BufferedMatrix: Introduction

Ben Bolstad
bmb@bmbolstad.com
http://bmbolstad.com

April 27, 2025

Contents

1 Introduction

This document is intended to introduce the BufferedMatrix package and how it may be used. It is very important to know that this package provides infrastructure rather than an analysis methods as most other BioConductor packages do. It is not intended to be used directly by end-users. Instead it is aimed at developers of other packages. The main purpose of this package is to provide the BufferedMatrix object. This document will explain how to use the BufferedMatrix object at the R level. There is also a C language interface to dealing with BufferedMatrix objects, but that will not be discussed here.

2 What is a BufferedMatrix?

A *BufferedMatrix* object stores numerical data in a tabular format, with most of the data primarily stored outside main memory on the file system. Figure 1 shows that a BufferedMatrix consists of data values arranged in rows and columns. The BufferedMatrix object implemented in this package is optimized for situations where the number of rows in much larger than the number of columns. The word *Buffered* is used because frequently used parts of the BufferedMatrix may be kept in main memory for increased speed. It is intended that users of the BufferedMatrix will be unaware of what is and is not in memory. Note that although the word *Matrix* is part of the name of the object it does not imply that you may treat a BufferedMatrix object exactly like a Matrix in all situations. For instance *BufferedMatrix* objects are passed by reference rather than by value, are not designed for using with Matrix algebra and can not automatically be passed to all pre-existing functions that expect ordinary matrices (particularly functions that use C code). More details about these issues will be discussed a little later in this document.

But first it is time to explore how to use the package. As with other packages we use `library` to load the package.

```
> library(BufferedMatrix)
```

To create a BufferedMatrix we use the `createBufferedMatrix` function. This function takes up to six arguments, but only the first one must be provided. To start with use:

```
> X <- createBufferedMatrix(10000)
```

to create a BufferedMatrix with 10000 rows and 0 columns. Notice how only the number of rows must be supplied when creating a *BufferedMatrix*. This is because BufferedMatrix objects fix the number of Rows they store, but allow you to dynamically add columns when needed. It is not possible however to remove columns from a BufferedMatrix. Just like all R objects typing the name of the object will show you some basic information (or the contents) of the object. In this case

```
> X
```

```
BufferedMatrix object
Matrix size:  10000 0
Buffer size:  1 1
Directory:    /private/var/folders/r0/l4fjk6cj5xj0j3brt4bplpl40000gt/T/RtmpkJrrWp/Rbuil
Prefix:       BM
Mode: Col mode
Read Only: FALSE
Memory usage :  211  bytes.
Disk usage :  0  bytes.
```

