Package ‘DelayedArray’

March 30, 2021

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, ordinary arrays and, data frames.

biocViews Infrastructure, DataRepresentation, Annotation, GenomeAnnotation

URL https://bioconductor.org/packages/DelayedArray

BugReports https://github.com/Bioconductor/DelayedArray/issues

Version 0.16.3

License Artistic-2.0

Encoding UTF-8

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

Maintainer Hervé Pagès <hpages.on.github@gmail.com>

Depends R (>= 3.4), methods, stats4, Matrix, BiocGenerics (>= 0.31.5), MatrixGenerics (>= 1.1.3), S4Vectors (>= 0.27.2), IRanges (>= 2.17.3)

Imports stats

LinkingTo S4Vectors

Suggests BiocParallel, HDF5Array (>= 1.17.12), genefilter, SummarizedExperiment, airway, pryr, DelayedMatrixStats, knitr, BiocStyle, RUnit

VignetteBuilder knitr

Collate utils.R sparseMatrix-utils.R compress_atomic_vector.R Nindex-utils.R array_selection.R aperm2.R bind-arrays.R Array-class.R extract_array.R ArrayGrid-class.R show-utils.R SparseArraySeed-class.R SparseArraySeed-utils.R read_block.R mapToGrid.R makeCappedVolumeBox.R AutoBlock-global-settings.R

AutoGrid.R blockApply.R DelayedOp-class.R showtree.R simplify.R DelayedArray-class.R DelayedArray-subsetting.R chunkGrid.R write_block.R realize.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_12

git_last_commit 42c3398

git_last_commit_date 2021-03-23

Date/Publication 2021-03-29

R topics documented:

`array selection` *Manipulation of array selections*

Description

NOTE: The tools documented in this man page are primarily intended for developers or advanced users curious about the internals of the **DelayedArray** package. End users typically don't need them for their regular use of DelayedArray objects.

An *array selection* is just an index into an array-like object that contains the information of which array elements are selected. This index can take various forms but 3 special forms are particularly useful and extensively used thoughout the **DelayedArray** framework: *linear index* (also referred to as *L-index* or *Lindex*), *matrix index* (also referred to as *M-index* or *Mindex*), *N-dimensional index* (also referred to as *N-index* or *Nindex*). See Details section below for more information.

Two utility functions are provided at the moment to convert back and forth between *L-indices* and *M-indices*. More will be added in the future to convert between other types of array indices.

Usage

```
## Convert back and forth between L-indices and M-indices:
Lindex2Mindex(Lindex, dim, use.names=FALSE)
Mindex2Lindex(Mindex, dim, use.names=FALSE, as.integer=FALSE)
```

Arguments

`Lindex` An *L-index*. See Details section below.

`Mindex` An *M-index*. See Details section below.

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

`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 one row per element in `Lindex` (or per row in `Mindex`) and one column per dimension in the underlying array.

`use.names` Should the names (or rownames) on the input be propagated to the output?

`as.integer` Set to `TRUE` to force `Mindex2Lindex` to return the L-index as an integer vector. Dangerous!

By default, i.e. when `as.integer=FALSE`, `Mindex2Lindex` will return the L-index either as an integer or numeric vector. It will choose the former only if it's safe, that is, only if all the values in the L-index "fit" in the integer type. More precisely:

- If `dim` is not a matrix (i.e. is a vector) or if it's a matrix with a single row: `Mindex2Lindex` returns an integer or numeric vector depending on whether `prod(dim)` is <= `.Machine$integer.max` (2^31 - 1) or not.
- Otherwise `Mindex2Lindex` returns a numeric vector.

Note that with these rules, `Mindex2Lindex` can return a numeric vector even if an integer vector could have been used.

Use `as.integer=TRUE` only in situations where you know that all the L-index values are going to "fit" in the integer type. `Mindex2Lindex` will return garbage if they don't.

Details

The 3 special forms of array indices extensively used thoughout the **DelayedArray** framework:

1. *Linear index* (or *L-index* or *Lindex*): A numeric vector with no NAs where each value is >= 1 and <= the length of the array-like object. When using an L-index to subset an array-like object, the returned value is a vector-like object (i.e. no dimensions) of the same length as the L-index.

   Example:

   ```
   a <- array(101:124, 4:2)
   Lindex <- c(7, 2, 24, 2)
   a[Lindex]
   ```

2. *Matrix index* (or *M-index* or *Mindex*): An integer matrix with one column per dimension in the array-like object and one row per array element in the selection. No NAs. The values in each column must be >= 1 and <= the extent of the array-like object along the corresponding dimension. When using an M-index to subset an array-like object, the returned value is a vector-like object (i.e. no dimensions) of length the number of rows in the M-index.

   Example:

   ```
   a <- array(101:124, 4:2)
   Mindex <- rbind(c(3, 2, 1),
                   c(2, 1, 1),
                   c(4, 3, 2),
                   c(2, 1, 1))
   a[Mindex]
   ```

   Note that this is the type of index returned by `base::arrayInd`.

3. *N-dimensional* (or *N-index* or *Nindex*): A list with one list element per dimension in the array-like object. Each list element must be a subscript describing the selection along the corresponding dimension of the array-like object. IMPORTANT: A `NULL` subscript is interpreted as a *missing* subscript ("missing" like in `a[ , , 1:2]`), that is, as a subscript that runs along the full extend of the corresponding dimension of the array-like object. This means that before an N-index can be used in a call to `[`, `[<-`, `[[` or `[[<-`, the `NULL` list elements in it must be replaced with objects of class `"name"`. When using an N-index to subset an array-like object, the returned value is another array-like object of dimensions the lengths of the selections along each dimensions.

   Examples:

   ```
   a <- array(101:124, 4:2)

   ## Normalized N-index:

   Nindex <- list(c(1, 4, 1), NULL, 1)
   ## Same as a[c(1, 4, 1), , 1, drop=FALSE]:
   DelayedArray:::subset_by_Nindex(a, Nindex)

   Nindex <- list(integer(0), NULL, 1)
   ## Same as a[integer(0), , 1, drop=FALSE]:
   DelayedArray:::subset_by_Nindex(a, Nindex)

   ## Non-normalized N-index:
   ```

```
Nindex <- list(-3, NULL, 1)
Nindex <- DelayedArray:::normalizeNindex(Nindex, a)
## Same as a[-3, , 1, drop=FALSE]:
DelayedArray:::subset_by_Nindex(a, Nindex)

Nindex <- list(IRanges(2, 4), NULL, 1)
Nindex <- DelayedArray:::normalizeNindex(Nindex, a)
## Same as a[2:4, , 1, drop=FALSE]:
DelayedArray:::subset_by_Nindex(a, Nindex)

dimnames(a)[[1]] <- LETTERS[1:4]
Nindex <- list(c("D", "B"), NULL, 1)
Nindex <- DelayedArray:::normalizeNindex(Nindex, a)
## Same as a[c("D", "B"), , 1, drop=FALSE]:
DelayedArray:::subset_by_Nindex(a, Nindex)
```

Value

`Lindex2Mindex` returns an M-index.

`Mindex2Lindex` returns an L-index.

See Also

`arrayInd` in the **base** package.

Examples

```
dim <- 4:2
Mindex2Lindex(c(4, 3, 1), dim)
Mindex2Lindex(c(4, 3, 2), dim)

Mindex <- 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))

Mindex2Lindex(Mindex, dim)

## With a matrix of dimensions:

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

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

Mindex2Lindex(Mindex, dims)
```

```
## Sanity checks:

dim <- c(33:30, 45L, 30L)
stopifnot(Mindex2Lindex(rep(1, 6), dim) == 1)
stopifnot(Mindex2Lindex(dim, dim) == prod(dim))

stopifnot(identical(Mindex2Lindex(arrayInd(1:120, 6:4), 6:4), 1:120))
stopifnot(identical(Mindex2Lindex(arrayInd(840:1, 4:7), 4:7), 840:1))
```

`ArrayGrid-class` *ArrayGrid and ArrayViewport objects*

Description

A *grid* is a partitioning of an array-like object into blocks (or viewports).

