Package 'DelayedArray'

October 16, 2019

Title A unified framework for working transparently with on-disk and in-memory array-like datasets

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 0.10.0

Encoding UTF-8

Author Hervé Pagès <hpages@fredhutch.org>, with contributions from Peter Hickey <peter.hickey@gmail.com> and Aaron Lun <infinite.monkeys.with.keyboards@gmail.com>

Maintainer Hervé Pagès <hpages@fredhutch.org>

biocViews Infrastructure, DataRepresentation, Annotation, GenomeAnnotation

Depends R (>= 3.4), methods, stats4, matrixStats, BiocGenerics (>= 0.27.1), S4Vectors (>= 0.21.7), IRanges (>= 2.17.3), BiocParallel

Imports stats, Matrix

LinkingTo S4Vectors

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

License Artistic-2.0

VignetteBuilder knitr

Collate utils.R compress_atomic_vector.R Nindex-utils.R aperm2.R linearInd.R bind-arrays.R Array-class.R extract_array.R ArrayGrid-class.R show-utils.R SparseArraySeed-class.R read_block.R mapToGrid.R makeCappedVolumeBox.R blockGrid.R DelayedOp-class.R showtree.R simplify.R DelayedArray-class.R chunkGrid.R block_processing.R RealizationSink-class.R realize.R DelayedArray-subsetting.R DelayedArray-utils.R DelayedMatrix-utils.R DelayedArray-stats.R DelayedMatrix-stats.R RleArraySeed-class.R RleArray-class.R zzz.R

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

git_branch RELEASE_3_9

git_last_commit 4781d07

git_last_commit_date 2019-05-02

Date/Publication 2019-10-15

R topics documented:

ArrayGrid-class   *ArrayGrid and ArrayViewport objects*

Description

ArrayGrid and ArrayViewport objects are used internally to support block processing of array-like objects.

Usage

```
## Constructor functions:
ArbitraryArrayGrid(tickmarks)
RegularArrayGrid(refdim, spacings=refdim)

downsample(x, ratio=1L)
```

Arguments

tickmarks	A list of integer vectors, one along each dimension of the reference array, representing the tickmarks along that dimension. Each integer vector must be sorted in ascending order. NAs or negative values are not allowed.
refdim	An integer vector containing the dimensions of the reference array.
spacings	An integer vector specifying the grid spacing along each dimension.
x	An ArrayGrid object.
ratio	An integer vector specifying the ratio of the downsampling along each dimension. Can be of length 1, in which case the same ratio is used along all the dimensions.

Value

- For `ArbitraryArrayGrid()`: An ArbitraryArrayGrid instance.
- For `RegularArrayGrid()`: A RegularArrayGrid instance.
- For `downsample()`: An ArrayGrid object on the same reference array than x.

See Also

- read_block.
- block_processing for more information about block processing of an array-like object.
- mapToGrid for mapping reference array positions to grid positions and vice-versa.
- chunkGrid.
- DelayedArray objects.
- array objects in base R.

Examples

```
## ---------------------------------------------------------------------
## A. ArrayGrid OBJECTS
## ---------------------------------------------------------------------

## Create a regularly-spaced grid on top of a 3700 x 100 x 33 array:
grid1 <- RegularArrayGrid(c(3700, 100, 33), c(250, 100, 10))

## Dimensions of the reference array:
refdim(grid1)

## Number of grid elements along each dimension of the reference array:
dim(grid1)

## Total number of grid elements:
length(grid1)

## First element in the grid:
grid1[[1L]]            # same as grid1[[1L, 1L, 1L]]

## Last element in the grid:
grid1[[length(grid1)]]  # same as grid1[[15L, 1L, 4L]]

## Dimensions of the grid elements:
```

```
dims(grid1)                # one row per grid element

## Lengths of the grid elements:
lengths(grid1)             # same as rowProds(dims(grid1))
stopifnot(sum(lengths(grid1)) == prod(refdim(grid1)))

maxlength(grid1)           # does not need to compute lengths(grid1)) first
                           # so is more efficient than max(lengths(grid1))
stopifnot(maxlength(grid1) == max(lengths(grid1)))

## Create an arbitrary-spaced grid on top of a 15 x 9 matrix:
grid2 <- ArbitraryArrayGrid(list(c(2L, 7:10, 13L, 15L), c(5:6, 6L, 9L)))

refdim(grid2)
dim(grid2)
length(grid2)
grid2[[1L]]                # same as grid2[[1L, 1L]]
grid2[[length(grid2)]]     # same as grid2[[15L, 9L]]

dims(grid2)
lengths(grid2)
array(lengths(grid2), dim(grid2))  # display the grid element lengths in
                                   # an array of same shape as grid2

stopifnot(sum(lengths(grid2)) == prod(refdim(grid2)))

maxlength(grid2)           # does not need to compute lengths(grid2)) first
                           # so is more efficient than max(lengths(grid2))
stopifnot(maxlength(grid2) == max(lengths(grid2)))

## Max (i.e. highest) resolution grid:
Hgrid <- RegularArrayGrid(6:4, c(1, 1, 1))
Hgrid
dim(Hgrid)                 # same as refdim(Hgrid)
stopifnot(identical(dim(Hgrid), refdim(Hgrid)))
stopifnot(all(lengths(Hgrid) == 1))

## Min (i.e. lowest) resolution grid:
Lgrid <- RegularArrayGrid(6:4, 6:4)
Lgrid
stopifnot(all(dim(Lgrid) == 1))
stopifnot(identical(dim(Lgrid[[1L]]), refdim(Lgrid)))
stopifnot(identical(dims(Lgrid), matrix(refdim(Lgrid), nrow=1)))

## ---------------------------------------------------------------------
## B. ArrayViewport OBJECTS
## ---------------------------------------------------------------------

## Grid elements are ArrayViewport objects:
grid1[[1L]]
class(grid1[[1L]])
grid1[[2L]]
grid1[[2L, 1L, 1L]]
grid1[[15L, 1L, 4L]]

## Construction of a standalong ArrayViewport object:
m0 <- matrix(1:30, ncol=5)
```

```
block_dim <- c(4, 3)
viewport1 <- ArrayViewport(dim(m0), IRanges(c(3, 2), width=block_dim))
viewport1

dim(viewport1)     # 'block_dim'
length(viewport1)  # number of array elements in the viewport
ranges(viewport1)

## ----------------------------------------------------------------------
## C. GRIDS CAN BE TRANSPOSED
## ----------------------------------------------------------------------

tgrid2 <- t(grid2)
dim(tgrid2)
refdim(tgrid2)

## Use aperm() if the grid has more than 2 dimensions:
tgrid1 <- aperm(grid1)
dim(tgrid1)
refdim(tgrid1)

aperm(grid1, c(3, 1, 2))
aperm(grid1, c(1, 3, 2))
aperm(grid1, c(3, 1))     # some dimensions can be dropped
aperm(grid1, c(3, 2, 3))  # and some can be repeated

## ----------------------------------------------------------------------
## D. DOWNSAMPLING AN ArrayGrid OBJECT
## ----------------------------------------------------------------------
## The elements (ArrayViewport) of an ArrayGrid object can be replaced
## with bigger elements obtained by merging adjacent elements. How many
## adjacent elements to merge along each dimension is specified via the
## 'ratio' vector (one integer per dimension). We call this operation
## "downsampling. It can be seen as reducing the "resolution" of a grid
## by the specified ratio (if we think of the grid elements as pixels).
downsample(grid2, 2)
downsample(grid2, 3)
downsample(grid2, 4)

## Downsampling preserves the dimensions of the reference array:
stopifnot(identical(refdim(downsample(grid2, 2)), refdim(grid2)))
stopifnot(identical(refdim(downsample(grid2, 3)), refdim(grid2)))
stopifnot(identical(refdim(downsample(grid2, 4)), refdim(grid2)))

## A big enough ratio will eventually produce the coarsest possible grid
## i.e. a grid with a single grid element covering the entire reference
## array:
grid3 <- downsample(grid2, 7)
length(grid3)
grid3[[1L]]
stopifnot(identical(dim(grid3[[1L]]), refdim(grid3)))

## Downsampling by a ratio of 1 is a no-op:
stopifnot(identical(downsample(grid2, 1), grid2))

## Using one ratio per dimension:
downsample(grid2, c(2, 1))
```

```
## Downsample a max resolution grid:
refdim <- c(45, 16, 20)
grid4 <- RegularArrayGrid(refdim, c(1, 1, 1))
ratio <- c(6, 1, 3)
stopifnot(identical(
    downsample(grid4, ratio),
    RegularArrayGrid(refdim, ratio)
))
```