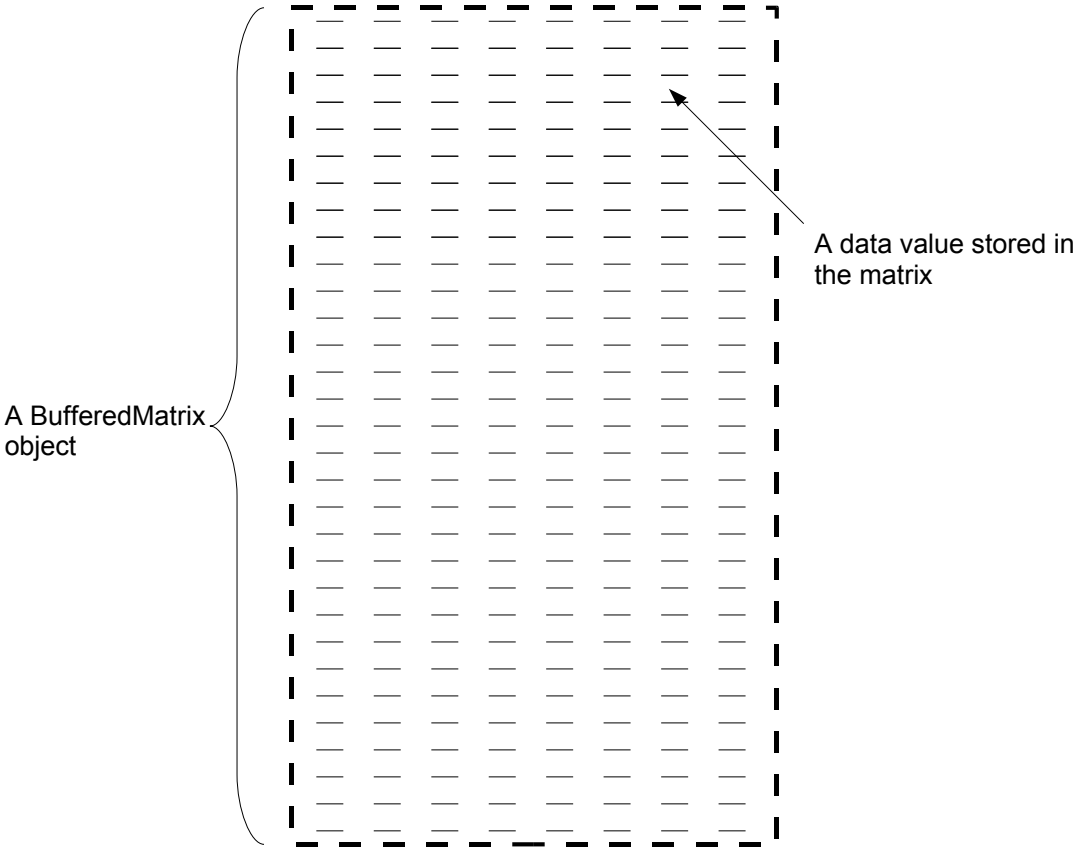

Figure 1: A BufferedMatrix stores data values in tabular format. In the figures the dashed lines means that the data is stored outside main memory on the filesystem

Notice that basic summary information about the matrix is displayed in this case. We will discuss what this information means a little later. We could use the `AddColumn` function to add columns to our matrix like this

```
> AddColumn(X)

BufferedMatrix object
Matrix size:  10000 1
Buffer size:  1 1
Directory:    /private/var/folders/r0/l4fjk6cj5xj0j3brt4bplpl40000gt/T/RtmpkJrrWp/Rbuil
Prefix:       BM
Mode: Col mode
Read Only: FALSE
Memory usage :  78.5  Kilobytes.
Disk usage :  78.1  Kilobytes.

> AddColumn(X)

BufferedMatrix object
Matrix size:  10000 2
Buffer size:  1 1
Directory:    /private/var/folders/r0/l4fjk6cj5xj0j3brt4bplpl40000gt/T/RtmpkJrrWp/Rbuil
Prefix:       BM
Mode: Col mode
Read Only: FALSE
Memory usage :  78.6  Kilobytes.
Disk usage :  156.2  Kilobytes.

> X

BufferedMatrix object
Matrix size:  10000 2
Buffer size:  1 1
Directory:    /private/var/folders/r0/l4fjk6cj5xj0j3brt4bplpl40000gt/T/RtmpkJrrWp/Rbuil
Prefix:       BM
Mode: Col mode
Read Only: FALSE
Memory usage :  78.6  Kilobytes.
Disk usage :  156.2  Kilobytes.
```

Of course it might have been more convinent just to instantiate our *BufferedMatrix* object from the beginning as having 10000 rows and 2 columns. This would be acomplished using

```
> X <- createBufferedMatrix(10000,2)
```

Of course *BufferedMatrix* objects use buffers to temporarily store some of its contents in main memory, rather than on the filesystem. There are two types of buffering provided by BufferedMatrix objects. This are *ColMode* and *RowMode*. The default mode when you create a BufferedMatrix is to be in ColMode. Figure 2 demonstrates the situation in ColMode. Specifically, the entire contents of a given number of columns are stored completely in RAM, with the rest of the data remaining on the file system. There is no requirement that these columns be contiguous. When most of your operations are to be done in a column-wise manner it is more efficient to be in ColMode.

