Package 'S4Arrays'

May 26, 2026

Title Foundation of array-like containers in Bioconductor

Description The S4Arrays package defines the Array virtual class to be extended by other S4 classes that wish to implement a container with an array-like semantic. It also provides: (1) low-level functionality meant to help the developer of such container to implement basic operations like display, subsetting, or coercion of their array-like objects to an ordinary matrix or array, and (2) a framework that facilitates block processing of array-like objects (typically on-disk objects).

biocViews Infrastructure, DataRepresentation

URL https://bioconductor.org/packages/S4Arrays

BugReports https://github.com/Bioconductor/S4Arrays/issues

Version 1.12.0

License Artistic-2.0

Encoding UTF-8

Depends R (>= 4.3.0), methods, Matrix, abind, BiocGenerics (>= 0.45.2), S4Vectors (>= 0.47.6), IRanges

Imports stats

LinkingTo S4Vectors

Suggests BiocParallel, SparseArray (>= 0.0.4), DelayedArray, HDF5Array, testthat, knitr, rmarkdown, BiocStyle

VignetteBuilder knitr

Collate utils.R rowsum.R abind.R aperm2.R array_selection.R Nindex-utils.R arep.R array_recycling.R Array-class.R dim-tuning-utils.R Array-subsetting.R Array-subassignment.R ArrayGrid-class.R mapToGrid.R extract_array.R type.R is_sparse.R read_block.R write_block.R show-utils.R Array-kronecker-methods.R zzz.R

git_url https://git.bioconductor.org/packages/S4Arrays

git_branch RELEASE_3_23

git_last_commit b1246fd

git_last_commit_date 2026-04-28

Repository Bioconductor 3.23

Date/Publication 2026-05-25

Author Hervé Pagès [aut, cre] (ORCID: <https://orcid.org/0009-0002-8272-4522>), Jacques Serizay [ctb]

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

Contents

`aperm2` *Generalized permutation of the dimensions of an array*

Description

`aperm2()` extends the functionality of `base::aperm()` by allowing dropping and/or adding *ineffective dimensions* (i.e. dimensions with an extent of 1) from/to the input array, in addition to permuting its dimensions.

Note that, like `base::aperm()`, `aperm2()` always preserves the length of the input array. However, unlike with `base::aperm()`, the array returned by `aperm2()` doesn't necessarily have the same number of dimensions as the input array.

Usage

```
aperm2(a, perm)
```

Arguments

a	An ordinary array.
perm	An integer vector, possibly containing NAs, indicating how the dimensions of the returned array should be mapped to those of the input array. More precisely, perm can be one of the following: • A permutation of the seq_along(dim(a)) vector, like for base::aperm(). Note that if the identity permutation is used (i.e. perm=seq_along(dim(a))), then aperm2() is a no-op (like with base::aperm()). • A permutation of a *subset* of the seq_along(dim(a)) vector. In this case the dimensions that are excluded must be *ineffective dimensions* i.e. each of them must have an extent of 1. In other words, only integers that belong to which(dim(a) == 1) can be missing from perm. In this case, the ineffective dimensions that are excluded will be dropped i.e. they won't be carried over to the returned array. • Additionally, any number of NAs can be inserted anywhere in a perm vector like one described above. In this case, ineffective dimensions will be added to the returned array. These added dimensions will materialize as additional 1's in the dim() vector of the returned array, at positions that match the positions of the NAs in perm. Note that if perm is missing, then aperm2(a) reverses the order of a's dimensions (i.e. perm gets set to rev(seq_along(dim(a)))), like base::aperm(a) does.

Value

An array with one dimension per element in the perm argument. The length of the returned array will always be the same as the length of the input array. (Note that for an array a, length(a) is prod(dim(a)).)

Note

The aperm() method for DelayedArray objects defined in the **DelayedArray** package implements the "aperm2 semantic", that is, it allows dropping and/or adding *ineffective dimensions* from/to the input DelayedArray object.

See Also

- aperm in the **base** package for the function that aperm2 is based on.
- aperm in the **BiocGenerics** package for the aperm *S4 generic function*.
- aperm,SVT_SparseArray-method in the **SparseArray** package and aperm,DelayedArray-method in the **DelayedArray** package for aperm() methods that implements the "aperm2 semantic".

