bigmemory: bigger, better, and platform-independent

John W. Emerson "Jay"

Associate Professor
Department of Statistics
Yale University
john.emerson@yale.edu
http://www.stat.yale.edu/~jay/

Collaborator:
Michael J Kane
Yale University

Abstract

The newly re-engineered package bigmemory uses the Boost Interprocess C++ library to provide platform independent support for massive matrices. These matrices may be allocated to shared memory with transparent read and write locking. In addition, bigmemory now supports file-backed matrices, ideal for applications exceeding available RAM.

Not all of the following slides will be presented during the talk, but we wanted to make them available online.

ASA 2009 Data Expo: Airline on-time performance

http://stat-computing.org/dataexpo/2009/

- Flight arrival and departure details for all* commercial flights within the USA, from October 1987 to April 2008.
- Nearly 120 million records, 29 variables (mostly integer-valued)
- We preprocessed the data, creating a single CSV file, recoding the carrier code, plane tail number, and airport codes as integers.

* Not really. Only for carriers with at least 1% of domestic flights in a given year.

Hardware used in the examples

Yale's "Bulldogi" cluster:

- 170 Dell Poweredge 1955 nodes
- 2 dual-core 3.0 Ghz 64-bit EM64T CPUs
- 16 GB RAM each node
- Gigabit ethernet with both NFS and a Lustre filesystem
- Managed via PBS

Yale

Bulldogi

This laptop (it ain't light):

- Dell Precision M6400
- Intel Core 2 Duo Extreme Edition
- 4 GB RAM (a deliberate choice)
- Plain-vanilla primary hard drive
- 64 GB solid state secondary drive

Laptop

REvolution computing

ASA 2009 Data Expo: Airline on-time performance

120 million flights by 29 variables ~ 3.5 billion elements. Too big for an R matrix (limited to 2^31 – 1 ~ 2.1 billion elements and likely to exceed available RAM, anyway).

Hadley Wickham's recommended approach: ***sqlite***

Upcoming alternative: ***ff***

We used version 2.1.0 (beta)

ff matrix limited to 2^31-1 elements;

ffdf data frame works, though.

Others: ***BufferedMatrix***, ***filehash***,

many database interface packages;

R.huge will no longer be supported.

Airline on-time performance via *bigmemory*

Via *bigmemory* (on CRAN): creating the filebacked big.matrix

Note: as part of the creation, I add an extra column that will be used for the calculated age of the aircraft at the time of the flight.

```
> x <- read.big.matrix(“AirlineDataAllFormatted.csv”,
                      header=TRUE, type=“integer”,
                      backingfile=“airline.bin”,
                      descriptorfile=“airline.desc”,
                      extraCols=“age”)
```

~ 25 minutes

Laptop

Airline on-time performance via *sqlite*

Via *sqlite* (http://sqlite.org/): preparing the database

```
Revo$ sqlite3 ontime.sqlite3
SQLite Version 3.6.10 …
sqlite> create table ontime (Year int, Month int,
             …, origin int, …, LateAircraftDelay int);
sqlite> .separator ,
sqlite> .import AirlineDataAllFormatted.csv ontime
sqlite> delete from ontime where typeof(year)==“text”;
sqlite> create index origin on ontime(origin);
sqlite> .quit
Revo$
```

~ 75 minutes excluding the **`create index`**.

Laptop

A first comparison: *bigmemory* vs *RSQLite*

Via *RSQLite* and *bigmemory*, a column minimum?

The result: *bigmemory* wins.

```
> library(bigmemory)
> x <- attach.big.matrix(
         dget(“airline.desc”) )
> system.time(colmin(x, 1))
   user  system elapsed
  0.236   0.372   7.564
> system.time(a <- x[,1])
   user  system elapsed
  0.852   1.060   1.910
> system.time(a <- x[,2])
   user  system elapsed
  0.800   1.508   9.246
```

```
> library(RSQLite)
> x <- attach.big.matrix(
+                dget(“airline.desc”) )
> ontime <- dbConnect(“SQLite”,
+              dbname=“ontime.sqlite3”)
> from_db <- function(sql) {
+               dbGetQuery(ontime, sql)
+            }
> system.time(from_db(
+      “select min(year) from ontime”))
   user  system elapsed
 45.722  14.672 129.098
> system.time(a <-
+   from_db(“select year from ontime”))
   user  system elapsed
 59.208  20.322 138.132
```