In the **DelayedArray** package, grids and viewports are formally represented by ArrayGrid and ArrayViewport objects, respectively.

There are two variants of ArrayGrid objects:

- RegularArrayGrid objects: for grids where all the blocks have the same geometry (except maybe for the edge blocks).
- ArbitraryArrayGrid objects: for grids where blocks don't necessarily have the same geometry.

ArrayGrid and ArrayViewport objects are used extensively in the context of block processing of array-like objects.

Usage

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

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

Arguments

`refdim`	An integer vector containing the dimensions of the reference array.
`spacings`	An integer vector specifying the grid spacing along each dimension.
`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.
`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.

Details

RegularArrayGrid and ArbitraryArrayGrid are concrete subclasses of ArrayGrid, which itself is a virtual class.

Note that an ArrayGrid or ArrayViewport object doesn't store any array data, only the geometry of the grid or viewport. This makes these objects extremely light-weight, even for grids made of millions of blocks.

Value

For `RegularArrayGrid()`, a RegularArrayGrid instance.

For `ArbitraryArrayGrid()`, an ArbitraryArrayGrid instance.

For `downsample()`, an ArrayGrid object on the same reference array than `x`.

See Also

- `read_block`.
- `blockApply` and family for convenient 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)
))
```

AutoBlock-global-settings

Control the geometry of automatic blocks

Description

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

Usage

```
getAutoBlockSize()
setAutoBlockSize(size=1e8)

getAutoBlockLength(type)

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

Arguments

`size`	The *auto block size* (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 500 x 1000 x 500 doubles has a length of 250 million elements and a size of 2 Gb (each double occupies 8 bytes of memory). The *auto 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 *auto block shape* (automatic block shape). See `makeCappedVolumeBox` for a description of the supported shapes. The *auto block shape* is set to `"hypercube"` at package startup and can be reset anytime to this value by calling `setAutoBlockShape()` with no argument.

Details

block size != *block length*

block length = number of array elements in a block (i.e. `prod(dim(block))`).

block size = *block length* * size of the individual elements in memory.

For example, for an integer array, *block size* (in bytes) is going to be 4 x *block length*. For a numeric array `x` (i.e. `type(x) == "double"`), it's going to be 8 x *block length*.

In its current form, block processing in the **DelayedArray** package must decide the geometry of the blocks before starting the walk on the blocks. It does this based on several criteria. Two of them are:

- The *auto block size*: maximum size (in bytes) of a block once loaded in memory.
- The `type()` of the array (e.g. `integer`, `double`, `complex`, etc...)

The *auto block size* setting and `type(x)` control the maximum length of the blocks. Other criteria control their shape. So for example if you set the *auto block size* to 8GB, this will cap the length of the blocks to 2e9 if your DelayedArray object `x` is of type `integer`, and to 1e9 if it's of type `double`.

Note that this simple relationship between *block size* and *block length* assumes that blocks are loaded in memory as ordinary (a.k.a. dense) matrices or arrays. With sparse blocks, all bets are off.

But the max block length is always taken to be the *auto block size* divided by `get_type_size(type())` whether the blocks are going to be loaded as dense or sparse arrays. If they are going to be loaded as sparse arrays, their memory footprint is very likely to be smaller than if they were loaded as dense arrays so this is safe (although probably not optimal).

It's important to keep in mind that the *auto block size* setting is a simple way for the user to put a cap on the memory footprint of the blocks. Nothing more. In particular it doesn't control the maximum amount of memory used by the block processing algorithm. Other variables can impact dramatically memory usage like parallelization (where more than one block is loaded in memory at any given time), what the algorithm is doing with the blocks (e.g. something like `blockApply(x,identity)` will actually load the entire array data in memory), what delayed operations are on `x`, etc... It would be awesome to have a way to control the maximum amount of memory used by a block processing algorithm as a whole but we don't know how to do that.

Value

`getAutoBlockSize`: The current *auto block size* in bytes as a single numeric value.

`setAutoBlockSize`: The new *auto block size* in bytes as an invisible single numeric value.

`getAutoBlockLength`: The *auto block length* as a single integer value.

`getAutoBlockShape`: The current *auto block shape* as a single string.

`setAutoBlockShape`: The new *auto block shape* as an invisible single string.

See Also

- `defaultAutoGrid` and family to generate automatic grids to use for block processing of array-like objects.
- `blockApply` and family for convenient block processing of an array-like object.
- The `makeCappedVolumeBox` utility to make *capped volume boxes*.

Examples

```
getAutoBlockSize()

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

m <- matrix(runif(600), ncol=12)
setAutoBlockSize(140)
getAutoBlockLength(type(m))
defaultAutoGrid(m)
lengths(defaultAutoGrid(m))
dims(defaultAutoGrid(m))

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

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

AutoGrid — *Generate automatic grids to use for block processing of array-like objects*

Description

`defaultAutoGrid()` is the default *automatic grid maker*. It creates a grid that is suitable for block processing of the array-like object passed to it.

`rowAutoGrid()` and `colAutoGrid()` are additional *automatic grid makers*, specific to the 2-dimensional case. They can be used to define blocks of full rows or full columns.

Usage

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

rowAutoGrid(x, nrow=NULL, block.length=NULL)
colAutoGrid(x, ncol=NULL, block.length=NULL)

## Replace default automatic grid maker with user-defined one:
getAutoGridMaker()
setAutoGridMaker(GRIDMAKER="defaultAutoGrid")
```

Arguments

`x`	An array-like or matrix-like object for `defaultAutoGrid`. A matrix-like object for `rowAutoGrid` and `colAutoGrid`.
`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.
`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 `defaultAutoGrid`, `rowAutoGrid`, or `colAutoGrid`. The *automatic grid maker* is set to `defaultAutoGrid` at package startup and can be reset anytime to this value by calling `setAutoGridMaker()` with no argument.

Details

By default, primary block processing functions `blockApply()` and `blockReduce()` use the grid returned by `defaultAutoGrid(x)` to process array-like object `x` block by block. This can be changed with `setAutoGridMaker()`.

Value

`defaultAutoGrid`: 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).

`rowAutoGrid`: A RegularArrayGrid object on reference array `x` where the grid elements define blocks made of full rows of `x`.

`colAutoGrid`: A RegularArrayGrid object on reference array `x` where the grid elements define blocks made of full columns of `x`.

See Also

- `setAutoBlockSize` and `setAutoBlockShape` to control the geometry of automatic blocks.
- `blockApply` and family for convenient block processing of an array-like object.
- ArrayGrid for the formal representation of grids and viewports.
- The `makeCappedVolumeBox` utility to make *capped volume boxes*.
- `chunkGrid`.
- `read_block` and `write_block`.