Examples

```
## ---------------------------------------------------------------------
## SOME EXAMPLES WITH A 4D ARRAY
```

```
## ------------------------------------------------------------------

a <- array(1:72, c(3, 6, 1, 4),
           dimnames=list(NULL, letters[1:6], NULL, LETTERS[1:4]))
a

## Permute first two dimensions:
aperm2(a, perm=c(2,1,3,4))

## Permute first and last dimensions:
aperm2(a, perm=c(4,2,3,1))

## Drop 3rd dimension:
aperm2(a, perm=c(1,2,4))

## Drop 3rd dimension and permute 2nd and last:
aperm2(a, perm=c(1,4,2))

## Drop 3rd dimension and cycle the order of the remaining ones:
aperm2(a, perm=c(2,4,1))

## Add one ineffective dimension:
aperm2(a, perm=c(NA,1,2,3,4))
aperm2(a, perm=c(1,NA,2,3,4))
aperm2(a, perm=c(1,2,NA,3,4))
aperm2(a, perm=c(1,2,3,NA,4))
aperm2(a, perm=c(1,2,3,4,NA))

## Add four ineffective dimensions:
aperm2(a, perm=c(NA,1,2,3,NA,NA,4,NA))

## Permute first and last dimensions and add one ineffective dimension:
aperm2(a, perm=c(4,2,3,NA,1))

## Drop 3rd dimension, cycle the order of the remaining ones, and add
## two ineffective dimensions:
aperm2(a, perm=c(2,4,NA,1,NA))

## No-op:
aperm2(a, perm=seq_along(dim(a)))

## Reverse the order of the dimensions (multidimensional transposition):
aperm2(a)  # same as 'aperm2(a, perm=rev(seq_along(dim(a))))'

## ------------------------------------------------------------------------
## COMPOSING aperm2() TRANSFORMATIONS
## ------------------------------------------------------------------------

## Applying two successive aperm() transformations, first with 'perm'
## set to 'perm1' then set to 'perm2', is equivalent to applying a
## single aperm() transformation with 'perm' set to 'perm1[perm2]'.
##
## More formally:
```

```
##     aperm(aperm(a, perm=perm1), perm=perm2)
## is equivalent to:
##     aperm(a, perm=perm1[perm2])
##
## Note that this also applies to aperm2()!

## Examples with aperm():

perm1 <- c(2,4,3,1)
perm2 <- c(4,3,2,1)
perm3 <- c(2,1,4,3)

a12 <- aperm(aperm(a, perm=perm1), perm=perm2)
stopifnot(identical(a12, aperm(a, perm=perm1[perm2])))

a13 <- aperm(aperm(a, perm=perm1), perm=perm3)
stopifnot(identical(a13, aperm(a, perm=perm1[perm3])))

a23 <- aperm(aperm(a, perm=perm2), perm=perm3)
stopifnot(identical(a23, aperm(a, perm=perm2[perm3])))

a123 <- aperm(aperm(aperm(a, perm=perm1), perm=perm2), perm=perm3)
stopifnot(identical(a123, aperm(a, perm=perm1[perm2][perm3])))
stopifnot(identical(a123, aperm(a, perm=perm1[perm2[perm3]])))

## Examples with aperm2():

perm1 <- c(2,4,1)
perm2 <- c(1,3,NA,2,NA)
perm3 <- c(5,4,2,1)

a12 <- aperm2(aperm2(a, perm=perm1), perm=perm2)
stopifnot(identical(a12, aperm2(a, perm=perm1[perm2])))

a123 <- aperm2(aperm2(aperm2(a, perm=perm1), perm=perm2), perm=perm3)
stopifnot(identical(a123, aperm2(a, perm=perm1[perm2][perm3])))
stopifnot(identical(a123, aperm2(a, perm=perm1[perm2[perm3]])))

## ---------------------------------------------------------------------
## REVERSIBILITY OF THE aperm2() TRANSFORMATION
## ---------------------------------------------------------------------

## An aperm() or aperm2() transformation is always reversible.
## The 'perm' vector to use to achieve the reverse transformation
## can be inferred from the initial 'perm' vector using the following
## helper function ('n' must be the number of dimensions of
## the original array):
build_rev_perm <- function(perm, n=length(perm)) {
    rev_perm <- rep.int(NA_integer_, n)
    na_idx <- which(!is.na(perm))
    rev_perm[perm[na_idx]] <- na_idx
    rev_perm
}
```

```
## Examples:

perm <- c(2,4,NA,1,NA)
rev_perm <- build_rev_perm(perm, n=length(dim(a)))
stopifnot(identical(aperm2(aperm2(a, perm=perm), perm=rev_perm), a))

## The "composed" 'perm' vector achieves identity:
perm[rev_perm]

## Sanity checks:

perm <- seq_len(10)
stopifnot(identical(build_rev_perm(perm), perm))

perm <- c(2:5,1L)
rev_perm <- build_rev_perm(perm)
stopifnot(identical(perm[rev_perm], seq_along(perm)))

perm <- c(5L,NA,2:4,NA,NA,1L)
rev_perm <- build_rev_perm(perm, n=6)
stopifnot(identical(perm[rev_perm], c(1:5,NA)))
```

`arep` *Replicate array elements*

Description

`arep_times()` and `arep_each()` are multidimensional versions of `base::rep( , times=)` and `base::rep( , each=)`, respectively.

They're both generic functions with default methods that work on any array-like object that supports `[` (single-bracket subsetting).

Usage

```
arep_times(x, times)
arep_each(x, each)
```

Arguments

`x`	An array-like object, that is, an ordinary array or any object with dimensions.
`times`, `each`	An integer vector *parallel* to `dim(x)` i.e. with one element per dimension in `x`.

Value

An array-like object of the same class as `x` and with dimensions `dim(x) * times` for `arep_times` or `dim(x) * each` for `arep_each`. The dimnames on `x` are propagated, if any.

See Also

- base::rep in the **base** package.
- array and matrix objects in base R.

Examples

```
m <- matrix(1:10, nrow=2)
arep_times(m, c(4, 2))
arep_each(m, c(4, 2))

## Note that the output array is 'prod(times)' (or 'prod(each)') times
## bigger than the input array!
```

---

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 **SparseArray** or **DelayedArray** packages. End users typically don't need them for their regular use of SparseArray or 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 in the context of the **SparseArray** and **DelayedArray** packages: *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 that are extensively used in the context of the **SparseArray** and **DelayedArray** packages:

1. *Linear index* (or *L-index* or *Lindex*): A numeric vector 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. 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 (i.e. non-NULL subscripts are integer
## vectors with positive values only):

Nindex <- list(c(1, 4, 1), NULL, 1)
## Same as a[c(1, 4, 1), , 1, drop=FALSE]:
S4Arrays:::subset_by_Nindex(a, Nindex)

Nindex <- list(integer(0), NULL, 1)
## Same as a[integer(0), , 1, drop=FALSE]:
S4Arrays:::subset_by_Nindex(a, Nindex)

## Non-normalized N-index:

Nindex <- list(-3, NULL, c(TRUE, FALSE, FALSE))
Nindex <- S4Arrays:::normalize_Nindex(Nindex, a)
## Same as a[-3, , 1, drop=FALSE]:
S4Arrays:::subset_by_Nindex(a, Nindex)

Nindex <- list(IRanges(2, 4), NULL, 1)
Nindex <- S4Arrays:::normalize_Nindex(Nindex, a)
## Same as a[2:4, , 1, drop=FALSE]:
S4Arrays:::subset_by_Nindex(a, Nindex)

dimnames(a)[[1]] <- LETTERS[1:4]
Nindex <- list(c("D", "B"), NULL, 1)
Nindex <- S4Arrays:::normalize_Nindex(Nindex, a)
## Same as a[c("D", "B"), , 1, drop=FALSE]:
S4Arrays:::subset_by_Nindex(a, Nindex)
```

Value

Lindex2Mindex returns an M-index.

Mindex2Lindex returns an L-index.

See Also

arrayInd in the **base** package.

Examples

```
## ------------------------------------------------------------------------
## M-index vs L-index
## ------------------------------------------------------------------------

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

## The same "array selection" can be represented by an M-index or
## an L-index. Here we use both representations to select the same
## 4 array elements:
Mindex <- rbind(c(3, 2, 1),
                c(2, 1, 1),
                c(4, 3, 2),
                c(2, 1, 1))
a[Mindex]

Lindex <- c(7, 2, 24, 2)
a[Lindex]

## Sanity check:
stopifnot(identical(a[Mindex], a[Lindex]))

## ------------------------------------------------------------------------
## Convert back and forth between M-index and L-index representation
## ------------------------------------------------------------------------

Mindex2Lindex(Mindex, dim(a))  # L-index

Lindex2Mindex(Lindex, dim(a))  # M-index

## Sanity checks:
storage.mode(Mindex) <- storage.mode(Lindex) <- "integer"
stopifnot(identical(Mindex2Lindex(Mindex, dim(a)), Lindex))
stopifnot(identical(Lindex2Mindex(Lindex, dim(a)), Mindex))

