Lazy frames: quickly extract subsets from large text files

Bryan W. Lewis
blewis@illposed.net

October 23, 2011

1 Introduction

I've been working with some moderately-sized text files recently. The files are each over two gigabytes with about 20 million rows, with columns separated by commas (CSV) or tabs. My computer has plenty of memory for R to load each file, but it takes a while and I'm impatient.

Now, I don't really need the entire data set in memory for my work. I just need to filter the data a bit and then sample from the rows. I think that this situation is typical enough–wanting fast access to subsets of large text files–that I wrote this package for it.

The `lazy.frame` package lets me quickly and efficiently work with subsets from a text file without loading the entire file into memory. A `lazy.frame` is a data frame promise. It presents a text file as a kind of simple data frame, without first loading the file into memory. Lazy frames load data from their backing files on demand. They are essentially wrappers for the `read.table` function with a few extra convenience functions. I probably should have called this "promise.frame," but I liked the sound of "lazy.frame" better.

There are several compelling R packages for working directly with file-backed data: The LaF package is quite similar to lazy.frames, reading data from CSV files on demand into data frames. But `LaF` is not cross-platform, has more limited file handling, differs from the `read.table` syntax, and most importantly lacks the filtering functions available in lazy frames. The bigmemory package by Emerson and Kane provides a memory mapped matrix object, free from R indexing constraints, and a comprehensive suite of fast analysis functions. The nicely simple and powerful (a superb combination!) mmap package by Jeff Ryan defines a data frame-like memory mapped object. And the venerable ff package by Adler, Oehlschlägel, et. al. defines a variety of memory mapped data frame-like objects and functions. All of these packages have really interesting features. Most of them are designed to facilitate working with objects larger than the physical RAM available on a computer.

But recall, my data sets easily fit into the RAM on my computer (RAM is really cheap)! My main irritation is the bottleneck incurred by parsing the entire data set, which isn't really avoided by the above packages (although the packages do include methods to help expedite loading data from text files).

A notable alternate method for efficiently processing very large text files splits the files manually into a set of files each containing a subset of rows of the original. Lazy frames provide the same capability, with much less work on the part of the user (no manual file processing).

Of course, lazy frames aren't a panacea and have limitations discussed below. The benefit of using lazy frames diminishes as the size of the extracted subsets grow. Thus, lazy frames are very good for extracting relatively small subsets. For *really* large data sets, or for more sophisticated operations involving all the data, `bigmemory` is a better option. Lazy frames work well with text files with between roughly a million and a hundred million or so rows.

2 Using Lazy Frames

Lazy frames are *good* for very efficiently extracting small subsets from large delimited text files. They are *bad* for use by computations that need all of the data–for that either pay the price up frontand load the data, or use one of the alternate file-backed methods discussed in the Introduction.

I can think of at least two applications that lazy frames are good for:

1. Quickly filtering a raw data set to get to a subset of interest (discarding the rest).
2. Developing models for imbalanced data sets, which involves filtering and specialized bootstrapping.

The second application is one approach to modeling an outcome that occurs only rarely in the data. Such problems arise in fraud detection and many other areas. One approach to constructing models of rare outcomes is to use a bootstrap technique that selects approximately equal resampled population sizes from the rare cases and majority cases.

There is another interesting aspect of the second application related to parallel computation. If the bootstrapped function is computationally expensive, lazy frames can help overlay I/O and computation–that is, keeping one process busy with selecting the next resampled subset, while another process evaluates the function on the current subset.

2.1 Overview

A lazy frame is basically a data frame promise that loads data on demand. Lazy frames are created with the `lazy.frame` function. Its options are mostly equivalent to the options for `read.table`–lazy frames directly use `read.table` to parse their backing data files.

The example shown in Listing 1 writes the `iris` data set to a CSV file and creates a lazy frame from that file. Any standard column delimiter may be used in place of comma. Lazy frames support all of the `read.table` options. In particular, the example uses `header=TRUE`, indicating that the first data file row contains column names, and `row.names=1`, indicating that the first column in the data file contains row names.

Note that I'll use the variable `x` defined in Listing 1 in subsequent examples.