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 <- defaultAutoGrid(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, colAutoGrid(M, block.length=120))
sum3

sum4 <- block_sum(M, rowAutoGrid(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))

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

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

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

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

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

## ---------------------------------------------------------------------
## REPLACE DEFAULT AUTOMATIC GRID MAKER WITH USER-DEFINED ONE
```

```
## ---------------------------------------------------------------------
getAutoGridMaker()
setAutoGridMaker(function(x) colAutoGrid(x, ncol=5))
getAutoGridMaker()

blockApply(m, function(block) currentViewport())

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

---

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 from 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)
```

blockApply *blockApply() and family*

Description

A family of convenience functions to walk on the blocks of an array-like object and process them.

Usage

```
## Main looping functions:

blockApply(x, FUN, ..., grid=NULL, as.sparse=FALSE,
           BPPARAM=getAutoBPPARAM(), verbose=NA)

blockReduce(FUN, x, init, ..., BREAKIF=NULL, grid=NULL, as.sparse=FALSE,
            verbose=NA)

## Lower-level looping functions:
viewportApply(grid, FUN, ..., BPPARAM=getAutoBPPARAM(), verbose=NA)
viewportReduce(FUN, grid, init, ..., BREAKIF=NULL, verbose=NA)

## Retrieve grid context for the current block/viewport:
effectiveGrid(envir=parent.frame(2))
currentBlockId(envir=parent.frame(2))
currentViewport(envir=parent.frame(2))

## Get/set automatic parallel back-end:
getAutoBPPARAM()
setAutoBPPARAM(BPPARAM=NULL)

## For testing/debugging callback functions:
set_grid_context(effective_grid, current_block_id, envir=parent.frame(1))
```

Arguments

x
: An array-like object, typically a DelayedArray object or derivative.

FUN
: For blockApply and blockReduce, FUN is the callback function to apply to each block of x. It must be able to accept as input any of the blocks of x.

  IMPORTANT: If as.sparse is set to FALSE, all blocks will be passed to FUN as ordinary arrays. If it's set to TRUE, they will be passed as SparseArraySeed objects. If it's set to NA, then is_sparse(x) determines how they will be passed to FUN.

  For viewportApply() and viewportReduce(), FUN is the callback function to apply to each **viewport** in grid. It must be able to accept as input any of the viewports in grid.

  For blockReduce(), init <-FUN(block,init) will be performed on each block so FUN must take at least two arguments (typically block and init but the names can differ) and should normally return a value of the same type as its 2nd argument (init).

  The same applies for viewportReduce(), except that init <-FUN(viewport,init) will be performed on each **viewport**.

`...`	Optional arguments to `FUN`.
`grid`	An ArrayGrid object that defines the blocks (or viewports) to walk on. For `blockApply()` and `blockReduce()` the supplied grid must be compatible with the geometry of `x`. If not specified, an automatic grid is used. By default `defaultAutoGrid(x)` is called to generate an automatic grid. The *automatic grid maker* can be changed with `setAutoGridMaker()`. See `?setAutoGridMaker` for more information.
`as.sparse`	Passed to the internal calls to `read_block`. See `?read_block` for more information.
`BPPARAM`	A `NULL`, in which case blocks are processed sequentially, or a BiocParallelParam instance (from the **BiocParallel** package), in which case they are processed in parallel. The specific BiocParallelParam instance determines the parallel back-end to use. See `?BiocParallelParam` in the **BiocParallel** package for more information about parallel back-ends.
`verbose`	Whether block processing progress should be displayed or not. If set to `NA` (the default), verbosity is controlled by `DelayedArray:::get_verbose_block_processing()`. Setting `verbose` to `TRUE` or `FALSE` overrides this.
`init`	The value to pass to the first call to `FUN(block,init)` (or `FUN(viewport,init)`) when `blockReduce()` (or `viewportReduce()`) starts the walk. Note that `blockReduce()` and `viewportReduce()` always operate sequentially.
`BREAKIF`	An optional callback function that detects a break condition. Must return `TRUE` or `FALSE`. At each iteration `blockReduce()` (and `viewportReduce()`) will call it on the result of `init <-FUN(block,init)` (on the result of `init <-FUN(viewport,init)` for `viewportReduce()`) and exit the walk if `BREAKIF(init)` returned `TRUE`.
`envir`	Do not use (unless you know what you are doing).
`effective_grid, current_block_id`	See Details below.

Details

`effectiveGrid()`, `currentBlockId()`, and `currentViewport()` return the "grid context" for the block/viewport being currently processed. By "grid context" we mean:

- The *effective grid*, that is, the user-supplied grid or `defaultAutoGrid(x)` if the user didn't supply any grid.
- The *current block id* (a.k.a. block rank).
- The *current viewport*, that is, the ArrayViewport object describing the position of the current block w.r.t. the effective grid.

Note that `effectiveGrid()`, `currentBlockId()`, and `currentViewport()` can only be called (with no arguments) from **within** the callback functions `FUN` and/or `BREAKIF` passed to `blockApply()` and family.

If you need to be able to test/debug your callback function as a standalone function, set an arbitrary *effective grid* and *current block id* by calling

```
set_grid_context(effective_grid, current_block_id)
```

right before calling the callback function.

Value

For `blockApply()` and `viewportApply()`, a list with one list element per block/viewport visited.

For `blockReduce()` and `viewportReduce()`, the result of the last call to `FUN`.

For `effectiveGrid()`, the grid (ArrayGrid object) being effectively used.

For `currentBlockId()`, the id (a.k.a. rank) of the current block.

For `currentViewport()`, the viewport (ArrayViewport object) of the current block.

See Also

- `defaultAutoGrid` and family to generate automatic grids to use for block processing of array-like objects.
- ArrayGrid for the formal representation of grids and viewports.
- `read_block` and `write_block`.
- `MulticoreParam`, `SnowParam`, and `bpparam`, from the **BiocParallel** package.
- DelayedArray objects.

Examples

```
m <- matrix(1:60, nrow=10)
m_grid <- defaultAutoGrid(m, block.length=16, block.shape="hypercube")

## ---------------------------------------------------------------------
## blockApply()
## ---------------------------------------------------------------------
blockApply(m, identity, grid=m_grid)
blockApply(m, sum, grid=m_grid)

blockApply(m, function(block) {block + currentBlockId()*1e3}, grid=m_grid)
blockApply(m, function(block) currentViewport(), grid=m_grid)
blockApply(m, dim, grid=m_grid)

## The grid does not need to be regularly spaced:
a <- array(runif(8000), dim=c(25, 40, 8))
a_tickmarks <- list(c(7L, 15L, 25L), c(14L, 22L, 40L), c(2L, 8L))
a_grid <- ArbitraryArrayGrid(a_tickmarks)
a_grid
blockApply(a, function(block) sum(log(block + 0.5)), grid=a_grid)

## See block processing in action:
blockApply(m, function(block) sum(log(block + 0.5)), grid=m_grid,
           verbose=TRUE)

## Use parallel evaluation:
library(BiocParallel)
if (.Platform$OS.type != "windows") {
    BPPARAM <- MulticoreParam(workers=4)
} else {
    ## MulticoreParam() is not supported on Windows so we use
    ## SnowParam() on this platform.
    BPPARAM <- SnowParam(4)
}
blockApply(m, function(block) sum(log(block + 0.5)), grid=m_grid,
           BPPARAM=BPPARAM, verbose=TRUE)
```

```
## Note that blocks can be visited in any order!

