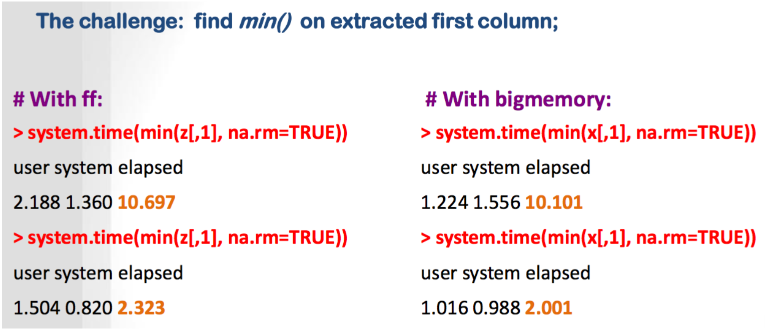
1. allows R to work with multiple HUGE datasets.
2. clean system; does not make a mess with a ton of files.
3. several optimizations show that `ff` has a bright future.

**Disadvantages**

1. few examples; somewhat difficult to introduce.
2. performing analysis may require some clever forethought since not all of the data is in RAM.
3. unzipping files on load is a huge bottleneck.

## bigmemory vs. ff

Which one to use is a matter of taste. Performance is about the same: the first row of numbers is the initial processing and the second uses caching:



**The challenge: find *min()* on extracted first column;**

```
# With ff:
> system.time(min(z[,1], na.rm=TRUE))
user system elapsed
2.188 1.360 10.697
> system.time(min(z[,1], na.rm=TRUE))
user system elapsed
1.504 0.820 2.323
```

```
# With bigmemory:
> system.time(min(x[,1], na.rm=TRUE))
user system elapsed
1.224 1.556 10.101
> system.time(min(x[,1], na.rm=TRUE))
user system elapsed
1.016 0.988 2.001
```

Source: `http://www.agrocampus-ouest.fr/math/useR-2009/slides/Emerson+Kane.pdf`

## MapReduce

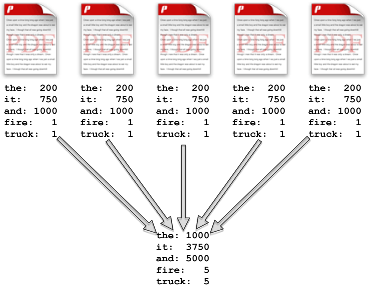
MapReduce is a way of dividing a large job into many smaller jobs producing an output, and then combining the individual outputs into one output. It is a classic *divide and conquer* approach that is *embarrasingly parallel* and can easily be *distributed* among many cores or a CPU, or among many CPUs and nodes. Oh, and it is patented by Google, its biggest user.
Two fundamental steps:

1. **map** step: perform some operation $f$, in parallel, on data and output a key/value pair for each record/row.
2. **reduce** step: group common elements and compute some summary statistic, one per group.

The notion of **map** and **reduce** comes from Lisp and functional programming, where `map` performs some operation on every element in a collection, and `reduce` collapses the results of that operation into one result for the collection.

## Example: Word Counts

Suppose we have 5 manuscripts and we want to count number of times each word occurs. (This is a common task in data mining and natural language processing). The **map** phase parses the text and produces output like `word: count`.
The **reduce** phase then groups records by `word` and sums (the reduce operation) the counts.



## mapReduce

`mapReduce` is a *pure R* implementation of MapReduce. A matter of fact, the authors of the package state that `mapReduce` is simply:

```
apply(map(data), reduce)
```

By default, `mapReduce` uses the same parallelization functionality as `sapply` which is not preferrable.

## mapReduce

The form of the `mapReduce` function is very simple.

```
mapReduce(map, ..., data, apply = sapply)
```

- `map` is an expression that yields a vector that partitions data into groups. Can use a single variable as well. This is a bit different from Hadoop's implementation.
- `...` is one *or more* reduce functions; typically summary statistics.
- `data` is a `data.frame`.
- `apply` is the parallelization toolkit to use: `papply`, `multicore`, `snow`.