Listing 1: Basic use.

```
> library("lazy.frame")
> data(iris)
> f = tempfile()
> write.table(iris, file=f, sep=",")

> x = lazy.frame(f, header=TRUE, row.names=1)
> head(x)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa

> dim(x)
[1] 150   5
```

Subsets of lazy frames are normal data frames. Indexing works mostly like normal data frames with a few exceptions:

- The $ operator is not supported for indexing columns.
- Leaving the row index blank to select all rows is not supported in the same way as with normal data frames–instead this returns a lazy frame again. If you *really* want all the rows, explicitly specify a start and end index (but this kind of defeats the purpose of using lazy frames!).

Listing 2 shows some examples of extracting subsets from the variable `x` defined in Listing 1.

Listing 2: Indexing

```
> s = sample(nrow(x),5,replace=TRUE)
> x[s, ]
    Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
83           5.8         2.7          3.9         1.2 versicolor
89           5.6         3.0          4.1         1.3 versicolor
147          6.3         2.5          5.0         1.9  virginica
123          7.7         2.8          6.7         2.0  virginica
60           5.2         2.7          3.9         1.4 versicolor

> x[1:3,c("Petal.Length","Petal.Width")]
  Petal.Length Petal.Width
1          1.4         0.2
2          1.4         0.2
3          1.3         0.2
```

2.2 Special comparison operations

Lazy frames provide a few very basic comparison operations that work on single columns. The operations are useful for basic data filtering. These operations *only* apply to one column at a time and are limited to the comparisons: $< \quad > \quad \leq \quad \geq \quad != \quad ==$ for numeric, integer, and character values.

Recall that comparisons on data frame columns return a vector of Boolean values with the result of the comparison for each row. Unlike data frames, lazy frames return a set of numeric row indices for which the comparisons are true (or `NULL` if all rows evaluated false), just like the `which` command.

Listing 3 shows an example that picks out rows of the iris data set with `Sepal.length < 4.5`.

Listing 3: Indexing

```
> x[x[,"Sepal.Length"] < 4.5, ]
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
9           4.4         2.9          1.4         0.2  setosa
14          4.3         3.0          1.1         0.1  setosa
39          4.4         3.0          1.3         0.2  setosa
43          4.4         3.2          1.3         0.2  setosa
```

3 Quirks and Limitations

Lazy frames are simple-minded cousins of data frames. They act like data frames in many ways, but also exhibit significant deviations from data frame behavior summarized here.

- Lazy frames are read only.
- Column name indexing with $ is not supported.
- Comparison operations that involve all rows are limited to basic comparisons for a single column only, and only for numeric, integer, and character values.
- Comparison operations return a set of indices like `which` instead of a vector of Boolean values.
- Row names are supported, but they must be from a column in the data file (they can't be independently specified).
- Factor variables are supported–but for them to make sense, the levels must be manually specified using the `column_attr` function.
- Lazy frames shouldn't be used directly by functions that expect data frames–use subsets of lazy frames instead.

4 Examples

I present a few examples that compare indexing operations on lazy frames with indexing operations on data frames read in by `read.table`. The machines used in each example are described below. In order to minimize disk caching effects between tests, the command

```
echo 3 > /proc/sys/vm/drop_caches
```

(wiping clean the Linux disk memory cache) was issued just before each test.

4.1 A medium-sized example

The example used a CSV file with about 18 million rows and 27 columns. Two of the columns were character, three double precision numeric, and the remaining integer valued.

The experiments in this section were conducted on a fairly old and slow 2 GHz, four CPU core AMD Opetron computer with 12 GB of DDR-2 RAM running Ubuntu 9.10 GNU/Linux and R version 2.12.1. The data files resided on a Fusion-io ioXtreme solid state disk rated at 700 MB/s data read rate and 80 μs read latency in the first set of tests.

I used `read.table` with and without defining column classes to read the data into a data frame from an uncompressed file. As expected, specifying column classes in `read.table` reduced the load time by more than 20% in this example, and greatly reduced the maximum memory consumption during loading from almost 8 GB to under 5 GB (note that the data set itself only requires about 2 GB to store in R). Without column classes, it took over 11 minutes to load the data in. Specifying column classes reduced that to about 9 minutes.

Once loaded, I extracted a subset of about 95 thousand rows in which the 20th column had values greater than zero. It took about 27 seconds to extract the subset.

Lazy frame took only about 4 seconds to "load" the same file, and about 53 seconds to extract the same row subset. Thus, we see the penalty of lazily loading data from the file–it took about twice as long to extract the subset in this example. But, we avoided the substantial initial load time almost completely. And, the maximum memory used by the R session was limited to about the 18 MB memory required to hold the subset, substantially reducing required memory overhead. Indeed, the lazy frame example runs fine on a machine with 4 GB RAM.

