Working with large arrays in R

A look at HDF5Array/RleArray/DelayedArray objects

Hervé Pagès
hpages@fredhutch.org

Bioconductor conference
Boston

July 2017

1. Motivation and challenges
2. Memory footprint
3. RleArray and HDF5Array objects
4. Hands-on
5. DelayedArray/HDF5Array: Future developments

1. Motivation and challenges

2. Memory footprint

3. RleArray and HDF5Array objects

4. Hands-on

5. DelayedArray/HDF5Array: Future developments

Motivation and challenges

R ordinary **matrix** or **array** is not suitable for big datasets:

- 10x Genomics dataset (single cell experiment): 30,000 genes x 1.3 million cells = 36.5 billion values
- in an ordinary integer matrix ==> 136G in memory!

Need for alternative containers:

- but at the same time, the object should be (almost) as easy to manipulate as an ordinary matrix or array
- *standard R matrix/array API*: `dim`, `dimnames`, `t`, `is.na`, `==`, `+`, `log`, `cbind`, `max`, `sum`, `colSums`, etc...
- not limited to 2 dimensions ==> also support arrays of arbitrary number of dimensions

2 approaches: **in-memory data** vs **on-disk data**

In-memory data

- a 30k x 1.3M matrix might still fit in memory if the data can be efficiently compressed
- example: sparse data (small percentage of non-zero values) ==> *sparse representation* (storage of non-zero values only)
- example: data with long runs of identical values ==> *RLE compression (Run Length Encoding)*
- choose the *smallest type* to store the values: `raw` (1 byte) < `integer` (4 bytes) < `double` (8 bytes)
- if using *RLE compression*:
  - choose the *best orientation* to store the values: *by row* or *by column* (one might give better compression than the other)
  - store the data by chunk ==> opportunity to pick up *best type* and *best orientation* on a chunk basis (instead of for the whole data)
- size of 30k x 1.3M matrix in memory can be reduced from 136G to 16G!

Examples of in-memory containers

dgCMatrix container from the *Matrix* package:

- sparse matrix representation
- non-zero values stored as `double`

RleArray and **RleMatrix** containers from the *DelayedArray* package:

- use RLE compression
- arbitrary number of dimensions
- type of values: any R atomic type (`integer`, `double`, `logical`, `complex`, `character`, and `raw`)

On-disk data

However...

- if data is too big to fit in memory (even after compression) ==> must use *on-disk representation*
- challenge: should still be (almost) as easy to manipulate as an ordinary matrix! (*standard R matrix/array API*)

Examples of on-disk containers

Direct manipulation of an **HDF5 dataset** via the *rhdf5* API. Low level API!

HDF5Array and **HDF5Matrix** containers from the *HDF5Array* package:

Provide access to the HDF5 dataset via an API that mimics the standard R matrix/array API

1 Motivation and challenges

2 Memory footprint

3 RleArray and HDF5Array objects

4 Hands-on

5 DelayedArray/HDF5Array: Future developments

Memory footprint

The "airway" dataset

```
library(airway)
data(airway)
m <- unname(assay(airway))
dim(m)

## [1] 64102     8

typeof(m)

## [1] "integer"
```

```
head(m, n=4)

##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
## [1,]  679  448  873  408 1138 1047  770  572
## [2,]    0    0    0    0    0    0    0    0
## [3,]  467  515  621  365  587  799  417  508
## [4,]  260  211  263  164  245  331  233  229

tail(m, n=4)

##          [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
## [64099,]    0    0    0    0    0    0    0    0
## [64100,]    0    0    0    0    0    0    0    0
## [64101,]    0    0    0    0    0    0    0    0
## [64102,]    0    0    0    0    0    0    0    0

sum(m != 0) / length(m)

## [1] 0.3863803
```

Memory footprint

dgCMatrix vs **RleMatrix** vs **HDF5Matrix**

```
library(pryr)  # for object_size()
object_size(m)

## 2.05 MB

library(Matrix)
object_size(as(m, "dgCMatrix"))

## 2.38 MB

library(DelayedArray)
object_size(as(m, "RleMatrix"))

## 2.22 MB

object_size(as(t(m), "RleMatrix"))

## 1.74 MB

library(HDF5Array)
object_size(as(m, "HDF5Matrix"))

## 1.99 kB
```

Memory footprint

Some limitations of the sparse matrix implementation in the *Matrix* package:

- non-zero values always stored as `double`, the most memory consuming type
- number of non-zero values must be $< 2^{31}$
- limited to 2 dimensions: no support for arrays of arbitrary number of dimensions

1 Motivation and challenges

2 Memory footprint

3 RleArray and HDF5Array objects

4 Hands-on

5 DelayedArray/HDF5Array: Future developments

RleArray and HDF5Array objects

RleMatrix/RleArray and HDF5Matrix/HDF5Array provide:

- support all R atomic types
- no limits in size (but each dimension must be $< 2^{31}$)
- arbitrary number of dimensions

And also:

- **delayed operations**
- **block-processing** (behind the scene)
- TODO: multicore block-processing (sequential only at the moment)

Delayed operations

We start with HDF5Matrix object `M`:

```
M <- as(m, "HDF5Matrix")
M

## <64102 x 8> HDF5Matrix object of type "integer":
##          [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
##     [1,]  679  448  873  408 1138 1047  770  572
##     [2,]    0    0    0    0    0    0    0    0
##     [3,]  467  515  621  365  587  799  417  508
##     [4,]  260  211  263  164  245  331  233  229
##     [5,]   60   55   40   35   78   63   76   60
##      ...    .    .    .    .    .    .    .    .
## [64098,]    0    0    0    0    0    0    0    0
## [64099,]    0    0    0    0    0    0    0    0
## [64100,]    0    0    0    0    0    0    0    0
## [64101,]    0    0    0    0    0    0    0    0
## [64102,]    0    0    0    0    0    0    0    0
```

RleArray and HDF5Array objects

Subsetting is delayed:

```
M2 <- M[10:12, 1:5]
M2

## <3 x 5> DelayedMatrix object of type "integer":
##      [,1] [,2] [,3] [,4] [,5]
## [1,]  394  236  464  175  658
## [2,]  172  168  264  118  241
## [3,] 2112 1867 5137 2657 2735
```

```
seed(M2)

## An object of class "HDF5ArraySeed"
## Slot "filepath":
## [1] "/tmp/RtmpV0heQX/HDF5Array_dump/auto00002.h5"
##
## Slot "name":
## [1] "/HDF5ArrayAUT000002"
##
## Slot "dim":
## [1] 64102     8
##
## Slot "first_val":
## [1] 679
##
## Slot "chunkdim":
## [1] 15000     1
```

RleArray and HDF5Array objects

Transposition is delayed:

```
M3 <- t(M2)
M3

## <5 x 3> DelayedMatrix object of type "integer":
##      [,1] [,2] [,3]
## [1,]  394  172 2112
## [2,]  236  168 1867
## [3,]  464  264 5137
## [4,]  175  118 2657
## [5,]  658  241 2735
```

```
seed(M3)

## An object of class "HDF5ArraySeed"
## Slot "filepath":
## [1] "/tmp/RtmpV0heQX/HDF5Array_dump/auto00002.h5"
##
## Slot "name":
## [1] "/HDF5ArrayAUTO00002"
##
## Slot "dim":
## [1] 64102     8
##
## Slot "first_val":
## [1] 679
##
## Slot "chunkdim":
## [1] 15000     1
```

`cbind()` / `rbind()` are delayed:

```
M4 <- cbind(M3, M[1:5, 6:8])
M4

## <5 x 6> DelayedMatrix object of type "integer":
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]  394  172 2112 1047  770  572
## [2,]  236  168 1867    0    0    0
## [3,]  464  264 5137  799  417  508
## [4,]  175  118 2657  331  233  229
## [5,]  658  241 2735   63   76   60
```

```
seed(M4)  # Error! (more than one seed)
```

RleArray and HDF5Array objects

All the operations in the following groups are delayed:

- `Arith` (+, -, ...)
- `Compare` (==, <, ...)
- `Logic` (&, |)
- `Math` (log, sqrt)
- and more ...

```
M5 <- M == 0
M5

## <64102 x 8> DelayedMatrix object of type "logical":
##            [,1]  [,2]  [,3] ...  [,7]  [,8]
##     [1,] FALSE FALSE FALSE    . FALSE FALSE
##     [2,]  TRUE  TRUE  TRUE    .  TRUE  TRUE
##     [3,] FALSE FALSE FALSE    . FALSE FALSE
##     [4,] FALSE FALSE FALSE    . FALSE FALSE
##     [5,] FALSE FALSE FALSE    . FALSE FALSE
##      ...     .     .     .    .     .     .
## [64098,]  TRUE  TRUE  TRUE    .  TRUE  TRUE
## [64099,]  TRUE  TRUE  TRUE    .  TRUE  TRUE
## [64100,]  TRUE  TRUE  TRUE    .  TRUE  TRUE
## [64101,]  TRUE  TRUE  TRUE    .  TRUE  TRUE
## [64102,]  TRUE  TRUE  TRUE    .  TRUE  TRUE
```

```
seed(M5)

## An object of class "HDF5ArraySeed"
## Slot "filepath":
## [1] "/tmp/RtmpV0heQX/HDF5Array_dump/auto00002.h5"
##
## Slot "name":
## [1] "/HDF5ArrayAUTO00002"
##
## Slot "dim":
## [1] 64102     8
##
## Slot "first_val":
## [1] 679
##
## Slot "chunkdim":
## [1] 15000     1
```

RleArray and HDF5Array objects

```
M6 <- round(M[11:14, ] / M[1:4, ], digits=3)
M6

## <4 x 8> DelayedMatrix object of type "double":
##       [,1]  [,2]  [,3] ...  [,7]  [,8]
## [1,] 0.253 0.375 0.302   . 0.201 0.309
## [2,]   Inf   Inf   Inf   .   Inf   Inf
## [3,] 1.122 0.948 1.027   . 1.182 0.935
## [4,] 0.273 0.242 0.802   . 0.575 0.751
```

```
seed(M6)  # Error! (more than one seed)
```

Realization

Delayed operations can be **realized** by coercing the DelayedMatrix object to HDF5Array:

```
M6a <- as(M6, "HDF5Array")
M6a

## <4 x 8> HDF5Matrix object of type "double":
##        [,1]  [,2]  [,3] ...  [,7]  [,8]
## [1,] 0.253 0.375 0.302   . 0.201 0.309
## [2,]   Inf   Inf   Inf   .   Inf   Inf
## [3,] 1.122 0.948 1.027   . 1.182 0.935
## [4,] 0.273 0.242 0.802   . 0.575 0.751
```

```
seed(M6a)

## An object of class "HDF5ArraySeed"
## Slot "filepath":
## [1] "/tmp/RtmpV0heQX/HDF5Array_dump/auto00003.h5"
##
## Slot "name":
## [1] "/HDF5ArrayAUTO00003"
##
## Slot "dim":
## [1] 4 8
##
## Slot "first_val":
## [1] 0.253
##
## Slot "chunkdim":
## [1] 4 8
```

RleArray and HDF5Array objects

... or by coercing it to RleArray:

```
M6b <- as(M6, "RleArray")
M6b

## <4 x 8> RleMatrix object of type "double":
##       [,1]  [,2]  [,3] ...  [,7]  [,8]
## [1,] 0.253 0.375 0.302   . 0.201 0.309
## [2,]   Inf   Inf   Inf   .   Inf   Inf
## [3,] 1.122 0.948 1.027   . 1.182 0.935
## [4,] 0.273 0.242 0.802   . 0.575 0.751
```

```
seed(M6b)

## An object of class "ChunkedRleArraySeed"
## Slot "breakpoints":
## [1] 32
##
## Slot "type":
## [1] "double"
##
## Slot "chunks":
## <environment: 0x227deb70>
##
## Slot "DIM":
## [1] 4 8
##
## Slot "DIMNAMES":
## [[1]]
## NULL
##
## [[2]]
## NULL
```

Controlling where HDF5 datasets are realized

HDF5 dump management utilities: a set of utilities to control where HDF5 datasets are written to disk.

```
setHDF5DumpFile("~/mydata/M6c.h5")
setHDF5DumpName("M6c")
M6c <- as(M6, "HDF5Array")
```

```
seed(M6c)

## An object of class "HDF5ArraySeed"
## Slot "filepath":
## [1] "/home/biocbuild/mydata/M6c.h5"
##
## Slot "name":
## [1] "M6c"
##
## Slot "dim":
## [1] 4 8
##
## Slot "first_val":
## [1] 0.253
##
## Slot "chunkdim":
## [1] 4 8

h5ls("~/mydata/M6c.h5")

##   group name       otype dclass   dim
## 0     /  M6c H5I_DATASET  FLOAT 4 x 8
```