## ------------------------------------------------------------------------
## More Mindex2Lindex() 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))
```

`Array-class` *Array objects*

Description

Array is a virtual class with no slots intended to be extended by concrete subclasses with an array-like semantic.

Details

Some examples of Array derivatives:

- SparseArray objects implemented in the **SparseArray** package.
- DelayedArray objects implemented in the **DelayedArray** package.
- ArrayGrid and ArrayViewport objects implemented in this package (the **S4Arrays** package).

See Also

- array and matrix objects in base R.

Examples

```
showClass("Array")  # virtual class with no slots
```

Array-kronecker-methods

Kronecker products on Array objects

Description

The **S4Arrays** package implements `kronecker()` methods for Array objects that work out-of-the-box on Array derivatives that support `[` and `*`.

Note that `kronecker` is a generic function defined in the **methods** package but documented in the **base** package. See `?base::kronecker`.

Usage

```
## S4 method for signature 'Array,ANY'
kronecker(X, Y, FUN="*", make.dimnames=FALSE, ...)
## S4 method for signature 'ANY,Array'
kronecker(X, Y, FUN="*", make.dimnames=FALSE, ...)
## S4 method for signature 'Array,Array'
kronecker(X, Y, FUN="*", make.dimnames=FALSE, ...)

## The workhorse behind the three above methods.
kronecker2(X, Y, FUN="*", make.dimnames=FALSE, ...)
```

Arguments

`X`, `Y`
: Array-like objects. Alternatively `X` and/or `Y` can be vectors, in which case they are converted to 1D-arrays with `as.array()`.

  Note that `X` and `Y` are expected to have the same number of dimensions. However, when they don't, *ineffective dimensions* (i.e. dimensions with an extent of 1) are added to the object with less dimensions.

  For the S4 methods, at least one of `X` or `Y` must be an Array derivative.

`FUN`, `make.dimnames`, `...`
: See `?base::kronecker` for a description of these arguments.

Details

The three `kronecker()` methods listed above delegate to `kronecker2()`, a re-implementation of `base::kronecker()`.

`kronecker2()` is semantically equivalent to `base::kronecker`. However, unlike the latter which calls `as.array()` on `X` and `Y` internally, the former *operates natively* on the input objects regardless of their internal representations, as long as they are array-like objects that support `[` (single-bracket subsetting) and `*`. In particular, when `X` and `Y` use the same internal representations, the returned object will also use that representation. In other words, the output object will have the same class as the input objects (*endomorphism*).

The *endomorphism* property is particularly important when the input objects are sparse (e.g. SVT_SparseArray objects from the **SparseArray** package) or when they use an on-disk representation (e.g. DelayedArray objects from the **DelayedArray** package). For example, if `X` and `Y` are DelayedArray objects, the returned object is another DelayedArray object. Also in that case, calling `kronecker2()` is virtually instantaneous because all the operations that the function performs internally on `X` and `Y` by the are delayed!

Value

An array-like object with dimensions `dim(X) * dim(Y)`.

See Also

- `base::kronecker` in the **base** package.
- The "Kronecker product" page on Wikipedia: https://en.wikipedia.org/wiki/Kronecker_product
- The Array class.
- SparseArray objects implemented in the **SparseArray** package.
- DelayedArray objects implemented in the **DelayedArray** package.
- TENxMatrix objects implemented in the **HDF5Array** package.

Examples

```
## ---------------------------------------------------------------------
## SIMPLE kronecker2() EXAMPLES
## ---------------------------------------------------------------------

m1 <- matrix(1:10, nrow=2)     # 2 x 5 matrix
m2 <- matrix(101:106, nrow=3)  # 3 x 2 matrix
kronecker2(m1, m2)             # 6 x 10 matrix

a1 <- array(1:16, dim=c(4, 2, 2))
a2 <- array(1:30, dim=c(3, 5, 2))
kronecker2(a1, a2)    # 12 x 10 x 4 array

## The Kronecker product is **not** commutative:
m1 <- matrix(LETTERS[1:10], nrow=2)
m2 <- matrix(letters[1:6], nrow=3)
kronecker2(m1, m2, paste, sep="*")
kronecker2(m2, m1, paste, sep="*")

## Sanity checks:
stopifnot(
  identical(kronecker2(m1, m2, paste0), kronecker(m1, m2, paste0)),
  identical(kronecker2(m2, m1, paste0), kronecker(m2, m1, paste0))
)

## ---------------------------------------------------------------------
## USING kronecker() ON Array DERIVATIVES
## ---------------------------------------------------------------------
```

```
## The user should typically avoid direct calls to kronecker2() and
## stick to kronecker(). Because this is a generic function, it will
## dispatch to the appropriate method based on the classes of the input
## objects. If one of them is an Array derivative, kronecker2() will
## be called thanks to the methods defined in the S4Arrays package and
## listed in the Usage section above.

## With SparseMatrix objects (Array derivatives):
library(SparseArray)
sm1 <- poissonSparseMatrix(300, 15, density=0.25)
sm2 <- poissonSparseMatrix(80, 500, density=0.15)
kronecker2(sm1, sm2)  # 24000 x 7500 SparseMatrix object

## With TENxMatrix objects (DelayedArray derivatives, therefore also
## Array derivatives):
library(HDF5Array)
M1 <- writeTENxMatrix(sm1)  # 300 x 15 TENxMatrix object
M2 <- writeTENxMatrix(sm2)  # 80 x 500 TENxMatrix object
K <- kronecker2(M1, M2)     # instantaneous! (all operations are delayed)
showtree(K)  # show delayed operations details

## ---------------------------------------------------------------------
## VERIFYING THE MIXED-PRODUCT PROPERTY (JUST FOR FUN!)
## ---------------------------------------------------------------------
## See https://en.wikipedia.org/wiki/Kronecker_product for details
## about "The mixed-product property".