---

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)
```

---

blockGrid   *Define grids to use in the context of block processing*

Description

blockGrid() is the primary utility function to use to define a grid that is suitable for block processing of an array-like object.

rowGrid() and colGrid() are additional functions, specific to the 2-dimensional case. They can be used to define blocks of full rows or full columns.

A family of utilities is provided to control the automatic block size (or length) and shape.

Usage

```
## Define grids to use in the context of block processing:

blockGrid(x, block.length=NULL, chunk.grid=NULL, block.shape=NULL)

rowGrid(x, nrow=NULL, block.length=NULL)
colGrid(x, ncol=NULL, block.length=NULL)

## Control the automatic block size (or length) and shape:

getAutoBlockSize()
setAutoBlockSize(size=1e8)

getAutoBlockLength(type)

getAutoBlockShape()
setAutoBlockShape(shape=c("hypercube",
                         "scale",
                         "first-dim-grows-first",
                         "last-dim-grows-first"))
```

Arguments

x	An array-like or matrix-like object for blockGrid. A matrix-like object for rowGrid and colGrid.
block.length	The length of the blocks i.e. the number of array elements per block. By default the automatic block length (returned by getAutoBlockLength(type(x))) is used. Depending on how much memory is available on your machine, you might want to increase (or decrease) the automatic block length by adjusting the automatic block size with setAutoBlockSize().
chunk.grid	The grid of physical chunks. By default [chunkGrid](x) is used.
block.shape	A string specifying the shape of the blocks. See [makeCappedVolumeBox](#) for a description of the supported shapes. By default getAutoBlockShape() is used.
nrow	The number of rows of the blocks. The bottommost blocks might have less. See examples below.
ncol	The number of columns of the blocks. The rightmost blocks might have less. See examples below.

size   The automatic block size in bytes. Note that, except when the type of the array
                data is "character" or "list", the size of a block is its length multiplied by
                the size of an array element. For example, a block of 500x1000x500 doubles
                has a length of 250 million elements and a size of 2 Gb (each double occupies 8
                bytes of memory).

                The automatic block size is set to 100 Mb at package startup and can be reset
                anytime to this value by calling setAutoBlockSize() with no argument.

type   A string specifying the type of the array data.

shape   A string specifying the automatic block shape. See [makeCappedVolumeBox](#) for
                a description of the supported shapes.

                The automatic block shape is set to "hypercube" at package startup and can be
                reset anytime to this value by calling setAutoBlockShape() with no argument.

Details

By default, primary block processing functions [blockApply](#)() and [blockReduce](#)() use the grid
returned by blockGrid(x) to process array-like object x block by block. This can be changed with
[setAutoGridMaker](#)(). See ?[setAutoGridMaker](#) for more information.

Value

blockGrid: An [ArrayGrid](#) object on reference array x. The grid elements define the blocks that
will be used to process x by block. The grid is *optimal* in the sense that:

1. It's *compatible* with the grid of physical chunks a.k.a. *chunk grid*. This means that, when
   the chunk grid is known (i.e. when [chunkGrid](#)(x) is not NULL or chunk.grid is supplied),
   every block in the grid contains one or more *full* chunks. In other words, chunks never cross
   block boundaries.

2. Its *resolution* is such that the blocks have a length that is as close as possibe to (but does not
   exceed) block.length. An exception is made when some chunks already have a length that
   is >= block.length, in which case the returned grid is the same as the chunk grid.

Note that the returned grid is regular (i.e. is a [RegularArrayGrid](#) object) unless the chunk grid is not
regular (i.e. is an [ArbitraryArrayGrid](#) object).

rowGrid: A [RegularArrayGrid](#) object on reference array x where the grid elements define blocks
made of full rows of x.

colGrid: A [RegularArrayGrid](#) object on reference array x where the grid elements define blocks
made of full columns of x.

getAutoBlockSize: The current automatic block size in bytes as a single numeric value.

setAutoBlockSize: The new automatic block size in bytes as an invisible single numeric value.

getAutoBlockLength: The automatic block length as a single integer value.

getAutoBlockShape: The current automatic block shape as a single string.

setAutoBlockShape: The new automatic block shape as an invisible single string.

See Also

- [blockApply](#) and family to process an array-like object block by block.
- [ArrayGrid](#) objects.
- The [makeCappedVolumeBox](#) utility to make *capped volume boxes*.
- [chunkGrid](#).

- Advanced users: RealizationSink objects for writing an array-like object block by block to disk (or to memory).

Examples

```
## ------------------------------------------------------------------------
## A VERSION OF sum() THAT USES BLOCK PROCESSING
## ------------------------------------------------------------------------

block_sum <- function(a, grid)
{
    sums <- lapply(grid, function(viewport) sum(read_block(a, viewport)))
    sum(unlist(sums))
}

## On an ordinary matrix:
m <- matrix(runif(600), ncol=12)
m_grid <- blockGrid(m, block.length=120)
sum1 <- block_sum(m, m_grid)
sum1

## On a DelayedArray object:
library(HDF5Array)
M <- as(m, "HDF5Array")
sum2 <- block_sum(M, m_grid)
sum2

sum3 <- block_sum(M, colGrid(M, block.length=120))
sum3

sum4 <- block_sum(M, rowGrid(M, block.length=80))
sum4

## Sanity checks:
sum0 <- sum(m)
stopifnot(identical(sum1, sum0))
stopifnot(identical(sum2, sum0))
stopifnot(identical(sum3, sum0))
stopifnot(identical(sum4, sum0))

## ------------------------------------------------------------------------
## blockGrid()
## ------------------------------------------------------------------------
grid <- blockGrid(m, block.length=120)
grid
as.list(grid)  # turn the grid into a list of ArrayViewport objects
table(lengths(grid))
stopifnot(maxlength(grid) <= 120)

grid <- blockGrid(m, block.length=120,
                     block.shape="first-dim-grows-first")
grid
table(lengths(grid))
stopifnot(maxlength(grid) <= 120)

grid <- blockGrid(m, block.length=120,
                     block.shape="last-dim-grows-first")
```

```
grid
table(lengths(grid))
stopifnot(maxlength(grid) <= 120)

blockGrid(m, block.length=100)
blockGrid(m, block.length=75)
blockGrid(m, block.length=25)
blockGrid(m, block.length=20)
blockGrid(m, block.length=10)

## ---------------------------------------------------------------------
## rowGrid() AND colGrid()
## ---------------------------------------------------------------------
rowGrid(m, nrow=10)  # 5 blocks of 10 rows each
rowGrid(m, nrow=15)  # 3 blocks of 15 rows each plus 1 block of 5 rows
colGrid(m, ncol=5)   # 2 blocks of 5 cols each plus 1 block of 2 cols

## See ?RealizationSink for an advanced example of user-implemented
## block processing using colGrid() and a realization sink.

