matter: Rapid prototyping with data on disk

Kylie A. Bemis

December 18, 2018

Contents

1 Introduction

matter is designed for rapid prototyping of new statistical methods when working with larger-than-memory datasets on disk. Unlike related packages *bigmemory* [1] and *ff* [2], which also work with file-backed larger-than-memory datasets, *matter* aims to offer strict control over memory and maximum flexibility with on-disk data structures, so it can be easily adapted to domain-specific file formats, including user-customized file formats.

The vignettes of this package are organized as follows:

- "Rapid prototyping with data on disk": This is the main vignette describing *matter* and its general use, design, and extensibility. It walks through basic data manipulation, data structures, and demonstrates an example S4 class extending *matter*.

- "Supplementary 1 - Simulations and comparative benchmarks": This supplementary vignette re-works the simulated statistical analysis examples from this vignette using *bigmemory* and *ff* and provides some basic benchmarks and comparisons of the three packages.
- "Supplementary 2 - 3D mass spectrometry imaging case study": This supplementary vignette demonstrates using *matter* for principal components analysis on large experimental data, along with in-depth comparisons with *bigmemory* and *ff* on real data.

2 Installation

matter can be installed from Bioconductor using the following commands in *R*.

```
> install.packages("BiocManager")
> BiocManager::install("matter")
```

3 Basic use and data manipulation

matter matrices and vectors can be initialized similarly to ordinary *R* matrices. When no file is given, a new temporary file is created in the default temporary file directory, which will be cleaned up later by either *R* or the operating system.

Here, we initialize a *matter* matrix with 10 rows and 5 columns. The resulting object is a subclass of the `matter` class, and stores file metadata that gives the location of the data on disk. In many cases, it can be treated as an ordinary *R* matrix.

```
> x <- matter_mat(data=1:50, nrow=10, ncol=5, datamode="double")
> x

An object of class 'matter_matc'
  <10 row, 5 column> matrix
    sources: 1
    datamode: numeric
    6.2 KB real memory
    400 bytes virtual memory

> x[]

      [,1] [,2] [,3] [,4] [,5]
 [1,]    1   11   21   31   41
 [2,]    2   12   22   32   42
 [3,]    3   13   23   33   43
 [4,]    4   14   24   34   44
 [5,]    5   15   25   35   45
 [6,]    6   16   26   36   46
 [7,]    7   17   27   37   47
 [8,]    8   18   28   38   48
 [9,]    9   19   29   39   49
[10,]   10   20   30   40   50
```

As seen above, this is a small toy example in which the in-memory metadata actually takes up more space than the size of the data stored on disk. For much larger datasets, the in-memory metadata will be a small fraction of the total size of the dataset on disk.

matter's matrices and vectors can be indexed into like ordinary *R* matrices and vectors.

```
> x[1:4,]

     [,1] [,2] [,3] [,4] [,5]
[1,]    1   11   21   31   41
[2,]    2   12   22   32   42
[3,]    3   13   23   33   43
[4,]    4   14   24   34   44

> x[,3:4]

      [,1] [,2]
 [1,]   21   31
 [2,]   22   32
 [3,]   23   33
 [4,]   24   34
 [5,]   25   35
 [6,]   26   36
 [7,]   27   37
 [8,]   28   38
 [9,]   29   39
[10,]   30   40
```

We can assign names to `matter_vec` vectors and row and column names to `matter_mat` matrices.

```
> rownames(x) <- 1:10
> colnames(x) <- letters[1:5]
> x[]

    a  b  c  d  e
1   1 11 21 31 41
2   2 12 22 32 42
3   3 13 23 33 43
4   4 14 24 34 44
5   5 15 25 35 45
6   6 16 26 36 46
7   7 17 27 37 47
8   8 18 28 38 48
9   9 19 29 39 49
10 10 20 30 40 50
```

matter provides methods for calculating summary statistics for its vectors and matrices, including some methods that do not exist in base *R*, such as `colVars`, which uses a memory-efficient running variance calculation that is accurate for large floating-point datasets [3].

```
> colSums(x)

  a   b   c   d   e
 55 155 255 355 455
```

```
> colSums(x[])

  a   b   c   d   e
 55 155 255 355 455

> colVars(x)

       a        b        c        d        e
9.166667 9.166667 9.166667 9.166667 9.166667

> apply(x, 2, var)

       a        b        c        d        e
9.166667 9.166667 9.166667 9.166667 9.166667
```

One of the major advantages of the flexibility of *matter* is being able to treat data from multiple files as a single dataset. This is particularly useful if analysing data from a domain where each sample in an experiment generates large files, such as high-resolution, high-throughput mass spectrometry imaging.