Listing 4: Extract a subset from a lazy frame.

```
> library("lazy.frame")
> t1 = proc.time()
> x  = file.frame(file="test.csv")
> print(proc.time() - t1)
   user  system elapsed
   2.34    2.05    4.39

> print(gc())
         used (Mb) gc trigger (Mb) max used (Mb)
Ncells 140517  7.6     350000 18.7   350000 18.7
Vcells 130910  1.0     786432  6.0   531925  4.1

> print(dim(x))
[1] 17826159       27

> t1 = proc.time()
```

```
> y  = x[x[,20]>0, ]
> print(proc.time() - t1)
   user  system elapsed
 40.870  11.770  52.709

> print(dim(y))
[1] 95166    27
```

Listing 5: Extract a subset from a data frame loaded with `read.table`.

```
> t1 = proc.time()
> x  = read.table(file="test.csv",header=FALSE,sep=",",stringsAsFactors=FALSE)
> print(proc.time() - t1)
   user  system elapsed
648.380  33.350 682.699

> print(gc())
            used   (Mb) gc trigger   (Mb)   max used   (Mb)
Ncells    138089    7.4     667722   35.7     380666   20.4
Vcells 285413776 2177.6  832606162 6352.3 1034548528 7893.0

> print(dim(x))
[1] 17826159       27

> t1 = proc.time()
> y  = x[x[,20]>0, ]
> print(proc.time() - t1)
  user  system elapsed
 27.87    2.41   30.31

> print(dim(y))
[1] 95166    27
```

Listing 6: Extract a subset from a data frame loaded with read.table with defined column classes.

```
> cc = c("numeric","integer","integer","integer","integer",
         "integer","integer","integer","integer","character",
         "character","integer","integer","integer","integer",
         "integer","integer","integer","integer","integer",
         "integer","integer","integer","numeric","integer",
         "numeric","integer")
> t1 = proc.time()
> x  = read.table(file="test.csv",header=FALSE,sep=",",stringsAsFactors=FALSE,
   colClasses=cc)
> print(proc.time() - t1)
   user  system elapsed
443.290  82.780 526.141

> print(gc())
            used   (Mb) gc trigger   (Mb)  max used   (Mb)
Ncells    138519    7.4     350000   18.7    350000   18.7
Vcells 285348278 2177.1  649037152 4951.8 641872298 4897.1

> print(dim(x))
[1] 17826159       27

> t1 = proc.time()
> y  = x[x[,20]>0, ]
> print(proc.time() - t1)
   user  system elapsed
 28.410   2.180  30.593

> print(dim(y))
[1] 95166    27
```

4.2 A pretty large example

This example used the concatenated airline data set from http://stat-computing.org/dataexpo/2009/. The data consists of flight arrival and departure details for all commercial flights within the USA, from October 1987 to April 2008. It's a relatively large example with about 120 million rows and 29 columns taking up 12 GB (uncompressed).

This example used an Amazon high-memory EC2 instance with 32 GB RAM and four Intel Xeon CPU X5550 cores operating at 2.67 GHz, running Ubuntu GNU/Linux 11.04 and R version 2.12.1. Except where otherwise indicated, the data file was placed in a RAM-based tmpfs file system to completely eliminate disk effects.

Despite plenty of available memory, I was unable to load the data set into R which always failed after about 16 minutes with the error:

Listing 7: Extract a subset from a data frame loaded with `read.table` with defined column classes.

```
Error in scan(file, what, nmax, sep, dec, quote, skip, nlines, na.strings,  :
  could not allocate memory (2048 Mb) in C function 'R_AllocStringBuffer'
```

The same error occurred after moving the uncompressed raw data file to disk in order to free up a maximum amount of RAM (nearly 32 GB).

Frustrated, I took the advice of the American Statistical Association in the Data Expo challenge and imported the data into sqlite to be read by R using the RSQLite package. This process is documented here: http://stat-computing.org/dataexpo/2009/sqlite.html. The raw CSV file was loaded from RAM to eliminate disk I/O effects.

The sqlite3 data load time without creating indices took about 18 minutes.

The sqlite3 data load time, including creation of a single index on the year field took about 26 minutes.

Lazy frame