Package 'DelayedArray'

October 15, 2018

Title Delayed operations on array-like objects

Description Wrapping an array-like object (typically an on-disk object) in a DelayedArray object allows one to perform common array operations on it without loading the object in memory. In order to reduce memory usage and optimize performance, operations on the object are either delayed or executed using a block processing mechanism. Note that this also works on in-memory array-like objects like DataFrame objects (typically with Rle columns), Matrix objects, and ordinary arrays and data frames.

Version 0.6.6

Encoding UTF-8

Author Hervé Pagès

Maintainer Hervé Pagès <hpages@fredhutch.org>

biocViews Infrastructure, DataRepresentation, Annotation, GenomeAnnotation

Depends R (>= 3.4), methods, stats4, matrixStats, BiocGenerics (>= 0.25.1), S4Vectors (>= 0.17.43), IRanges (>= 2.11.17), BiocParallel

Imports stats

Suggests Matrix, HDF5Array, genefilter, SummarizedExperiment, airway, pryr, knitr, BiocStyle, RUnit

License Artistic-2.0

VignetteBuilder knitr

Collate utils.R bind-arrays.R Array-class.R ArrayGrid-class.R extract_array.R show-utils.R DelayedOp-class.R showtree.R DelayedArray-class.R realize.R block_processing.R DelayedArray-utils.R DelayedMatrix-utils.R DelayedArray-stats.R DelayedMatrix-stats.R RleArray-class.R zzz.R

git_url https://git.bioconductor.org/packages/DelayedArray

git_branch RELEASE_3_7

git_last_commit bdb0ac0

git_last_commit_date 2018-09-10

Date/Publication 2018-10-15

R topics documented:

`bind-arrays` *Bind arrays along their rows or columns*

Description

Bind array-like objects with an arbitrary number of dimensions along their rows (`arbind`) or columns (`acbind`).

Usage

```
arbind(...)
acbind(...)
```

Arguments

`...`	The array-like objects to bind.

Value

An array-like object, typically of the same class as the input objects if they all have the same class.

See Also

- `DelayedArray` in this package for arbind/acbind'ing DelayedArray objects.
- `rbind` and `cbind` in the **base** package for the corresponding operations on matrix-like objects.
- The **abind** package on CRAN.

Examples

```
a1 <- array(1:60, c(3, 5, 4),
            dimnames=list(NULL, paste0("M1y", 1:5), NULL))
a2 <- array(101:240, c(7, 5, 4),
            dimnames=list(paste0("M2x", 1:7), paste0("M2y", 1:5), NULL))
a3 <- array(10001:10100, c(5, 5, 4),
            dimnames=list(paste0("M3x", 1:5), NULL, paste0("M3z", 1:4)))

arbind(a1, a2, a3)
```

block_processing *Block processing of an array*

Description

A set of utilities for processing an array-like object block by block.

Details

Coming soon...

See Also

- DelayedArray objects.
- `realize` for realizing a DelayedArray object in memory or on disk.

DelayedArray-class *DelayedArray objects*

Description

Wrapping an array-like object (typically an on-disk object) in a DelayedArray object allows one to perform common array operations on it without loading the object in memory. In order to reduce memory usage and optimize performance, operations on the object are either delayed or executed using a block processing mechanism.

Usage

```
DelayedArray(seed)  # constructor function
seed(x)             # seed getter
nseed(x)            # seed counter
path(object, ...)   # path getter
type(x)
```

Arguments

`seed`	An array-like object.
`x, object`	A DelayedArray object. For `type()`, `x` can also be any array-like object, that is, any object for which `dim(x)` is not NULL.
`...`	Additional arguments passed to methods.

In-memory versus on-disk realization

To *realize* a DelayedArray object (i.e. to trigger execution of the delayed operations carried by the object and return the result as an ordinary array), call `as.array` on it. However this realizes the full object at once *in memory* which could require too much memory if the object is big. A big DelayedArray object is preferrably realized *on disk* e.g. by calling `writeHDF5Array` on it (this function is defined in the **HDF5Array** package) or coercing it to an HDF5Array object with `as(x, "HDF5Array")`. Other on-disk backends can be supported. This uses a block-processing strategy so that the full object is not realized at once in memory. Instead the object is processed block by block i.e. the blocks are realized in memory and written to disk one at a time. See ?`writeHDF5Array` in the **HDF5Array** package for more information about this.