## mapReduce Demonstration

```
data(iris)
mapReduce(
  map=Species,
  mean.sepal.length=mean(Sepal.Length),
  max.sepal.length=max(Sepal.Length) ,
  data = iris
)
```

```
           mean.sepal.length max.sepal.length
setosa                 5.006              5.8
versicolor             5.936              7.0
virginica              6.588              7.9
```

**Demonstration here.**

## Hadoop

Hadoop is an open-source implementation of MapReduce that has gained a lot of traction in the data mining community.

`http://hadoop.apache.org`

Cloudera and Yahoo provide their own implementations of Hadoop, which may be easier to use in the cloud:

`http://www.cloudera.com/developers/downloads/hadoop-distro/`

`http://developer.yahoo.com/hadoop/distribution/`

## Hadoop Infrastructure

Hadoop is a separate open source project, so will not discuss it in detail.

1. Hadoop allows use of parallelism to solve problems using big data.
2. Hadoop is most effective with a cluster and Hadoop assigns certain machines critical tasks.
3. Hadoop uses its own filesystem by default, called HDFS.
4. Map and reduce functions are the same as mentioned in the Intro to Map-Reduce slide.
5. Map, reduce, job control, logging etc. methods are written in Java.

We will exploit some workarounds to deal with point #5.

## Hadoop Installation

Installing Hadoop on most systems is simple.

1. Download the tarball from
   http://www.apache.org/dyn/closer.cgi/hadoop/core/
2. Extract using `tar xf`.
3. Add the Hadoop `bin` directory to your path.
4. Set environment variable `JAVA_HOME` to point to the location of system Java (Oracle/Sun preferred).

Hadoop is then ready to run **locally**. To run on a cluster requires some more configuration beyond the scope of this talk.

## HadoopStreaming

Typically, Hadoop jobs are written in Java. This was a roadblock to many developers that would incur time having to port code over to Java.

Hadoop is distributed with *Hadoop Streaming*, which allows map/reduce jobs to be written in *any* language as long as it can read and write from `stdin` and `stdout` respectively.

This includes R. Cue the `HadoopStreaming` package!

## HadoopStreaming

One way to easily use Map/Reduce in R is with the HadoopStreaming package. The workflow is as follows:

1. Open a connection: to a file, STDIN, or a pipe.
2. Write map and reduce functions either in one file, or in separate files.
3. Run the job on data either from the command line using R only, or with Hadoop.

## Data Mining with R

The data for this demonstration is a large 14GB file containing a tweet ID and the content of a user's tweet.

```
tweetID tweet_text
```

The typical "hello world" example for map/reduce is word counting, so let's construct the a list of words and the number of times they appear over all tweets in the sample.

## `HadoopStreaming`: The Map Function

The map function takes a line of input, does something with it, and outputs a key/value pair that is sorted and passed to the reducer. For this example, I split the text of the tweet and output each word as it appears, and the number 1 to denote that the word was seen once.

The **key** is the word, and the **value** is the intermediate count (1 in this case).

## `HadoopStreaming`: The Reduce Function

Before data enters the reduce phase, it is sorted on its key. The reduce function takes a key/value pair and performs an aggregation on each value associated with the same key and writes it to disk (or HDFS with Hadoop) in some format specified by the user.

In the output, the “**key**” is a word and the “**value**” is the number of times that word appeared in all tweets in the file.

## HadoopStreaming: Anatomy of a HadoopStreaming Job

First, we create a file that will contain the map and reduce functions.

```
#! /usr/bin/env Rscript    #allows script to be
    EXECUTABLE.

library(HadoopStreaming)   #need the package.

#user can create own command line arguments.
#By default, certain arguments are already parsed for you
    .
opts <- c()
#Gets arguments from the environment
op <- hsCmdLineArgs(opts, openConnections=TRUE)
```

## HadoopStreaming: Anatomy of a HadoopStreaming Job

When running an R script, we can pass command line arguments. For example:

```
./mapReduce.R -m
```

| Short Name | Character | Argument | Type | Description |
|---|---|---|---|---|
| mapper | m | None | logical | Runs the mapper. |
| reducer | r | None | logical | Runs the reducer. |
| infile | i | Required | character | Input file, otherwise STDIN. |
| outfile | o | Required | character | Output file, otherwise STDOUT. |

Some selected arguments are above.

