Package 'SparseArray'

May 14, 2024

Title High-performance sparse data representation and manipulation in R

Description The SparseArray package provides array-like containers for efficient in-memory representation of multidimensional sparse data in R (arrays and matrices). The package defines the SparseArray virtual class and two concrete subclasses: COO_SparseArray and SVT_SparseArray. Each subclass uses its own internal representation of the nonzero multidimensional data: the ``COO layout'' and the ``SVT layout'', respectively. SVT_SparseArray objects mimic as much as possible the behavior of ordinary matrix and array objects in base R. In particular, they suppport most of the ``standard matrix and array API'' defined in base R and in the matrixStats package from CRAN.

biocViews Infrastructure, DataRepresentation

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

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

Version 1.5.3

License Artistic-2.0

Encoding UTF-8

Depends R (>= 4.3.0), methods, Matrix, BiocGenerics (>= 0.43.1), MatrixGenerics (>= 1.11.1), S4Vectors, S4Arrays (>= 1.1.6)

Imports utils, stats, matrixStats, IRanges, XVector

LinkingTo S4Vectors, IRanges, XVector

Suggests DelayedArray, testthat, knitr, rmarkdown, BiocStyle

VignetteBuilder knitr

Collate utils.R options.R thread-control.R sparseMatrix-utils.R SparseArray-class.R COO_SparseArray-class.R SVT_SparseArray-class.R extract_sparse_array.R read_block_as_sparse.R SparseArray-dim-tuning.R SparseArray-aperm.R SparseArray-subsetting.R SparseArray-subassignment.R SparseArray-abind.R SparseArray-summarization.R SparseArray-Ops-methods.R

SparseArray-Math-methods.R SparseArray-Complex-methods.R
SparseArray-misc-methods.R SparseMatrix-mult.R
matrixStats-methods.R rowsum-methods.R randomSparseArray.R
readSparseCSV.R zzz.R

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

git_branch devel

git_last_commit 4737f33

git_last_commit_date 2024-05-12

Repository Bioconductor 3.20

Date/Publication 2024-05-13

Author Hervé Pagès [aut, cre],
Vince Carey [fnd],
Rafael A. Irizarry [fnd],
Jacques Serizay [ctb]

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

Contents

COO_SparseArray-class *COO_SparseArray objects*

Description

The COO_SparseArray class is a container for efficient in-memory representation of multidimensional sparse arrays. It uses the *COO layout* to represent the nonzero data internally.

A COO_SparseMatrix object is a COO_SparseArray object of 2 dimensions.

IMPORTANT NOTE: COO_SparseArray and COO_SparseMatrix objects are now superseded by the new and more efficient SVT_SparseArray and SVT_SparseMatrix objects.

Usage

```
## Constructor function:
COO_SparseArray(dim, nzcoo=NULL, nzdata=NULL, dimnames=NULL, check=TRUE)

## Getters (in addition to dim(), length(), and dimnames()):
nzcoo(x)
nzdata(x)
```

Arguments

dim	The dimensions (specified as an integer vector) of the COO_SparseArray or COO_SparseMatrix object to create.
nzcoo	A matrix containing the array coordinates of the nonzero elements. This must be an integer matrix of array coordinates like one returned by base::arrayInd or S4Arrays::Lindex2Mindex, that is, a matrix with length(dim) columns and where each row is an n-tuple representing the coordinates of an array element.
nzdata	A vector (atomic or list) of length nrow(nzcoo) containing the nonzero elements.
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 COO_SparseArray or COO_SparseMatrix object.

Value

- For COO_SparseArray(): A COO_SparseArray or COO_SparseMatrix object.
- For nzcoo(): A matrix with one column per dimension containing the *array coordinates* of the nonzero elements.
- For nzdata(): A vector *parallel* to nzcoo(x) (i.e. with one element per row in nzcoo(x)) containing the nonzero elements.

See Also

- The new SVT_SparseArray class for a replacement of of the COO_SparseArray class.
- The SparseArray class for the virtual parent class of COO_SparseArray and SVT_SparseArray.
- S4 classes dgCMatrix and lgCMatrix defined in the **Matrix** package, for the de facto standard of sparse matrix representations in the R ecosystem.
- `base::arrayInd` in the **base** package.
- `S4Arrays::Lindex2Mindex` in the **S4Arrays** package for an improved (faster) version of `base::arrayInd`.
- Ordinary array objects in base R.