Accessors

DelayedArray objects support the same set of getters as ordinary arrays i.e. `dim()`, `length()`, and `dimnames()`. In addition, they support `seed()`, `nseed()`, `path()`, and `type()`. `type()` is the DelayedArray equivalent of `typeof()` (or `storage.mode()`) for ordinary arrays. Note that, for convenience and consistency, `type()` also supports ordinary arrays and, more generally, any array-like object, that is, any object `x` for which `dim(x)` is not NULL.

`dimnames()`, `seed()`, and `path()` also work as setters.

Subsetting

A DelayedArray object can be subsetted with `[` like an ordinary array but with the following differences:

- *Multi-dimensional single bracket subsetting* (i.e. subsetting of the form `x[i_1, i_2, ..., i_n]` with one (possibly missing) subscript per dimension) returns a DelayedArray object where the subsetting is actually delayed. So it's a very light operation.
- *Linear single bracket subsetting* (a.k.a. 1D-style subsetting, that is, subsetting of the form `x[i]`) only works if subscript `i` is a numeric vector at the moment. Furthermore, `i` cannot contain NAs and all the indices in it must be >= 1 and <= `length(x)` for now. It returns an atomic vector of the same length as `i`. This is NOT a delayed operation.

Subsetting with `[[` is supported but only the *linear* form of it at the moment i.e. the `x[[i]]` form where `i` is a *single* numeric value >= 1 and <= `length(x)`. It is equivalent to `x[i]`.

DelayedArray objects support only 2 forms of subassignment at the moment: `x[i] <- value` and `x[] <- value`. The former is supported only when the subscript `i` is a logical DelayedArray object with the same dimensions as `x` and when `value` is a *scalar* (i.e. an atomic vector of length 1). The latter is supported only when `value` is an atomic vector and `length(value)` is a divisor of `nrow(x)`. Both are delayed operations so are very light.

Single value replacement (`x[[...]] <- value`) is not supported.

Binding

Binding DelayedArray objects along the rows (or columns) is supported via the `rbind` and `arbind` (or `cbind` and `acbind`) methods for DelayedArray objects. All these operations are delayed.

See Also

- `type` to get the type of the elements of an array-like object.
- `realize` for realizing a DelayedArray object in memory or on disk.
- DelayedArray-utils for common operations on DelayedArray objects.

- DelayedArray-stats for statistical functions on DelayedArray objects.
- cbind in the **base** package for rbind/cbind'ing ordinary arrays.
- acbind in this package (**DelayedArray**) for arbind/acbind'ing ordinary arrays.
- RleArray objects.
- HDF5Array objects in the **HDF5Array** package.
- DataFrame objects in the **S4Vectors** package.
- array objects in base R.