Below, we create a second matrix, and show its data is stored in a separate file. We then combine the matrices, and the result can be treated as a single matrix, despite originating from multiple files. Combining the matrices does not create new data or change the existing data on disk.

```
> y <- matter_mat(data=51:100, nrow=10, ncol=5, datamode="double")
> paths(x)

[1] "/tmp/RtmpCwSGHP/filefd1f30dbeab0.bin"

> paths(y)

[1] "/tmp/RtmpCwSGHP/filefd1f794e5d76.bin"

> z <- cbind(x, y)
> z

An object of class 'matter_matc'
  <10 row, 10 column> matrix
    sources: 2
    datamode: numeric
    11.6 KB real memory
    800 bytes virtual memory

> z[]

    a  b  c  d  e
1   1 11 21 31 41 51 61 71 81  91
2   2 12 22 32 42 52 62 72 82  92
3   3 13 23 33 43 53 63 73 83  93
4   4 14 24 34 44 54 64 74 84  94
5   5 15 25 35 45 55 65 75 85  95
6   6 16 26 36 46 56 66 76 86  96
7   7 17 27 37 47 57 67 77 87  97
8   8 18 28 38 48 58 68 78 88  98
9   9 19 29 39 49 59 69 79 89  99
10 10 20 30 40 50 60 70 80 90 100
```

Note that matrices in *matter* are either stored in a column-major or a row-major format. The default is to use the column-major format, as *R* does. Column-major matrices are optimized for fast column-access, and assume that each column is stored contiguously or mostly-contiguously on disk. Conversely, row-major matrices are optimized for fast row-access, and make the same assumption for rows.

Since *matter* does support both column-major and row-major formats, transposing a matrix is a trivial operation in *matter* that only needs to change the matrix metadata, and doesn't touch the data on disk.

```
> t(x)

An object of class 'matter_matr'
  <5 row, 10 column> matrix
    sources: 1
    datamode: numeric
    6.5 KB real memory
    400 bytes virtual memory

> rbind(t(x), t(y))

An object of class 'matter_matr'
  <10 row, 10 column> matrix
    sources: 2
    datamode: numeric
    11.6 KB real memory
    800 bytes virtual memory
```

Note that this is equivalent to `t(cbind(x, y))`.

Below, we inspect the metadata associated with the different columns of `x` using the `atomdata` method.

```
> atomdata(x)

  group_id source_id datamode offset extent index_offset index_extent
1        1         1   double      0     10            0           10
2        2         1   double     80     10            0           10
3        3         1   double    160     10            0           10
4        4         1   double    240     10            0           10
5        5         1   double    320     10            0           10
```

This shows the "atoms" that compose the `matter` object. An atom in *matter* is a single contiguous segment of a file on disk. In this case, each atom corresponds to a different column. Note that each atom has a byte offset and an extent (i.e., length) associated with it.

Now we show how to create a `matter` object from a pre-existing file. We will first create vectors corresponding to the second column of `x` and third column of `y`.

```
> x2 <- matter_vec(offset=80, extent=10, paths=paths(x), datamode="double")
> y3 <- matter_vec(offset=160, extent=10, paths=paths(y), datamode="double")
> cbind(x2, y3)[]

     [,1] [,2]
 [1,]   71   71
 [2,]   72   72
```

```
 [3,]   73   73
 [4,]   74   74
 [5,]   75   75
 [6,]   76   76
 [7,]   77   77
 [8,]   78   78
 [9,]   79   79
[10,]   80   80
```

```
> cbind(x[,2], y[,3])
```

```
   [,1] [,2]
1    11   71
2    12   72
3    13   73
4    14   74
5    15   75
6    16   76
7    17   77
8    18   78
9    19   79
10   20   80
```