showHDF5DumpLog()

```
showHDF5DumpLog()

## [2018-08-01 21:23:46] #1 In file '/tmp/RtmpV0heQX/HDF5Array_dump/auto00001.h5':  creation of dataset '/HDF5ArrayAUTO00001'
(64102x8:integer, chunkdims=15000x1, level=6)
## [2018-08-01 21:23:46] #2 In file '/tmp/RtmpV0heQX/HDF5Array_dump/auto00002.h5':  creation of dataset '/HDF5ArrayAUTO00002'
(64102x8:integer, chunkdims=15000x1, level=6)
## [2018-08-01 21:23:47] #3 In file '/tmp/RtmpV0heQX/HDF5Array_dump/auto00003.h5':  creation of dataset '/HDF5ArrayAUTO00003'
(4x8:double, chunkdims=4x8, level=6)
## [2018-08-01 21:23:47] #4 In file '/home/biocbuild/mydata/M6c.h5':  creation of dataset 'M6c' (4x8:double, chunkdims=4x8, level=6)
```

Block processing

The following operations are NOT delayed. They are implemented via a *block processing* mechanism that loads and processes one block at a time:

- operations in the `Summary` group (`max`, `min`, `sum`, `any`, `all`)
- `mean`
- Matrix row/col summarization operations (`col/rowSums`, `col/rowMeans`, ...)
- `anyNA`, `which`
- `apply`
- and more ...

```
DelayedArray:::set_verbose_block_processing(TRUE)

## [1] FALSE

colSums(M)

## Processing block 1/1 ...  OK

## [1] 20637971 18809481 25348649 15163415 24448408 30818215 19126151 21164133
```

Control the block size:

```
getOption("DelayedArray.block.size")

## [1] 4500000

options(DelayedArray.block.size=1e6)
colSums(M)

## [1] 20637971 18809481 25348649 15163415 24448408 30818215 19126151 21164133
```

1 Motivation and challenges

2 Memory footprint

3 RleArray and HDF5Array objects

4 Hands-on

5 DelayedArray/HDF5Array: Future developments

Hands-on

1. Load the "airway" dataset.

2. It's wrapped in a SummarizedExperiment object. Get the count data as an ordinary matrix.

3. Wrap it in an HDF5Matrix object: (1) using `writeHDF5Array()`; then (2) using coercion.

4. When using coercion, where has the data been written on disk?

5. See `?setHDF5DumpFile` for how to control the location of "automatic" HDF5 datasets. Try to control the destination of the data when coercing.

Hands-on

6. Use `showHDF5DumpLog()` to see all the HDF5 datasets written to disk during the current session.

7. Try some operations on the HDF5Matrix object: (1) some delayed ones; (2) some non-delayed ones (block processing).

8. Use `DelayedArray:::set_verbose_block_processing(TRUE)` to see block processing in action.

9. Control the block size via `DelayedArray.block.size` global option.

Hands-on

10. Stick the HDF5Matrix object back in the SummarizedExperiment object. The resulting object is an "HDF5-backed SummarizedExperiment object".

11. The HDF5-backed SummarizedExperiment object can be manipulated (almost) like an in-memory SummarizedExperiment object. Try `[`, `cbind`, `rbind` on it.

12. The *SummarizedExperiment* package provides `saveHDF5SummarizedExperiment` to save a SummarizedExperiment object (HDF5-backed or not) as an HDF5-backed SummarizedExperiment object. Try it.

1 Motivation and challenges

2 Memory footprint

3 RleArray and HDF5Array objects

4 Hands-on

5 DelayedArray/HDF5Array: Future developments

Block processing improvements

Block genometry: (1) better by default, (2) let the user have more control on it

Support multicore

Expose it: `blockApply()`

HDF5Array improvements

Store the `dimnames` in the HDF5 file (in *HDF5 Dimension Scale datasets* - `https://www.hdfgroup.org/HDF5/Tutor/h5dimscale.html`)

Use better default chunk geometry when realizing an HDF5Array object

Block processing should take advantage of the chunk geometry (e.g. `realize()` should use blocks that are clusters of chunks)

Unfortunately: not possible to support multicore realization at the moment (HDF5 does not support concurrent writing to a dataset yet)

Future developments

RleArray improvements

Let the user have more control on the chunk geometry when constructing/realizing an RleArray object

Like for HDF5Array objects, block processing should take advantage of the chunk geometry

Support multicore realization

Provide C/C++ low-level API for direct row/column access from C/C++ code (e.g. from the *beachmat* package)