Examples

```
## ---------------------------------------------------------------------
## A. WRAP AN ORDINARY ARRAY IN A DelayedArray OBJECT
## ---------------------------------------------------------------------
a <- array(runif(1500000), dim=c(10000, 30, 5))
A <- DelayedArray(a)
A
## The seed of A is treated as a "read-only" object so won't change when
## we start operating on A:
stopifnot(identical(a, seed(A)))
type(A)

## Multi-dimensional single bracket subsetting:
m <- a[11:20 , 5, ]  # a matrix
M <- A[11:20 , 5, ]  # a DelayedMatrix object
stopifnot(identical(m, as.array(M)))

## Linear single bracket subsetting:
A[11:20]
A[A <= 1e-5]

## Subassignment:
A[A < 0.2] <- NA
a[a < 0.2] <- NA
stopifnot(identical(a, as.array(A)))

## Other operations:
toto <- function(x) (5 * x[ , , 1] ^ 3 + 1L) * log(x[, , 2])
b <- toto(a)
head(b)

B <- toto(A)  # very fast! (operations are delayed)
B

cs <- colSums(b)
CS <- colSums(B)
stopifnot(identical(cs, CS))

## ---------------------------------------------------------------------
## B. WRAP A DataFrame OBJECT IN A DelayedArray OBJECT
## ---------------------------------------------------------------------

## Generate random coverage and score along an imaginary chromosome:
cov <- Rle(sample(20, 5000, replace=TRUE), sample(6, 5000, replace=TRUE))
score <- Rle(sample(100, nrun(cov), replace=TRUE), runLength(cov))
```

```
DF <- DataFrame(cov, score)
A2 <- DelayedArray(DF)
A2
seed(A2)  # 'DF'

## Coercion of a DelayedMatrix object to DataFrame produces a DataFrame
## object with Rle columns:
as(A2, "DataFrame")
stopifnot(identical(DF, as(A2, "DataFrame")))

t(A2)  # transposition is delayed so is very fast and memory efficient
colSums(A2)

## ---------------------------------------------------------------------
## C. A HDF5Array OBJECT IS A (PARTICULAR KIND OF) DelayedArray OBJECT
## ---------------------------------------------------------------------
library(HDF5Array)
A3 <- as(a, "HDF5Array")   # write 'a' to an HDF5 file
A3
is(A3, "DelayedArray")     # TRUE
seed(A3)                   # an HDF5ArraySeed object

B3 <- toto(A3)             # very fast! (operations are delayed)
B3                         # not an HDF5Array object anymore because
                           # now it carries delayed operations
CS3 <- colSums(B3)
stopifnot(identical(cs, CS3))

## ---------------------------------------------------------------------
## D. PERFORM THE DELAYED OPERATIONS
## ---------------------------------------------------------------------
as(B3, "HDF5Array")        # "realize" 'B3' on disk

## If this is just an intermediate result, you can either keep going
## with B3 or replace it with its "realized" version:
B3 <- as(B3, "HDF5Array")  # no more delayed operations on new 'B3'
seed(B3)
path(B3)

## For convenience, realize() can be used instead of explicit coercion.
## The current "realization backend" controls where realization
## happens e.g. in memory if set to NULL or in an HDF5 file if set
## to "HDF5Array":
D <- cbind(B3, exp(B3))
D
setRealizationBackend("HDF5Array")
D <- realize(D)
D
## See '?realize' for more information about "realization backends".

## ---------------------------------------------------------------------
## E. BIND DelayedArray OBJECTS
## ---------------------------------------------------------------------

## rbind/cbind

library(HDF5Array)
```

```
toy_h5 <- system.file("extdata", "toy.h5", package="HDF5Array")
h5ls(toy_h5)

M1 <- HDF5Array(toy_h5, "M1")
M2 <- HDF5Array(toy_h5, "M2")

M12 <- rbind(M1, t(M2))
M12
colMeans(M12)

## arbind/acbind

example(acbind)  # to create arrays a1, a2, a3

A1 <- DelayedArray(a1)
A2 <- DelayedArray(a2)
A3 <- DelayedArray(a3)
A <- arbind(A1, A2, A3)
A

## Sanity check:
stopifnot(identical(arbind(a1, a2, a3), as.array(A)))

## ---------------------------------------------------------------------
## F. MODIFY THE PATH OF A DelayedArray OBJECT
## ---------------------------------------------------------------------
## This can be useful if the file containing the array data is on a
## shared partition but the exact path to the partition depends on the
## machine from which the data is being accessed.
## For example:

## Not run:
A <- HDF5Array("/path/to/lab_data/my_precious_data.h5")
path(A)

## Operate on A...
## Now A carries delayed operations.
## Make sure path(A) still works:
path(A)

## Save A:
save(A, file="A.rda")

## A.rda should be small (it doesn't contain the array data).
## Send it to a co-worker that has access to my_precious_data.h5.

## Co-worker loads it:
load("A.rda")
path(A)

## A is broken because path(A) is incorrect for co-worker:
A  # error!

## Co-worker fixes the path (in this case this is better done using the
## dirname() setter rather than the path() setter):
dirname(A) <- "E:/other/path/to/lab_data"
```

```
## A "works" again:
A

## End(Not run)

## ---------------------------------------------------------------------
## G. WRAP A SPARSE MATRIX IN A DelayedArray OBJECT
## ---------------------------------------------------------------------
## Not run:
library(Matrix)
M <- 75000L
N <- 1800L
p <- sparseMatrix(sample(M, 9000000, replace=TRUE),
                  sample(N, 9000000, replace=TRUE),
                  x=runif(9000000), dims=c(M, N))
P <- DelayedArray(p)
P
p2 <- as(P, "sparseMatrix")
stopifnot(identical(p, p2))

## The following is based on the following post by Murat Tasan on the
## R-help mailing list:
##   https://stat.ethz.ch/pipermail/r-help/2017-May/446702.html

## As pointed out by Murat, the straight-forward row normalization
## directly on sparse matrix 'p' would consume too much memory:
row_normalized_p <- p / rowSums(p^2)  # consumes too much memory
## because the rowSums() result is being recycled (appropriately) into a
## *dense* matrix with dimensions equal to dim(p).

## Murat came up with the following solution that is very fast and memory
## efficient:
row_normalized_p1 <- Diagonal(x=1/sqrt(Matrix::rowSums(p^2)))

## With a DelayedArray object, the straight-forward approach uses a
## block processing strategy behind the scene so it doesn't consume
## too much memory.

## First, let's see the block processing in action:
DelayedArray:::set_verbose_block_processing(TRUE)
## and set block size to a bigger value than the default:
getOption("DelayedArray.block.size")
options(DelayedArray.block.size=80e6)

row_normalized_P <- P / sqrt(DelayedArray::rowSums(P^2))

## Increasing the block size increases the speed but also memory usage:
options(DelayedArray.block.size=200e6)
row_normalized_P2 <- P / sqrt(DelayedArray::rowSums(P^2))
stopifnot(all.equal(row_normalized_P, row_normalized_P2))

## Back to sparse representation:
DelayedArray:::set_verbose_block_processing(FALSE)
row_normalized_p2 <- as(row_normalized_P, "sparseMatrix")
stopifnot(all.equal(row_normalized_p1, row_normalized_p2))

options(DelayedArray.block.size=10e6)
```

```
## End(Not run)
```