We can even combine multiple on-disk vectors together before binding them all into a matrix.

```
> z <- cbind(c(x2, y3), c(y3, x2))
> atomdata(z)
```

```
  group_id source_id datamode offset extent index_offset index_extent
1        1         2   double    160     10            0           10
2        1         1   double     80     10           10           20
3        2         2   double    160     10            0           10
4        2         1   double     80     10           10           20
```

```
> z[]
```

```
      [,1] [,2]
 [1,]   71   71
 [2,]   72   72
 [3,]   73   73
 [4,]   74   74
 [5,]   75   75
 [6,]   76   76
 [7,]   77   77
 [8,]   78   78
 [9,]   79   79
[10,]   80   80
[11,]   11   11
[12,]   12   12
[13,]   13   13
[14,]   14   14
[15,]   15   15
[16,]   16   16
```

```
[17,]   17   17
[18,]   18   18
[19,]   19   19
[20,]   20   20
```

This is a quick and easy way to build a dataset from many files, or even many segments of many files. Even if the resulting matrix would fit into memory, using *matter* can be a tidy, efficient way of reading complex binary data from multiple files into *R*.

Lastly, it is straightforward to coerce common base R types to their `matter` object equivalents using `as.matter`. This includes raw, logical, integer, numeric, and character vectors, integer and numeric matrices, and data frames.

```
> v1 <- 1:10
> v2 <- as.matter(v1)
> v2

An object of class 'matter_vec'
  <10 length> vector
    sources: 1
    datamode: integer
    6 KB real memory
    40 bytes virtual memory

> v2[]

 [1]  1  2  3  4  5  6  7  8  9 10

> m1 <- diag(3)
> m2 <- as.matter(m1)
> m2

An object of class 'matter_matc'
  <3 row, 3 column> matrix
    sources: 1
    datamode: numeric
    6.1 KB real memory
    72 bytes virtual memory

> m2[]

     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1

> s1 <- letters[1:10]
> s2 <- as.matter(s1)
> s2

An object of class 'matter_str'
  <10 length> string
    sources: 1
    datamode: raw
    12.9 KB real memory
```

```
    10 bytes virtual memory
    encoding: unknown

> s2[]

 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"

> df1 <- data.frame(a=v1, b=s1, stringsAsFactors=FALSE)
> df2 <- as.matter(df1)
> df2

An object of class 'matter_df'
  <10 row, 2 column> data frame
    22.4 KB real memory: 0 variables
    50 bytes virtual memory: 2 variables

             a           b
  <matter_vec> <matter_str>
1            1           a
2            2           b
3            3           c
4            4           d
5            5           e
6            6           f
[and 4 more rows]

> df2[]

    a b
1   1 a
2   2 b
3   3 c
4   4 d
5   5 e
6   6 f
7   7 g
8   8 h
9   9 i
10 10 j
```

4 Linear regression for on-disk datasets

matter is designed to provide a statistical computing environment for larger-than-memory datasets on disk. To facilitate this, *matter* provides a method for fitting of linear models for `matter` matrices through the *biglm* package [4]. *matter* provides a wrapper for *biglm*'s `bigglm` function that works with `matter_mat` matrices, which we demonstrate below.

First, we simulate some data appropriate for linear regression.

```
> set.seed(81216)
> n <- 1.5e7
```

```
> p <- 9
> b <- runif(p)
> names(b) <- paste0("x", 1:p)
> data <- matter_mat(nrow=n, ncol=p + 1, datamode="double")
> colnames(data) <- c(names(b), "y")
> data[,p + 1] <- rnorm(n)
> for ( i in 1:p ) {
+   xi <- rnorm(n)
+   data[,i] <- xi
+   data[,p + 1] <- data[,p + 1] + xi * b[i]
+ }
> data

An object of class 'matter_matc'
  <15000000 row, 10 column> matrix
    sources: 1
    datamode: numeric
    13.4 KB real memory
    1.2 GB virtual memory

> head(data)

              x1        x2         x3          x4         x5         x6
[1,] -0.45330471 0.5995144 -0.1392395  0.36748584  1.4000923  0.5555708
[2,] -1.60355974 0.5862366 -0.5421275 -0.36101120 -0.4930582  0.7549443
[3,]  0.22920974 0.5138377 -1.7860077  1.53126322  0.3557548 -0.6093811
[4,] -1.38862865 0.1411892  0.3166607 -0.08396404  0.9629351  0.3443397
[5,] -0.36473656 0.4315282  1.1860328 -1.13518455  0.5386445  1.1426125
[6,] -0.07204838 0.2744724 -0.6730541  0.03472469  0.2138691  0.5923886
             x7          x8         x9          y
[1,] -2.4031764 -0.57037899 -0.4356390  0.2280728
[2,] -0.1348020  0.05384544 -0.5209713  0.3358334
[3,]  1.0381120  0.72976777  0.9689488  3.8910764
[4,] -1.5310565 -0.44875206 -1.1320185 -0.8646491
[5,]  0.2239818  1.40000992 -0.9843404  1.8709778
[6,]  0.4852140 -0.29082018  1.0831832  1.5140973
```