## ---------------------------------------------------------------------
## CONTROL THE DEFAULT BLOCK SIZE (OR LENGTH) AND SHAPE
## ---------------------------------------------------------------------
getAutoBlockSize()

getAutoBlockLength("double")
getAutoBlockLength("integer")
getAutoBlockLength("logical")
getAutoBlockLength("raw")

setAutoBlockSize(140)
getAutoBlockLength(type(m))
blockGrid(m)
lengths(blockGrid(m))
dims(blockGrid(m))

getAutoBlockShape()
setAutoBlockShape("scale")
blockGrid(m)
lengths(blockGrid(m))
dims(blockGrid(m))

## Reset automatic block size and shape to factory settings:
setAutoBlockSize()
setAutoBlockShape()
```

---

block_processing   *Block processing of an array-like object*

Description

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

Usage

```
blockApply(x, FUN, ..., grid=NULL, BPPARAM=getAutoBPPARAM())
blockReduce(FUN, x, init, BREAKIF=NULL, grid=NULL)

effectiveGrid(block)
currentBlockId(block)
currentViewport(block)

getAutoGridMaker()
setAutoGridMaker(GRIDMAKER="blockGrid")
getAutoBPPARAM()
setAutoBPPARAM(BPPARAM=NULL)
```

Arguments

x	An array-like object.
FUN	Coming soon...
...	Coming soon...
grid	Coming soon...
BPPARAM	Coming soon...
init	Coming soon...
BREAKIF	Coming soon...
block	Coming soon...
GRIDMAKER	The function to use as automatic grid maker, that is, the function that will be used by blockApply() and blockReduce() to make a grid when no grid is supplied via their grid argument. The function will be called on array-like object x and must return an ArrayGrid object, say grid, that is compatible with x i.e. such that refdim(grid) is identical to dim(x).
	GRIDMAKER can be specified as a function or as a single string naming a function. It can be a user-defined function or a pre-defined grid maker like blockGrid, rowGrid, or colGrid.
	The automatic grid maker is set to blockGrid at package startup and can be reset anytime to this value by calling setAutoGridMaker() with no argument.

Details

Coming soon...

See Also

- blockGrid to define grids to use in the context of block processing of array-like objects.

- MulticoreParam, SnowParam, and bpparam, from the **BiocParallel** package.

- read_block.

- DelayedArray objects.

- Advanced users: RealizationSink objects for writing an array-like object block by block to disk (or to memory).

Examples

```
## ---------------------------------------------------------------------
## blockApply()
## ---------------------------------------------------------------------

## Coming soon...

## ---------------------------------------------------------------------
## blockReduce()
## ---------------------------------------------------------------------

## Coming soon...

## ---------------------------------------------------------------------
## CONTROL THE DEFAULT GRID MAKER
## ---------------------------------------------------------------------
getAutoGridMaker()
setAutoGridMaker(function(x) colGrid(x, ncol=5))
getAutoGridMaker()

m <- matrix(runif(600), ncol=12)
blockApply(m, currentViewport)

## Reset automatic grid maker to factory settings:
setAutoGridMaker()
```

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
type(x)
```

Arguments

seed   An array-like object.

x   Typically a DelayedArray object. More generally type() is expected to work
                on any array-like object (that is, any object for which dim(x) is not NULL), or
                any ordinary vector (i.e. atomic or non-atomic).

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 type(), nseed(), seed(), and path().

type() is the DelayedArray equivalent of typeof() (or storage.mode()) for ordinary arrays and vectors. Note that, for convenience and consistency, type() also supports ordinary arrays and vectors. It should also support 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. One notable exception to this is when drop=TRUE and the result has only one dimension, in which case it is returned as an ordinary vector (atomic or list). Note that NAs in the subscripts are not supported.

- *Linear single bracket subsetting* (a.k.a. 1D-style subsetting, that is, subsetting of the form x[i]) only works if the 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 (block processing is triggered).

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][[1]].

Subassignment to a DelayedArray object with [<- is also supported like with an ordinary array, but with the following restrictions:

- *Multi-dimensional subassignment* (i.e. subassignment of the form x[i_1,i_2,...,i_n] <-value with one (possibly missing) subscript per dimension) only accepts a replacement value (a.k.a. right value) that is an array-like object (e.g. ordinary array, dgCMatrix object, DelayedArray object, etc...) or an ordinary vector (atomic or list) of length 1.

- *Linear subassignment* (a.k.a. 1D-style subassignment, that is, subassignment of the form x[i] <-value) only works if the subscript i is a logical DelayedArray object of the same dimensions as x and if the replacement value is an ordinary vector (atomic or list) of length 1.

- *Filling with a vector*, that is, subassignment of the form x[] <-v where v is an ordinary vector (atomic or list), is only supported if the length of the vector is a divisor of nrow(x).

These 3 forms of subassignment are implemented as delayed operations so are very light.

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

See Also

- showtree for DelayedArray accessors nseed, seed, and path.
- realize for realizing a DelayedArray object in memory or on disk.
- block_processing for more information about block processing of an array-like object.
- DelayedArray-utils for common operations on DelayedArray objects.
- DelayedMatrix-utils for common operations on DelayedMatrix objects.
- DelayedArray-stats for statistical functions on DelayedArray objects.
- DelayedMatrix-stats for DelayedMatrix row/col summarization.
- 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 DelayedArray object is **always** treated as a
## "read-only" object so will never be modified by the operations
## we perform on A:
stopifnot(identical(a, seed(A)))
type(A)

## Multi-dimensional single bracket subsetting:
m <- a[11:20 , 5, -3]  # an ordinary matrix
M <- A[11:20 , 5, -3]  # a DelayedMatrix object
stopifnot(identical(m, as.array(M)))

## Linear single bracket subsetting:
A[11:20]
A[A <= 1e-5]
stopifnot(identical(a[a <= 1e-5], A[A <= 1e-5]))

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

A[2:5, 1:2, ] <- array(1:40, c(4, 2, 5))
a[2:5, 1:2, ] <- array(1:40, c(4, 2, 5))
stopifnot(identical(a, as.array(A)))

## Other operations:
crazy <- function(x) (5 * x[ , , 1] ^ 3 + 1L) * log(x[, , 2])
```

```
b <- crazy(a)
head(b)

B <- crazy(A)  # very fast! (all 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. AN 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 <- crazy(A3)            # very fast! (all 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. 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:
library(HDF5Array)
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)


## ----------------------------------------------------------------------
## F. 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 block processing in action:
DelayedArray:::set_verbose_block_processing(TRUE)
## and check the automatic block size:
getAutoBlockSize()

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

## Increasing the block size increases the speed but also memory usage:
setAutoBlockSize(2e8)
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))

setAutoBlockSize()

## 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.
- DelayedMatrix-stats for DelayedMatrix row/col summarization.
- 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:

- rbind and cbind
- 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
- 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`
- unique, `table`
- all the members of the `Summary` group
- mean
- apply