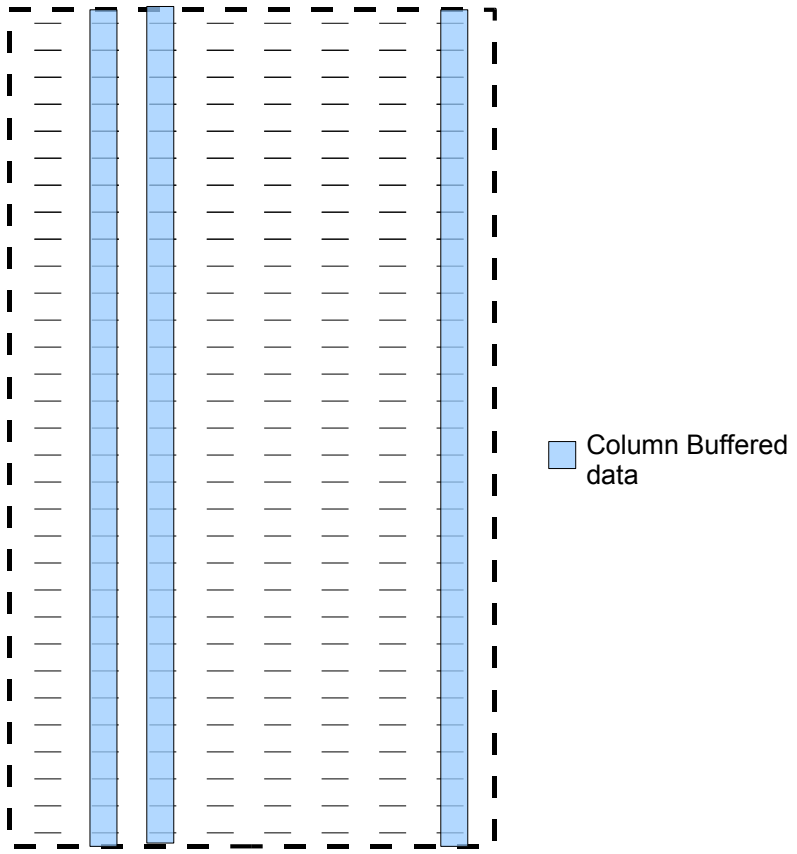

Figure 2: Column Buffers hold selected columns in RAM. This speeds up access to data. Column Buffers are always active

The other main buffering mode is RowMode. Figure 3 illustrates the situation when in RowMode. Note that when in row mode a block of contiguous rows across all columns of the BufferedMatrix is kept in main memory. RowMode is designed for situations where you need to access a fixed set of closely spaced rows of the matrix. Note that RowMode

is considered a secondary buffering mode and that the column buffers are also present when in this mode.

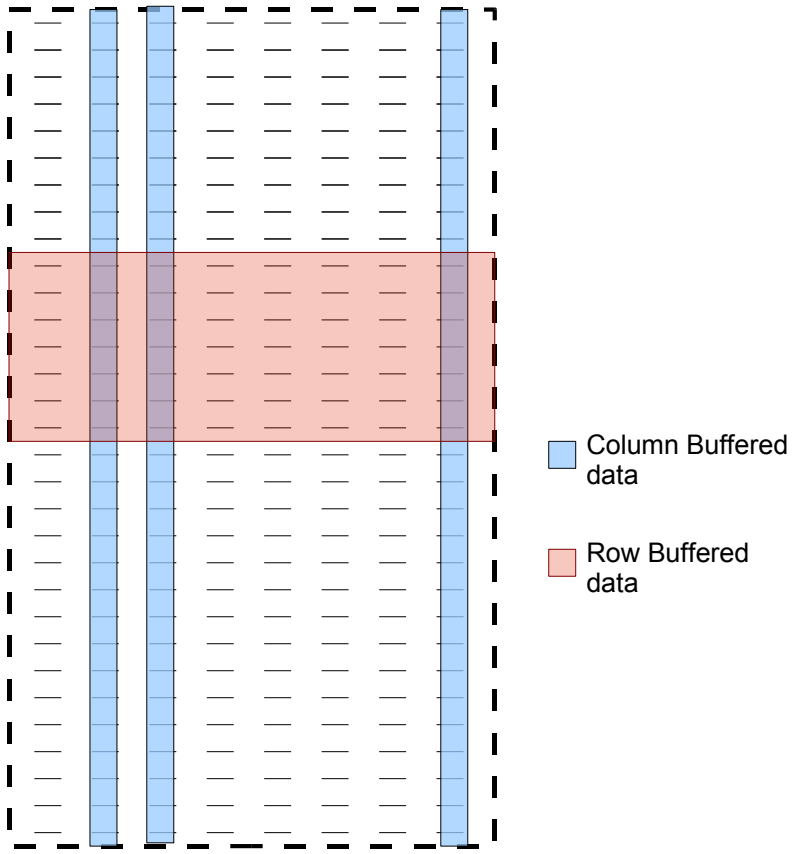

Figure 3: Row Buffers hold a selected contiguous block of rows in RAM. Row Buffers exist only when the BufferedMatrix is in RowMode.