## We verify the property on 4 random matrices:
A <- matrix(runif(1000), ncol=40)
B <- matrix(runif(800), ncol=100)
C <- matrix(runif(600), nrow=40)
D <- matrix(runif(5000), nrow=100)

kAB <- kronecker2(A, B)
kCD <- kronecker2(C, D)
kAB_x_kCD <- kAB %*% kCD
A_x_C <- A %*% C
B_x_D <- B %*% D
stopifnot(all.equal(kAB_x_kCD, kronecker2(A_x_C, B_x_D)))

## The mixed-product property also for the element-wise product
## (Hadamard product). We verify this on 4 random arrays:
A <- array(1:60, dim=5:3)
B <- array(101:180, dim=c(2,10,4))
C <- array(runif(60), dim=5:3)
D <- array(runif(80), dim=c(2,10,4))

kAB <- kronecker2(A, B)
kCD <- kronecker2(C, D)
kAB_o_kCD <- kAB * kCD
A_o_C <- A * C
B_o_D <- B * D
stopifnot(all.equal(kAB_o_kCD, kronecker2(A_o_C, B_o_D)))
```

Array-subassignment *Low-level internal generics involved in subassignment of Array derivatives*

Description

The **S4Arrays** package defines a small set of low-level generic functions that are involved in subassignment operations on Array derivatives. They are not intended to be used directly by the end user.

See Also

- The Array class.

Array-subsetting *Low-level internal generics involved in subsetting of Array derivatives*

Description

The **S4Arrays** package defines a small set of low-level generic functions that are involved in subsetting operations on Array derivatives. They are not intended to be used directly by the end user.

See Also

- The Array class.

ArrayGrid-class *ArrayGrid and ArrayViewport objects*

Description

A *grid* is a partitioning of an array-like object into block-shaped regions called *viewports*.

The **S4Arrays** package defines two S4 classes to formally represent grids and viewports: the ArrayGrid and ArrayViewport classes. Note that ArrayGrid and ArrayViewport objects are used extensively in the context of block processing of array-like objects.

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.

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` to read a block of data from an array-like object.
- `blockApply` and family, in the **DelayedArray** package, for convenient block processing of an array-like object.
- `mapToGrid` for mapping reference array positions to grid positions and vice-versa.
- array and matrix 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]]
stopifnot(is(grid1[[1L]], "ArrayViewport"))
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)
))
```

`array_recycling` *Multidimensional array recycling*

Description

In base R, arithmetic and other binary operations between a matrix or array and a vector don't let the user control how the latter should be recycled: it's always recycled along the first dimension of the

matrix or array. This has led users to use various tricks when they need recycling along the second dimension (a.k.a. "horizontal recycling"), like the popular "double-transposition" trick: `t(t(m) / colSums(m))`. However this is not only inelegant, it's also inefficient.

`as_tile()` is meant to address that.

It also allows arithmetic and other binary operations between two arrays of distinct dimensions (rejected as "non-conformable" by arithmetic operations in base R), typically between a small one (the tile) and a bigger one, as long as their geometries are compatible.

Usage

```
as_tile(x, along=1L, dim=NULL)
```

Arguments

`x`	An array-like object or a vector.
`along`	Can only be used when `x` is a vector. Must be a single positive integer indicating the "orientation" of the tile to be created, that is, the dimension along which `x` will be recycled by arithmetic operations and other binary operations.
`dim`	`NULL` or the dimensions (supplied as an integer vector) of the tile to be created.

Value

A tile object (tile is an extension of array).

See Also

- `base::colSums` in the **base** package.
- array and matrix objects in base R.

Examples

```
## ---------------------------------------------------------------------
## 2D EXAMPLES
## ---------------------------------------------------------------------
m0 <- matrix(1:54, nrow=6)
x <- c(-1, 0, 100)

## Arithmetic operations in base R recycle 'x' along the first dimension
## of the matrix ("vertical recycling"):
m0 * x

## To recycle 'x' along the second dimension of the matrix ("horizontal
## recycling"), we turn it into an "horizontal" tile:
t <- as_tile(x, along=2)
t

m0 * t

## The above produces the same result as the double-transposition trick
## but is more efficient (and also makes code easier to read):
```

```
stopifnot(identical(m0 * t, t(t(m0) * x)))

## A less artificial example:
cs0 <- colSums(m0)
m <- m0 / as_tile(cs0, along=2)

stopifnot(all.equal(colSums(m), rep(1, ncol(m))))  # sanity check

## Using an arbitrary 2D tile:

t <- m0[1:2, 1:3]
t
## Unfortunately arithmetic operations in base R refuse to operate on
## arrays that don't have the same dimensions:
## Not run:
  m0 / t  # ERROR! (non-conformable arrays)

## End(Not run)

## Wrapping 't' in a tile object makes this work:
m0 / as_tile(t)

## ---------------------------------------------------------------------
## 3D EXAMPLES
## ---------------------------------------------------------------------
## Note that colSums() supports multidimensional arrays. In this case
## the user can use the 'dims' argument to control how the array should
## be sliced into "columns". See '?base::colSums' for the details.

a <- array(runif(300), dim=c(10, 5, 6))

## Using 'dims=2' indicates that the columns are the 2D slices obtained
## by slicing the array along its 3rd dimension. With this slicing, each
## column is a 10 x 5 matrix.
cs2 <- colSums(a, dims=2)
cs2  # vector of length 6 (one value per 2D slice)

t <- as_tile(cs2, along=3)
a / t

stopifnot(all.equal(colSums(a / t, dims=2), rep(1, 6)))  # sanity check

## By default (i.e. when 'dims=1') the array is considered to be made
## of 5*6 columns of length 10.
cs1 <- colSums(a)
cs1  # 5 x 6 matrix

t <- as_tile(cs1, dim=c(1L, dim(cs1)))
a / t

## Sanity check:
stopifnot(all.equal(colSums(a / t), matrix(1, nrow=5, ncol=6)))
```

bind-arrays *Combine multidimensional array-like objects*

Description

Bind multidimensional array-like objects along any dimension.

NOTE: This man page is for the `abind` *S4 generic function* defined in the **S4Arrays** package. See `?abind::abind` for the default method (defined in the **abind** package). Bioconductor packages can define specific methods for objects not supported by the default method.

Usage

```
## Bind array-like objects along any dimension:
abind(..., along=NULL, rev.along=NULL)

## Bind array-like objects along their first or second dimension:
arbind(...)
acbind(...)
```

Arguments

`...`	The array-like objects to bind.
`along`, `rev.along`	See `?abind::abind` for a description of these arguments.

Value

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

See Also

- `abind::abind` in the **abind** package for the default `abind` method.
- `rbind` and `cbind` in the **base** package for the corresponding operations on matrix-like objects.

Examples

```
a1 <- array(1:60, c(3, 5, 4),
            dimnames=list(NULL, paste0("A1y", 1:5), NULL))
a2 <- array(101:240, c(7, 5, 4),
            dimnames=list(paste0("A2x", 1:7), paste0("A2y", 1:5), NULL))
a3 <- array(10001:10100, c(5, 5, 4),
            dimnames=list(paste0("A3x", 1:5), NULL, paste0("A3z", 1:4)))

abind(a1, a2, a3, along=1)  # same as 'arbind(a1, a2, a3)'

m2 <- matrix(1:35, nrow=7)
abind(m2, a2, along=3)
abind(a2, m2, along=3)
```

```
abind(m2, m2+0.5, rev.along=0)  # same as 'abind(m2, m2+0.5, along=3)'
```

---

dim-tuning-utils *Internal "dim tuning" utilities*

Description

Internal "dim tuning" utilities not meant to be used directly by the end user.