DelayedArray-stats *Statistical functions on DelayedArray objects*

Description

Statistical functions on DelayedArray objects.

All these functions are implemented as delayed operations.

Usage

```
## --- The Normal Distribution ----- ##

## S4 method for signature 'DelayedArray'
dnorm(x, mean=0, sd=1, log=FALSE)
## S4 method for signature 'DelayedArray'
pnorm(q, mean=0, sd=1, lower.tail=TRUE, log.p=FALSE)
## S4 method for signature 'DelayedArray'
qnorm(p, mean=0, sd=1, lower.tail=TRUE, log.p=FALSE)

## --- The Binomial Distribution --- ##

## S4 method for signature 'DelayedArray'
dbinom(x, size, prob, log=FALSE)
## S4 method for signature 'DelayedArray'
pbinom(q, size, prob, lower.tail=TRUE, log.p=FALSE)
## S4 method for signature 'DelayedArray'
qbinom(p, size, prob, lower.tail=TRUE, log.p=FALSE)

## --- The Poisson Distribution ---- ##

## S4 method for signature 'DelayedArray'
dpois(x, lambda, log=FALSE)
## S4 method for signature 'DelayedArray'
ppois(q, lambda, lower.tail=TRUE, log.p=FALSE)
## S4 method for signature 'DelayedArray'
qpois(p, lambda, lower.tail=TRUE, log.p=FALSE)

## --- The Logistic Distribution --- ##

## S4 method for signature 'DelayedArray'
dlogis(x, location=0, scale=1, log=FALSE)
## S4 method for signature 'DelayedArray'
plogis(q, location=0, scale=1, lower.tail=TRUE, log.p=FALSE)
## S4 method for signature 'DelayedArray'
qlogis(p, location=0, scale=1, lower.tail=TRUE, log.p=FALSE)
```

Arguments

`x, q, p`	A DelayedArray object.
`mean, sd, log, lower.tail, log.p, size, prob, lambda, location, scale`	See `?stats::dnorm`, `?stats::dbinom`, `?stats::dpois`, and `?stats::dlogis`, for a description of these arguments.

See Also

- `dnorm`, `dbinom`, `dpois`, and `dlogis` in the **stats** package for the corresponding operations on ordinary arrays or matrices.
- DelayedArray objects.
- HDF5Array objects in the **HDF5Array** package.
- array objects in base R.

Examples

```
a <- array(4 * runif(1500000), dim=c(10000, 30, 5))
A <- DelayedArray(a)
A

A2 <- dnorm(A + 1)[ , , -3]  # very fast! (operations are delayed)
A2

a2 <- as.array(A2)           # "realize" 'A2' in memory (as an ordinary
                             # array)

DelayedArray(a2) == A2       # DelayedArray object of type "logical"
stopifnot(all(DelayedArray(a2) == A2))


library(HDF5Array)
A3 <- as(A2, "HDF5Array")    # "realize" 'A2' on disk (as an HDF5Array
                             # object)

A3 == A2                     # DelayedArray object of type "logical"
stopifnot(all(A3 == A2))

## See '?DelayedArray' for general information about DelayedArray objects
## and their "realization".
```

DelayedArray-utils *Common operations on DelayedArray objects*

Description

Common operations on DelayedArray objects.