Laptop

Airline on-time performance via *ff*

Example: *ff* (Dan Adler et.al., Beta version 2.1.0)

```
> library(bigmemory)
> library(filehash)
> x <- attach.big.matrix(dget(“airline.desc”))
> y1 <- ff(x[,1], filename="ff1")
> y2 <- ff(x[,2], filename="ff2")
…
> y30 <- ff(x[,30], filename="ff30")
> z <- ffdf(y1,y2,y3,y4,y5,y6,y7,y8,y9,y10,
+            y11,y12,y13,y14,y15,y16,y17,y18,y19,y20,
+            y21,y22,y23,y24,y25,y26,y27,y28,y29,y30)
```

With apologies to Adler et.al, we couldn’t figure out how to do this more elegantly, but it worked (and, more quickly – 7 minutes, above – than you’ll see with the subsequent two examples with other packages). As we noted last year at UseR!, an function like ***read.big.matrix()*** would greatly benefit ***ff***.

Laptop

Airline on-time performance via *ff*

Example: *ff* (Dan Adler et.al., Beta version 2.1.0)

The challenge: R's *min()* on extracted first column; caching.

The result: they're about the same.

```
# 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
```

Laptop

Airline on-time performance via *ff*

Example: *ff* (Dan Adler et.al., Beta version 2.1.0)

The challenge: alternating *min()* on first and last rows.

The result: maybe an edge to *bigmemory*, but do we care?

```
> # With bigmemory:
> system.time(min(x[1,],na.rm=TRUE))
   user  system elapsed
  0.004   0.000   0.071
> system.time(min(x[nrow(x),],
              na.rm=TRUE))
   user  system elapsed
  0.000   0.000   0.001
> system.time(min(x[1,],na.rm=TRUE))
   user  system elapsed
  0.000   0.000   0.001
> system.time(min(x[nrow(x),],
              na.rm=TRUE))
   user  system elapsed
  0.000   0.000   0.001
```

```
> # With ff:
> system.time(min(z[1,],na.rm=TRUE))
   user  system elapsed
  0.040   0.000   0.115
> system.time(min(z[nrow(z),],
+             na.rm=TRUE))
   user  system elapsed
  0.032   0.000   0.099
> system.time(min(z[1,],na.rm=TRUE))
   user  system elapsed
  0.020   0.000   0.024
> system.time(min(z[nrow(z),],
              na.rm=TRUE))
   user  system elapsed
  0.036   0.000   0.080
```

Laptop

Airline on-time performance via *ff*

Example: *ff* (Dan Adler et.al., Beta version 2.1.0)

The challenge: random extractions, two runs (time two):

```
> theserows <- sample(nrow(x), 10000)
> thesecols <- sample(ncol(x), 10)
>
> # With ff:
> system.time(a <- z[theserows,
+                    thesecols])
   user  system elapsed
  0.092   1.796  60.574
> system.time(a <- z[theserows,
+                    thesecols])
   user  system elapsed
  0.040   0.384   4.069

> # With bigmemory:
> system.time(a <- x[theserows,
+                    thesecols])
   user  system elapsed
  0.020   1.612  64.136
> system.time(a <- x[theserows,
+                    thesecols])
   user  system elapsed
  0.020   0.024   1.323
```

```
> theserows <- sample(nrow(x), 100000)
> thesecols <- sample(ncol(x), 10)
>
> # With ff:
> system.time(a <- z[theserows,
+                    thesecols])
   user  system elapsed
  0.352   3.305  78.161
> system.time(a <- z[theserows,
+                    thesecols])
   user  system elapsed
  0.340   3.156  77.623

> # With bigmemory:
> system.time(a <- x[theserows,
+                    thesecols])
   user  system elapsed
  0.248   2.752  78.935
> system.time(a <- x[theserows,
+                    thesecols])
   user  system elapsed
  0.248   2.676  78.973
```