---

extract_array *extract_array*

Description

extract_array is an internal generic function not intended to be used directly by the end user. It has methods defined for array, data.frame, DataFrame objects, and other array-like objects.

Note that extract_array is part of the *seed contract* as defined in the *Implementing A DelayedArray Backend* vignette from the **DelayedArray** package.

Usage

```
## The extract_array() S4 generic:

extract_array(x, index)

## extract_array() methods defined in the S4Arrays package:

## S4 method for signature 'ANY'
extract_array(x, index)

## S4 method for signature 'array'
extract_array(x, index)

## S4 method for signature 'data.frame'
extract_array(x, index)

## S4 method for signature 'DataFrame'
extract_array(x, index)
```

Arguments

x	An array-like object. This can be an ordinary array, a SparseArray object from the **SparseArray** package, a dgCMatrix object from the **Matrix** package, a DelayedArray object from the **DelayedArray** package, or any object with an array semantic (i.e. an object for which `dim(x)` is not NULL). Note that data.frame and DataFrame objects are also supported.
index	An unnamed list of integer vectors, one per dimension in `x`. Each vector is called a *subscript* and can only contain positive integers that are valid 1-based indices along the corresponding dimension in `x`. Empty or missing subscripts are allowed. They must be represented by list elements set to `integer(0)` or `NULL`, respectively. The subscripts cannot contain NAs or non-positive values. Individual subscripts are allowed to contain duplicated indices.

Details

`extract_array()` methods need to support empty or missing subscripts. For example, if `x` is an M x N matrix-like object, then `extract_array(x, list(NULL, integer(0)))` must return an M x 0 ordinary matrix, and `extract_array(x, list(integer(0), integer(0)))` a 0 x 0 ordinary matrix.

Also subscripts are allowed to contain duplicated indices so things like `extract_array(x, list(c(1:3, 3:1), 2L))` need to be supported.

Finally, for maximum efficiency, `extract_array()` methods should not try to do anything with the dimnames on `x`.

Value

An *ordinary* array of the same `type()` as `x`. For example, if `x` is an object representing an M x N matrix of complex numbers (i.e. `type(x) == "complex"`), then `extract_array(x, list(NULL, 2L))` must return the 2nd column in `x` as an M x 1 *ordinary* matrix of `type()` `"complex"`.

See Also

- `S4Arrays::type` to get the type of the elements of an array-like object.
- array and data.frame objects in base R.
- SparseArray objects implemented in the **SparseArray** package.
- DelayedArray objects implemented in the **DelayedArray** package.
- DataFrame objects implemented in the **S4Vectors** package.

Examples

```
extract_array
showMethods("extract_array")

## extract_array() works on array-like objects like SparseArray objects,
```

```
## dgCMatrix objects, DataFrame objects, etc...

## --- On a SparseArray object ---

library(SparseArray)
a <- array(0L, 5:3)
a[c(1:2, 8, 10, 15:17, 20, 24, 40, 56:60)] <- (1:15)*10L
svt <- as(a, "SparseArray")
svt

extract_array(svt, list(NULL, c(4L,2L,4L), 1L))
extract_array(svt, list(NULL, c(4L,2L,4L), 2:3))
extract_array(svt, list(NULL, c(4L,2L,4L), integer(0)))

## Sanity checks:
stopifnot(
  identical(extract_array(svt, list(NULL, c(4L,2L,4L), 1L)),
            as.array(svt)[ , c(4L,2L,4L), 1L, drop=FALSE]),
  identical(extract_array(svt, list(NULL, c(4L,2L,4L), 2:3)),
            as.array(svt)[ , c(4L,2L,4L), 2:3]),
  identical(extract_array(svt, list(NULL, c(4L,2L,4L), integer(0))),
            as.array(svt)[ , c(4L,2L,4L), integer(0)])
)

## --- On a dgCMatrix object ---

library(Matrix)
m <- a[ , , 1]
dgcm <- as(m, "dgCMatrix")
dgcm

extract_array(dgcm, list(NULL, c(4L,2L,4L)))

## Sanity check:
stopifnot(
  identical(extract_array(dgcm, list(NULL, c(4L,2L,4L))),
            as.matrix(dgcm)[ , c(4L,2L,4L)])
)

## --- On a data.frame or DataFrame object ---

df <- data.frame(a=44:49, b=letters[1:6], c=c(TRUE, FALSE))
DF <- as(df, "DataFrame")

extract_array(df, list(4:2, c(1L,3L)))
extract_array(DF, list(4:2, c(1L,3L)))

## Sanity check:
target <- as.matrix(df)[4:2, c(1L,3L)]
dimnames(target) <- NULL
stopifnot(
  identical(extract_array(df, list(4:2, c(1L,3L))), target),
  identical(extract_array(DF, list(4:2, c(1L,3L))), target)
```

```
)
```

`is_sparse` *Check for sparse representation*

Description

`is_sparse` indicates whether an object (typically array-like) uses a sparse representation of the data or not.

Note that this is about *data representation* and not about the data itself. For example, `is_sparse()` always returns `FALSE` on an *ordinary* matrix, even if the matrix contains 99% zeros, because the data in such a matrix is always stored in a dense form. OTOH `is_sparse()` always returns `TRUE` on a SparseArray derivative from the **SparseArray** package, or on a dgCMatrix object from the **Matrix** package, even if the data contains no zeros, because these objects use a sparse representation of the data.

Usage

```
is_sparse(x)
```

Arguments

`x`	Any object, but will typically be an array-like object. Examples of array-like objects: ordinary arrays, SparseArray objects from the **SparseArray** package, dgCMatrix objects from the **Matrix** package, DelayedArray objects from the **DelayedArray** package, or any object with an array semantic (i.e. an object for which `dim(x)` is not NULL).

Value

`TRUE` or `FALSE`

See Also

- `read_block` to read a block of data from an array-like object.
- array and matrix objects in base R.
- dgCMatrix objects implemented in the **Matrix** package.

Examples

```
m <- matrix(0L, nrow=50, ncol=20)
stopifnot(identical(is_sparse(m), FALSE))

dgc <- as(m + runif(1000), "dgCMatrix")
stopifnot(identical(is_sparse(dgc), TRUE))
```

mapToGrid *Map reference array positions to grid positions and vice-versa*

Description

Use `mapToGrid()` to map a set of reference array positions to grid positions. Use `mapToRef()` for the reverse mapping.