Details

The operations currently supported on DelayedArray objects are:

Delayed operations:

- all the members of the `Ops`, `Math`, and `Math2` groups
- `sweep`
- `!`
- `is.na`, `is.finite`, `is.infinite`, `is.nan`
- `lengths`
- `nchar`, `tolower`, `toupper`, `grepl`, `sub`, `gsub`
- `pmax2` and `pmin2`
- `t`
- `rbind` and `cbind` (documented in DelayedArray)
- statistical functions like `dnorm`, `dbinom`, `dpois`, and `dlogis` (for the Normal, Binomial, Poisson, and Logistic distribution, respectively) and related functions (documented in DelayedArray-stats)

Block-processed operations:

- `anyNA`, `which`
- all the members of the `Summary` group
- `mean`
- `apply`
- matrix multiplication (%*%) of an ordinary matrix by a DelayedMatrix object
- matrix row/col summarization [DelayedMatrix objects only]: `rowSums`, `colSums`, `rowMeans`, `colMeans`, `rowMaxs`, `colMaxs`, `rowMins`, `colMins`, `rowRanges`, and `colRanges`

See Also

- `is.na`, `!`, `mean`, `apply`, and `%*%` in the **base** package for the corresponding operations on ordinary arrays or matrices.
- `rowSums` in the **base** package and `rowMaxs` in the **matrixStats** package for row/col summarization of an ordinary matrix.
- `setRealizationBackend` for how to set a *realization backend*.
- `writeHDF5Array` in the **HDF5Array** package for writing an array-like object to an HDF5 file and other low-level utilities to control the location of automatically created HDF5 datasets.
- DelayedArray objects.
- DelayedArray-stats for statistical functions on DelayedArray objects.
- HDF5Array objects in the **HDF5Array** package.
- `S4groupGeneric` in the **methods** package for the members of the `Ops`, `Math`, and `Math2` groups.
- array objects in base R.

Examples

```
library(HDF5Array)
toy_h5 <- system.file("extdata", "toy.h5", package="HDF5Array")
h5ls(toy_h5)

M1 <- HDF5Array(toy_h5, "M1")
range(M1)
M1 >= 0.5 & M1 < 0.75
log(M1)

M2 <- HDF5Array(toy_h5, "M2")
pmax2(M2, 0)

sweep(M2, 2, colMeans(M2))

M3 <- rbind(M1, t(M2))
M3

## ---------------------------------------------------------------------
## MATRIX MULTIPLICATION
## ---------------------------------------------------------------------

## Matrix multiplication is not delayed: the output matrix is realized
## block by block. The current "realization backend" controls where
## realization happens e.g. in memory if set to NULL or in an HDF5 file
## if set to "HDF5Array". See '?realize' for more information about
## "realization backends".
## The output matrix is returned as a DelayedMatrix object with no delayed
## operations on it. The exact class of the object depends on the backend
## e.g. it will be HDF5Matrix with "HDF5Array" backend.

m <- matrix(runif(50000), ncol=nrow(M1))

## Set backend to NULL for in-memory realization:
setRealizationBackend()
P1 <- m %*% M1
P1

## Set backend to HDF5Array for realization in HDF5 file:
setRealizationBackend("HDF5Array")

## With the HDF5Array backend, the output matrix will be written to an
## automatic location on disk:
getHDF5DumpFile()  # HDF5 file where the output matrix will be written
lsHDF5DumpFile()

P2 <- m %*% M1
P2

lsHDF5DumpFile()

## Use setHDF5DumpFile() and setHDF5DumpName() from the HDF5Array package
## to control the location of automatically created HDF5 datasets.

stopifnot(identical(as.array(P1), as.array(P2)))
```

```
## ---------------------------------------------------------------------
## MATRIX ROW/COL SUMMARIZATION
## ---------------------------------------------------------------------

rowSums(M1)
colSums(M1)

rowMeans(M1)
colMeans(M1)

rmaxs <- rowMaxs(M1)
cmaxs <- colMaxs(M1)

rmins <- rowMins(M1)
cmins <- colMins(M1)

rranges <- rowRanges(M1)
cranges <- colRanges(M1)

stopifnot(identical(cbind(rmins, rmaxs, deparse.level=0), rranges))
stopifnot(identical(cbind(cmins, cmaxs, deparse.level=0), cranges))
```

`DelayedOp-class` *DelayedOp objects*

Description

In a DelayedArray object the delayed operations are stored as a tree of DelayedOp objects. Each node in the tree is represented by a DelayedOp object.

DelayedOp objects are used inside DelayedArray objects and are not intended to be manipulated directly by the end user.