This creates a 1.2 GB dataset on disk, but only about 12 KB of metadata is stored in memory.

Now we calculate some statistical summaries using *matter*'s `apply` method for `matter_mat` matrices.

```
> apply(data, 2, mean)

           x1            x2            x3            x4            x5
 2.962621e-04 -2.596339e-04 -2.729651e-04  3.014581e-05 -5.893552e-05
           x6            x7            x8            x9             y
-2.835383e-04 -1.309537e-04 -9.810476e-05 -1.404680e-04 -3.225581e-04

> apply(data, 2, var)

       x1        x2        x3        x4        x5        x6        x7
1.0003094 0.9996336 0.9990518 1.0003654 0.9999593 0.9995961 0.9999286
       x8        x9         y
```

```
1.0001395 0.9996875 4.4527319
```

We could also have used `colMeans` and `colVars`, which are specialized to be faster and more memory efficient.

Now we fit the linear model to the data using the `bigglm` method for `matter_mat` matrices. Note that it requires a formula, and (unfortunately) it does not allow `y ~ .`, so all variables must be stated explicitly.

```
> fm <- as.formula(paste0("y ~ ", paste0(names(b), collapse=" + ")))
> bigglm.out <- bigglm(fm, data=data, chunksize=10000)
```

```
> summary(bigglm.out)

Large data regression model: bigglm(formula, getNextDataChunk, ...)
Sample size =  1.5e+07
              Coef    (95%    CI)    SE      p
(Intercept) 0.0004 -0.0001 0.0009 3e-04 0.1001
x1          0.1689  0.1684 0.1695 3e-04 0.0000
x2          0.9572  0.9566 0.9577 3e-04 0.0000
x3          0.3801  0.3796 0.3806 3e-04 0.0000
x4          0.6042  0.6037 0.6048 3e-04 0.0000
x5          0.5198  0.5193 0.5203 3e-04 0.0000
x6          0.6926  0.6921 0.6931 3e-04 0.0000
x7          0.8374  0.8369 0.8380 3e-04 0.0000
x8          0.4616  0.4610 0.4621 3e-04 0.0000
x9          0.5782  0.5777 0.5788 3e-04 0.0000

> cbind(coef(bigglm.out)[-1], b)

                     b
x1 0.1689408 0.1689486
x2 0.9571547 0.9574388
x3 0.3800765 0.3802078
x4 0.6042379 0.6043915
x5 0.5198087 0.5194832
x6 0.6926179 0.6927430
x7 0.8374374 0.8373628
x8 0.4615518 0.4617963
x9 0.5782414 0.5775168
```

On a 2012 retina MacBook Pro with 2.6 GHz Intel CPU, 16 GB RAM, and 500 GB SSD, fitting the linear model takes 40 seconds and uses an additional 400 MB of memory overhead. The max amount of memory used while fitting the model was only 650 MB, for the 1.2 GB dataset. This memory usage can be controlled further by using the `chunksize` argument in `bigglm` or by specifying a different `chunksize` for the `matter` object.

5 Principal components analysis for on-disk datasets

Because *matter* provides basic linear algebra for on-disk `matter_mat` matrices with in-memory R matrices, it opens up the possibility for the use of many iterative statistical methods which can operate on only small portions of the data at a time.

For example, `matter_mat` matrices are compatible with the *irlba* package, which performs efficient, bounded-memory singular value decomposition (SVD) of matrices, and which can therefore be used for efficient principal components analysis (PCA) of large datasets [5].

For convenience, *matter* provides a `prcomp` method for performing PCA on `matter_mat` matrices using the `irlba` method from the *irlba* package, as demonstrated below.