Laptop

Airline on-time performance via *filehash*

Example: *filehash* (Roger Peng, on CRAN)

```
> library(bigmemory)
> library(filehash)
> x <- attach.big.matrix(dget(“airline.desc”))
> dbCreate(“filehashairline”, type=“RDS”)
> fhdb <- dbInit(“filehashairline”, type=“RDS”)
> for (i in 1:ncol(x))
+   dbInsert(fhdb, colnames(x)[i], x[,i])   # About 15 minutes.
```

```
> system.time(min(fhdb$Year))
   user  system elapsed
 11.317   0.236  11.584
> system.time(min(fhdb$Year))
   user  system elapsed
 11.744   0.236  11.987
```

```
> system.time(min(x[,"Year"]))
  user  system elapsed
 1.128   1.616   9.758
> system.time(min(x[,"Year"]))
  user  system elapsed
 0.900   0.984   1.891
> system.time(colmin(x, "Year"))
  user  system elapsed
 0.184   0.000   0.183
```

***filehash* is quite memory-efficient on disk!**

Laptop

Airline on-time performance via *BufferedMatrix*

Example: *BufferedMatrix* (Ben Bolstad, on BioConductor)

```
> library(bigmemory)
> library(BufferedMatrix)
> x <- attach.big.matrix(dget(“airline.desc”))
> y <- createBufferedMatrix(nrow(x), ncol(x))
> for (i in 1:ncol(x)) y[,i] <- x[,i]
```

More than 90 minutes to fill the *BufferedMatrix*; inefficient (only 8-byte numeric is supported); not persistent.

```
> system.time(colmin(x))
   user  system elapsed
  4.576   4.560 113.289
> system.time(colMin(y))
   user  system elapsed
 20.926  71.492 966.952
```

```
> system.time(colmin(x, na.rm=TRUE))
   user  system elapsed
 11.264   9.645 256.911
> system.time(colMin(y, na.rm=TRUE))
   user  system elapsed
 39.515  70.436 941.229
```

Laptop

More basics of *bigmemory*

A ***big.matrix*** is a lot like a ***matrix***…

```
> library(bigmemory)
> xdesc <- dget("airline.desc")
> x <- attach.big.matrix(xdesc)
> dim(x)
[1] 118914458        30
> colnames(x)
 [1] "Year"               "Month"              "DayofMonth"
 [4] "DayOfWeek"          "DepTime"            "CRSDepTime"
 [7] "ArrTime"            "CRSArrTime"         "UniqueCarrier"
[10] "FlightNum"          "TailNum"            "ActualElapsedTime"
[13] "CRSElapsedTime"     "AirTime"            "ArrDelay"
[16] "DepDelay"           "Origin"             "Dest"
… (rows omitted for this slide)
> tail(x, 1)
             Year             Month        DayofMonth         DayOfWeek
             2008                 4                17                 4
          DepTime        CRSDepTime           ArrTime        CRSArrTime
              381               375               472               754
    UniqueCarrier         FlightNum           TailNum ActualElapsedTime
               11              1211              2057                91
   CRSElapsedTime           AirTime          ArrDelay          DepDelay
               99                64                -2                 6
           Origin              Dest          Distance            TaxiIn
               63                35               430                15
… (rows omitted for this slide)
```

More basics of *bigmemory*

```
> ##########################################################
> # Can we get all flights from JFK to SFO?  Sure!
>
> a <- read.csv("AirportCodes.csv")
> a <- na.omit(a)
> JFK <- a$index[a$airport=="JFK"]
> SFO <- a$index[a$airport=="SFO"]
>
> gc(reset=TRUE)
         used (Mb) gc trigger  (Mb) max used (Mb)
Ncells 214256 11.5     407500  21.8   214256 11.5
Vcells 169064  1.3   29629238 226.1   169064  1.3
> system.time(
+   y <- x[x[,"Origin"]==JFK & x[,"Dest"]==SFO,]
+ )
   user  system elapsed
   6.50    5.23   11.74
> dim(y)
[1] 99867    30
> gc()
          used (Mb) gc trigger   (Mb)  max used   (Mb)
Ncells  214242 11.5     407500   21.8    220478   11.8
Vcells 1667071 12.8  220757362 1684.3 241395930 1841.8
> rm(y)
```