`showtree` and `simplify` can be used to visualize and simplify this tree.

Usage

```
isNoOp(x)
```

Arguments

x	A DelayedSubset, DelayedAperm, or DelayedDimnames object.

Details

6 types of nodes are currently supported. Each type is a DelayedOp subclass:

```
Node type    Outdegree  Operation
-----------------------------------------------------------------------
DelayedSubset        1  Multi-dimensional single bracket subsetting
DelayedAperm         1  Extended aperm() (can drop dimensions)
DelayedUnaryIsoOp    1  Unary op that preserves the geometry
DelayedDimnames      1  Set dimnames
DelayedNaryIsoOp     N  N-ary op that preserves the geometry
DelayedAbind         N  abind()
```

All the nodes are array-like objects that must comply with the *seed contract* i.e. they must support `dim()`, `dimnames()`, and `extract_array()`. See ?`extract_array` for more information about the *seed contract*.

`isNoOp()` can be called on a DelayedSubset, DelayedAperm, or DelayedDimnames object and will return `TRUE` if the object represents a no-op.

Note

The DelayedOp virtual class and its 6 concrete subclasses are for internal use only and are not exported.

See Also

- DelayedArray objects.
- `showtree` to visualize, simplify, and inspect the tree of delayed operations in a DelayedArray object.
- `extract_array`.

---

realize *Realize a DelayedArray object*

Description

Realize a DelayedArray object in memory or on disk. Get or set the *realization backend* for the current session with `getRealizationBackend` or `setRealizationBackend`.

Usage

```
supportedRealizationBackends()
getRealizationBackend()
setRealizationBackend(BACKEND=NULL)

realize(x, ...)

## S4 method for signature 'ANY'
realize(x, BACKEND=getRealizationBackend())
```

Arguments

`x`	The array-like object to realize.
`...`	Additional arguments passed to methods.
`BACKEND`	`NULL` (the default), or a single string specifying the name of the backend. When the backend is set to `NULL`, `x` is realized in memory as an ordinary array by just calling `as.array` on it.

Details

The *realization backend* controls where/how realization happens e.g. as an ordinary array if set to `NULL`, as an RleArray object if set to `"RleArray"`, or in an HDF5 file if set to `"HDF5Array"`.

Value

`realize(x)` returns a DelayedArray object. More precisely, it returns `DelayedArray(as.array(x))` when the backend is set to `NULL` (the default). Otherwise it returns an instance of the class associated with the specified backend (which should extend DelayedArray).

See Also

- DelayedArray objects.
- RleArray objects.
- HDF5Array objects in the **HDF5Array** package.
- array objects in base R.

Examples

```
library(HDF5Array)
toy_h5 <- system.file("extdata", "toy.h5", package="HDF5Array")
h5ls(toy_h5)
M1 <- HDF5Array(toy_h5, "M1")
M2 <- HDF5Array(toy_h5, "M2")
M3 <- rbind(log(M1), t(M2))

supportedRealizationBackends()
getRealizationBackend()  # backend is set to NULL
realize(M3)  # realization as ordinary array

setRealizationBackend("RleArray")
getRealizationBackend()  # backend is set to "RleArray"
realize(M3)  # realization as RleArray object

setRealizationBackend("HDF5Array")
getRealizationBackend()  # backend is set to "HDF5Array"
realize(M3)  # realization in HDF5 file
```

RleArray-class *RleArray objects*

Description

The RleArray class is an array-like container where the values are stored in a run-length encoding format. RleArray objects support delayed operations and block processing.

Usage

```
RleArray(rle, dim, dimnames=NULL, chunksize=NULL)  # constructor function
```

Arguments

`rle`	An Rle object.
`dim`	The dimensions of the object to be created, that is, an integer vector of length one or more giving the maximal indices in each dimension.

dimnames	Either NULL or the names for the dimensions. This must a list of length the number of dimensions. Each list element must be either NULL or a character vector along the corresponding dimension.
chunksize	Experimental. Don't use!

Details

RleArray extends DelayedArray. All the operations available on DelayedArray objects work on RleArray objects.

See Also

- Rle objects in the **S4Vectors** package.
- DelayedArray objects.
- DelayedArray-utils for common operations on DelayedArray objects.
- `realize` for realizing a DelayedArray object in memory or on disk.
- HDF5Array objects in the **HDF5Array** package.
- DataFrame objects in the **S4Vectors** package.
- array objects in base R.