See Also

- `cbind` in the **base** package for rbind/cbind'ing ordinary arrays.
- `arbind` and `acbind` in this package (**DelayedArray**) for binding ordinary arrays of arbitrary dimensions along their rows or columns.
- `is.na`, `!`, `table`, `mean`, `apply`, and `%*%` in the **base** package for the corresponding operations on ordinary arrays or matrices.
- DelayedMatrix-utils for common operations on DelayedMatrix objects.
- DelayedArray-stats for statistical functions on DelayedArray objects.
- DelayedMatrix-stats for DelayedMatrix row/col summarization.
- 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

```
## ---------------------------------------------------------------------
## BIND DelayedArray OBJECTS
## ---------------------------------------------------------------------
## DelayedArray objects can be bound along their 1st (rows) or 2nd
## (columns) dimension with rbind() or cbind(). These operations are
## equivalent to arbind() and acbind(), respectively, and are all
## delayed.

## On 2D objects:
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))        # delayed
M12
colMeans(M12)                  # block-processed

## On objects with more than 2 dimensions:
example(arbind)  # to create arrays a1, a2, a3

A1 <- DelayedArray(a1)
A2 <- DelayedArray(a2)
A3 <- DelayedArray(a3)
A123 <- rbind(A1, A2, A3)      # delayed
```

```
    A123

    ## On 1D objects:
    v1 <- array(11:15, 5, dimnames=list(LETTERS[1:5]))
    v2 <- array(letters[1:3])
    V1 <- DelayedArray(v1)
    V2 <- DelayedArray(v2)
    V12 <- rbind(V1, V2)
    V12

    ## Not run: cbind(V1, V2)  # Error! (the objects to cbind() must have at least 2
                    # dimensions)

    ## End(Not run)

    ## Note that base::rbind() and base::cbind() do something completely
    ## different on ordinary arrays that are not matrices. They treat them
    ## as if they were vectors:
    rbind(a1, a2, a3)
    cbind(a1, a2, a3)
    rbind(v1, v2)
    cbind(v1, v2)

    ## Also note that DelayedArray objects of arbitrary dimensions can be
    ## stored inside a DataFrame object as long as they all have the same
    ## first dimension (nrow()):
    DF <- DataFrame(M=I(tail(M1, n=5)), A=I(A3), V=I(V1))
    DF[-3, ]
    DF2 <- rbind(DF, DF)
    DF2$V

    ## Sanity checks:
    m1 <- as.matrix(M1)
    m2 <- as.matrix(M2)
    stopifnot(identical(rbind(m1, t(m2)), as.matrix(M12)))
    stopifnot(identical(arbind(a1, a2, a3), as.array(A123)))
    stopifnot(identical(arbind(v1, v2), as.array(V12)))
    stopifnot(identical(rbind(DF$M, DF$M), DF2$M))
    stopifnot(identical(rbind(DF$A, DF$A), DF2$A))
    stopifnot(identical(rbind(DF$V, DF$V), DF2$V))

    ## ---------------------------------------------------------------------
    ## MORE OPERATIONS
    ## ---------------------------------------------------------------------

    M1 >= 0.5 & M1 < 0.75        # delayed
    log(M1)                      # delayed
    pmax2(M2, 0)                 # delayed

    ## table() is block-processed:
    a4 <- array(sample(50L, 2000000L, replace=TRUE), c(200, 4, 2500))
    A4 <- as(a4, "HDF5Array")
    table(A4)
    a5 <- array(sample(20L, 2000000L, replace=TRUE), c(200, 4, 2500))
    A5 <- as(a5, "HDF5Array")
    table(A5)
```

```
A4 - 2 * A5                      # delayed
table(A4 - 2 * A5)              # block-processed

## range() is block-processed:
range(A4 - 2 * A5)
range(M1)

cmeans <- colMeans(M2)         # block-processed
sweep(M2, 2, cmeans)           # delayed
```

DelayedMatrix-stats   *DelayedMatrix row/col summarization*

Description

Only a small number of row/col summarization methods are provided by the **DelayedArray** package.

See the **DelayedMatrixStats** package for an extensive set of row/col summarization methods.

Usage

```
## S4 method for signature 'DelayedMatrix'
rowSums(x, na.rm=FALSE, dims=1)
## S4 method for signature 'DelayedMatrix'
colSums(x, na.rm=FALSE, dims=1)

## S4 method for signature 'DelayedMatrix'
rowMeans(x, na.rm=FALSE, dims=1)
## S4 method for signature 'DelayedMatrix'
colMeans(x, na.rm=FALSE, dims=1)

## S4 method for signature 'DelayedMatrix'
rowMaxs(x, rows=NULL, cols=NULL, na.rm=FALSE, dim.=dim(x))
## S4 method for signature 'DelayedMatrix'
colMaxs(x, rows=NULL, cols=NULL, na.rm=FALSE, dim.=dim(x))

## S4 method for signature 'DelayedMatrix'
rowMins(x, rows=NULL, cols=NULL, na.rm=FALSE, dim.=dim(x))
## S4 method for signature 'DelayedMatrix'
colMins(x, rows=NULL, cols=NULL, na.rm=FALSE, dim.=dim(x))

## S4 method for signature 'DelayedMatrix'
rowRanges(x, rows=NULL, cols=NULL, na.rm=FALSE, dim.=dim(x))
## S4 method for signature 'DelayedMatrix'
colRanges(x, rows=NULL, cols=NULL, na.rm=FALSE, dim.=dim(x))
```

Arguments

x   A DelayedMatrix object.

na.rm   Should missing values (including NaN) be omitted from the calculations?

dims, rows, cols, dim.

                     These arguments are not supported and should not be used.

Details

All these operations are block-processed.

See Also

- The **DelayedMatrixStats** package for more row/col summarization methods for DelayedMatrix objects.

- rowSums in the **base** package and rowMaxs in the **matrixStats** package for row/col summarization of an ordinary matrix.

- DelayedMatrix-utils for other common operations on DelayedMatrix objects.

- DelayedMatrix objects.

- matrix 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")

M12 <- rbind(M1, t(M2))        # delayed

## All these operations are block-processed.

rowSums(M12)
colSums(M12)

rowMeans(M12)
colMeans(M12)

rmaxs <- rowMaxs(M12)
cmaxs <- colMaxs(M12)

rmins <- rowMins(M12)
cmins <- colMins(M12)

rranges <- rowRanges(M12)
cranges <- colRanges(M12)

## Sanity checks:
m12 <- rbind(as.matrix(M1), t(as.matrix(M2)))
stopifnot(identical(rowSums(M12), rowSums(m12)))
stopifnot(identical(colSums(M12), colSums(m12)))
stopifnot(identical(rowMeans(M12), rowMeans(m12)))
stopifnot(identical(colMeans(M12), colMeans(m12)))
stopifnot(identical(rmaxs, rowMaxs(m12)))
stopifnot(identical(cmaxs, colMaxs(m12)))
stopifnot(identical(rmins, rowMins(m12)))
stopifnot(identical(cmins, colMins(m12)))
stopifnot(identical(rranges, cbind(rmins, rmaxs, deparse.level=0)))
stopifnot(identical(cranges, cbind(cmins, cmaxs, deparse.level=0)))
```

DelayedMatrix-utils   *Common operations on DelayedMatrix objects*

Description

Common operations on DelayedMatrix objects.

Details

In addition to the operations supported on DelayedArray objects, DelayedMatrix objects support the following operations:

Delayed operations:

- `t`

Block-processed operations:

- `rowsum` and `colsum`
- matrix multiplication (%*%) of an ordinary matrix by a DelayedMatrix object
- matrix row/col summarization (see ?`DelayedMatrix-stats`)

See Also

- rowsum in the **base** package for computing column sums across rows of an ordinary matrix for each level of a grouping variable.
- DelayedArray-utils for common operations on DelayedArray objects.
- DelayedArray-stats for statistical functions on DelayedArray objects.
- DelayedMatrix-stats for DelayedMatrix row/col summarization.
- 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.
- HDF5Array objects in the **HDF5Array** package.
- array objects in base R.

Examples

```
## ---------------------------------------------------------------------
## rowsum() / colsum()
## ---------------------------------------------------------------------
library(HDF5Array)
set.seed(123)
m0 <- matrix(runif(14400000), ncol=2250,
             dimnames=list(NULL, sprintf("C%04d", 1:2250)))
M0 <- writeHDF5Array(m0, chunkdim=c(200, 250))
dimnames(M0) <- dimnames(m0)

## --- rowsum() ---

group <- sample(90, nrow(M0), replace=TRUE)  # define groups of rows
```

```
rs <- rowsum(M0, group)
rs[1:5, 1:8]
rs2 <- rowsum(M0, group, reorder=FALSE)
rs2[1:5, 1:8]

## Let's see block processing in action:
DelayedArray:::set_verbose_block_processing(TRUE)
setAutoBlockSize(2e6)
rs3 <- rowsum(M0, group)
setAutoBlockSize()
DelayedArray:::set_verbose_block_processing(FALSE)

## Sanity checks:
stopifnot(all.equal(rowsum(m0, group), rs))
stopifnot(all.equal(rowsum(m0, group, reorder=FALSE), rs2))
stopifnot(all.equal(rs, rs3))

## --- colsum() ---

group <- sample(30, ncol(M0), replace=TRUE)  # define groups of cols
cs <- colsum(M0, group)
cs[1:5, 1:7]
cs2 <- colsum(M0, group, reorder=FALSE)
cs2[1:5, 1:7]