## ---------------------------------------------------------------------
## blockReduce()
## ---------------------------------------------------------------------
FUN <- function(block, init) anyNA(block) || init
blockReduce(FUN, m, init=FALSE, grid=m_grid, verbose=TRUE)

m[10, 1] <- NA
blockReduce(FUN, m, init=FALSE, grid=m_grid, verbose=TRUE)

## With early bailout:
blockReduce(FUN, m, init=FALSE, BREAKIF=identity, grid=m_grid,
            verbose=TRUE)

## Note that this is how the anyNA() method for DelayedArray objects is
## implemented.

## ---------------------------------------------------------------------
## viewportReduce()
## ---------------------------------------------------------------------
## The man page for write_block() contains several examples of how to
## use viewportReduce() to write array blocks to a "realization sink".
## See '?write_block'
```

`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:

- *N-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 is when `drop=TRUE` and the result has only one dimension, in which case it is *realized* as an ordinary vector (atomic or list). Note that NAs in the subscripts are not supported.
- *1D-style single bracket subsetting* (i.e. subsetting of the form `x[i]`) only works if the subscript `i` is a numeric or logical vector, or a logical array-like object with the same dimensions as `x`, or a numeric matrix with one column per dimension in `x`. When `i` is a numeric vector, all the indices in it must be >= 1 and <= `length(x)`. NAs in the subscripts are not supported. This is NOT a delayed operation (block processing is triggered) i.e. the result is *realized* as an ordinary vector (atomic or list). One exception is when `x` has only one dimension and `drop` is set to `FALSE`, in which case the subsetting is *delayed*.

Subsetting with `[[` is supported but only the 1D-style form of it at the moment, that is, subsetting of the form `x[[i]]` 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:

- *N-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.
- *1D-style 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.
- blockApply and family for convenient 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)

## N-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)))

## 1D-style 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 "automatic 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
setAutoRealizationBackend("HDF5Array")
D <- realize(D)
D
## See '?setAutoRealizationBackend' 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:
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()  # reset automatic block size to factory settings

## 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`
- `type<-`
- `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

type(M2) <- "integer"          # delayed
M2

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

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

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

Arguments

x	A DelayedMatrix object.
na.rm	Should missing values (including NaN) be omitted from the calculations?
dims, rows, cols	These arguments are not supported. Don't use them.

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)

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

rmaxs <- rowMaxs(M12)
cmaxs <- colMaxs(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(rmins, rowMins(m12)))
stopifnot(identical(cmins, colMins(m12)))
stopifnot(identical(rmaxs, rowMaxs(m12)))
stopifnot(identical(cmaxs, colMaxs(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.
- `setAutoRealizationBackend` for how to set a *automatic 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 "automatic realization backend" controls
## where realization happens e.g. in memory as an ordinary matrix if not
## (i.e. set to NULL) or in an HDF5 file if set to "HDF5Array".
## See '?setAutoRealizationBackend' 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:
setAutoRealizationBackend()
P1 <- m %*% M1
P1

## Set backend to HDF5Array for realization in HDF5 file:
setAutoRealizationBackend("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 this tree is a DelayedOp object that represents a delayed operation.

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`.

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 defaultAutoGrid 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

- `defaultAutoGrid` and family to generate automatic grids to use for block processing of array-like objects.
- ArrayGrid for the formal representation of grids and viewports.

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 array blocks*

Description

Use read_block to read a block from an array-like object. The function is typically used in the context of block processing of array-like objects (typically DelayedArray objects but not necessarily).

Usage

```
read_block(x, viewport, as.sparse=FALSE)
```

Arguments

x
: An array-like object, typically a DelayedArray object or derivative.

viewport
: An ArrayViewport object compatible with x, that is, such that refdim(viewport) is identical to dim(x).

as.sparse
: Can be FALSE, TRUE, or NA.

  If FALSE (the default), the block is returned as an ordinary array (a.k.a. dense array). If TRUE, it's returned as a SparseArraySeed object. Using as.sparse=NA is equivalent to using as.sparse=is_sparse(x) and is the most efficient way to read a block. (This might become the default in the future.)

  Note that when returned as a 2D SparseArraySeed object with numeric or logical data, a block can easily and efficiently be coerced to a sparseMatrix derivative from the **Matrix** package with as(block,"sparseMatrix"). This will return a dgCMatrix object if type(block) is "double" or "integer", or a lgCMatrix object if it's "logical".

Value

The data from x that belongs to the block delimited by the specified viewport. The data is returned as an ordinary (dense) array or as a SparseArraySeed object. In both cases it has the same dimensions as the `viewport`.

See Also

- ArrayViewport objects.
- SparseArraySeed objects.
- `write_block`.
- `blockApply` and family for convenient block processing of an array-like object.
- `defaultAutoGrid` and family to generate automatic grids to use for block processing of array-like objects.
- dgCMatrix-class and lgCMatrix-class objects in the **Matrix** package.
- DelayedArray objects.
- array objects in base R.

Examples

```
## ---------------------------------------------------------------------
## TYPICAL USE
## ---------------------------------------------------------------------
## read_block() is typically used in combination with write_block().
## See '?write_block' for typical uses of the read_block/write_block
## combo.

## ---------------------------------------------------------------------
## VERY BASIC (BUT ALSO VERY ARTIFICIAL) EXAMPLE 1:
## Read a block from an ordinary matrix
## ---------------------------------------------------------------------
m1 <- matrix(1:30, ncol=5)
m1

## Define the viewport on 'm1' to read the data from:
block1_dim <- c(4, 3)
viewport1 <- ArrayViewport(dim(m1), IRanges(c(3, 2), width=block1_dim))
viewport1

## Read the block:
block1 <- read_block(m1, viewport1)  # same as m1[3:6, 2:4, drop=FALSE]
block1

## Sanity checks:
stopifnot(identical(dim(viewport1), dim(block1)))
stopifnot(identical(m1[3:6, 2:4, drop=FALSE], block1))

## ---------------------------------------------------------------------
## VERY BASIC (BUT ALSO VERY ARTIFICIAL) EXAMPLE 2:
## Read a block from a sparse matrix
## ---------------------------------------------------------------------
m2 <- rsparsematrix(12, 20, density=0.2,
                    rand.x=function(n) sample(25, n, replace=TRUE))
m2
```

```
## Define the viewport on 'm2' to read the data from:
block2_dim <- c(2, 20)
viewport2 <- ArrayViewport(dim(m2), IRanges(c(1, 1), width=block2_dim))
viewport2

## By default, read_block() always returns an ordinary matrix or array:
block2 <- read_block(m2, viewport2)
block2

## It is recommended to use 'as.sparse=NA' rather than 'as.sparse=TRUE'
## or 'as.sparse=FALSE' to let read_block() pick up the optimal
## representation:
block2b <- read_block(m2, viewport2, as.sparse=NA)
class(block2b)  # a SparseArraySeed object
as(block2b, "sparseMatrix")

## For comparison, using 'as.sparse=NA' on 'm1' still returns the
## block as an ordinary matrix (a.k.a. dense matrix):
read_block(m1, viewport1, as.sparse=NA)

## Sanity checks:
stopifnot(identical(dim(viewport2), dim(block2)))
stopifnot(identical(dim(viewport2), dim(block2b)))
stopifnot(identical(block2, as.array(block2b)))

## ---------------------------------------------------------------------
## VERY BASIC (BUT ALSO VERY ARTIFICIAL) EXAMPLE 3:
## Read a block from a 3D array
## ---------------------------------------------------------------------
a3 <- array(1:60, 5:3)

## Define the viewport on 'a3' to read the data from:
block3_dim <- c(2, 4, 1)
viewport3 <- ArrayViewport(dim(a3), IRanges(c(1, 1, 3), width=block3_dim))
viewport3

## Read the block:
block3 <- read_block(a3, viewport3)  # same as a3[1:2, 1:4, 3, drop=FALSE]
block3

## Note that unlike [, read_block() never drops dimensions.

## Sanity checks:
stopifnot(identical(dim(viewport3), dim(block3)))
stopifnot(identical(a3[1:2, 1:4, 3, drop=FALSE], block3))
```

`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=getAutoRealizationBackend())
```

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 *automatic realization backend* by default i.e. the backend returned by `getAutoRealizationBackend()`.