Examples

```
rle <- Rle(sample(6L, 500000, replace=TRUE), 8)
a <- array(rle, dim=c(50, 20, 4000))  # array() expands the Rle object
                                      # internally with as.vector()

A <- RleArray(rle, dim=c(50, 20, 4000))  # Rle object is NOT expanded
A

object.size(a)
object.size(A)

stopifnot(identical(a, as.array(A)))

as(A, "Rle")  # deconstruction

toto <- function(x) (5 * x[ , , 1] ^ 3 + 1L) * log(x[, , 2])
b <- toto(a)
head(b)

B <- toto(A)  # very fast! (operations are delayed)
B

stopifnot(identical(b, as.array(B)))

cs <- colSums(b)
CS <- colSums(B)
stopifnot(identical(cs, CS))

## Coercion of a DelayedMatrix object to DataFrame produces a DataFrame
## object with Rle columns:
as(B, "DataFrame")

## Coercion of an RleList object to RleArray only works if all the list
```

```
## elements in the RleList have the same length. Column names are taken
## from the names of the list elements.
rle_list <- RleList(A=Rle(sample(3L, 100, replace=TRUE)),
                    B=Rle(sample(3L, 100, replace=TRUE)))
C <- as(rle_list, "RleArray")
C
stopifnot(identical(as(C, "RleList"), rle_list))
```

---

showtree *Visualize, simplify, and inspect a tree of delayed operations*

Description

NOTE: The tools documented in this man page are primarily intended for developers. End users of DelayedArray objects will typically not need them.

`showtree` can be used to visualize the tree of delayed operations carried by a DelayedArray object.

`simplify` can be used to simplify this tree.

`contentIsPristine` can be used to know whether the operations in this tree leave the values of the array elements intact or not.

`netSubsetAndAperm` returns an object that represents the *net subsetting* and *net dimension rearrangement* of all the operations in this tree.

Usage

```
showtree(x, show.node.dim=TRUE)
simplify(x, incremental=FALSE)

contentIsPristine(x)
netSubsetAndAperm(x, as.DelayedOp=FALSE)
```

Arguments

`x`	Typically a DelayedArray object but can also be a DelayedOp object. Additionally `showtree` accepts a list where each element is a DelayedArray or DelayedOp object.
`show.node.dim`	`TRUE` or `FALSE`. If `TRUE` (the default), the nodes dimensions and data type are displayed.
`incremental`	For internal use.
`as.DelayedOp`	`TRUE` or `FALSE`. Control the form of the returned object. See details below.

Details

`netSubsetAndAperm` is only supported on a DelayedArray object `x` with a single seed i.e. if `nseed(x) == 1`.

The mapping between the elements of `x` and the elements of its seed is affected by the following delayed operations carried by `x`: `[`, `drop()`, and `aperm()`. `x` can carry any number of each of these operations in any order but their net result can always be described by a *net subsetting* followed by a *net dimension rearrangement*.

`netSubsetAndAperm(x)` returns an object that represents the *net subsetting* and *net dimension rearrangement*. The `as.DelayedOp` argument controls in what form this object should be returned:

- If `as.DelayedOp` is `FALSE` (the default), the returned object is a list of subscripts that describes the *net subsetting*. The list contains one subscript per dimension in the seed. Each subscript can be either a vector of positive integers or a `NULL`. A `NULL` indicates a *missing subscript*. In addition, if `x` carries delayed operations that rearrange its dimensions (i.e. operations that drop and/or permute some of the original dimensions), the *net dimension rearrangement* is described in a `dimmap` attribute added to the list. This attribute is an integer vector parallel to `dim(x)` that reports how the dimensions of `x` are mapped to the dimensions of its seed.
- If `as.DelayedOp` is `TRUE`, the returned object is a linear tree with 2 DelayedOp nodes and a leaf node. The leaf node is the seed of `x`. Walking the tree from the seed, the 2 DelayedOp nodes are of type DelayedSubset and DelayedAperm, in that order (this reflects the order in which the operations apply). More precisely, the returned object is a DelayedAperm object with one child (the DelayedSubset object), and one grandchid (the seed of `x`). The DelayedSubset and DelayedAperm nodes represent the *net subsetting* and *net dimension rearrangement*, respectively. Either or both of them can be a no-op.

Note that the returned object describes how the elements of `x` map to their corresponding element in `seed(x)`.

Value

The simplified object for `simplify`.

`TRUE` or `FALSE` for `contentIsPristine`.

An ordinary list (possibly with the `dimmap` attribute on it) for `netSubsetAndAperm`. Unless `as.DelayedOp` is set to `TRUE`, in which case a DelayedAperm object is returned (see Details section above for more information).