## Sanity checks:
stopifnot(all.equal(colsum(m0, group), cs))
stopifnot(all.equal(cs, t(rowsum(t(m0), group))))
stopifnot(all.equal(cs, t(rowsum(t(M0), group))))
stopifnot(all.equal(colsum(m0, group, reorder=FALSE), cs2))
stopifnot(all.equal(cs2, t(rowsum(t(m0), group, reorder=FALSE))))
stopifnot(all.equal(cs2, t(rowsum(t(M0), group, reorder=FALSE))))

## ---------------------------------------------------------------------
## MATRIX MULTIPLICATION
## ---------------------------------------------------------------------
library(HDF5Array)
toy_h5 <- system.file("extdata", "toy.h5", package="HDF5Array")
h5ls(toy_h5)
M1 <- HDF5Array(toy_h5, "M1")

## 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(dim(P1), dim(P2)),
          all.equal(as.array(P1), as.array(P2)))
```

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, inspect, and simplify this tree.

Usage

```
is_noop(x)
```

Arguments

x   A DelayedSubset, DelayedAperm, or DelayedDimnames object.

Details

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

```
Node type                        Represented operation
------------------------------------------------------------------
DelayedOp (VIRTUAL)
------------------------------------------------------------------
* DelayedUnaryOp (VIRTUAL)
  o DelayedSubset                Multi-dimensional single bracket
                                 subsetting.
  o DelayedAperm                 Extended aperm() (can drop and/or
                                 add ineffective dimensions).
  o DelayedUnaryIsoOp (VIRTUAL)  Unary op that preserves the
```

```
                                geometry.
        - DelayedUnaryIsoOpStack      Simple ops stacked together.
        - DelayedUnaryIsoOpWithArgs   One op with vector-like arguments
                                      along the dimensions of the input.
        - DelayedSubassign            Multi-dimensional single bracket
                                      subassignment.
        - DelayedDimnames             Set/replace the dimnames.
      ----------------------------------------------------------------
   * DelayedNaryOp (VIRTUAL)
     o DelayedNaryIsoOp             N-ary op that preserves the
                                    geometry.
     o DelayedAbind                 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*.

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

Note

The DelayedOp virtual class and its 8 concrete subclasses are for internal use only and never exposed to the end user.

See Also

- DelayedArray objects.

- showtree to visualize and access the leaves of a tree of delayed operations carried by a DelayedArray object.

- simplify to simplify the tree of delayed operations carried by a DelayedArray object.

- extract_array.

---

linearInd   *Converting array indices into linear indices*

Description

linearInd performs the reverse conversion of base::arrayInd, that is, it converts so-called *array indices* (i.e. n-uplets) into *linear indices*.

Usage

```
linearInd(aind, dim)
```

Arguments

aind   Typically a numeric matrix like one returned by base::arrayInd, that is, a
                matrix where each row is an n-uplet representing an array index. Each array
                index must describe a position relative to the implicit array i.e. to the array
                whose dimensions are specified via the dim argument.

                For convenience, aind can also be specified as a vector with one element per
                dimension in the implicit array, in which case it will be treated like a 1-row
                matrix.

                Note that no bounds checking is performed, that is, values in the j-th column of
                aind can be < 1 or > dim[j].

dim   An integer vector containing the dimensions of the underlying array.

                Note that dim can also be an integer matrix, in which case it must have the same
                shape as aind, that is, 1 row per row in aind and 1 column per dimension.

Value

An integer vector with one element per row in aind if aind is a matrix.

A single integer if aind is a vector.

See Also

arrayInd in the **base** package for the reverse conversion.

Examples

```
dim <- 4:2
linearInd(c(4, 3, 1), dim)
linearInd(c(4, 3, 2), dim)

aind <- rbind(c(1, 1, 1),
              c(2, 1, 1),
              c(3, 1, 1),
              c(4, 1, 1),
              c(1, 2, 1),
              c(1, 1, 2),
              c(4, 3, 2))

linearInd(aind, dim)

## With a matrix of dimensions:

dims <- rbind(c(4L, 3L),
              c(5L, 3L),
              c(6L, 3L))

aind <- rbind(c(1,  2),
              c(1,  2),
              c(1,  2))

linearInd(aind, dims)

## Sanity checks:

dim <- c(33:30, 45L, 30L)
```

```
stopifnot(linearInd(rep(1, 6), dim) == 1)
stopifnot(linearInd(dim, dim) == prod(dim))

stopifnot(identical(linearInd(arrayInd(1:120, 6:4), 6:4), 1:120))
stopifnot(identical(linearInd(arrayInd(840:1, 4:7), 4:7), 840:1))
```

---

makeCappedVolumeBox   *Utilities to make capped volume boxes*

Description

makeCappedVolumeBox returns the dimensions of the biggest multidimensional box (a.k.a. hyper-rectangle) that satisfies 3 constraints: (1) its volume is capped, (2) it fits in the *constraining box*, (3) it has the specified shape.

makeRegularArrayGridOfCappedLengthViewports makes a [RegularArrayGrid](#) object with grid elements that are capped volume boxes with the specified constraints.

These are low-level utilities used internally to support [blockGrid](#) and family.

Usage

```
makeCappedVolumeBox(maxvol, maxdim, shape=c("hypercube",
                                            "scale",
                                            "first-dim-grows-first",
                                            "last-dim-grows-first"))

makeRegularArrayGridOfCappedLengthViewports(refdim,
                          viewport_len,
                          viewport_shape=c("hypercube",
                                           "scale",
                                           "first-dim-grows-first",
                                           "last-dim-grows-first"))
```

Arguments

maxvol	The maximum volume of the box to return.
maxdim	The dimensions of the constraining box.
shape	The shape of the box to return.
refdim	The dimensions of the reference array of the grid to return.
viewport_len	The maximum length of the elements (a.k.a. viewports) of the grid to return.
viewport_shape	The shape of the elements (a.k.a. viewports) of the grid to return.

Details

makeCappedVolumeBox returns the dimensions of a box that satisfies the following constraints:

1. The volume of the box is as close as possibe to (but no bigger than) maxvol.

2. The box fits in the *constraining box* i.e. in the box whose dimensions are specified via maxdim.

3. The box has a non-zero volume if the *constraining box* has a non-zero volume.

4. The shape of the box is as close as possible to the requested shape.

The supported shapes are:

- hypercube: The box should be as close as possible to an *hypercube* (a.k.a. *n-cube*), that is, the ratio between its biggest and smallest dimensions should be as close as possible to 1.

- scale: The box should have the same proportions as the *constraining box*.

- first-dim-grows-first: The box will be grown along its 1st dimension first, then along its 2nd dimension, etc...

- last-dim-grows-first: Like first-dim-grows-first but starting along the last dimension.

See Also

- [blockGrid](#) to define grids to use in the context of block processing of array-like objects.

- [ArrayGrid](#) objects.