## HadoopStreaming: Anatomy of a HadoopStreaming Job

op is populated with a bunch of command line arguments for you. If `-m` is passed to a script, `op$mapper` is TRUE and the mapper is run.

```
if (op$mapper) {
    mapper <- function(x) {
        #tokenize each tweet
        words <- unlist(strsplit(x, "[[:punct:][:space
            :]]+"))
        words <- words[!(words=='')]
        #Create a data frame with 1 column: the words.
        df <- data.frame(Word=words)
        #Add a column called count, initialized to 1.
        df[,'Count'] = 1
        #Send this out to the console for the reducer.
        hsWriteTable(df[,c('Word','Count')], file=op
            $outcon, sep=',')
    }
    #Read a line from IN.
    hsLineReader(op$incon, chunkSize=op$chunksize, FUN=
        mapper)
}
```

## HadoopStreaming: Anatomy of a HadoopStreaming Job

op is populated with a bunch of command line arguments for you. If -r is passed to a script, op$reducer is TRUE and the reducer is run.

```r
else if (op$reducer) {
    #Define the reducer function.
    #It just prints the word, the sum of the counts
    #separated by comma.
    reducer <- function(d) {
        cat(d[1,'Word'], sum(d$Count), '\n', sep=',')
    }
    #Define the column names and types for output.
    cols = list(Word='', Count=0)
    hsTableReader(op$incon, cols, chunkSize=op$chunkSize,
         skip=0,
        sep=',', keyCol='Word', singleKey=T, ignoreKey=F,
        FUN=reducer)
}
```

## HadoopStreaming: Anatomy of a HadoopStreaming Job

Finally, we clean after ourselves and close the connections to the in and out connections.

```r
if (!is.na(opts$infile)) {
  close(opt$incon)
}

if (!is.na(opt$outfile)) {
  close(opt$outcon)
}
```

## HadoopStreaming: Running a HadoopStreaming Job

HadoopStreaming allows the user to pass data to map/reduce from the command line using a pipe, or using a file. To input data using a pipe:

```
cat twitter.tsv | ./count.R -m | sort | ./count.R -r
```

To input data using a file:

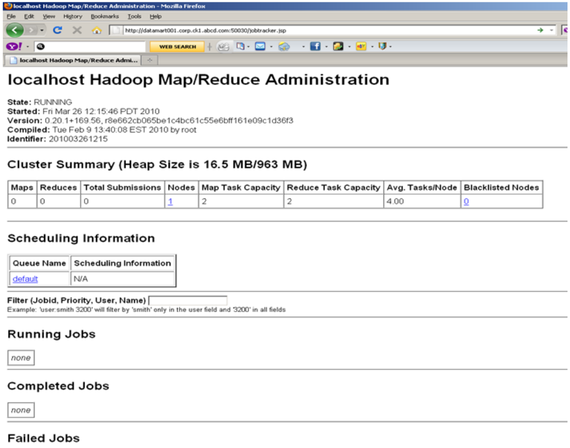
```
./count.R -m -i twitter.tsv | sort | ./count.R -r
```

## HadoopStreaming: Running a HadoopStreaming Job

Or we can run the script using, you know, Hadoop Streaming!

```
hadoop jar $HADOOP_HOME/contrib/streaming/hadoop-0.20.2-streaming.jar \
     -input /home/ryan/hdfs/in \
     -output ~/hdfs/out \
     -mapper "count.R -m" \
     -reducer "count.R -r" \
     -file ./count.R
```

Hadoop produces a lot of status and progress output and provides a web interface that you can explore when using it.


localhost Hadoop Map/Reduce Administration - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://datamart001.corp.ck1.abcd.com:50030/jobtracker.jsp
WEB SEARCH
localhost Hadoop Map/Reduce Admi...

localhost Hadoop Map/Reduce Administration

**State:** RUNNING
**Started:** Fri Mar 26 12:15:46 PDT 2010
**Version:** 0.20.1+169.56, r8e662cb065be1c4bc61c55e6bff161e09c1d36f3
**Compiled:** Tue Feb 9 13:40:08 EST 2010 by root
**Identifier:** 201003261215