See Also

- DelayedArray objects.
- DelayedOp objects.

Examples

```
## ---------------------------------------------------------------------
## showtree()
## ---------------------------------------------------------------------
m1 <- matrix(runif(150), nrow=15, ncol=10)
M1 <- DelayedArray(m1)

## By default, the tree of delayed operations carried by a DelayedArray
## object gets simplified each time a delayed operation is added to it.
## This can be disabled via a global option:
options(DelayedArray.simplify=FALSE)
M2 <- log(t(M1[5:1, c(TRUE, FALSE)] + 10))[-1, ]
showtree(M2)

## Note that as part of the simplification process, some operations
## can be reordered:
options(DelayedArray.simplify=TRUE)
M2 <- log(t(M1[5:1, c(TRUE, FALSE)] + 10))[-1, ]
showtree(M2)

## In the above example, the tree is linear i.e. all the operations
## are represented by unary nodes. The simplest way to know if a
```

```
## tree is linear is by counting its leaves with nseed():
nseed(M2)  # only 1 leaf means the tree is linear

options(DelayedArray.simplify=FALSE)

dimnames(M1) <- list(letters[1:15], LETTERS[1:10])
showtree(M1)

m2 <- matrix(1:20, nrow=10)
Y <- cbind(t(M1[ , 10:1]), DelayedArray(m2), M1[6:15, "A", drop=FALSE])
showtree(Y)
showtree(Y, show.node.dim=FALSE)
nseed(Y)  # the tree is not linear

Z <- t(Y[10:1, ])[1:15, ] + 0.4 * M1
showtree(Z)
nseed(Z)

Z@seed@seeds
Z@seed@seeds[[2]]@seed            # reaching to M1
Z@seed@seeds[[1]]@seed@seed@seed  # reaching to Y

## ---------------------------------------------------------------------
## contentIsPristine()
## ---------------------------------------------------------------------
a <- array(1:120, c(4, 5, 2))
A <- DelayedArray(a)

stopifnot(contentIsPristine(A))
stopifnot(contentIsPristine(A[1, , ]))
stopifnot(contentIsPristine(t(A[1, , ])))
stopifnot(contentIsPristine(cbind(A[1, , ], A[2, , ])))
dimnames(A) <- list(LETTERS[1:4], letters[1:5], NULL)
stopifnot(contentIsPristine(A))

contentIsPristine(log(A))     # FALSE
contentIsPristine(A - 11:14)  # FALSE
contentIsPristine(A * A)      # FALSE

## ---------------------------------------------------------------------
## netSubsetAndAperm()
## ---------------------------------------------------------------------
a <- array(1:120, c(4, 5, 2))
M <- aperm(DelayedArray(a)[ , -1, ] / 100)[ , , 3] + 99:98
M
showtree(M)

netSubsetAndAperm(M)  # 1st dimension was dropped, 2nd and 3rd
                      # dimension were permuted (transposition)

op2 <- netSubsetAndAperm(M, as.DelayedOp=TRUE)
op2                   # 2 nested delayed operations
op1 <- op2@seed
class(op1)            # DelayedSubset
class(op2)            # DelayedAperm
op1@index
op2@perm
```

```
DelayedArray(op2)     # same as M from a [, drop(), and aperm() point of
                      # view but the individual array elements are now
                      # reset to their original values i.e. to the values
                      # they have in the seed
stopifnot(contentIsPristine(DelayedArray(op2)))

## A simple function that returns TRUE if a DelayedArray object carries
## no "net subsetting" and no "net dimension rearrangement":
is_aligned_with_seed <- function(x)
{
    if (nseed(x) != 1L)
        return(FALSE)
    op2 <- netSubsetAndAperm(x, as.DelayedOp=TRUE)
    op1 <- op2@seed
    isNoOp(op1) && isNoOp(op2)
}

M <- DelayedArray(a[ , , 1])
is_aligned_with_seed(log(M + 11:14) > 3)            # TRUE
is_aligned_with_seed(M[4:1, ])                      # FALSE
is_aligned_with_seed(M[4:1, ][4:1, ])               # TRUE
is_aligned_with_seed(t(M))                          # FALSE
is_aligned_with_seed(t(t(M)))                       # TRUE
is_aligned_with_seed(t(0.5 * t(M[4:1, ])[ , 4:1]))  # TRUE

options(DelayedArray.simplify=TRUE)
```

Index