Examples

```
## ---------------------------------------------------------------------
## makeCappedVolumeBox()
## ---------------------------------------------------------------------

maxdim <- c(50, 12)  # dimensions of the "constraining box"

## "hypercube" shape:
makeCappedVolumeBox(40, maxdim)
makeCappedVolumeBox(120, maxdim)
makeCappedVolumeBox(125, maxdim)
makeCappedVolumeBox(200, maxdim)

## "scale" shape:
makeCappedVolumeBox(40, maxdim, shape="scale")
makeCappedVolumeBox(160, maxdim, shape="scale")

## "first-dim-grows-first" and "last-dim-grows-first" shapes:
makeCappedVolumeBox(120, maxdim, shape="first-dim-grows-first")
makeCappedVolumeBox(149, maxdim, shape="first-dim-grows-first")
makeCappedVolumeBox(150, maxdim, shape="first-dim-grows-first")

makeCappedVolumeBox(40, maxdim, shape="last-dim-grows-first")
makeCappedVolumeBox(59, maxdim, shape="last-dim-grows-first")
makeCappedVolumeBox(60, maxdim, shape="last-dim-grows-first")

## ---------------------------------------------------------------------
## makeRegularArrayGridOfCappedLengthViewports()
## ---------------------------------------------------------------------

grid1a <- makeRegularArrayGridOfCappedLengthViewports(maxdim, 40)
grid1a
as.list(grid1a)  # turn the grid into a list of ArrayViewport objects
table(lengths(grid1a))
stopifnot(maxlength(grid1a) <= 40)  # sanity check

grid1b <- makeRegularArrayGridOfCappedLengthViewports(maxdim, 40,
                                            "first-dim-grows-first")
grid1b
```

```
as.list(grid1b)  # turn the grid into a list of ArrayViewport objects
table(lengths(grid1b))
stopifnot(maxlength(grid1b) <= 40)  # sanity check

grid2a <- makeRegularArrayGridOfCappedLengthViewports(maxdim, 120)
grid2a
as.list(grid2a)  # turn the grid into a list of ArrayViewport objects
table(lengths(grid2a))
stopifnot(maxlength(grid2a) <= 120)  # sanity check

grid2b <- makeRegularArrayGridOfCappedLengthViewports(maxdim, 120,
                                        "first-dim-grows-first")
grid2b
as.list(grid2b)  # turn the grid into a list of ArrayViewport objects
table(lengths(grid2b))
stopifnot(maxlength(grid2b) <= 120)  # sanity check

grid3a <- makeRegularArrayGridOfCappedLengthViewports(maxdim, 200)
grid3a
as.list(grid3a)  # turn the grid into a list of ArrayViewport objects
table(lengths(grid3a))
stopifnot(maxlength(grid3a) <= 200)  # sanity check

grid3b <- makeRegularArrayGridOfCappedLengthViewports(maxdim, 200,
                                        "first-dim-grows-first")
grid3b
as.list(grid3b)  # turn the grid into a list of ArrayViewport objects
table(lengths(grid3b))
stopifnot(maxlength(grid3b) <= 200)  # sanity check
```

---

read_block   *Read/write blocks of array data*

Description

2 utilities for reading/writing blocks from/to an array-like object.

Usage

```
read_block(x, viewport)
write_block(x, viewport, block)
```

Arguments

x	An array-like object.
viewport	An ArrayViewport object.
block	An ordinary array of the same dimensions as viewport.

See Also

- ArrayViewport objects.
- blockGrid to define grids to use in the context of block processing of array-like objects.
- block_processing for more information about block processing of an array-like object.

- [DelayedArray](#) objects.
- [array](#) objects in base R.

Examples

```
m0 <- matrix(1:30, ncol=5)

block_dim <- c(4, 3)
viewport1 <- ArrayViewport(dim(m0), IRanges(c(3, 2), width=block_dim))
viewport1

block1 <- read_block(m0, viewport1)
block1

## No-op:
write_block(m0, viewport1, block1)
stopifnot(identical(m0, write_block(m0, viewport1, block1)))

write_block(m0, viewport1, block1 + 100L)

viewport2 <- ArrayViewport(dim(m0), IRanges(c(1, 3), width=block_dim))
write_block(m0, viewport2, block1 + 100L)

## Using a grid:
grid0 <- RegularArrayGrid(dim(m0), spacings=c(3L, 2L))
grid0
length(grid0)  # number of blocks defined by the grid
read_block(m0, grid0[[3L]])  # read 3rd block
read_block(m0, grid0[[1L, 3L]])

## Walk on the grid, colum by column:
m1 <- m0
for (b in seq_along(grid0)) {
    viewport <- grid0[[b]]
    block <- read_block(m1, viewport)
    block <- b * 1000L + block
    m1 <- write_block(m1, viewport, block)
}
m1

## Walk on the grid, row by row:
m2 <- m0
for (i in seq_len(dim(grid0)[[1]])) {
  for (j in seq_len(dim(grid0)[[2]])) {
    viewport <- grid0[[i, j]]
    block <- read_block(m2, viewport)
    block <- (i * 10L + j) * 1000L + block
    m2 <- write_block(m2, viewport, block)
  }
}
m2
```

RealizationSink   *RealizationSink objects*

Description

Get or set the *realization backend* for the current session with getRealizationBackend or setRealizationBackend.

Advanced users: Create a RealizationSink object with the backend-agnostic RealizationSink() constructor. Use this object to write an array-like object block by block to disk (or to memory).

Usage

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

RealizationSink(dim, dimnames=NULL, type="double")
```

Arguments

BACKEND	NULL (the default), or a single string specifying the name of a *realization backend*.
dim	The dimensions (specified as an integer vector) of the RealizationSink object to create.
dimnames	The dimnames (specified as a list of character vectors or NULLs) of the RealizationSink object to create.
type	The type of the data that will be written to the RealizationSink object to create.

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 as an [HDF5Array](#) object if set to "HDF5Array".

Value

supportedRealizationBackends: A data frame with 1 row per supported *realization backend*.

getRealizationBackend: NULL or a single string specifying the name of the *realization backend* currently in use.

RealizationSink: A RealizationSink object for the current *realization backend*.

See Also

- [write_block](#).
- [blockGrid](#) to define grids to use in the context of block processing of array-like objects.
- [DelayedArray](#) objects.
- [RleArray](#) objects.
- [HDF5Array](#) objects in the **HDF5Array** package.
- [HDF5-dump-management](#) in the **HDF5Array** package to control the location and physical properties of automatically created HDF5 datasets.
- [array](#) objects in base R.

Examples

```
## ---------------------------------------------------------------------
## A. supportedRealizationBackends() AND FAMILY
## ---------------------------------------------------------------------
supportedRealizationBackends()
getRealizationBackend()  # backend is set to NULL

setRealizationBackend("HDF5Array")
supportedRealizationBackends()
getRealizationBackend()  # backend is set to "HDF5Array"

## ---------------------------------------------------------------------
## B. A SIMPLE (AND VERY ARTIFICIAL) RealizationSink() EXAMPLE
## ---------------------------------------------------------------------
getHDF5DumpChunkLength()
setHDF5DumpChunkLength(500L)
getHDF5DumpChunkShape()

sink <- RealizationSink(c(120L, 50L))
dim(sink)
chunkdim(sink)

grid <- blockGrid(sink, block.length=600)
for (b in seq_along(grid)) {
    viewport <- grid[[b]]
    block <- 101 * b + runif(length(viewport))
    dim(block) <- dim(viewport)
    write_block(sink, viewport, block)
}

## Always close the RealizationSink object when you are done writing to
## it and before coercing it to DelayedArray:
close(sink)
A <- as(sink, "DelayedArray")
A

## ---------------------------------------------------------------------
## C. AN ADVANCED EXAMPLE OF USER-IMPLEMENTED BLOCK PROCESSING USING
##    colGrid() AND A REALIZATION SINK
## ---------------------------------------------------------------------
## Say we have 2 matrices with the same number of columns. Each column
## represents a biological sample:
library(HDF5Array)
R <- as(matrix(runif(75000), ncol=1000), "HDF5Array")   # 75 rows
G <- as(matrix(runif(250000), ncol=1000), "HDF5Array")  # 250 rows

## Say we want to compute the matrix U obtained by applying the same
## binary functions FUN() to all samples i.e. U is defined as:
##
##   U[ , j] <- FUN(R[ , j], G[ , j]) for 1 <= j <= 1000
##
## Note that FUN() should return a vector of constant length, say 200,
## so U will be a 200x1000 matrix. A naive implementation would be:
##
##   pFUN <- function(r, g) {
##       stopifnot(ncol(r) == ncol(g))  # sanity check
```

```
##          sapply(seq_len(ncol(r)), function(j) FUN(r[ , j], g[ , j]))
##    }
##
## But because U is going to be too big to fit in memory, we can't
## just do pFUN(R, G). So we want to compute U block by block and
## write the blocks to disk as we go. The blocks will be made of full
## columns. Also since we need to walk on 2 matrices at the same time
## (R and G), we can't use blockApply() or blockReduce() so we'll use
## a "for" loop.

