bigmemoRy: an R package supporting massive data sets

Michael J. Kane
Yale University

John W. Emerson
Yale University

Abstract

Multi-gigabyte data sets challenge and frustrate R users even on well-equipped hardware. C programming can provide memory efficiency and speed improvements, but is cumbersome for interactive data analysis and lacks the flexibility and power of R's rich statistical programming environment. The new package **bigmemoRy** bridges this gap, implementing persistent massive objects in memory (managed in R but implemented in C) and supporting the manipulation and exploration of these objects.

Keywords: R, C, memory, data, statistics, data mining.

1. IntRoduction

A numeric matrix containing 100 million rows and 3 columns consumes approximately 2.4 gigabytes (GB) of memory in the R statistical programming environment (1). Such massive, multi-gigabyte data sets challenge and frustrate R users even on well-equipped hardware. The C programming language allows quick, memory-efficient operations on massive objects, without the memory overhead of many R operations. Unfortunately, the C language is not well-suited for interactive data exploration, lacking the flexibility, power, and convenience of R's rich environment.

The new package **bigmemoRy** bridges the gap between R and C, implementing persistent massive objects in memory (managed in R but implemented in C) and supporting their manipulation and exploration. An API is also provided, allowing savvy C programmers to extend the functionality of **bigmemoRy**.

As of 2007, typical high-end personal computers (PCs) have 1-4 GB of random access memory (RAM) and run 32-bit operating systems. A small number of PCs might have more than 4 GB of memory and 64-bit operating systems, and such configurations are now common on servers and high-performance computing clusters. At Google, for example, Daryl Pregibon's group uses 64-bit Linux workstations with up to 32 GB of RAM. Massive data sets are increasingly common; the Netflix Prize[1] competition, for example, involves the analysis of approximately 100 million customer ratings of movies, and a basic data structure would be a 100 million by 5 matrix of integers (movie ID, customer ID, rating, year, and month). And Daryl's group studies massive subsets of terrabytes (though perhaps not googols) of data.

Fast-forward to year 2015, eight years hence. A naive application of Moore's Law projects a sixteen-fold increase (four doublings) in hardware capacity. R will naturally support much

[1] http://www.netflixprize.com

larger data analyses, taking advantage of additional hardware capacity. But these increases only tempt us to push the envelope at the frontiers of applied research.

With **bigmemoRy**, we can exploit simultaneously the strengths of R and C to the full capacity of our hardware. Most significantly, R users of **bigmemoRy** don't need to be C experts (and don't have to use C at all, in most cases). And C programmers can make use of R as a convenient interface, without needing to become experts in the environment. Thus, **bigmemoRy** offers something for everyone.

2. Some backgRound

Data frames and matrices in R were designed for data sets much smaller in size than the computer's memory limit. They are flexible and easy to use, with typical manipulations executed quickly on smaller data sets. They suit the needs of the vast majority of R users.

A second category of data sets are those requiring more memory than a machine's RAM. CRAN and Bioconductor packages such as **DBI**, **RJDBC**, **RMySQL**, **RODBC**, **ROracle**, **TSMySQL**, **filehashSQLite**, **TSSQLite**, **pgUtils**, and **Rdbi** communicate with previously implemented databases using SQL statements. Other packages, such as **filehash** and **ff**, provide a convenient `data.frame`-like interface. While they all help manage massive data sets, the user is often forced to wait for disk accesses.

This paper addresses a third category of data sets. These can be massive data sets (perhaps requiring several gigabytes of memory on typical computers, as of 2007) but not larger than the total available RAM. In this case, disk accesses are unnecessary. However, there may not be enough RAM to handle the overhead of using a `data.frame` or `matrix`. The next section will outline some of R's limitations for this type of data set. The `BigMatrix` class has been created to fill this niche.

3. DangeRs of data.frame and matrix

We love R's data frames and matrices. The observations made in this section are not complaints, because there are excellent reasons behind choices made in the design and implementation of the programming environment. Here, we illustrate some dangers of R's memory management. We will use a large matrix of integers as an example, motivated by the Netflix Prize competition.