First, we simulate some data appropriate for principal components analysis.

```
> set.seed(81216)
> n <- 1.5e6
> p <- 100
> data <- matter_mat(nrow=n, ncol=p, datamode="double")
> for ( i in 1:10 )
+   data[,i] <- (1:n)/n + rnorm(n)
> for ( i in 11:20 )
+   data[,i] <- (n:1)/n + rnorm(n)
> for ( i in 21:p )
+   data[,i] <- rnorm(n)
> data

An object of class 'matter_matc'
  <1500000 row, 100 column> matrix
    sources: 1
    datamode: numeric
    12.6 KB real memory
    1.2 GB virtual memory
```

This again creates a 1.2 GB dataset on disk, but only about 12 KB of metadata is stored in memory. More metadata is stored compared to the previous example, because the matrix has more columns, and it is stored as a column-major `matter_matc` matrix with independent metadata for each column.

Note that, in the simulated data, only the first twenty variables show systematic variation, with the first ten variables varying distinctly from the next ten variables.

First we calculate the variance for each column.

```
> var.out <- colVars(data)
> plot(var.out, type='h', ylab="Variance")
```

This takes only 7 seconds and uses less than 30 KB of additional memory. The maximum amount of memory used while calculating the variance for all columns of the 1.2 GB dataset is only 27 MB.

Note that the `irlba` function has an optional argument `mult` which allows specification of a custom matrix multiplication method, for use with packages such as *bigmemory* and *ff*. This is especially useful since it allows a `transpose=TRUE` argument, so that the

identity `t(t(B) %*% A)` can be used in place of `t(A) %*% B)` when transposition is an expensive operation. However, this is not necessary for *matter*, since transposition is a trivial operation for `matter_mat` matrices.

Implicit scaling and centering is supported in *matter*, so the data can be scaled and centered without changing the data on disk or the additional overhead of storing a scaled version of the data.

Unlike the default `prcomp` method for ordinary *R* matrices, this version supports an argument `n` for specifying the number of principal components to calculate.

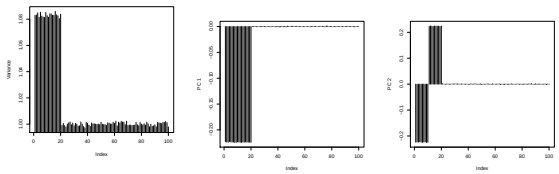
```
> prcomp.out <- prcomp(data, n=2, center=FALSE, scale.=FALSE)
```

On a 2012 retina MacBook Pro with 2.6 GHz Intel CPU, 16 GB RAM, and 500 GB SSD, calculating the first two principal components takes 100 seconds and uses an additional 450 MB of memory overhead. The max amount of memory used during the computation was only 700 MB, for the 1.2 GB dataset.

Now we plot the first two principal components.

```
> plot(prcomp.out$rotation[,1], type='h', ylab="PC 1")
```

```
> plot(prcomp.out$rotation[,2], type='h', ylab="PC 2")
```



(a) Sample variance **(b)** PC1 loadings **(c)** PC2 loadings

Figure 1: **Principal components analysis for on-disk dataset**

As shown in the plots of the first and second principal components, the first PC shows that most of the variation in the data occurs in the first twenty variables, while the second PC distinguishes the first ten variables from the next ten variables.

6 Design and implementation

The *matter* package is designed with several goals in mind. Like the *bigmemory* and *ff* packages, it seeks to make statistical methods scalable to larger-than-memory datasets by utilizing data-on-disk. Unlike those packages, it seeks to make domain-specific file formats (such as Analyze 7.5 and imzML for MS imaging experiments) accessible from disk directly without additional file conversion. It seeks to have a minimal memory footprint, and require minimal developer effort to use, while maintaining computational efficiency wherever possible.

6.1 S4 classes

matter utilizes S4 classes to implement on-disk matrices in a way so that they can be seamlessly accessed as if they were ordinary *R* matrices. These are the `atoms` class and the `matter` class. The `atoms` class is not exported to the user, who only interacts with the `matter` class and `matter` objects to create and manipulate on-disk matrices.