Usage

```
mapToGrid(Mindex, grid, linear=FALSE)

mapToRef(major, minor, grid, linear=FALSE)
```

Arguments

`Mindex`	An *M-index* containing *absolute* positions, that is, positions with respect to the underlying array (i.e. to the reference array of `grid`). 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. Note that no bounds checking is performed, that is, values in the j-th column of `Mindex` can be < 1 or > `refdim(grid)[j]`. What those values will be mapped to is undefined.
`grid`	An ArrayGrid object.
`linear`	`TRUE` or `FALSE`. Controls the format of the output for `mapToGrid` and the input for `mapToRef`. By default (i.e. when `linear` is `FALSE`), the major and minor indices returned by `mapToGrid` (or taken by `mapToRef`) are both *M-indices* (a.k.a. *matrix indices*). When `linear` is set to `TRUE`, they are both returned (or taken) as *L-indices* (a.k.a. *linear indices*).
`major`, `minor`	The `major` and `minor` components as returned by `mapToGrid`.

Value

- For `mapToGrid()`: A list with 2 components, `major` and `minor`.
  Each row in input matrix `Mindex` is an n-tuple that contains the coordinates of an *absolute* position.
  By default (i.e. when `linear` is `FALSE`), the 2 components of the returned list are integer matrices of the same dimensions as the input matrix. A row in the `major` (or `minor`) matrix is called a "major n-tuple" (or "minor n-tuple"). So for each "input position" (i.e. for each row in the input matrix), 2 n-tuples are returned: the "major n-tuple" and the "minor n-tuple". The "major n-tuple" contains the coordinates of the "input position" *in the grid coordinate system*, that is, the coordinates of the grid element where the position falls in. The "minor n-tuple" represents where exactly the "input position" falls *inside* the grid element reported by the "major n-tuple". The coordinates in the "minor n-tuple" are *relative* to this grid element.

When `linear` is `TRUE`, the `major` and `minor` components are returned as linear indices. In this case, both are integer vectors containing 1 linear index per "input position".

- For `mapToRef()`: A numeric matrix like one returned by `base::arrayInd` describing positions relative to the reference array of `grid`.

See Also

- ArrayGrid for the formal representation of grids and viewports.
- `Lindex2Mindex` and `Mindex2Lindex` for converting back and forth between *linear indices* and *matrix indices*.
- array and matrix objects in base R.

Examples

```
## 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)))

## Create a set of reference array positions:
Mindex <- rbind(c( 2, 5),  # bottom right corner of 1st grid element
                c( 3, 1),  # top left corner of 2nd grid element
                c(14, 9),  # top right corner of last grid element
                c(15, 7),  # bottom left corner of last grid element
                c(15, 9))  # bottom right corner of last grid element

## Map them to grid positions:
majmin <- mapToGrid(Mindex, grid2)
majmin

## Reverse mapping:
Mindex2 <- mapToRef(majmin$major, majmin$minor, grid2)
stopifnot(all.equal(Mindex2, Mindex))

majmin <- mapToGrid(Mindex, grid2, linear=TRUE)
majmin
Mindex2 <- mapToRef(majmin$major, majmin$minor, grid2, linear=TRUE)
stopifnot(all.equal(Mindex2, Mindex))

## Map all the valid positions:
all_positions <- seq_len(prod(refdim(grid2)))
Mindex <- arrayInd(all_positions, refdim(grid2))
majmin <- data.frame(mapToGrid(Mindex, grid2, linear=TRUE))
majmin

## Sanity checks:
min_by_maj <- split(majmin$minor,
                    factor(majmin$major, levels=seq_along(grid2)))
stopifnot(identical(lengths(min_by_maj, use.names=FALSE), lengths(grid2)))
stopifnot(all(mapply(isSequence, min_by_maj, lengths(min_by_maj))))
Mindex2 <- mapToRef(majmin$major, majmin$minor, grid2, linear=TRUE)
stopifnot(identical(Mindex2, Mindex))
```

```
## More mapping:
grid4 <- RegularArrayGrid(c(50, 20), spacings=c(15L, 9L))
Mindex <- rbind(c( 1,  1),
                c( 2,  1),
                c( 3,  1),
                c(16,  1),
                c(16,  2),
                c(16, 10),
                c(27, 18))

mapToGrid(Mindex, grid4)
mapToGrid(Mindex, grid4, linear=TRUE)
```

---

read_block *Read array blocks*

Description

Use read_block to read a block of data from an array-like object.

Note that this function is typically used in the context of block processing of on-disk objects (e.g. DelayedArray objects), often in combination with write_block.

Usage

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

## Internal generic function used by read_block() when is_sparse(x)
## is FALSE:
read_block_as_dense(x, viewport)
```

Arguments

x	An array-like object. This can be an ordinary array, a SparseArray object from the **SparseArray** package, a dgCMatrix object from the **Matrix** package, a DelayedArray object from the **DelayedArray** package, or any object with an array semantic (i.e. an object for which dim(x) is not NULL).
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 block is returned as an ordinary array (a.k.a. dense array). If TRUE, it's returned as a SparseArray object. If NA (the default), the block is returned as an ordinary array if is_sparse(x) is FALSE and as a SparseArray object otherwise. In other words, using as.sparse=NA is equivalent to using as.sparse=is_sparse(x). This preserves sparsity and is the most efficient way to read a block.

Note that when returned as a 2D SparseArray 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"`, and a lgCMatrix object if it's `"logical"`.

Details

`read_block()` delegates to 2 internal generic functions for reading a block:

- `read_block_as_dense`: used when `is_sparse(x)` is `FALSE`.
- `read_block_as_sparse` (defined in the **SparseArray** package): used when `is_sparse(x)` is `TRUE`.

Note that these 2 internal generic functions are not meant to be called directly by the end user. The end user should always call the higher-level user-facing `read_block()` function instead.

Value

A block of data. More precisely, the data from `x` that belongs to the block delimited by the specified viewport.

The block of data is returned either as an ordinary (dense) array or as a SparseArray object from the **SparseArray** package.

Note that the returned block of data is guaranteed to have the same type as `x` and the same dimensions as the `viewport`. More formally, if `block` is the value returned by `read_block(x, viewport)`, then:

```
identical(type(block), type(x))
```

and

```
identical(dim(block), dim(viewport))
```

are always `TRUE`.

See Also

- ArrayGrid for ArrayGrid and ArrayViewport objects.
- `is_sparse` to check whether an object uses a sparse representation of the data or not.
- SparseArray objects implemented in the **SparseArray** package.
- `S4Arrays::type` to get the type of the elements of an array-like object.
- The `read_block_as_sparse` internal generic function defined in the **SparseArray** package and used by `read_block()` when `is_sparse(x)` is `TRUE`.
- `write_block` to write a block of data to an array-like object.
- `blockApply` and family, in the **DelayedArray** package, for convenient block processing of an array-like object.
- dgCMatrix and lgCMatrix objects implemented in the **Matrix** package.
- DelayedArray objects implemented in the **DelayedArray** package.
- array and matrix objects in base R.

Examples

```
## Please note that, although educative, the examples below are somewhat
## artificial and do not illustrate real-world usage of read_block().
## See '?RealizationSink' in the DelayedArray package for more realistic
## read_block/write_block examples.