Cluster Summary (Heap Size is 16.5 MB/963 MB)

| Maps | Reduces | Total Submissions | Nodes | Map Task Capacity | Reduce Task Capacity | Avg. Tasks/Node | Blacklisted Nodes |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 2 | 2 | 4.00 | 0 |

Scheduling Information

| Queue Name | Scheduling Information |
|---|---|
| default | N/A |

**Filter (Jobid, Priority, User, Name)**
Example: 'user:smith 3200' will filter by 'smith' only in the user field and '3200' in all fields

Running Jobs

*none*

Completed Jobs

*none*

Failed Jobs

## HadoopStreaming: Running a HadoopStreaming Job

Output looks as follows.

```
a,13
about,2
action,1
acutely,1
adapting,1
affairs,1
after,1
again,2
ah,2
Ah,1
```

## HadoopStreaming vs. Rhipe

Rhipe is another R interface for Hadoop that provides a more "native" feel to it.

1. incorporates an rhlapply function similar to the standard apply variants.
2. uses Google Protocol Buffers.
3. seems to have great flexibility in modifying Hadoop parameters.
4. has more use cases and flexibility in how to run jobs and how to transmit and receive data.
5. but... is more complicated to use and requires a more in-depth knowledge of Hadoop, which is beyond the scope of the group's purpose.

## The Hadoop Barnyard

Hadoop also has several other projects that run on top of it:

- **Pig**: data-flow (query) language and execution framework for parallel computing.
- **ZooKeeper**: high-performance coordination service for distributed applications.
- **Hive**: data warehouse infrastructure providing summarization and ad-hoc querying.
- **HBase**: a scalable, distributed database supporting structured data storage for large tables.
- **Avro**: data serialization providing dynamic integration with scripting languages.
- **Chukwa**: data collection system for managing large distributed systems.

## The Hadoop Barnyard

There are other packages that can interface with Hadoop, but are part of a different family.

- **Mahout**: suite of scalable machine learning libraries.
- **Nutch**: provides web search and crawling application software on top of Lucene.
- **Lucene**: an efficient indexer for information retrieval.

## When to Use What?

This is my personal opinion based on experience with both R and Hadoop.

- For datasets with size in the range 10GB, `bigmemory` and `ff` handle themselves well.
- For "larger" datasets, use Hadoop (integrated with R?)
- Not enough research on data on the scale of TB or PB in R. Hadoop is superior here.

## Other Solutions: Hardware

There are other solutions to *some* of these problems.

- Buy more RAM... lots of it... lots of fast RAM
- Let your system run out of memory and configure it to use solid state drives (SSD) for swap?
- Use a very expensive SAS drive.
- use a GPU.

## Other Solutions: Software

- (use the packages we discussed today)
- use databases that can be called from R.
- if data is sparse, use sparse matrix packages `sparseM` or `slam`.
- use a different language like C/C++/FORTRAN and interface with R.
- use a different language entirely.
- Revolution R is beta testing big dataset support.
- use Hadoop or Amazon EC2 or Elastic MapReduce with(out) R
- use SAS

## In Conclusion

1. R provides ways to deal with big data.
2. They are fairly easy to use.
3. Worth learning as R gains popularity and datasets grow huge in size.

## Keep in Touch!

My email: `ryan@stat.ucla.edu`

My blog: `http://www.bytemining.com`

Follow me on Twitter: `@datajunkie`

## The End

Questions?

Thank You!

## References

Most examples and descriptions can from package documentation. Other references not previously mentioned:

1. "Big Data in R" by G Stat: March 24, 2010.
   `http://statistics.org.il/wp-content/uploads/2010/04/Big_Memory%20V0.pdf`
2. "The `ff` Package: Handling Large Data Sets in R with Memory Mapped Pages of Binary Flat Files" by Adler, Nenadic, Zucchini, Glaser: UseR! 2007.
   `http://user2007.org/program/presentations/adler.pdf`
3. "The Bigmemory Project" by Michael Kane and John Emerson: April 29, 2010.
   `http://cran.r-project.org/web/packages/bigmemory/vignettes/Overview.pdf`
4. "Airline On-time Performance." Data Expo 2009.
   `http://stat-computing.org/dataexpo/2009/the-data.html`.