6.1.1 `atoms`: contiguous sectors of data on disk

By analogy to *R*'s notion of "atomic" vectors, the `atoms` class uses the notion of contiguous "atomic" sectors of disk. Each "atom" in an `atoms` object gives the location of one block of contiguous data on disk, as denoted by a file path, an byte offset from the beginning of the file, a data type, and the number of data elements (i.e., the length) of the atom. An `atoms` object may consist of many atoms from multiple files and multiple locations on disk, while ultimately representing a single vector or row or column of a matrix. The "atoms" may be arranged into group, corresponding to rows or columns of a matrix, margins of an array, elements of a list, etc.

Structure:

- `natoms`: the number of atoms
- `ngroups`: the number of groups
- `group_id`: which group each atom belongs to
- `source_id`: the ID's of the file paths where each atom is located
- `datamode`: the type of data (short, int, long, float, double) for each atom
- `offset`: each atom's byte offset from the beginning of the file
- `extent`: the length of each atom
- `index_offset`: the cumulative index of the first element of each atom
- `index_extent`: the cumulative one-past-the-end index of each atom

The `atoms` class has a C++ backend in the `Atoms` C++ class.

6.1.2 `matter`: vectors and matrices stored on disk

A `matter` object is made of one or more `atoms` objects, and represents a vector or matrix. It includes additional metadata such as dimensions and row names or column names.

Structure:

- `data`: one or more `atoms` objects
- `datamode`: the type of data (integer, numeric) for the represented vector or matrix
- `paths`: the paths to the files used by the `atoms` objects
- `filemode`: should the files be open for read/write, or read-only?
- `chunksize`: how large the chunk sizes should be for calculations that operate on chunks of the dataset

- `length`: the total length of the dataset
- `dim`: the extent of each dimension (for a matrix)
- `names`: the names of the data elements (e.g., for a vector)
- `dimnames`: the names of the dimensions (e.g., for a matrix)
- `ops`: delayed operations registered to the object

A `matter_vec` vector contains a single `atoms` object that represents all of the atoms of the vector. The `matter_mat` matrix class has two subtypes for column-major (`matter_matc`) and row-major (`matter_matr`) matrices. A column-major `matter_matc` matrix has one `atoms` object for each column, while a row-major `matter_matr` matrix has one `atoms` object for each row.

The `matter` class has a C++ backend in the `Matter` C++ class.

6.2 C++ classes

matter utilizes a C++ backend to access the data on disk and transform it into the appropriate representation in R. Although these classes correspond to S4 classes in *R*, and are responsible for most of the computational work, all of the required metadata is stored in the S4 classes in *R*, and are simply read by the C++ classes. This means that *matter* never depends on external pointers, which makes it trivial to share `matter` vectors and `matter` matrices between *R* sessions that have access to the same filesystem.

6.2.1 `Atoms`: contiguous sectors of data on disk

The `Atoms` C++ class is responsible for reading and writing the data on disk, based on its metadata. For computational efficiency, it tries to perform sequential reads over random read/writes whenever possible, while minimizing the total number of atomic read/writes on disk.

6.2.2 `Matter`: vectors and matrices stored on disk

The `Matter` C++ class is responsible for transforming the data read by the `Atoms` class into a format appropriate for R. This may include re-arranging contiguous data that has been read sequentially into a different order, either due to the inherent organization of the dataset, or as requested by the user in R.

6.2.3 `MatterIterator`: iterate over virtual disk objects

The `MatterIterator` C++ class acts similarly to an iterator, and allows buffered iteration over a `Matter` object. It can either iterate over the whole dataset (for both vectors and matrices), or over a single column for column-major matrices, or over a single row for row-major matrices.

A `MatterIterator` object will load portions of the dataset (as many elements as the `chunksize` at once) into memory, and then free that portion of the data and load a new chunk, as necessary. This buffering is handled automatically by the class, and code can treat it as a regular iterator. This allows seamless and simple iteration over `Matter` objects while maintaining strict control over the memory footprint of the calculation.

7 Extending with new S4 classes

The *matter* package is intended to be extensible to any uncompressed, open-source format. For example, the *Cardinal* package uses *matter* to attach Analyze 7.5 and imzML datasets, which are popular open-source data formats in mass spectrometry imaging. The flexibility of *matter* in terms of specifying custom file formats with a high degree of control distinguishes it from other packages for working with large, on-disk datasets in R.

In this section, we demonstrate how one could create a custom S4 class for genomics sequencing data.