As previously mentioned before, all *BufferedMatrix* objects start in ColMode. To switch to RowMode you can use the `RowMode` function like this:

```
> RowMode(X)

<pointer: 0x600001618000>
```

To switch back to ColMode use

```
> ColMode(X)

<pointer: 0x600001618000>
```

When you create the BufferedMatrix you can control the size of these Buffers in terms of number of columns or number of rows. In particular, suppose we wanted a 10000 by 5 matrix with 1 column buffered and 500 rows buffered when in row mode, this could be done by

```
> X <- createBufferedMatrix(10000,5,bufferrows=500,buffercols=1)
```

You can also use `set.buffer.dim` function to set or change these after the matrix has been create. For instance to change it to 2 columns buffered and 100 rows use:

```
> set.buffer.dim(X,100,2)

<pointer: 0x600001618060>
```

There are two additional arguments to `createBufferedMatrix`. The first is `prefix` which is a string that will be used to start each temporary file name used for the filesystem storage of the *BufferedMatrix*, by default this is "BM". The second is `directory` which specifies where these temporary files are to be stored. By default this directory is the current working directory.

There are several functions to get this, and additional, information about an already created *BufferedMatrix*.

```
> memory.usage(X)

[1] 160909

> disk.usage(X)

[1] 4e+05

> nrow(X)

[1] 10000

> ncol(X)

[1] 5

> dim(X)

[1] 10000     5

> buffer.dim(X)

[1] 100   2
```

```
> prefix(X)

[1] "BM"

> directory(X)

[1] "/private/var/folders/r0/l4fjk6cj5xj0j3brt4bplpl40000gt/T/RtmpkJrrWp/Rbuild18de64bd

> is.RowMode(X)

[1] FALSE

> is.ColMode(X)

[1] TRUE
```

There is one other important mode that you may switch a *BufferedMatrix* in and out of. This is *ReadOnly* mode. What this means is that you can access data values stored in the matrix but you can not change them. This provides speed-up in some situations because the current buffer does not need to be flushed out to the filesystem when an attempt is made to access a value currently not stored in the buffer. The function `ReadOnlyMode` is used to toggle between the two states.

```
> ReadOnlyMode(X)

<pointer: 0x600001618060>

> is.ReadOnlyMode(X)

[1] TRUE

> ReadOnlyMode(X)

<pointer: 0x600001618060>

> is.ReadOnlyMode(X)

[1] FALSE
```