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

- `getAutoRealizationBackend` and `setAutoRealizationBackend` for getting and setting the current *automatic 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()
getAutoRealizationBackend()  # backend is set to NULL
realize(M3)  # realization as ordinary array

setAutoRealizationBackend("RleArray")
getAutoRealizationBackend()  # backend is set to "RleArray"
realize(M3)  # realization as RleArray object

setAutoRealizationBackend("HDF5Array")
getAutoRealizationBackend()  # 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 (i.e.
## an object of length > .Machine$integer.max) 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:

toy_Rle <- function() {
  run_lens <- c(sample(4), sample(rep(c(1:19, 40) * 3, 6e4)), sample(4))
  run_vals <- sample(700, length(run_lens), replace=TRUE) / 5
  Rle(run_vals, run_lens)
}
rle_list <- lapply(1:80, function(j) toy_Rle())  # takes about 20 sec.

## Cumulative length of all the Rle objects is > .Machine$integer.max:
sum(lengths(rle_list))  # 3.31e+09

## Feed 'rle_list' to the RleArray() constructor:
dim <- c(14395, 320, 719)
A <- RleArray(rle_list, dim)
A

## Because all the Rle objects in 'rle_list' 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 'rle_list':
M <- RleArray(rle_list)
M
stopifnot(identical(as(rle_list, "RleList"), as(M, "RleList")))

## ---------------------------------------------------------------------
## E. CHANGING THE TYPE OF AN RleArray OBJECT FROM "double" TO "integer"
## ---------------------------------------------------------------------

## An RleArray object is an in-memory object so it can be useful to
## reduce its memory footprint. For an object of type "double" this can
## be done by changing its type to "integer" (integers are half the size
## of doubles in memory). Of course this only makes sense if this results
## in a loss of precision that is acceptable.
## On an ordinary array (or matrix) 'a', this is simply a matter of
## doing 'storage.mode(a) <- "integer"'. However, with a DelayedArray
## object, things are a little bit different. Let's do this on a subset
## of the RleMatrix object 'M' created in the previous section.

M1 <- as(M[1:6e5, ], "RleMatrix")
rm(M)

## First of all, it's important to be aware that object.size() (from
## package utils) is NOT reliable on RleArray objects! This is because
## the data in an RleArray object is stored in an environment and
## object.size() stubbornly refuses to take the content of an environment
## into account when computing its size:
object.size(list2env(list(aa=1:10)))   # 56 bytes
object.size(list2env(list(aa=1:1e6)))  # always 56 bytes!
```

```
## So we'll use object_size() instead (from package pryr):
library(pryr)
object_size(list2env(list(aa=1:10)))   # 264 B
object_size(list2env(list(aa=1:1e6)))  # 4 MB
object_size(list2env(list(aa=as.double(1:1e6))))  # 8 MB

object_size(M1)  # 16.7 MB

type(M1) <- "integer"  # Delayed!
M1                     # Note the class: it's no longer RleMatrix!
                       # (That's because the object now carries delayed
                       # operations.)

## Because changing the type is a delayed operation, the memory footprint
## of the object has not changed yet (remember that the original data in
## a DelayedArray object is stored in its "seed" and its seed is never
## modified **in-place**, that is, no operation on the object will ever
## modify its seed):
object_size(M1)  # Still the same (well, a very tiny more, because the
                 # object is now carrying one more delayed operation,
                 # the `type<-` operation)

## To effectively reduce the memory footprint of the object, a new object
## needs to be created. This is achieved simply by **realizing** M1 as a
## (new) RleArray object. Note that this realization will use block
## processing:

DelayedArray:::set_verbose_block_processing(TRUE)  # See block processing
                                                   # in action.
getAutoBlockSize()      # Automatic block size (100 Mb by default).
setAutoBlockSize(20e6)  # Set automatic block size to 20 Mb.

M2 <- as(M1, "RleArray")
DelayedArray:::set_verbose_block_processing(FALSE)
setAutoBlockSize()      # Reset automatic block size to factory settings.


M2
object_size(M2)  # 6.91 MB (Less than half the original size! This is
                 # because RleArray objects use some internal tricks to
                 # reduce memory footprint even more when the data in
                 # their seed is of type "integer".)

## Finally note that the 2-step approach described here (i.e.
## type(A) <- "integer" followed by realization) is generic and works
## on any kind of DelayedArray object or derivative. In particular,
## after doing 'type(A) <- "integer"', 'A' can be realized as anything
## as long as the realization backend is supported (e.g. could be
## 'as(A, "HDF5Array")' or 'as(A, "TENxMatrix")') and realization will
## always use block processing so the array data will never be fully
## loaded in memory.
```

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 or advanced users curious about the internals of the **DelayedArray** package. End users typically don't need them for their regular use of DelayedArray objects.

In a DelayedArray object, the delayed operations are stored as a tree of DelayedOp objects. See ?DelayedOp for more information about this tree.

`simplify` can be used to simplify the tree of delayed operations in a DelayedArray object.

`isPristine` can be used to know whether a DelayedArray object is *pristine* or not. A DelayedArray object is considered *pristine* when it carries no delayed operation. Note that an object that carries delayed operations that do nothing (e.g. `A + 0`) is not considered *pristine*.

`contentIsPristine` can be used to know whether the delayed operations in a DelayedArray object *touch* its array elements or not.

`netSubsetAndAperm` returns an object that represents the *net subsetting* and *net dimension rearrangement* of all the delayed operations in a DelayedArray object.

Usage

```
simplify(x, incremental=FALSE)

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

Arguments

`x`	Typically a DelayedArray object but can also be a DelayedOp object (except for `isPristine`).
`incremental`	For internal use.
`ignore.dimnames`	`TRUE` or `FALSE`. When `TRUE`, the object is considered *pristine* even if its dimnames have been modified and no longer match the dimnames of its seed (in which case the object carries a single delayed operations of type DelayedDimnames).
`as.DelayedOp`	`TRUE` or `FALSE`. Controls 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

## ---------------------------------------------------------------------
## isPristine()
## ---------------------------------------------------------------------
m <- matrix(1:20, ncol=4, dimnames=list(letters[1:5], NULL))
M <- DelayedArray(m)

isPristine(M)                # TRUE
isPristine(log(M))           # FALSE
isPristine(M + 0)            # FALSE
isPristine(t(M))             # FALSE
isPristine(t(t(M)))          # TRUE
isPristine(cbind(M, M))      # FALSE
isPristine(cbind(M))         # TRUE

dimnames(M) <- NULL
isPristine(M)                # FALSE
isPristine(M, ignore.dimnames=TRUE)  # TRUE
isPristine(t(t(M)), ignore.dimnames=TRUE)  # TRUE
isPristine(cbind(M, M), ignore.dimnames=TRUE)  # FALSE

## ---------------------------------------------------------------------
## contentIsPristine()
## ---------------------------------------------------------------------
a <- array(1:40, 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(1:40, 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, nzindex=NULL, nzdata=NULL, dimnames=NULL, check=TRUE)
```

```
## Getters (in addition to dim(), length(), and dimnames()):
nzindex(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.
`nzindex`	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(nzindex)` containing the nonzero data.
`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.
`check`	Should the object be validated upon construction?
`x`	A SparseArraySeed object for the `nzindex`, `nzdata`, and `sparsity` getters. An array-like object for `dense2sparse`.
`sas`	A SparseArraySeed object.

Value

- For `SparseArraySeed()`: A SparseArraySeed instance.
- For `nzindex()`: 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

- SparseArraySeed-utils for native operations on SparseArraySeed objects.
- The `read_block` function.
- `blockApply` and family for convenient block processing of an array-like object.
- `extract_array`.
- DelayedArray objects.
- `arrayInd` in the **base** package.
- array objects in base R.