We begin by creating a small toy example of a FASTQ file with only two sample reads from a larger library [6] and writing them to disk.

```
> seqs <- c("@SRR001666.1 071112_SLXA-EAS1_s_7:5:1:817:345 length=72",
+         "GGGTGATGGCCGCTGCCGATGGCGTCAAATCCCACCAAGTTACCCTTAACAACTTAAGGGTTTTCAAATAGA",
+         "+SRR001666.1 071112_SLXA-EAS1_s_7:5:1:817:345 length=72",
+         "IIIIIIIIIIIIIIIIIIIIIIIIIIIII9IG9ICIIIIIIIIIIIIIIIIIIIDIIIIIII>IIIIII/",
+         "@SRR001666.2 071112_SLXA-EAS1_s_7:5:1:801:338 length=72",
+         "GTTCAGGGATACGACGTTTGTATTTTAAGAATCTGAAGCAGAAGTCGATGATAATACGCGTCGTTTTATCAT",
+         "+SRR001666.2 071112_SLXA-EAS1_s_7:5:1:801:338 length=72",
+         "IIIIIIIIIIIIIIIIIIIIIIIIIIIIII6IBIIIIIIIIIIIIIIIIIIIIIGII>IIIII-I)8I")
> file <- tempfile()
> writeLines(seqs, file)
```

We create a new S4 class called `Fastq`, with slots for a sequence identifier, the raw sequence of letters, and the quality values for the sequence, all of which will be objects of the `matter_str` class for on-disk strings.

We also create generic functions and S4 methods to access these slots. Note that if we were creating this class for a package, it would be wiser to import the existing generic functions from packages *Biostrings* and *ShortRead*. For the purpose of demonstration, we define them here.

```
> setClass("Fastq", slots=c(
+         id = "matter_str",
+         sread = "matter_str",
+         quality = "matter_str"))
> setGeneric("id", function(x) standardGeneric("id"))

[1] "id"

> setGeneric("sread", function(object, ...) standardGeneric("sread"))

[1] "sread"

> setGeneric("quality", function(object, ...) standardGeneric("quality"))
```

```
[1] "quality"
```

```
> setMethod("id", "Fastq", function(x) x@id)
> setMethod("sread", "Fastq", function(object) object@sread)
> setMethod("quality", "Fastq", function(object) object@quality)
```

Now we write a function for constructing the new class. First we attach the file as a flat `matter_vec` raw byte vector, and calculate the byte offsets of the new lines in the file. This is done by calling `which(raw == charToRaw('\n'))`, which requires parsing the whole vector to do the elementwise comparisons. Note that *matter* will automatically perform this operation in chunks in efficient C++ code. For a large dataset, this means that only `chunksize(raw)` data elements are ever loaded into memory at once.

```
> attachFastq <- function(file) {
+         length <- file.info(file)$size
+         raw <- matter_vec(paths=file, length=length, datamode="raw")
+         newlines <- which(raw == charToRaw('\n')) # parses the file in chunks
+         if ( newlines[length(newlines)] == length )
+                 newlines <- newlines[-length(newlines)]
+         byte_start <- c(0L, newlines)
+         byte_end <- c(newlines, length) - 1L # don't include the '\n'
+         line_offset <- byte_start
+         line_extent <- byte_end - byte_start
+         id <- matter_str(paths=file,
+                 offset=1L + line_offset[c(TRUE,FALSE,FALSE,FALSE)], # skip the '@'
+                 extent=line_extent[c(TRUE,FALSE,FALSE,FALSE)] - 1L) # adjust for '@'
+         sread <- matter_str(paths=file,
+                 offset=line_offset[c(FALSE,TRUE,FALSE,FALSE)],
+                 extent=line_extent[c(FALSE,TRUE,FALSE,FALSE)])
+         quality <- matter_str(paths=file,
+                 offset=line_offset[c(FALSE,FALSE,FALSE,TRUE)],
+                 extent=line_extent[c(FALSE,FALSE,FALSE,TRUE)])
+         new("Fastq", id=id, sread=sread, quality=quality)
+ }
```