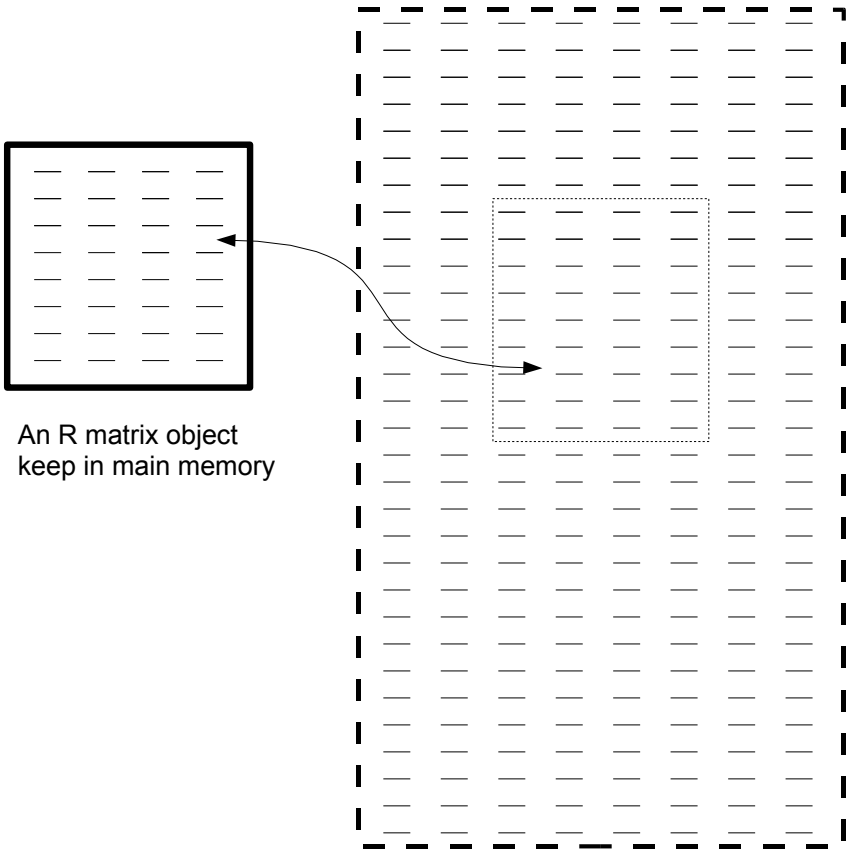

Figure 4: Subsetting operators pull data out of the BufferedMatrix object and into R matrix objects or vice versa.

3 Accessing and Manipulating data stored within a BufferedMatrix

Data is accessed from *BufferedMatrix* objects using similar indexing procedures to those you would use with a standard *matrix* object. In particular you can use the `[` aind `[<-` operators to see or replace data in an Buffered Matrix. Figure 4 demonstrates how indexing and assigment operations are set up to occur with *BufferedMatrix* objects. Specifically, when you index part of the BufferedMatrix an ordinary R matrix is created and the the data requested copied from the BufferedMatrix. This matrix is then no longer linked to the data in the BufferedMatrix.

For example

```
> X <- createBufferedMatrix(20,2)
> X[1:20,] <- 1:40
> B <- X[1:5,]
> B

     [,1] [,2]
[1,]    1   21
[2,]    2   22
[3,]    3   23
[4,]    4   24
[5,]    5   25

> B[1:2,] <- B[1:2,]^2
> B

     [,1] [,2]
[1,]    1  441
[2,]    4  484
[3,]    3   23
[4,]    4   24
[5,]    5   25

> X[1:5,]

     [,1] [,2]
[1,]    1   21
[2,]    2   22
[3,]    3   23
[4,]    4   24
[5,]    5   25
```

Similarly, assignments using the indexing operators copy the data from the R *matrix* or *vector* into the specified location of the *BufferedMatrix*.

```
> X[1:5,] <- B
> X[1:5,]
```

```
     [,1] [,2]
[1,]    1  441
[2,]    4  484
[3,]    3   23
[4,]    4   24
[5,]    5   25
```

As with ordinary R *matrix* objects we may have column or row name indices. For example:

```
> rownames(X)
```

```
NULL
```

```
> colnames(X)
```

```
NULL
```

```
> rownames(X) <- letters[1:20]
> colnames(X) <- month.abb[1:2]
```

and these can be used interchangably with numerical indexing values.

```
> X[c("a","b"),"Jan"]
```

```
  Jan
a   1
b   4
```

```
> X["t",2] <- 0
```

BufferedMatrix objects also support logical indexing. eg

```
> X[rep(c(TRUE,FALSE),10),1]
```

```
  Jan
a   1
c   3
e   5
g   7
```

```
i   9
k  11
m  13
o  15
q  17
s  19
```

Sometimes it might be useful to create another *BufferedMatrix* out of an existing one. For this purpose the `subBufferedMatrix` command may be used. This function operates in a similar manner to indexing operators, but instead of copying the data into an R matrix another `BufferedMatrix` is produced. Figure 5 illustates this procedure.