## ---------------------------------------------------------------------
## BASIC EXAMPLE 1: READ A BLOCK FROM AN ORDINARY MATRIX (DENSE)
## ---------------------------------------------------------------------
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

## Use 'as.sparse=TRUE' to read the block as sparse object:
block1b <- read_block(m1, viewport1, as.sparse=TRUE)
block1b
is_sparse(block1b)  # TRUE
class(block1b)      # an SVT_SparseArray object

## Sanity checks:
stopifnot(identical(type(m1), type(block1)))
stopifnot(identical(dim(viewport1), dim(block1)))
stopifnot(identical(m1[3:6, 2:4, drop=FALSE], block1))
stopifnot(is(block1b, "SparseArray"))
stopifnot(identical(type(m1), type(block1b)))
stopifnot(identical(dim(viewport1), dim(block1b)))
stopifnot(identical(block1, as.array(block1b)))

## ---------------------------------------------------------------------
## BASIC 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
is_sparse(m2)  # TRUE

## 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() preserves sparsity:
block2 <- read_block(m2, viewport2)
block2
```

```
is_sparse(block2)  # TRUE
class(block2)      # an SVT_SparseArray object

## Use 'as.sparse=FALSE' to force read_block() to return an ordinary
## matrix or array:
block2b <- read_block(m2, viewport2, as.sparse=FALSE)
block2b
as(block2b, "sparseMatrix")

## Sanity checks:
stopifnot(is(block2, "SparseArray"))
stopifnot(identical(type(m2), type(block2)))
stopifnot(identical(dim(viewport2), dim(block2)))
stopifnot(identical(type(m2), type(block2b)))
stopifnot(identical(dim(viewport2), dim(block2b)))
stopifnot(identical(block2b, as.array(block2)))

## ---------------------------------------------------------------------
## BASIC EXAMPLE 3: READ A BLOCK FROM A 3D ARRAY
## ---------------------------------------------------------------------
a3 <- array(1:60, dim=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(type(a3), type(block3)))
stopifnot(identical(dim(viewport3), dim(block3)))
stopifnot(identical(a3[1:2, 1:4, 3, drop=FALSE], block3))

## ---------------------------------------------------------------------
## BASIC EXAMPLE 4: READ AND PROCESS BLOCKS DEFINED BY A GRID
## ---------------------------------------------------------------------
a4 <- array(runif(120), dim=6:4)

## Define a grid of 2x3x2 blocks on 'a4':
grid4 <- RegularArrayGrid(dim(a4), spacings=c(2,3,2))
grid4
nblock <- length(grid4)  # number of blocks

## Walk on the grid and print the corresponding blocks:
for (bid in seq_len(nblock)) {
    viewport <- grid4[[bid]]
    block <- read_block(a4, viewport)
    cat("====== Block ", bid, "/", nblock, " ======\n", sep="")
```

```
    print(block)
}

## Walk on the grid and compute the sum of each block:
block_sums <- sapply(grid4,
    function(viewport) sum(read_block(a4, viewport))
)
block_sums

## Sanity checks:
stopifnot(identical(length(block_sums), nblock))
stopifnot(all.equal(sum(block_sums), sum(a4)))

## ---------------------------------------------------------------------
## THE read_block/write_block COMBO
## ---------------------------------------------------------------------
## See '?write_block' for examples that use the read_block/write_block
## combo.
```

`rowsum` *Compute column/row sums of a matrix-like object, for groups of rows/columns*

Description

`rowsum()` computes column sums across rows of a numeric matrix-like object for each level of a grouping variable.

`colsum()` computes row sums across columns of a numeric matrix-like object for each level of a grouping variable.

NOTE: This man page is for the `rowsum` and `colsum` *S4 generic functions* defined in the **S4Arrays** package. See `?base::rowsum` for the default `rowsum()` method (defined in the **base** package). Bioconductor packages can define specific methods for objects (typically matrix-like) not supported by the default methods.

Usage

```
rowsum(x, group, reorder=TRUE, ...)

colsum(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 method.

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

Specific methods defined in Bioconductor packages should behave as consistently as possible with the default methods.

See Also

- base::rowsum for the default rowsum method.
- showMethods for displaying a summary of the methods defined for a given generic function.
- selectMethod for getting the definition of a specific method.
- rowsum,DelayedMatrix-method in the **DelayedArray** package for an example of a specific rowsum method (defined for DelayedMatrix objects).

Examples

```
rowsum  # note the dispatch on the 'x' arg only
showMethods("rowsum")
selectMethod("rowsum", "ANY")     # the default rowsum() method

colsum  # note the dispatch on the 'x' arg only
showMethods("colsum")
selectMethod("colsum", "ANY")     # the default colsum() method
selectMethod("colsum", "matrix")  # colsum() method for ordinary matrices
```

---

type *Get the type of the elements of an array-like object*

Description

The **S4Arrays** package defines a couple of type() methods to get the type of the *elements* of a matrix-like or array-like object.

Usage

```
## S4 method for signature 'ANY'
type(x)

## S4 method for signature 'DataFrame'
type(x)
```

Arguments

x	For the default type() method: An array-like object. This can be an ordinary array, a SparseArray object from the **SparseArray** package, a dgCMatrix object from the **Matrix** package, a DelayedArray object from the **DelayedArray** package, or any object with an array semantic (i.e. an object for which dim(x) is not NULL). For the method for DataFrame objects: A DataFrame derivative for which as.data.frame(x) preserves the number of columns. See below for more information.

Details

Note that for an ordinary matrix or array x, type(x) is the same as typeof(x).

On an array-like object x that is not an ordinary array, type(x) is *semantically equivalent* to typeof(as.array(x)). However, the actual implementation is careful to avoid turning the full array-like object x into an ordinary array, as this would tend to be very inefficient in general. For example, doing so on a big DelayedArray object could easily eat all the memory available on the machine.

On a DataFrame object, type(x) only works if as.data.frame(x) preserves the number of columns, in which case it is *semantically equivalent* to typeof(as.matrix(as.data.frame(x))). Here too, the actual implementation is careful to avoid turning the full object into a data frame, then into a matrix, for efficiency reasons.

Value

A single string indicating the type of the array elements in x.