Examples

```
## ---------------------------------------------------------------------
## EXAMPLE 1
## ---------------------------------------------------------------------
dim1 <- 5:3
nzcoo1 <- Lindex2Mindex(sample(60, 8), 5:3)
nzdata1 <- 11.11 * seq_len(nrow(nzcoo1))
coo1 <- COO_SparseArray(dim1, nzcoo1, nzdata1)
coo1

nzcoo(coo1)
nzdata(coo1)
type(coo1)
sparsity(coo1)

as.array(coo1)  # back to a dense representation

#as.matrix(coo1)  # error!

## ---------------------------------------------------------------------
## EXAMPLE 2
## ---------------------------------------------------------------------
m2 <- matrix(c(5:-2, rep.int(c(0L, 99L), 11)), ncol=6)
coo2 <- as(m2, "COO_SparseArray")
class(coo2)
dim(coo2)
length(coo2)
nzcoo(coo2)
nzdata(coo2)
type(coo2)
sparsity(coo2)

stopifnot(identical(as.matrix(coo2), m2))

t(coo2)
stopifnot(identical(as.matrix(t(coo2)), t(as.matrix(coo2))))

## ---------------------------------------------------------------------
## COERCION FROM/TO dg[C|R]Matrix OR lg[C|R]Matrix OBJECTS
```

```
## ---------------------------------------------------------------------
## dg[C|R]Matrix and lg[C|R]Matrix objects are defined in the Matrix
## package.

## dgCMatrix/dgRMatrix:

M2C <- as(coo2, "dgCMatrix")
stopifnot(identical(M2C, as(m2, "dgCMatrix")))

coo2C <- as(M2C, "COO_SparseArray")
## 'coo2C' is the same as 'coo2' except that 'nzdata(coo2C)' has
## type "double" instead of "integer":
stopifnot(all.equal(coo2, coo2C))
typeof(nzdata(coo2C))  # double
typeof(nzdata(coo2))   # integer

M2R <- as(coo2, "dgRMatrix")
stopifnot(identical(M2R, as(m2, "dgRMatrix")))
coo2R <- as(M2R, "COO_SparseArray")
stopifnot(all.equal(as.matrix(coo2), as.matrix(coo2R)))

## lgCMatrix/lgRMatrix:

m3 <- m2 == 99  # logical matrix
coo3 <- as(m3, "COO_SparseArray")
class(coo3)
type(coo3)

M3C <- as(coo3, "lgCMatrix")
stopifnot(identical(M3C, as(m3, "lgCMatrix")))
coo3C <- as(M3C, "COO_SparseArray")
identical(as.matrix(coo3), as.matrix(coo3C))

M3R <- as(coo3, "lgRMatrix")
#stopifnot(identical(M3R, as(m3, "lgRMatrix")))
coo3R <- as(M3R, "COO_SparseArray")
identical(as.matrix(coo3), as.matrix(coo3R))

## ---------------------------------------------------------------------
## A BIG COO_SparseArray OBJECT
## ---------------------------------------------------------------------
nzcoo4 <- cbind(sample(25000, 600000, replace=TRUE),
                sample(195000, 600000, replace=TRUE))
nzdata4 <- runif(600000)
coo4 <- COO_SparseArray(c(25000, 195000), nzcoo4, nzdata4)
coo4
sparsity(coo4)
```

---

extract_sparse_array *extract_sparse_array*

Description

`extract_sparse_array()` is an internal generic function that is the workhorse behind the default `read_block_as_sparse()` method. It is not intended to be used directly by the end user.

It is similar to the `extract_array()` internal generic function defined in the **S4Arrays** package, with the major difference that, in the case of `extract_sparse_array()`, the extracted array data is returned as a SparseArray object instead of an ordinay array.