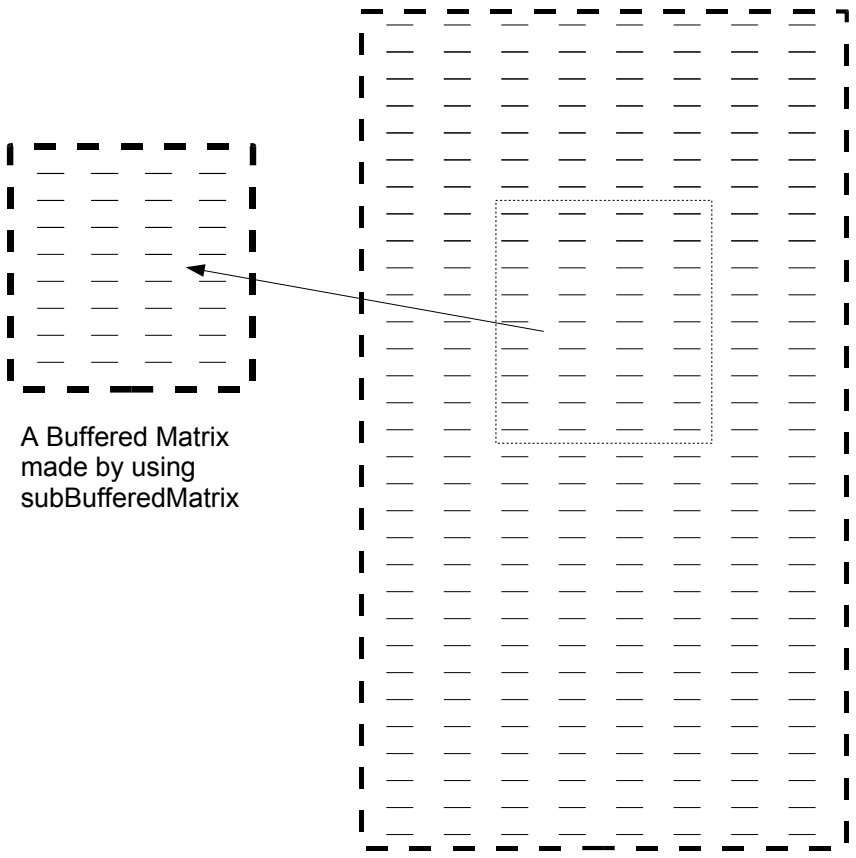

Figure 5: Using subBufferedMatrix creates a new BufferedMatrix containing a subset of the data stored in the BufferedMatrix

```
> Y <- subBufferedMatrix(X,1:5,1:2)
> Y
```

```
BufferedMatrix object
Matrix size:  5 2
Buffer size:  1 1
Directory:    /private/var/folders/r0/l4fjk6cj5xj0j3brt4bplpl40000gt/T/RtmpkJrrWp/Rbuil
Prefix:       BM
Mode: Col mode
Read Only: FALSE
Memory usage :  533  bytes.
Disk usage :  80  bytes.
```

4 Functions for summarizing a BufferedMatrix

Sometimes it is useful to generate summary statistic values from a *BufferedMatrix* object. While it would be possible to do this by writing your own code using loops and indexing a number of more optimized functions have been provided for this purpose. These functions concentrate on getting the maximum, minimum, means, variances and so forth. The try to minimize buffer misses as much as possible.

```
> X <- createBufferedMatrix(10,3)
> X[1:10,] <- (1:30)^2
> Max(X)

[1] 900

> Min(X)

[1] 1

> mean(X)

[1] 315.1667

> Sum(X)

[1] 9455

> Var(X)

[1] 79106.83

> Sd(X)

[1] 281.2594
```

It may also be useful to get these values for each column or each row.

```
> rowMeans(X)

 [1] 187.6667 210.6667 235.6667 262.6667 291.6667 322.6667 355.6667 390.6667
 [9] 427.6667 466.6667

> colMeans(X)

[1]  38.5 248.5 658.5

> rowSums(X)

 [1]  563  632  707  788  875  968 1067 1172 1283 1400

> colSums(X)

[1]  385 2485 6585

> rowVars(X)

 [1]  51733.33  60933.33  70933.33  81733.33  93333.33 105733.33 118933.33
 [8] 132933.33 147733.33 163333.33

> colVars(X)

[1]  1167.833  8867.833 23901.167

> rowSd(X)

 [1] 227.4496 246.8468 266.3331 285.8904 305.5050 325.1666 344.8671 364.6002
 [9] 384.3609 404.1452

> colSd(X)

[1]  34.17358  94.16917 154.60002

> rowMax(X)

 [1] 441 484 529 576 625 676 729 784 841 900

> colMax(X)

[1] 100 400 900

> rowMin(X)

 [1]   1   4   9  16  25  36  49  64  81 100

> colMin(X)

[1]   1 121 441
```