Now we can call our `attachFastq` function to parse the file and create an object of our new class `Fastq`.

```
> fq <- attachFastq(file)
> fq
```

```
An object of class "Fastq"
Slot "id":
An object of class 'matter_str'
  <2 length> string
    sources: 1
    datamode: raw
    6.2 KB real memory
    108 bytes virtual memory
    encoding: unknown

Slot "sread":
```

```
An object of class 'matter_str'
  <2 length> string
    sources: 1
    datamode: raw
    6.2 KB real memory
    144 bytes virtual memory
    encoding: unknown

Slot "quality":
An object of class 'matter_str'
  <2 length> string
    sources: 1
    datamode: raw
    6.2 KB real memory
    144 bytes virtual memory
    encoding: unknown
```

```
> id(fq)[1]
```

```
[1] "SRR001666.1 071112_SLXA-EAS1_s_7:5:1:817:345 length=72"
```

```
> id(fq)[2]
```

```
[1] "SRR001666.2 071112_SLXA-EAS1_s_7:5:1:801:338 length=72"
```

```
> sread(fq)[1]
```

```
[1] "GGGTGATGGCCGCTGCCGATGGCGTCAAATCCCACCAAGTTACCCTTAACAACTTAAGGGTTTTCAAATAGA"
```

```
> sread(fq)[2]
```

```
[1] "GTTCAGGGATACGACGTTTGTATTTTAAGAATCTGAAGCAGAAGTCGATGATAATACGCGTCGTTTTATCAT"
```

```
> quality(fq)[1]
```

```
[1] "IIIIIIIIIIIIIIIIIIIIIIIIIIIII9IG9ICIIIIIIIIIIIIIIIIIIIDIIIIIII>IIIIII/"
```

```
> quality(fq)[2]
```

```
[1] "IIIIIIIIIIIIIIIIIIIIIIIIIIIII6IBIIIIIIIIIIIIIIIIIIIIIGII>IIIII-I)8I"
```

Although this example used only a small toy data file, the same class and parsing function could be applied to much larger files. Very large files would take some time to parse and index the newlines, but the our `Fastq` class would remain memory-efficient.

8 Session info

- R version 3.5.1 Patched (2018-07-12 r74967), `x86_64-apple-darwin15.6.0`
- Locale: `C/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8`
- Running under: `OS X El Capitan 10.11.6`
- Matrix products: default
- BLAS: `/Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRblas.0.dylib`

- LAPACK: `/Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRlapack.dylib`
- Base packages: base, datasets, grDevices, graphics, methods, stats, utils
- Other packages: DBI 1.0.0, biglm 0.9-1, matter 1.8.3
- Loaded via a namespace (and not attached): BiocGenerics 0.28.0, BiocManager 1.30.4, BiocStyle 2.10.0, Matrix 1.2-15, Rcpp 1.0.0, compiler 3.5.1, digest 0.6.18, evaluate 0.12, grid 3.5.1, htmltools 0.3.6, irlba 2.3.2, knitr 1.21, lattice 0.20-38, parallel 3.5.1, rmarkdown 1.11, tools 3.5.1, xfun 0.4, yaml 2.2.0

References

[1] Michael J. Kane, John Emerson, and Stephen Weston. Scalable strategies for computing with massive data. *Journal of Statistical Software*, 55(14):1–19, 2013. URL: http://www.jstatsoft.org/v55/i14/.

[2] Daniel Adler, Christian GlÃďser, Oleg Nenadic, Jens OehlschlÃďgel, and Walter Zucchini. *ff: memory-efficient storage of large data on disk and fast access functions*, 2014. R package version 2.2-13. URL: https://CRAN.R-project.org/package=ff.

[3] B P Welford. Note on a Method for Calculating Corrected Sums of Squares and Products. *Technometrics*, 4(3):1–3, August 1962.

[4] Thomas Lumley. *biglm: bounded memory linear and generalized linear models*, 2013. R package version 0.9-1. URL: https://CRAN.R-project.org/package=biglm.

[5] Jim Baglama and Lothar Reichel. *irlba: Fast Truncated SVD, PCA and Symmetric Eigendecomposition for Large Dense and Sparse Matrices*, 2015. R package version 2.0.0. URL: https://CRAN.R-project.org/package=irlba.

[6] Srx000430: Illumina sequencing of escherichia coli str. k-12 substr. mg1655 genomic paired-end library. URL: https://www.ncbi.nlm.nih.gov/sra/SRR001666.