Slower and less memory-efficient than our alternative: ***mwhich()***, coming up next…

mwhich()

```
> ##########################################################
> # mwhich() for fast, no-overhead "multi-which"
>
> gc(reset=TRUE)
          used (Mb) gc trigger   (Mb) max used (Mb)
Ncells 214238 11.5     407500   21.8   214238 11.5
Vcells 169034  1.3  176605889 1347.4   169034  1.3
> system.time(
+   y <- x[mwhich(x, cols=c("Origin", "Dest"),
+                     vals=list(JFK, SFO),
+                     comps="eq",
+                     op="AND"), ]
+ )
   user  system elapsed
  5.270   0.020   5.308
> dim(y)
[1] 99867    30
> gc()
           used (Mb) gc trigger  (Mb) max used (Mb)
Ncells  214277 11.5     407500  21.8   235659 12.6
Vcells 1667109 12.8  113027768 862.4  3271422 25.0
> rm(y)
```

Fast, no memory overhead!

Bulldogi

mwhich(): useful with R matrices, too!

```
> #####################################################
> # mwhich() works on a matrix, too, but I can't
> # hold all the data as an R matrix, even if I had
> # the RAM (see earlier comment on size). On a subset:
>
> xx <- x[,15:18]
> gc(reset=TRUE)
            used   (Mb) gc trigger   (Mb)  max used   (Mb)
Ncells    203561   10.9     407500   21.8    203561   10.9
Vcells 237996106 1815.8  499861463 3813.7 237996106 1815.8
> system.time(
+   y <- xx[mwhich(x, cols=c("Origin", "Dest"),
+                    vals=list(JFK, SFO),
+                    comps="eq",
+                    op="AND"), ]
+ )
   user  system elapsed
  5.220   0.000   5.219
> dim(y)
[1] 99867     4
> gc()
            used   (Mb) gc trigger   (Mb)  max used   (Mb)
Ncells    203566   10.9     407500   21.8    213419   11.4
Vcells 238195846 1817.3  499861463 3813.7 238448239 1819.3
```

Just as fast as with a ***big.matrix***, with no memory overhead beyond the ***matrix*** itself.

Bulldogi

Airline on-time performance: solving a problem

For each plane in the data set, what was the first month (in months A.D.) of service?

No surprises… yet.

```
> date()
[1] "Fri Jun 19 13:27:23 2009"
> library(bigmemory)
> xdesc <- dget("airline.desc")
> x <- attach.big.matrix(xdesc)
> numplanes <- length(unique(x[,"TailNum"])) - 1
> planeStart <- rep(0, numplanes)
> for (i in 1:numplanes) {
+   y <- x[mwhich(x, "TailNum", i, 'eq'),
+          c("Year", "Month"), drop=FALSE]  # Note this.
+   minYear <- min(y[,"Year"], na.rm=TRUE)
+   these <- which(y[,"Year"]==minYear)
+   minMonth <- min(y[these,"Month"], na.rm=TRUE)
+   planeStart[i] <- 12*minYear + minMonth
+ }
> date()
[1] "Fri Jun 19 22:27:36 2009"
```

~ 9 hours

Laptop

Introducing ***foreach, iterators, doMC, doSNOW, doNWS***

- Brand new, coming out of REvolution Computing

- The brainchildren of Steve Weston (who produced a subset of the slides immediately following this one)

- The following are called “parallel backends”:

  - ***doMC*** makes use of ***multicore*** (Simon Urbanek)

  - ***doSNOW*** makes use of ***snow*** (Luke Tierney, A.J. Rossini, Na Li, and H. Sevcikova)