Examples

```
## ---------------------------------------------------------------------
## EXAMPLE 1
## ---------------------------------------------------------------------
dim1 <- 5:3
nzindex1 <- Lindex2Mindex(sample(60, 8), 5:3)
nzdata1 <- 11.11 * seq_len(nrow(nzindex1))
sas1 <- SparseArraySeed(dim1, nzindex1, nzdata1)

dim(sas1)        # the dimensions of the implicit array
length(sas1)     # the length of the implicit array
nzindex(sas1)
nzdata(sas1)
type(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)
class(sas2)
dim(sas2)
length(sas2)
nzindex(sas2)
nzdata(sas2)
type(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))))

## ---------------------------------------------------------------------
## COERCION FROM/TO dgCMatrix OR lgCMatrix OBJECTS
## ---------------------------------------------------------------------
## dgCMatrix and lgCMatrix objects are defined in the Matrix package.

M2 <- as(sas2, "dgCMatrix")
stopifnot(identical(M2, as(m2, "dgCMatrix")))

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

```
typeof(nzdata(sas2))   # integer

m3 <- m2 == 99  # logical matrix
sas3 <- dense2sparse(m3)
class(sas3)
type(sas3)
M3 <- as(sas3, "lgCMatrix")
stopifnot(identical(M3, as(m3, "lgCMatrix")))

sas3b <- as(M3, "SparseArraySeed")
stopifnot(identical(sas3, sas3b))

## ---------------------------------------------------------------------
## 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:
nzindex4 <- cbind(sample(25000, 600000, replace=TRUE),
                  sample(195000, 600000, replace=TRUE))
nzdata4 <- runif(600000)
sas4 <- SparseArraySeed(c(25000, 195000), nzindex4, nzdata4)
sparsity(sas4)

M4 <- DelayedArray(sas4)
M4
colSums(M4[ , 1:20])
```

`SparseArraySeed-utils` *Operate natively on SparseArraySeed objects*

Description

Some utilities to operate natively on SparseArraySeed objects. Mostly for internal use by the **DelayedArray** package e.g. they support block processed methods for sparse DelayedArray objects like `sum()`, `mean()`, `which()`, etc...

Usage

```
## S4 method for signature 'SparseArraySeed'
is.na(x)

## S4 method for signature 'SparseArraySeed'
is.infinite(x)

## S4 method for signature 'SparseArraySeed'
```

```
is.nan(x)

## S4 method for signature 'SparseArraySeed'
tolower(x)

## S4 method for signature 'SparseArraySeed'
toupper(x)

## S4 method for signature 'SparseArraySeed'
nchar(x, type="chars", allowNA=FALSE, keepNA=NA)

## S4 method for signature 'SparseArraySeed'
anyNA(x, recursive=FALSE)

## S4 method for signature 'SparseArraySeed'
which(x, arr.ind=FALSE, useNames=TRUE)


## <>-<>-<> "Summary" group generic <>-<>-<>

## S4 method for signature 'SparseArraySeed'
max(x, ..., na.rm=FALSE)

## S4 method for signature 'SparseArraySeed'
min(x, ..., na.rm=FALSE)

## S4 method for signature 'SparseArraySeed'
range(x, ..., finite=FALSE, na.rm=FALSE)

## S4 method for signature 'SparseArraySeed'
sum(x, ..., na.rm=FALSE)

## S4 method for signature 'SparseArraySeed'
prod(x, ..., na.rm=FALSE)

## S4 method for signature 'SparseArraySeed'
any(x, ..., na.rm=FALSE)

## S4 method for signature 'SparseArraySeed'
all(x, ..., na.rm=FALSE)


## <>-<>-<> others <>-<>-<>

## S4 method for signature 'SparseArraySeed'
mean(x, na.rm=FALSE)
```

Arguments

`x` A SparseArraySeed object.

`type, allowNA, keepNA`
See `?base::nchar` for a description of these arguments.

recursive	See ?base::anyNA for a description of this argument.
arr.ind	See ?base::which for a description of this argument.
useNames	Ignored.
...	Unsupported.
na.rm	TRUE or FALSE (the default). Should NA's and NaN's be removed?
finite	TRUE or FALSE (the default). Should non-finite values be removed?

Value

See corresponding functions in the **base** package.

See Also

- SparseArraySeed objects.

Examples

```
## Create a SparseArraySeed object:
dim1 <- 5:3
nzindex1 <- Lindex2Mindex(sample(60, 14), 5:3)
sas1 <- SparseArraySeed(dim1, nzindex1, nzdata=sample(0:13))

## Apply native operations:
sum(sas1)
range(sas1)
mean(sas1)

## Sanity checks:
stopifnot(identical(sum(as.array(sas1)), sum(sas1)))
stopifnot(identical(range(as.array(sas1)), range(sas1)))
stopifnot(identical(mean(as.array(sas1)), mean(sas1)))
```

---

sparseMatrix-utils *sparseMatrix utilities*

Description

Some utilities to operate natively on sparseMatrix objects (e.g. dgCMatrix and lgCMatrix objects) from the **Matrix** package. Mostly for internal use by the **DelayedArray** package.

Usage

```
## rowsum() and colsum() S4 generics:

#rowsum(x, group, reorder=TRUE, ...)
#colsum(x, group, reorder=TRUE, ...)

## Default methods:

## S4 method for signature 'ANY'
rowsum(x, group, reorder=TRUE, ...)
```

```
## S4 method for signature 'ANY'
colsum(x, group, reorder=TRUE, ...)

## rowsum() method for dgCMatrix objects:

## S4 method for signature 'dgCMatrix'
rowsum(x, group, reorder=TRUE, ...)
```

Arguments

x A numeric matrix-like object.

group, reorder, ...
See ?base::rowsum for a description of these arguments.

Value

See ?base::rowsum for the value returned by the default rowsum method.

The default colsum method returns t(rowsum(t(x),group,reorder=reorder,...)).

See Also

- DelayedMatrix-utils in this package for the rowsum and colsum methods defined for DelayedMatrix objects.
- base::rowsum in the **base** package for the default rowsum method.
- dgCMatrix objects in the **Matrix** package.

Examples

```
m0 <- rsparsematrix(1e5, 800, density=0.15)  # sparse representation
m <- as.matrix(m0)                           # dense representation
group <- sample(20, nrow(m), replace=TRUE)

## 'rowsum(m0)' is about 4x faster than 'rowsum(m)':
rs0 <- rowsum(m0, group)
rs <- rowsum(m, group)

stopifnot(identical(rs0, rs))
```

---

write_block *Write array blocks*

Description

Use write_block to write a block of array data to a RealizationSink object. The function is typically used in the context of block processing of array-like objects (typically DelayedArray objects but not necessarily).

Usage

```
write_block(sink, viewport, block)