## Before we can write the "for" loop, we need 4 things:

## 1) Two grids of blocks, one on R and one on G. The blocks in the
##    2 grids must contain the same number of columns. We arbitrarily
##    choose to use blocks of 150 columns:
R_grid <- colGrid(R, ncol=150)
G_grid <- colGrid(G, ncol=150)

## 2) The function pFUN(). It will take 2 blocks as input, 1 from R
##    and 1 from G, apply FUN() to all the samples in the blocks,
##    and return a matrix with one columns per sample:
pFUN <- function(r, g) {
    stopifnot(ncol(r) == ncol(g))  # sanity check
    ## Return a matrix with 200 rows with random values. Completely
    ## artificial sorry. A realistic example would actually need to
    ## apply the same binary function to r[ ,j] and g[ , j] for
    ## 1 <= j <= ncol(r).
    matrix(runif(200 * ncol(r)), nrow=200)
}

## 3) A RealizationSink object where to write the matrices returned
##    by pFUN() as we go. Note that instead of creating a realization
##    sink by calling a backend-specific sink constructor (e.g.
##    HDF5Array:::HDF5RealizationSink), we use the backend-agnostic
##    constructor RealizationSink() and set the current realization
##    backend to HDF5:

setRealizationBackend("HDF5Array")
U_sink <- RealizationSink(c(200L, 1000L))

## 4) Finally, we create a grid on U_sink with blocks that contain the
##    same number of columns as the corresponding blocks in R and G:

U_grid <- colGrid(U_sink, ncol=150)

## Note that the 3 grids should have the same number of blocks:
stopifnot(length(U_grid) == length(R_grid))
stopifnot(length(U_grid) == length(G_grid))

## Now we can procede. We write a loop where we walk on R and G at
## the same time, block by block, apply pFUN(), and write the output
## of pFUN() to U_sink:
for (b in seq_along(U_grid)) {
    R_block <- read_block(R, R_grid[[b]])
    G_block <- read_block(G, G_grid[[b]])
    U_block <- pFUN(R_block, G_block)
    write_block(U_sink, U_grid[[b]], U_block)
```

```
}

close(U_sink)
U <- as(U_sink, "DelayedArray")

## A note about parallelization: even though concurrent block reading
## from the same object is supported, concurrent writing to a sink is
## not supported yet. So the above code cannot be parallelized at the
## moment.
```

---

realize   *Realize a DelayedArray object*

Description

Realize a [DelayedArray](#) object in memory or on disk.

Usage

```
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	A single string specifying the name of the *realization backend*. Use the current realization backend by default i.e. the backend returned by [getRealizationBackend](#)().

Value

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

- [getRealizationBackend](#) and [setRealizationBackend](#) for getting and setting the current *realization backend*.

- [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 a [DelayedArray](#) subclass for representing an in-memory Run Length Encoded array-like dataset.

All the operations available for [DelayedArray](#) objects work on RleArray objects.

Usage

```
## Constructor function:
RleArray(data, dim, dimnames, chunksize=NULL)
```

Arguments

data	An [Rle](#) object, or an ordinary list of Rle objects, or an [RleList](#) object, or a [DataFrame](#) object where all the columns are Rle objects. More generally speaking, data can be any list-like object where all the list elements are Rle objects.
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	The *dimnames* of the object to be created. Must be NULL or 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!

Value

An RleArray object.

See Also

- Rle and DataFrame objects in the **S4Vectors** package and RleList objects in the **IRanges** 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.

- The RleArraySeed helper class.

Examples

```
## ---------------------------------------------------------------------
## A. BASIC EXAMPLE
## ---------------------------------------------------------------------

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

A <- RleArray(data, 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])
m1 <- toto(a)
head(m1)

M1 <- toto(A)  # very fast! (operations are delayed)
M1

stopifnot(identical(m1, as.array(M1)))

cs <- colSums(m1)
CS <- colSums(M1)
stopifnot(identical(cs, CS))

## Coercing a DelayedMatrix object to DataFrame produces a DataFrame
## object with Rle columns:
as(M1, "DataFrame")

## ---------------------------------------------------------------------
## B. MAKING AN RleArray OBJECT FROM A LIST-LIKE OBJECT OF Rle OBJECTS
## ---------------------------------------------------------------------

## From a DataFrame object:
DF <- DataFrame(A=Rle(sample(3L, 100, replace=TRUE)),
                B=Rle(sample(3L, 100, replace=TRUE)),
                C=Rle(sample(3L, 100, replace=TRUE) - 0.5),
```

```
                        row.names=sprintf("ID%03d", 1:100))

    M2 <- RleArray(DF)
    M2

    A3 <- RleArray(DF, dim=c(25, 6, 2))
    A3

    M4 <- RleArray(DF, dim=c(25, 12), dimnames=list(LETTERS[1:25], NULL))
    M4

    ## From an ordinary list:
    ## If all the supplied Rle objects have the same length and if the 'dim'
    ## argument is not specified, then the RleArray() constructor returns an
    ## RleMatrix object with 1 column per Rle object. If the 'dimnames'
    ## argument is not specified, then the names on the list are propagated
    ## as the colnames of the returned object.
    data <- as.list(DF)
    M2b <- RleArray(data)
    A3b <- RleArray(data, dim=c(25, 6, 2))
    M4b <- RleArray(data, dim=c(25, 12), dimnames=list(LETTERS[1:25], NULL))

    data2 <- list(Rle(sample(3L, 9, replace=TRUE)) * 11L,
                  Rle(sample(3L, 15, replace=TRUE)))
    ## Not run:
      RleArray(data2)  # error! (cannot infer the dim)

    ## End(Not run)
    RleArray(data2, dim=c(4, 6))

    ## From an RleList object:
    data <- RleList(data)
    M2c <- RleArray(data)
    A3c <- RleArray(data, dim=c(25, 6, 2))
    M4c <- RleArray(data, dim=c(25, 12), dimnames=list(LETTERS[1:25], NULL))

    data2 <- RleList(data2)
    ## Not run:
      RleArray(data2)  # error! (cannot infer the dim)

    ## End(Not run)
    RleArray(data2, dim=4:2)

    ## Sanity checks:
    data0 <- as.vector(unlist(DF, use.names=FALSE))
    m2 <- matrix(data0, ncol=3, dimnames=dimnames(M2))
    stopifnot(identical(m2, as.matrix(M2)))
    rownames(m2) <- NULL
    stopifnot(identical(m2, as.matrix(M2b)))
    stopifnot(identical(m2, as.matrix(M2c)))
    a3 <- array(data0, dim=c(25, 6, 2))
    stopifnot(identical(a3, as.array(A3)))
    stopifnot(identical(a3, as.array(A3b)))
    stopifnot(identical(a3, as.array(A3c)))
    m4 <- matrix(data0, ncol=12, dimnames=dimnames(M4))
    stopifnot(identical(m4, as.matrix(M4)))
    stopifnot(identical(m4, as.matrix(M4b)))
```

```
  stopifnot(identical(m4, as.matrix(M4c)))

  ## ---------------------------------------------------------------------
  ## C. COERCING FROM RleList OR DataFrame TO RleMatrix
  ## ---------------------------------------------------------------------

  ## Coercing an RleList object to RleMatrix only works if all the list
  ## elements in the former have the same length.
  x <- RleList(A=Rle(sample(3L, 20, replace=TRUE)),
               B=Rle(sample(3L, 20, replace=TRUE)))
  M <- as(x, "RleMatrix")
  stopifnot(identical(x, as(M, "RleList")))

  x <- DataFrame(A=x[[1]], B=x[[2]], row.names=letters[1:20])
  M <- as(x, "RleMatrix")
  stopifnot(identical(x, as(M, "DataFrame")))

  ## ---------------------------------------------------------------------
  ## D. CONSTRUCTING A LARGE RleArray OBJECT
  ## ---------------------------------------------------------------------

  ## The RleArray() constructor does not accept a long Rle object at the
  ## moment:
  ## Not run:
    RleArray(Rle(5, 3e9), dim=c(3, 1e9))  # error!

  ## End(Not run)

  ## The workaround is to supply a list of Rle objects instead:
  data <- lapply(1:500, function(j) Rle(runif(99), 1e6 + 99:1))
  dim <- c(6750, 73337, 100)
  A <- RleArray(data, dim)
  A

  ## Because all the Rle objects in 'data' have the same length, we can
  ## call RleArray() on it without specifying the 'dim' argument. This
  ## returns an RleMatrix object where each column corresponds to an Rle
  ## object in 'data':
  M <- RleArray(data)
  M
  stopifnot(identical(as(data, "RleList"), as(M, "RleList")))
```

RleArraySeed-class   *RleArraySeed objects*

Description

RleArraySeed is a low-level helper class for representing an in-memory Run Length Encoded array-like dataset. RleArraySeed objects are not intended to be used directly. Most end-users should create and manipulate RleArray objects instead. See ?RleArray for more information.