  - ***doNWS*** makes use of NetWorkSpaces (***nws***, REvolution Computing following from Scientific Computing Associates)

foreach, iterators

```
> library(foreach)
Loading required package: iterators
Loading required package: codetools
> foreach (i=1:3) %do% { sqrt(i) }
[[1]]
[1] 1

[[2]]
[1] 1.414214

[[3]]
[1] 1.732051

> foreach (i=1:3, .combine=c) %do% { sqrt(i) }
[1] 1.000000 1.414214 1.732051
> foreach (i=1:3, .combine='+') %do% { sqrt(i) }
[1] 4.146264
```

foreach on SMP via *doMC* (master plus 4 workers)

```
> date()
[1] "Thu Jun 18 21:39:09 2009"
> library(bigmemory)
> library(doMC)
Loading required package: foreach
Loading required package: iterators
Loading required package: codetools
Loading required package: multicore
> registerDoMC()
> xdesc <- dget("airline.desc")
> x <- attach.big.matrix(xdesc)
> numplanes <- length(unique(x[,"TailNum"])) - 1
> planeStart <- foreach(i=1:numplanes, .combine=c) %dopar% {
+   require(bigmemory)
+   x <- attach.big.matrix(xdesc)
+   y <- x[mwhich(x, "TailNum", i, 'eq'),
+          c("Year", "Month"), drop=FALSE]
+   minYear <- min(y[,"Year"], na.rm=TRUE)
+   these <- which(y[,"Year"]==minYear)
+   minMonth <- min(y[these,"Month"], na.rm=TRUE)
+   12*minYear + minMonth
+ }
> date()
[1] "Fri Jun 19 00:14:36 2009"
```

VERY NEW!

- ~ 2.5 hours
- A new type of loop structure
- Some initialization

Package ***multicore*** by Simon Urbanek

Laptop

foreach on SMP via *doSNOW* (master plus 3 workers)

```
> date()
[1] "Fri Jun 19 09:10:22 2009"
> library(bigmemory)
> library(doSNOW)
Loading required package: foreach
Loading required package: iterators
Loading required package: codetools
Loading required package: snow
> cl <- makeSOCKcluster(3)
> registerDoSNOW(cl)
> xdesc <- dget("airline.desc")
> x <- attach.big.matrix(xdesc)
> numplanes <- length(unique(x[,"TailNum"])) - 1
> planeStart <- foreach(i=1:numplanes, .combine=c) %dopar% {
+   require(bigmemory)
+   x <- attach.big.matrix(xdesc)
+   y <- x[mwhich(x, "TailNum", i, 'eq'),
+          c("Year", "Month"), drop=FALSE]
+   minYear <- min(y[,"Year"], na.rm=TRUE)
+   these <- which(y[,"Year"]==minYear)
+   minMonth <- min(y[these,"Month"], na.rm=TRUE)
+   12*minYear + minMonth
+ }
> date()
[1] "Fri Jun 19 12:38:33 2009"
```

- ~ 3.5 hours
- Different parallel backend setup and registration
- Otherwise identical code to the ***doMC*** SMP version

Package ***snow*** by Luke Tierney,
A.J. Rossini, Na Li, and H. Sevcikova

Yale

REvolution computing

Laptop

foreach on SMP via ***doNWS*** (master plus 3 workers)

```
> date()
[1] "Thu Jun 18 17:42:52 2009"
> library(bigmemory)
> library(doNWS)
Loading required package: foreach
Loading required package: iterators
Loading required package: codetools
Loading required package: nws
> sl <- sleigh(workerCount=3)
> registerDoNWS(sl)
> xdesc <- dget("airline.desc")
> x <- attach.big.matrix(xdesc)
> numplanes <- length(unique(x[,"TailNum"])) - 1
> planeStart <- foreach(i=1:numplanes, .combine=c) %dopar% {
+   require(bigmemory)
+   x <- attach.big.matrix(xdesc)
+   y <- x[mwhich(x, "TailNum", i, 'eq'),
+          c("Year", "Month"), drop=FALSE]
+   minYear <- min(y[,"Year"], na.rm=TRUE)
+   these <- which(y[,"Year"]==minYear)
+   minMonth <- min(y[these,"Month"], na.rm=TRUE)
+   12*minYear + minMonth
+ }
> date()
[1] "Thu Jun 18 21:12:45 2009"
```