## Backend-agnostic RealizationSink constructor:
AutoRealizationSink(dim, dimnames=NULL, type="double", as.sparse=FALSE)

## Get/set the "automatic realization backend":
getAutoRealizationBackend()
setAutoRealizationBackend(BACKEND=NULL)
supportedRealizationBackends()
```

Arguments

`sink` A **writable** array-like object, typically a RealizationSink derivative. Some important notes:

- DelayedArray objects are NEVER writable, even when they don't carry delayed operations (e.g. HDF5Array objects from the **HDF5Array** package), even when they don't carry delayed operations and have all their data in memory (e.g. RleArray objects).
- RealizationSink is a virtual class so `sink` must be an instance of a **concrete** RealizationSink subclass (e.g. an HDF5RealizationSink object from the **HDF5Array** package).
- RealizationSink derivatives are considered array-like objects i.e. they have dimensions and possibly dimnames.

Although `write_block()` will typically be used on a RealizationSink derivative, it can also be used on an ordinary array or other writable in-memory array-like arrays like dgCMatrix objects from the **Matrix** package.

`viewport` An ArrayViewport object compatible with `sink`, that is, such that `refdim(viewport)` is identical to `dim(sink)`.

`block` An ordinary (dense) array or SparseArraySeed object of the same dimensions as `viewport`.

`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.

`as.sparse` Whether the data should be written as sparse or not to the RealizationSink object to create. Not all *realization backends* support this.

`BACKEND` `NULL` (the default), or a single string specifying the name of a realization backend e.g. `"HDF5Array"` or `"RleArray"` etc...

Details

*** The RealizationSink API ***

The DelayedArray package provides a simple API for writing blocks of array data to disk (or to memory): the "RealizationSink API". This API allows the developper to write code that is agnostic about the particular on-disk (or in-memory) format being used to store the data.

Here is how to use it:

1. Create a realization sink.
2. Write blocks of array data to the realization sink with one or several calls to `write_block()`.
3. Close the realization sink with `close()`.
4. Coerce the realization sink to DelayedArray.

A realization sink is formally represented by a RealizationSink object. Note that RealizationSink is a virtual class with various concrete subclasses like HDF5RealizationSink from the **HDF5Array** package, or RleRealizationSink. Each subclass implements the "RealizationSink API" for a specific realization backend.

To create a realization sink, use the specific constructor function. This function should be named as the class itself e.g. `HDF5RealizationSink()`.

To create a realization sink in a backend-agnostic way, use `AutoRealizationSink()`. It will create a RealizationSink object for the current *automatic realization backend* (see below).

Once writing to the realization sink is completed, the RealizationSink object must be closed (with `close(sink)`), then coerced to DelayedArray (with `as(sink,"DelayedArray")`. What specific DelayedArray derivative this coercion will return depends on the specific class of the RealizationSink object. For example, if `sink` is an HDF5RealizationSink object from the **HDF5Array** package, then `as(sink,"DelayedArray")` will return an HDF5Array instance (the HDF5Array class is a DelayedArray subclass).

*** The *automatic realization backend* ***

The *automatic realization backend* is a user-controlled global setting that indicates what specific RealizationSink object `AutoRealizationSink()` should return. In the context of block processing of a DelayedArray object, this controls where/how realization happens e.g. as an ordinary array if not set (i.e. set to `NULL`), or as an HDF5Array object if set to `"HDF5Array"`, or as an RleArray object if set to `"RleArray"`, etc...

Use `getAutoRealizationBackend()` or `setAutoRealizationBackend()` to get or set the *automatic realization backend*.

Use `supportedRealizationBackends()` to get the list of realization backends that are currently supported.

*** Cross realization backend compatibility ***

Two important things to keep in mind for developers aiming at writing code that is compatible across realization backends:

- Realization backends don't necessarily support concurrent writing.

  More precisely: Even though it is safe to assume that any DelayedArray object will support concurrent `read_block()` calls, it is not so safe to assume that any RealizationSink derivative will support concurrent calls to `write_block()`. For example, at the moment, HDF5RealizationSink objects do not support concurrent writing.

  This means that in order to remain compatible across realization backends, code that contains calls to `write_block()` should NOT be parallelized.

- Some realization backends are "linear write only", that is, they don't support *random write access*, only *linear write access*.

  Such backends will provide a relization sink where the blocks of data must be written in linear order (i.e. by ascending rank). Furthermore, the geometry of the blocks must also be compatible with *linear write access*, that is, they must have a "first-dim-grows-first" shape. Concretely this means that the grid used to walk on the relization sink must be created with something like:

  ```
  colAutoGrid(sink)
  ```

for a two-dimensional sink, or with something like:

```
defaultAutoGrid(sink, block.shape="first-dim-grows-first")
```

for a sink with an arbitrary number of dimensions.

See ?`defaultAutoGrid` for more information.

For obvious reasons, "linear write only" realization backends do not support concurrent writing.

Value

For `write_block()`, the modified array-like object `sink`.

For `AutoRealizationSink()`, a RealizationSink object for the current *automatic realization backend*.

For `getAutoRealizationBackend`, `NULL` (no backend set yet) or a single string specifying the name of the *automatic realization backend* currently in use.

For `supportedRealizationBackends`, a data frame with 1 row per supported realization backend.

See Also

- ArrayViewport objects.
- SparseArraySeed objects.
- `read_block`.
- `blockApply` and family for convenient block processing of an array-like object.
- `defaultAutoGrid` and family to generate automatic grids to use for block processing of array-like objects.
- HDF5RealizationSink objects in the **HDF5Array** package.
- HDF5-dump-management in the **HDF5Array** package to control the location and physical properties of automatically created HDF5 datasets.
- RleArray objects.
- DelayedArray objects.
- array objects in base R.

Examples

```
## ---------------------------------------------------------------------
## USING THE "RealizationSink API": EXAMPLE 1
## ---------------------------------------------------------------------

## -- STEP 1 --
## Create a realization sink. Note that instead of creating a
## realization sink by calling a backend-specific sink constructor
## (e.g. HDF5Array::HDF5RealizationSink), we set the "automatic
## realization backend" to "HDF5Array" and use backend-agnostic
## constructor AutoRealizationSink():
setAutoRealizationBackend("HDF5Array")
sink <- AutoRealizationSink(c(35L, 50L, 8L))
dim(sink)

## -- STEP 2 --
## Define the grid of viewports to walk on. Here we define a grid made
## of very small viewports on 'sink'. Note that, for real-world use cases,
```

```
## block processing will typically use grids made of much bigger
## viewports, usually obtained with defaultAutoGrid() or family.
## Also please note that this grid would not be compatible with "linear
## write only" realization backends. See "Cross realization backend
## compatibility" above in this man page for more information.
sink_grid <- RegularArrayGrid(dim(sink), spacings=c(20, 20, 4))

## -- STEP 3 --
## Walk on the grid, and, for each of viewport, write random data to it.
for (bid in seq_along(sink_grid)) {
    viewport <- sink_grid[[bid]]
    block <- array(runif(length(viewport)), dim=dim(viewport))
    sink <- write_block(sink, viewport, block)
}

## -- An alternative to STEP 3 --
FUN <- function(viewport, sink) {
    block <- array(runif(length(viewport)), dim=dim(viewport))
    write_block(sink, viewport, block)
}
sink <- viewportReduce(FUN, sink_grid, sink, verbose=TRUE)

## -- STEP 4 --
## Close the sink and turn it into a DelayedArray object:
close(sink)
A <- as(sink, "DelayedArray")
A

setAutoRealizationBackend()  # unset automatic realization backend

## ---------------------------------------------------------------------
## USING THE "RealizationSink API": EXAMPLE 2
## ---------------------------------------------------------------------

## Say we have a 3D array and want to collapse its 3rd dimension by
## summing the array elements that are stacked vertically, that is, we
## want to compute the matrix M obtained by doing sum(A[i, j, ]) for all
## valid i and j. This is very easy to do with an ordinary array:
collapse_3rd_dim <- function(a) apply(a, MARGIN=1:2, sum)

## or, in a slightly more efficient way:
collapse_3rd_dim <- function(a) {
    m <- matrix(0, nrow=nrow(a), ncol=ncol(a))
    for (z in seq_len(dim(a)[[3]]))
        m <- m + a[ , , z]
    m
}

## With a toy 3D array:
a <- array(runif(8000), dim=c(25, 40, 8))
dim(collapse_3rd_dim(a))
stopifnot(identical(sum(a), sum(collapse_3rd_dim(a))))  # sanity check

## Now say that A is so big that even M wouldn't fit in memory. This is
## a situation where we'd want to compute M block by block:

## -- STEP 1 --
```

```
## Create the 2D realization sink:
setAutoRealizationBackend("HDF5Array")
sink <- AutoRealizationSink(dim(a)[1:2])
dim(sink)

## -- STEP 2 --
## Define two grids: one for 'sink' and one for 'a'. Since we're going
## to walk on the two grids simultaneously, read a block from 'a' and
## write it to 'sink', we need to make sure that we define grids that
## are "aligned". More precisely the two grids must have the same number
## of viewports and the viewports in one must correspond to the viewports
## in the other one:
sink_grid <- colAutoGrid(sink, ncol=10)
a_spacings <- c(dim(sink_grid[[1L]]), dim(a)[[3]])
a_grid <- RegularArrayGrid(dim(a), spacings=a_spacings)
dims(sink_grid)  # dimensions of the individual viewports
dims(a_grid)     # dimensions of the individual viewports

## Here is how to check that the two grids are "aligned":
stopifnot(identical(length(sink_grid), length(a_grid)))
stopifnot(identical(dims(sink_grid), dims(a_grid)[ , -3]))

## -- STEP 3 --
## Walk on the two grids simultaneously:
for (bid in seq_along(sink_grid)) {
    ## Read block from 'a'.
    a_viewport <- a_grid[[bid]]
    block <- read_block(a, a_viewport)
    ## Collapse it.
    block <- collapse_3rd_dim(block)
    ## Write the collapsed block to 'sink'.
    sink_viewport <- sink_grid[[bid]]
    sink <- write_block(sink, sink_viewport, block)
}

## -- An alternative to STEP 3 --
FUN <- function(sink_viewport, sink) {
    ## Read block from 'a'.
    bid <- currentBlockId()
    a_viewport <- a_grid[[bid]]
    block <- read_block(a, a_viewport)
    ## Collapse it.
    block <- collapse_3rd_dim(block)
    ## Write the collapsed block to 'sink'.
    write_block(sink, sink_viewport, block)
}
sink <- viewportReduce(FUN, sink_grid, sink, verbose=TRUE)

## -- STEP 4 --
## Close the sink and turn it into a DelayedArray object:
close(sink)
M <- as(sink, "DelayedArray")
M

## Sanity check:
stopifnot(identical(collapse_3rd_dim(a), as.array(M)))
```

```
setAutoRealizationBackend()  # unset automatic realization backend

## ---------------------------------------------------------------------
## USING THE "RealizationSink API": AN ADVANCED EXAMPLE
## ---------------------------------------------------------------------

## 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 get to the "for" loop, we need 4 things:

## 1. Two grids of blocks, one on R and one on G. The blocks in the
##    two grids must contain the same number of columns. We arbitrarily
##    choose to use blocks of 150 columns:
R_grid <- colAutoGrid(R, ncol=150)
G_grid <- colAutoGrid(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:
setAutoRealizationBackend("HDF5Array")
U_sink <- AutoRealizationSink(c(200L, 1000L))
```

```
## 4. Finally, we create a grid on U_sink with viewports that contain
##    the same number of columns as the corresponding blocks in R and G:
U_grid <- colAutoGrid(U_sink, ncol=150)

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

## 5. Now we can proceed. We use a "for" loop to walk on R and G
##    simultaneously, block by block, apply pFUN(), and write the
##    output of pFUN() to U_sink:
for (bid in seq_along(U_grid)) {
    R_block <- read_block(R, R_grid[[bid]])
    G_block <- read_block(G, G_grid[[bid]])
    U_block <- pFUN(R_block, G_block)
    U_sink <- write_block(U_sink, U_grid[[bid]], U_block)
}

## An alternative to the "for" loop is to use viewportReduce():
FUN <- function(U_viewport, U_sink) {
    bid <- currentBlockId()
    R_block <- read_block(R, R_grid[[bid]])
    G_block <- read_block(G, G_grid[[bid]])
    U_block <- pFUN(R_block, G_block)
    write_block(U_sink, U_viewport, U_block)
}
U_sink <- viewportReduce(FUN, U_grid, U_sink, verbose=TRUE)

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

setAutoRealizationBackend()  # unset automatic realization backend

## ---------------------------------------------------------------------
## VERY BASIC (BUT ALSO VERY ARTIFICIAL) USAGE OF THE
## read_block()/write_block() COMBO
## ---------------------------------------------------------------------

###### On an ordinary matrix ######
m1 <- matrix(1:30, ncol=5)

## Define a viewport on 'm1':
block1_dim <- c(4, 3)
viewport1 <- ArrayViewport(dim(m1), IRanges(c(3, 2), width=block1_dim))

## Read/tranform/write:
block1 <- read_block(m1, viewport1)
write_block(m1, viewport1, block1 + 1000L)

## Define another viewport on 'm1':
viewport1b <- ArrayViewport(dim(m1), IRanges(c(1, 3), width=block1_dim))

## Read/tranform/write:
write_block(m1, viewport1b, block1 + 1000L)

## No-op:
```

```
m <- write_block(m1, viewport1, read_block(m1, viewport1))
stopifnot(identical(m1, m))

########## On a 3D array ##########
a3 <- array(1:60, 5:3)

## Define a viewport on 'a3':
block3_dim <- c(2, 4, 1)
viewport3 <- ArrayViewport(dim(a3), IRanges(c(1, 1, 3), width=block3_dim))

## Read/tranform/write:
block3 <- read_block(a3, viewport3)
write_block(a3, viewport3, block3 + 1000L)

## Define another viewport on 'a3':
viewport3b <- ArrayViewport(dim(a3), IRanges(c(3, 1, 3), width=block3_dim))

## Read/tranform/write:
write_block(a3, viewport3b, block3 + 1000L)

## No-op:
a <- write_block(a3, viewport3, read_block(a3, viewport3))
stopifnot(identical(a3, a))

## ---------------------------------------------------------------------
## LESS BASIC (BUT STILL VERY ARTIFICIAL) USAGE OF THE
## read_block()/write_block() COMBO
## ---------------------------------------------------------------------

grid1 <- RegularArrayGrid(dim(m1), spacings=c(3L, 2L))
grid1
length(grid1)  # number of blocks defined by the grid
read_block(m1, grid1[[3L]])  # read 3rd block
read_block(m1, grid1[[1L, 3L]])

## Walk on the grid, colum by column:
m1a <- m1
for (bid in seq_along(grid1)) {
    viewport <- grid1[[bid]]
    block <- read_block(m1a, viewport)
    block <- bid * 1000L + block
    m1a <- write_block(m1a, viewport, block)
}
m1a

## Walk on the grid, row by row:
m1b <- m1
for (i in seq_len(dim(grid1)[[1]])) {
  for (j in seq_len(dim(grid1)[[2]])) {
    viewport <- grid1[[i, j]]
    block <- read_block(m1b, viewport)
    block <- (i * 10L + j) * 1000L + block
    m1b <- write_block(m1b, viewport, block)
  }
}
m1b
```

```
## ---------------------------------------------------------------------
## supportedRealizationBackends() AND FAMILY
## ---------------------------------------------------------------------

getAutoRealizationBackend()  # no backend set yet

supportedRealizationBackends()
setAutoRealizationBackend("HDF5Array")
getAutoRealizationBackend()  # backend is set to "HDF5Array"
supportedRealizationBackends()

getHDF5DumpChunkLength()
setHDF5DumpChunkLength(500L)
getHDF5DumpChunkShape()

sink <- AutoRealizationSink(c(120L, 50L))
class(sink)  # HDF5-specific realization sink
dim(sink)
chunkdim(sink)

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

close(sink)
A <- as(sink, "DelayedArray")
A

setAutoRealizationBackend()  # unset automatic realization backend
```

Index