5 Applying your own function to the rows or columns of a BufferedMatrix

While many useful functions are provided for carrying out row-wise or column-wise operations, you may want to use your own function to do a different summarization. This may be done using the `rowApply` and `colApply` functions. For example suppose you wanted to write a function which computed the sum of the cube roots of the elements of a column. This would be done like this:

```
> sum.cube.root <- function(x){
+         sum(x^(1/3))
+ }
> colApply(X,sum.cube.root)

[1] 30.04092 61.92514 86.51216
```

these functions may also take additional arguments

```
> sum.arbitrary.power <- function(x,power=2){
+         sum(x^power)
+ }
> rowApply(X,sum.arbitrary.power,power=3)

 [1]  87537683 116365952 152863427 198636608 255546875 325739648 411675707
 [8] 516164672 642400643 794000000
```

Note that if your *BufferedMatrix* is large and you are not in RowMode, with a sufficiently sized row buffer, then calls to `rowApply` will be very slow.

It is also possible to use a function which returns more than one item. In this case, rather than returning a vector another *BufferedMatrix* will be returned.

```
> Y <- colApply(X,sort,decreasing=TRUE)
> is(Y,"BufferedMatrix")

[1] TRUE
```

Note that `colApply` and `rowApply` assume that your function returns a fixed length vector.

6 Elementwise transformation of BufferedMatrix objects

In some situations it might be useful to transform each element of your *BufferedMatrix*. For instance log or exponential transformations, square roots or arbitrary powers. Unlike when you ordinarily apply these functions to an R matrix, when you apply these function to *BufferedMatrix* objects, the object is not copied before being transformed. In otherwords these functions do not leave the *BuffferedMatrix* untouched.

```
> exp(X)

<pointer: 0x600001618120>

> log(X)

<pointer: 0x600001618120>

> sqrt(X)

<pointer: 0x600001618120>

> pow(X,2.0)
```

If you have a customized function that you want to apply in an element-wise fashion you can use the `ewApply` function. Note that the function you supply must return only a single value when it is given an argument of length 1 and must return a vector of length n if given a vector of length n. It is best that you design your input function to operate in a vectorized manner.

```
> my.function <- function(x){
+   x^2 +3*abs(x) - 9
+ }
> ewApply(X, my.function)

BufferedMatrix object
Matrix size:  10 3
Buffer size:  1 1
Directory:    /private/var/folders/r0/l4fjk6cj5xj0j3brt4bplpl40000gt/T/RtmpkJrrWp/Rbuil
Prefix:       BM
Mode: Col mode
Read Only: FALSE
Memory usage :  706  bytes.
Disk usage :  240  bytes.
```

7 Converting between BufferedMatrix and Matrix objects

It is possible to convert a *BufferedMatrix* into a *matrix* and vice versa. But in most cases you will not want to do this because your *BufferedMatrix* may be too large to keep completely in the RAM available to R.

To convert a *BufferedMatrix* to a *Matrix* use:

```
> Z <- as(X,"matrix")
> class(Z)

[1] "matrix" "array"
```

To make a *Matrix* become a *BufferedMatrix* use:

```
> A <- as(Z,"BufferedMatrix")
> class(A)

[1] "BufferedMatrix"
attr(,"package")
[1] "BufferedMatrix"
```