- ~ 3.5 hours

- A different parallel backend setup and registration

- Otherwise identical code to the ***doMC*** and ***doSNOW*** SMP versions

Laptop

foreach on cluster via ***doNWS*** (10 nodes by 3 processors)

```
# Cluster Setup:
# qsub -I -l nodes=10:ppn=3 -q sandbox
# Once launched, fire up R on master.
```

```
> date()
[1] "Thu Jun 18 18:10:37 2009"
> library(bigmemory)
> library(doNWS)
Loading required package: foreach
Loading required package: iterators
Loading required package: codetools
Loading required package: nws
> nodes <- pbsNodeList()[-1]
> sl <- sleigh(nodeList=nodes, launch=sshcmd)
> registerDoNWS(sl)
> xdesc <- dget("airline.desc")
> x <- attach.big.matrix(xdesc)
> numplanes <- length(unique(x[,"TailNum"])) - 1
> planeStart <- foreach(i=1:numplanes, .combine=c) %dopar% {
+   require(bigmemory)
+   x <- attach.big.matrix(xdesc)
+   y <- x[mwhich(x, "TailNum", i, 'eq'),
+          c("Year", "Month"), drop=FALSE]
+   minYear <- min(y[,"Year"], na.rm=TRUE)
+   these <- which(y[,"Year"]==minYear)
+   minMonth <- min(y[these,"Month"], na.rm=TRUE)
+   12*minYear + minMonth
+ }
> dput(planeStart, "planeStart30NWS.txt")
> date()
[1] "Thu Jun 18 18:51:23 2009"
```

- ~ 40 minutes (slower than expected – why?)

- No substantive code changes from the SMP version

- Different ***sleigh()*** (NetWorkSpaces) setup for cluster

Yale
Bulldogi

REvolution
computing

Big ***big.matrix***: no 2^31 row limitation

```
> R <- 3e9             # 3 billion rows
> C <- 2               # 2 columns
>
> R*C*8                # 48 GB total size
[1] 4.8e+10
>
> date()
[1] "Thu Jun 18 20:11:49 2009"
> x <- filebacked.big.matrix(R, C, type='double',
+                              backingfile='test.bin',
+                              descriptorfile='test.desc')
> x[1,] <- rnorm(C)
> x[nrow(x),] <- runif(C)
> summary(x[1,])
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
-1.7510 -1.2640 -0.7777 -0.7777 -0.2912  0.1953
> summary(x[nrow(x),])
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
0.04232 0.21080 0.37930 0.37930 0.54780 0.71630
> date()
[1] "Thu Jun 18 20:11:49 2009"
```

Bulldogi

The new package ***synchronicity***

- Locking has been removed from ***bigmemory*** itself (upcoming version 4.0 and onwards) so that packages can take advantage of synchronization mechanisms without having to install bigmemory.
  - exclusive locks
  - shared locks
  - timed locks
  - conditional locking
- Allows for the creation of Universal Unique Identifiers
- The following locking schemes have been implemented for use in ***bigmemory*** (version 4.0 and onwards).
  - no locking
  - read only (allows a ***big.matrix*** object to be read only)
  - column locking
  - row locking
- The architecture is flexible enough to allow a user to define his own mutex scheme for a ***big.matrix*** object.

Supporting linear algebra routines with *bigalgebra*

- ***bigalgebra*** (currently in development) suports linear algebra operations on ***R*** matrices as well as ***big.matrix*** objects (including various combinations) for the following operations:
  - matrix copy
  - scalar multiply
  - matrix addition
  - matrix multiplication
  - SVD
  - eigenvalues and eigenvectors
  - Cholesky fatorization
  - QR factorization
  - others?
- The routines are implemented in BLAS and LAPACK libraries

In summary: ***bigmemory*** and more

- **User-friendly, familiar interface (less user overhead than the other alternatives)**
- **Memory-efficient externalities (e.g. `mwhich`() cleverness)**
- **Shared memory will full mutexes (SMP)**
- **Distributed memory (locking to be supported via NetWorkSpaces soon; currently no mutexes)**
- **A developer tool, with access to pointers in C++ allowing integration with existing libraries (e.g. linear algebra routines).**

- ***foreach/iterators* plus *bigmemory*: a winning combination for massive data concurrent programming**

Supplemental Slides: ***`mwhich()`*** and ***bigmemory*** for developers

The following includes C++ templates, but there isn't much to learn if you want to develop analytics to be used with R matrices as well as *big.matrix* objects.