See Also

- The type generic function defined in the **BiocGenerics** package.
- SparseArray objects implemented in the **SparseArray** package.
- DelayedArray objects implemented in the **DelayedArray** package.
- DataFrame objects implemented in the **S4Vectors** package.

Examples

```
m <- matrix(rpois(54e6, lambda=0.4), ncol=1200)
type(m)   # integer

x1 <- as(m, "dgCMatrix")
type(x1)  # double

library(SparseArray)
x2 <- SparseArray(m)
type(x2)  # integer
```

write_block *Write array blocks*

Description

Use write_block to write a block of data to an array-like object.

Note that this function is typically used in the context of block processing of on-disk objects (e.g. DelayedArray objects), often in combination with read_block.

Usage

```
write_block(sink, viewport, block)
```

Arguments

sink	A **writable** array-like object. This is typically a RealizationSink derivative (RealizationSink is a virtual class defined in the **DelayedArray** package), but not necessarily. See ?RealizationSink in the **DelayedArray** package for more information about RealizationSink objects. Although write_block() will typically be used on a RealizationSink derivative, it can also be used on an ordinary array or other in-memory array-like object that supports subassignment (`[<-`), like a SparseArray object from the **SparseArray** package, or a dgCMatrix object from the **Matrix** package.
viewport	An ArrayViewport object compatible with sink, that is, such that refdim(viewport) is identical to dim(sink).
block	An array-like object of the same dimensions as viewport.

Value

The modified array-like object sink.

See Also

- ArrayGrid for ArrayGrid and ArrayViewport objects.
- SparseArray objects implemented in the **SparseArray** package.
- read_block to read a block of data from an array-like object.
- blockApply and family, in the **DelayedArray** package, for convenient block processing of an array-like object.
- RealizationSink objects implemented in the **DelayedArray** package for more realistic write_block examples.
- array and matrix objects in base R.

Examples

```
## Please note that, although educative, the examples below are somewhat
## artificial and do not illustrate real-world usage of write_block().
## See '?RealizationSink' in the DelayedArray package for more realistic
## read_block/write_block examples.

## ---------------------------------------------------------------------
## BASIC EXAMPLE 1: WRITE A BLOCK TO AN ORDINARY MATRIX (DENSE)
## ---------------------------------------------------------------------
m1 <- matrix(1:30, ncol=5)
m1

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

## Data to write:
block1 <- read_block(m1, viewport1) + 1000L

## Write the block:
m1A <- write_block(m1, viewport1, block1)
m1A

## Sanity checks:
stopifnot(identical(`[<-`(m1, 3:6, 2:4, value=block1), m1A))
m1B <- write_block(m1, viewport1, as(block1, "dgCMatrix"))
stopifnot(identical(m1A, m1B))

## ---------------------------------------------------------------------
## BASIC EXAMPLE 2: WRITE A BLOCK TO 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 write the data to:
block2_dim <- c(2, 20)
viewport2 <- ArrayViewport(dim(m2), IRanges(c(1, 1), width=block2_dim))
viewport2

## Data to write:
block2 <- matrix(1001:1040, nrow=2)

## Write the block:
m2A <- write_block(m2, viewport2, block2)
m2A

## Sanity checks:
stopifnot(identical(`[<-`(m2, 1:2, , value=block2), m2A))
m2B <- write_block(m2, viewport2, as(block2, "dgCMatrix"))
stopifnot(identical(m2A, m2B))
```

```
## ---------------------------------------------------------------------
## BASIC EXAMPLE 3: WRITE A BLOCK TO A 3D ARRAY
## ---------------------------------------------------------------------
a3 <- array(1:60, dim=5:3)

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

## Data to write:
block3 <- array(-(1:8), dim=block3_dim)

## Write the block:
a3A <- write_block(a3, viewport3, block3)
a3A

## Sanity checks:
stopifnot(identical(`[<-`(a3, 1:2, , 3, value=block3), a3A))
a3B <- write_block(a3, viewport3, as(block3, "SparseArray"))
stopifnot(identical(a3A, a3B))

## ---------------------------------------------------------------------
## BASIC EXAMPLE 4: WRITE BLOCKS DEFINED BY A GRID
## ---------------------------------------------------------------------
a4 <- array(NA_real_, dim=6:4)

## Define a grid of 2x3x2 blocks on 'a4':
grid4 <- RegularArrayGrid(dim(a4), spacings=c(2,3,2))
grid4
nblock <- length(grid4)  # number of blocks

## Walk on the grid and write blocks of random data:
for (bid in seq_len(nblock)) {
    viewport <- grid4[[bid]]
    block <- array(runif(length(viewport)), dim=dim(viewport))
    cat("====== Write block ", bid, "/", nblock, " ======\n", sep="")
    a4 <- write_block(a4, viewport, block)
}
a4

## ---------------------------------------------------------------------
## BASIC EXAMPLE 5: READ, PROCESS, AND WRITE BLOCKS DEFINED BY TWO GRIDS
## ---------------------------------------------------------------------
## 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. There are several ways to do this.

## 1. Here is a solution based on apply():

collapse_3rd_dim <- function(a) apply(a, MARGIN=1:2, sum)
```

```
## 2. Here is a slightly more efficient solution:

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
}

## 3. And here is a block-processing solution that involves two grids,
##    one for the sink, and one for the input:

a5 <- array(runif(8000), dim=c(25, 40, 8))  # input
m  <- array(NA_real_, dim=dim(a5)[1:2])     # sink

## Since we're going to walk on the two grids simultaneously, read a
## block from 'a5' and write it to 'm', 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:
m_grid  <- RegularArrayGrid(dim(m),  spacings=c(10, 10))
a5_grid <- RegularArrayGrid(dim(a5), spacings=c(10, 10, dim(a5)[[3]]))

## Let's check that our two grids are actually "aligned":
stopifnot(identical(length(m_grid), length(a5_grid)))
stopifnot(identical(dims(m_grid), dims(a5_grid)[ , 1:2, drop=FALSE]))

## Walk on the two grids simultaneously, and read/collapse/write blocks:
for (bid in seq_along(m_grid)) {
    ## Read block from 'a5'.
    a5_viewport <- a5_grid[[bid]]
    block <- read_block(a5, a5_viewport)
    ## Collapse it.
    block <- collapse_3rd_dim(block)
    ## Write the collapsed block to 'm'.
    m_viewport <- m_grid[[bid]]
    m <- write_block(m, m_viewport, block)
}

## Sanity checks:
stopifnot(identical(dim(a5)[1:2], dim(m)))
stopifnot(identical(sum(a5), sum(m)))
stopifnot(identical(collapse_3rd_dim(a5), m))

## See '?RealizationSink' in the DelayedArray package for a more
## realistic "array collapse" example where the blocks are written
## to a RealizationSink object.
```

Index