8 BufferedMatrix objects are pass-by reference

As previously mentioned when you pass an *BufferedMatrix* to a function you are passing it by reference. This means that if the function operates on the *BufferedMatrix* using a function that can change values stored within the *BufferedMatrix* then it is changed in the calling environment. This differs from the behaviour of a normal R matrix.

```
> X <- createBufferedMatrix(50,10)
> X[1:50,] <- 1:500
> Y <- as(X,"matrix")
> my.function <- function(a.matrix){
+   a.matrix[,1:10] <- a.matrix[,sample(1:10,10)]
+ }
> X[1:5,]

     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1   51  101  151  201  251  301  351  401   451
[2,]    2   52  102  152  202  252  302  352  402   452
[3,]    3   53  103  153  203  253  303  353  403   453
[4,]    4   54  104  154  204  254  304  354  404   454
[5,]    5   55  105  155  205  255  305  355  405   455
```

```
> my.function(X)
> X[1:5,]

     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]  101  451  201    1   51  251  401  301  351   151
[2,]  102  452  202    2   52  252  402  302  352   152
[3,]  103  453  203    3   53  253  403  303  353   153
[4,]  104  454  204    4   54  254  404  304  354   154
[5,]  105  455  205    5   55  255  405  305  355   155

> Y[1:5,]

     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1   51  101  151  201  251  301  351  401   451
[2,]    2   52  102  152  202  252  302  352  402   452
[3,]    3   53  103  153  203  253  303  353  403   453
[4,]    4   54  104  154  204  254  304  354  404   454
[5,]    5   55  105  155  205  255  305  355  405   455

> my.function(Y)
> Y[1:5,]

     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1   51  101  151  201  251  301  351  401   451
[2,]    2   52  102  152  202  252  302  352  402   452
[3,]    3   53  103  153  203  253  303  353  403   453
[4,]    4   54  104  154  204  254  304  354  404   454
[5,]    5   55  105  155  205  255  305  355  405   455
```

There is one exception to the pass-by reference rule, column and row names are passed by value, meaning that modifications to dimension names within a calling function have no effect on the dimension names in the calling environment. However, this may be changed in future releases so you should not rely on this behaviour.

In situations where you want to simulate call by value you can use the `duplicate` function. This will make a copy of the *BufferedMatrix*.

```
> X <- createBufferedMatrix(50,10)
> X[1:50,] <- 1:500
> my.function <- function(my.bufmat){
+   internal.bufmat <- duplicate(my.bufmat)
+   internal.bufmat[,1:10] <- internal.bufmat[,sample(1:10,10)]
+ }
> X[1:5,]
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1   51  101  151  201  251  301  351  401   451
[2,]    2   52  102  152  202  252  302  352  402   452
[3,]    3   53  103  153  203  253  303  353  403   453
[4,]    4   54  104  154  204  254  304  354  404   454
[5,]    5   55  105  155  205  255  305  355  405   455
```

```
> my.function(X)
> X[1:5,]
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    1   51  101  151  201  251  301  351  401   451
[2,]    2   52  102  152  202  252  302  352  402   452
[3,]    3   53  103  153  203  253  303  353  403   453
[4,]    4   54  104  154  204  254  304  354  404   454
[5,]    5   55  105  155  205  255  305  355  405   455
```

Table 1 summarizes which functions modify and those don't modify *BufferedMatrix* objects.

9 Future Enhancements

Currently *BufferedMatrix* objects can not properly serialized. That means you can not save and reload a BufferedMatrix between R sessions. This is an issue to be addressed in a future release of the BufferedMatrix package.

Functions that do not modify data stored in *BufferedMatrix* objects	Functions that do modify data stored in *BufferedMatrix* objects
`[`	`[<-`
`Max`	`exp`
`Min`	`log`
`mean`	`sqrt`
`Sum`	`pow`
`Var`	`ewApply`
`Sd`	`RowMode`
`rowMeans`	`ColMode`
`colMeans`	`ReadOnlyMode`
`rowSums`	`AddColumn`
`colSums`	
`rowVars`	
`colVars`	
`rowSd`	
`colSd`	
`rowMax`	
`colMax`	
`rowMin`	
`colMin`	
`colApply`	
`rowApply`	
`duplicate`	
`as`	
`ncol`	
`nrow`	
`is.ColMode`	
`is.RowMode`	
`set.buffer.dim`	
`prefix`	
`directory`	
`filenames`	
`subBufferedMatrix`	
`is.ReadOnlyMode`	
`memory.usage`	
`disk.usage`	

Table 1: A breakdown of functions that do and do not modify data stored in or how it is stored in *BufferedMatrix* objects.