Many R users are not aware that 4-byte integers are implemented in R. So if memory is at a premium and only integers are required, the astute and fastidious user would avoid the 2.24 GB memory consumption of

```
> x <- matrix(0, 1e+08, 3)
> round(object.size(x)/(1024^3), 2)

[1] 2.24
```

in favor of the 1.12 GB memory consumption of an integer matrix:

```
> x <- matrix(as.integer(0), 1e+08, 3)
> round(object.size(x)/(1024^3), 2)
```

```
[1] 1.12
```

Similar attention is needed in subsequent arithmetic, because

```
> x <- x + 1
```

coerces `x` into a matrix of 8-byte real numbers. The memory used is then back up to 2.24 GB, and the peak memory usage of this operation is approximately 3.35 GB (1.12 GB from the original `x`, and a new 2.24 GB vector). In contrast,

```
> x <- matrix(as.integer(0), 1e+08, 3)
> x <- x + as.integer(1)
```

has the desired consequence, adding 1 to every element of `x` while maintaining the integer type. The memory usage then remains at 1.12 GB, and the operation requires temporary memory overhead of an additional 1.12 GB. It's tempting to be critical of *any* memory overhead for such a simple operation, but this is often necessary in a high-level, interactive programming environment. In fact, R is being efficient in the previous example; the peak memory consumption of

```
> x <- matrix(as.integer(0), 1e+08, 3)
> x <- x + matrix(as.integer(1), 1e+08, 3)
```

is approximately 3.35 GB. Some overhead is necessary in order to provide flexibility in the programming environment, freeing the user from the explicit structures of programming languages like C. However, this overhead becomes cumbersome with massive data sets.

Similar challenges arise in function calls, where R uses the *call-by-value* convention instead of *call-by-reference*. Fortunately, R creates physical copies of objects only when apparently necessary (called *copy-on-demand*). So there is no extra memory overhead for

```
> x <- matrix(as.integer(0), 1e+08, 3)
> myfunc1 <- function(z) return(c(z[1, 1], nrow(z), ncol(z)))
> myfunc1(x)

[1]         0 100000000         3
```

which uses only 1.12 GB for the matrix `x`, while

```
> x <- matrix(as.integer(0), 1e+08, 3)
> myfunc2 <- function(z) {
+     z[1, 1] <- as.integer(5)
+     return(c(z[1, 1], nrow(z), ncol(z)))
+ }
> myfunc2(x)

[1]         5 100000000         3
```

temporarily creates a second 1.12 GB matrix, for a total peak memory usage of 2.24 GB.

These examples demonstrate the simultaneous strengths and weaknesses of R. As a programming environment, it frees the user from the rigidity of a programming language like C. The resulting power and flexibility has a cost: data structures can be inefficient, and even the simplest manipulations can create unexpected memory overhead. For most day-to-day use, the strengths far outweigh the weaknesses. But when working with massive data sets, the weaknesses can cause even the best efforts to grind to a halt.

4. bigmemoRy: the method behind the madness

We define an `S4` class `BigMatrix`, containing an external pointer (R type `externalptr`) and a type (a character string describing the type of the atomic element). The pointer gives the memory location of a C structure such as `BigIntMatrix` (the only structure yet implemented). The `BigIntMatrix` structure contains the number of rows and columns of the matrix, a vector of strings corresponding to matrix column names, and a pointer to an array of pointers to integers (of length equal to the number of columns of the matrix). The structure, including the massive vectors of data (one for each column of the matrix), are dynamically allocated using `malloc()`. We recognize that this is not the typical representation of a matrix as a massive vector, but it has some natural advantages for data exploration and analysis.

We provide a collection of functions for the class `BigMatrix`, designed to mimic traditional matrix functionality. For example, the R function `ncol()` automatically associates a `BigMatrix` with a new method function, returning the number of columns of the matrix via the C function `CGetIntNcol()`. Other examples are more subtle. For example, if `x` is a `BigMatrix` and `a` and `b` are traditional R vectors of row and column indices, then `x[a,b]` can be used to retrieve the appropriate submatrix of `x` (and is a traditional R matrix, not a `BigMatrix`). The same notation can be used to assign values to the submatrix, if the dimensionality of the assignment conforms to R's set of permissible matrix assignments.

An object of class `BigMatrix` contains a *reference* to a data structure. This has some noteworthy consequences. First, if a `BigMatrix` is passed to a function which changes its values, those changes will be reflected outside the function scope. Thus, *side-effects* have been introduced for `BigMatrix` objects. When an object of class `BigMatrix` is passed to a function, it should be thought of as being *called-by-reference*.