Details

No operation can be performed directly on an RleArraySeed object. It first needs to be wrapped in a DelayedArray object. The result of this wrapping is an RleArray object (an RleArray object is just an RleArraySeed object wrapped in a DelayedArray object).

See Also

- RleArray objects.

- Rle objects in the **S4Vectors** package.

---

showtree   *Visualize and access the leaves of a tree of delayed operations*

Description

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

Use nseed, seed, or path to access the number of seeds, the seed, or the seed path of a DelayedArray object, respectively.

Use seedApply to apply a function to the seeds of a DelayedArray object.

Usage

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

nseed(x)              # seed counter
seed(x)               # seed getter and setter
path(object, ...)     # path getter and setter

seedApply(x, FUN, ...)
```

Arguments

x, object	Typically a DelayedArray object but can also be a DelayedOp object or 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.
FUN	The function to be applied to each leaf in x.
...	Optional arguments to FUN for seedApply().
	Additional arguments passed to methods for path().

Value

The number of seeds contained in x for nseed.

The seed contained in x for seed.

The path of the seed contained in object for path.

A list of length nseed(x) for seedApply.

See Also

- simplify to simplify the tree of delayed operations carried by a DelayedArray object.

- DelayedOp objects.

- DelayedArray objects.

Examples

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

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
seed(M2)

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)  # the tree is not linear

## ---------------------------------------------------------------------
## seedApply()
## ---------------------------------------------------------------------
seedApply(Y, class)
seedApply(Y, dim)
```

---

simplify   *Simplify 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.

`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

```
simplify(x, incremental=FALSE)

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

Arguments

x   Typically a [DelayedArray](#) object but can also be a [DelayedOp](#) object.

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 array elements of x and the array 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 array elements of x map to their corresponding array 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

- showtree to visualize and access the leaves of a tree of delayed operations carried by a DelayedArray object.
- DelayedOp objects.
- DelayedArray objects.

Examples

```
## ---------------------------------------------------------------------
## Simplification of the tree of delayed operations
## ---------------------------------------------------------------------
m1 <- matrix(runif(150), nrow=15, ncol=10)
M1 <- DelayedArray(m1)
showtree(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)  # linear tree

## 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)  # linear tree

options(DelayedArray.simplify=FALSE)

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

m2 <- matrix(1:20, nrow=10)
Y <- cbind(t(M1[ , 10:1]), DelayedArray(m2), M1[6:15, "A", drop=FALSE])
showtree(Y)   # non-linear tree

Z <- t(Y[10:1, ])[1:15, ] + 0.4 * M1
showtree(Z)   # non-linear tree

Z@seed@seeds
Z@seed@seeds[[2]]@seed                        # reaching to M1
Z@seed@seeds[[1]]@seed@seed@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
    is_noop(op1) && is_noop(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)
```

SparseArraySeed-class   *SparseArraySeed objects*

Description

SparseArraySeed objects are used internally to support block processing of array-like objects.

Usage

```
## Constructor function:
SparseArraySeed(dim, aind=NULL, nzdata=NULL, check=TRUE)

## Getters (in addition to dim() and length()):
aind(x)
nzdata(x)
sparsity(x)

## Two low-level utilities:
dense2sparse(x)
sparse2dense(sas)
```

Arguments

dim   The dimensions (specified as an integer vector) of the SparseArraySeed object
            to create.

aind   A matrix containing the array indices of the nonzero data.

            This must be an integer matrix like one returned by base::arrayInd, that is,
            with length(dim) columns and where each row is an n-uplet representing an
            array index.

nzdata   A vector of length nrow(aind) containing the nonzero data.

check   Should the object be validated upon construction?

x   A SparseArraySeed object for the aind, nzdata, and sparsity getters.

            An array-like object for dense2sparse.

sas   A SparseArraySeed object.

Value

- For SparseArraySeed(): A SparseArraySeed instance.
- For aind(): The matrix containing the array indices of the nonzero data.
- For nzdata(): The vector of nonzero data.
- For sparsity(): The number of zero-valued elements in the implicit array divided by the
  total number of array elements (a.k.a. the length of the array).
- For dense2sparse(): A SparseArraySeed instance.
- For sparse2dense(): An ordinary array.

See Also

- The read_sparse_block function.
- block_processing for more information about block processing of an array-like object.
- extract_array.
- DelayedArray objects.
- arrayInd in the **base** package.
- array objects in base R.

Examples

```
## ---------------------------------------------------------------------
## EXAMPLE 1
## ---------------------------------------------------------------------
aind1 <- rbind(c(2,4,3), c(2,1,3), c(5,4,3), c(5,3,3),
               c(5,4,1), c(5,1,1), c(5,4,2), c(5,4,1))
nzdata1 <- 11.11 * seq_len(nrow(aind1))
sas1 <- SparseArraySeed(5:3, aind1, nzdata1)

dim(sas1)        # the dimensions of the implicit array
length(sas1)     # the length of the implicit array
aind(sas1)
nzdata(sas1)
sparsity(sas1)

sparse2dense(sas1)
as.array(sas1)   # same as sparse2dense(sas1)

## Not run:
as.matrix(sas1)  # error!

## End(Not run)
## ---------------------------------------------------------------------
## EXAMPLE 2
## ---------------------------------------------------------------------
m2 <- matrix(c(5:-2, rep.int(c(0L, 99L), 11)), ncol=6)
sas2 <- dense2sparse(m2)
dim(sas2)
length(sas2)
aind(sas2)
nzdata(sas2)
sparsity(sas2)

stopifnot(identical(sparse2dense(sas2), m2))

as.matrix(sas2)  # same as sparse2dense(sas2)

t(sas2)
stopifnot(identical(as.matrix(t(sas2)), t(as.matrix(sas2))))

## Go back and forth between SparseArraySeed and dgCMatrix objects:
M2 <- as(sas2, "dgCMatrix")
stopifnot(identical(M2, as(m2, "dgCMatrix")))

sas2b <- as(M2, "SparseArraySeed")
## 'sas2b' is the same as 'sas2' except that
## 'nzdata(sas2b)' is of type numeric instead of integer:
all.equal(sas2b, sas2)
typeof(nzdata(sas2b))  # numeric
typeof(nzdata(sas2))   # integer

## ---------------------------------------------------------------------
## SEED CONTRACT
## ---------------------------------------------------------------------
## SparseArraySeed objects comply with the "seed contract".
## In particular they support extract_array():
```

```
extract_array(sas1, list(c(5, 3:2, 5), NULL, 3))

## See '?extract_array' for more information about the "seed contract".

## This means that they can be wrapped in a DelayedArray object:
A1 <- DelayedArray(sas1)
A1

## A big very sparse DelayedMatrix object:
aind3 <- cbind(sample(25000, 600000, replace=TRUE),
               sample(195000, 600000, replace=TRUE))
nzdata3 <- runif(600000)
sas3 <- SparseArraySeed(c(25000, 195000), aind3, nzdata3)
sparsity(sas3)

M3 <- DelayedArray(sas3)
M3
colSums(M3[ , 1:20])
```

Index