bigmemory for developers

```
template<typename T, typename MatrixType>
SEXP MWhichMatrix(MatrixType mat, long nrow, SEXP selectColumn,
                   SEXP minVal, SEXP maxVal, SEXP chkMin, SEXP chkMax,
                   SEXP opVal, double C_NA)
{
  long numSc = GET_LENGTH(selectColumn);
  double *sc = NUMERIC_DATA(selectColumn);
  double *min = NUMERIC_DATA(minVal);
  double *max = NUMERIC_DATA(maxVal);
  int *chkmin = INTEGER_DATA(chkMin);
  int *chkmax = INTEGER_DATA(chkMax);

  double minV, maxV;
  int ov = INTEGER_VALUE(opVal);
  long count = 0;
  long i,j;
  double val;
  for (i=0; i < nrow; ++i) {
    for (j=0; j < numSc; ++j)  {
      // …
      val = (double) mat[(long)sc[j]-1][i];
      // …
```

bigmemory for developers

```
SEXP MWhichBigMatrix(SEXP bigMatAddr, SEXP selectColumn, SEXP minVal,
                     SEXP maxVal, SEXP chkMin, SEXP chkMax, SEXP opVal)
{
  BigMatrix *pMat =
    reinterpret_cast<BigMatrix*>(R_ExternalPtrAddr(bigMatAddr));
  if (pMat->separated_columns())
  {
    switch (pMat->matrix_type())
    {
      case 1:
        SepBigMatrixAccessor<char> mat(*pMat);
        return MWhichMatrix<char>(mat, pMat->nrow(), selectColumn,
          minVal, maxVal, chkMin, chkMax, opVal, NA_CHAR);
      //… (cases 2 and 4 omitted here)
      case 8:
        SepBigMatrixAccessor<double> mat(*pMat);
        return MWhichMatrix<double>(mat, pMat->nrow(), selectColumn,
          minVal, maxVal, chkMin, chkMax, opVal, NA_REAL);
    }
  } else // Same type of code, but with BigMatrixAccessor
```

bigmemory for developers

```
SEXP MWhichRIntMatrix(SEXP matrixVector, SEXP nrow, SEXP selectColumn,
  SEXP minVal, SEXP maxVal, SEXP chkMin, SEXP chkMax, SEXP opVal)
{
  long numRows = static_cast<long>(INTEGER_VALUE(nrow));
  BigMatrixAccessor<int> mat(INTEGER_DATA(matrixVector), numRows);

  return MWhichMatrix<int>(mat, numRows, selectColumn, minVal, maxVal,
    chkMin, chkMax, opVal, NA_INTEGER);
}

SEXP MWhichRNumericMatrix(SEXP matrixVector, SEXP nrow, SEXP selectColumn,
  SEXP minVal, SEXP maxVal, SEXP chkMin, SEXP chkMax, SEXP opVal)
{
  long numRows = static_cast<long>(INTEGER_VALUE(nrow));
  BigMatrixAccessor<double> mat(NUMERIC_DATA(matrixVector), numRows);

  return MWhichMatrix<double>(mat, numRows, selectColumn, minVal, maxVal,
    chkMin, chkMax, opVal, NA_REAL);
}
```

(Yes, these could have been a single function with a switch statement.)

bigmemory for developers

```
template<typename T>
class SepBigMatrixAccessor
{
  public:
    SepBigMatrixAccessor( BigMatrix &bm)
    {
      _ppMat = reinterpret_cast<T**>(bm.matrix());
    }

    inline T* operator[](const unsigned long col) {
      return _ppMat[col];
    }
  protected:
    T **_ppMat;
};
```

bigmemory for developers

```
template<typename T>
class BigMatrixAccessor    // We’ll rename this MatrixAccessor
{
  public:
    BigMatrixAccessor( T* pData, const unsigned long nrow)
    {
      _pMat = pData;
      _nrow = nrow;
    }
    BigMatrixAccessor( BigMatrix &bm )
    {
      _pMat = reinterpret_cast<T*>(bm.matrix());
      _nrow = bm.num_rows();
    }
    inline T* operator[](const unsigned long col) {
      return _pMat+_nrow*col;
    }
  protected:
    T *_pMat;
    long _nrow;
};
```

For handling an R **`matrix`**

For handling a **`big.matrix`**