The second consequence is that copies are not made by simple assignment. For example:

```
> library(bigmemoRy)
> x <- BigMatrix(nrow = 2, ncol = 2, init = 0)
> x[, ]

     [,1] [,2]
[1,]    0    0
[2,]    0    0

> y <- x
> y[1, 1] <- 1
> x[, ]
```

```
     [,1] [,2]
[1,]    1    0
[2,]    0    0
```

The assignment `y <- x` does not copy the contents of the `BigMatrix` object; it only copies the type information and memory location of the data. As a result, the variables `x` and `y` refer to the same data in memory. If an actual copy is required, the programmer must first create a new object of class `BigMatrix`, and then copy the contents.

Along with bracket operator, the core functions supporting `BigMatrix` objects are:

```
BigMatrix()    is.BigMatrix()   as.BigMatrix()   nrow()
ncol()         dim()            head()           tail()
print()        which.bm()       read.bm()        write.bm()
rm.cols.bm()   add.cols.bm()    hash.mat.bm()    dimnames()
```

Other basic functions are included, serving as templates for the development of new functions. These include:

```
colmin()      min()     max()       colmax()
colrange()    range()   colmean()   mean()
colvar()      colsd()   summary()   biglm.bm()
bigglm.bm()
```

5. Package bigmemoRy in use

One of the most important new functions is `which.bm()`. For a `BigMatrix`, `x`, the logical expression `x[,2]==0` would extract the second column of the matrix as a massive numeric vector in R, do the logical comparison in R, and produce a massive R vector of `TRUE` and `FALSE` values. Thus, the command `x[x[,2]==0,]` would generate considerable memory overhead. The command `x[which.bm(x, 2, 0),]` gives the identical result with only a vector of indices of length `sum(x[,2]==0)` existing temporarily in R. More complex comparisons are supported by `which.bm()`, including the specification of minimum and maximum test values and comparisons on multiple columns in conjunction with `AND` and `OR` operations. Even more complex comparisons could involve set operations in R on collections of indices returned by `which.bm()` from C.

We use the Netflix Prize data as an example. This data set includes 103,297,638 ratings and five integer variables: movie ID, customer ID, rating, year and month. As a regular R matrix, this would require approximately 4 GB of RAM, whereas only 2 GB is needed for the `BigMatrix`.

```
> x <- read.bm("ALL.txt", sep = "\t")
> colnames(x) = c("movie", "customer", "rating", "year", "month")
> cust.indices <- which.bm(x, "customer", 50)
> length(cust.indices)

[1] 625
```

Customer 50 provided 625 ratings. We can extract other subsets of the data. For example we really might want both customers 50 and 55 (an OR command):

```
> cust.indices <- which.bm(x, rep("customer", 2), c(50, 55), op = "OR")
> length(cust.indices)

[1] 1004
```

Support for Thomas Lumley's **biglm** package (2) is provided via the `biglm.bm` and `bigglm.bm` functions. In this example, the movie release year is used (as a factor) to try to predict customer ratings:

```
> lm.0 = biglm.bm(rating ~ year, data = x, fc = "year")
> print(summary(lm.0)$mat)

                  Coef        (95%         CI)           SE            p
(Intercept)  3.64455545  3.64426881  3.64484210 0.0001433232 0.000000e+00
year2004    -0.05304919 -0.05353212 -0.05256626 0.0002414644 0.000000e+00
year2003    -0.23921791 -0.23995324 -0.23848257 0.0003676688 0.000000e+00
year2001    -0.25430020 -0.25593524 -0.25266517 0.0008175193 0.000000e+00
year2002    -0.26335461 -0.26442104 -0.26228819 0.0005332123 0.000000e+00
year2000    -0.27979480 -0.28203994 -0.27754967 0.0011225672 0.000000e+00
year1999    -0.30754903 -0.35345442 -0.26164363 0.0229526996 6.107488e-41
```

This particular regression will not win the $1,000,000 Netflix prize. However, it does provide an interesting use of a `BigMatrix` object to manage and study several gigabytes of data.

6. Conclusion

bigmemoRy currently supports integer types and will soon be extended for double, short, and char data. The implementation of short and char types will provide the most significant memory efficiencies. The first version, including these data types, will be available from CRAN in January 2008.

The next version of **bigmemoRy** will implement the data structures in shared memory. At the moment, parallel use of R requires redundant copies of data for each R process. The shared memory version of a `BigMatrix` object will allow separate R processes on the same computer to share access to a single copy of the massive data set in memory. This will open the door for more powerful parallel analyses and data mining of massive data sets.

References

[1] R Development Core Team (2007). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org.

[2] Thomas Lumley (2005). biglm: bounded memory linear and generalized linear models. R package version 0.4.