Usage

```
extract_sparse_array(x, index)

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

Arguments

`x`	An array-like object for which `is_sparse(x)` is `TRUE`.
`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 NOT allowed to contain duplicated indices. This is an important difference with `extract_array`.

Details

`extract_sparse_array()` should *always* be called on an array-like object `x` for which `is_sparse(x)` is `TRUE`. Also it should *never* be called with duplicated indices in the individual list elements of the `index` argument.

For maximum efficiency, `extract_sparse_array()` methods should:

1. NOT check that `is_sparse(x)` is `TRUE`.
2. NOT check that the individual list elements in `index` contain no duplicated indices.
3. NOT try to do anything with the dimnames on `x`.
4. always operate natively on the sparse representation of the data in `x`, that is, they should never *expand* it into a dense representation (e.g. with *as.array()*).

Like for `extract_array()`, `extract_sparse_array()` methods need to support empty or missing subscripts. For example, if `x` is an M x N matrix-like object for which `is_sparse(x)` is `TRUE`, then `extract_sparse_array(x, list(NULL, integer(0)))` must return an M x 0 SparseArray derivative, and `extract_sparse_array(x, list(integer(0), integer(0)))` a 0 x 0 SparseArray derivative.

Value

A SparseArray derivative (COO_SparseArray or SVT_SparseArray) of the same `type()` as `x`. For example, if `x` is an object representing an M x N sparse matrix of complex numbers (i.e. `type(x) == "complex"`), then `extract_sparse_array(x, list(NULL, 2L))` must return the 2nd column in `x` as an M x 1 SparseArray derivative of `type()` `"complex"`.

See Also

- `is_sparse` in the **S4Arrays** package to check whether an object uses a sparse representation of the data or not.
- SparseArray objects.
- `S4Arrays::type` in the **S4Arrays** package to get the type of the elements of an array-like object.
- `read_block_as_sparse` to read array blocks as SparseArray objects.
- `extract_array` in the **S4Arrays** package.
- dgCMatrix objects implemented in the **Matrix** package.

Examples

```
extract_sparse_array
showMethods("extract_sparse_array")

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

m <- matrix(0L, nrow=6, ncol=4)
m[c(1:2, 8, 10, 15:17, 24)] <- (1:8)*10L
dgcm <- as(m, "dgCMatrix")
dgcm

extract_sparse_array(dgcm, list(3:6, NULL))
extract_sparse_array(dgcm, list(3:6, 2L))
extract_sparse_array(dgcm, list(3:6, integer(0)))

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

a <- array(0L, dim=5:3, dimnames=list(letters[1:5], NULL, LETTERS[1:3]))
a[c(1:2, 8, 10, 15:17, 20, 24, 40, 56:60)] <- (1:15)*10L
svt <- as(a, "SparseArray")
svt

extract_sparse_array(svt, list(NULL, 4:2, 1L))
extract_sparse_array(svt, list(NULL, 4:2, 2:3))
extract_sparse_array(svt, list(NULL, 4:2, integer(0)))
```

`matrixStats-methods` *SparseArray col/row summarization methods*

Description

The **SparseArray** package provides memory-efficient col/row summarization methods (a.k.a. matrixStats methods) for SparseArray objects, like `colSums()`, `rowSums()`, `colMedians()`, `rowMedians()`, `colVars()`, `rowVars()`, etc...

Note that these are *S4 generic functions* defined in the **MatrixGenerics** package, with methods for ordinary matrices defined in the **matrixStats** package. This man page documents the methods defined for SVT_SparseArray objects.

Usage

```
## N.B.: Showing ONLY the col*() methods (usage of row*() methods is
## the same):

## S4 method for signature 'SVT_SparseArray'
colAnyNAs(x, rows=NULL, cols=NULL, dims=1, ..., useNames=NA)

## S4 method for signature 'SVT_SparseArray'
colAnys(x, rows=NULL, cols=NULL, na.rm=FALSE, dims=1, ..., useNames=NA)

## S4 method for signature 'SVT_SparseArray'
colAlls(x, rows=NULL, cols=NULL, na.rm=FALSE, dims=1, ..., useNames=NA)

## S4 method for signature 'SVT_SparseArray'
colMins(x, rows=NULL, cols=NULL, na.rm=FALSE, dims=1, ..., useNames=NA)

## S4 method for signature 'SVT_SparseArray'
colMaxs(x, rows=NULL, cols=NULL, na.rm=FALSE, dims=1, ..., useNames=NA)

## S4 method for signature 'SVT_SparseArray'
colRanges(x, rows=NULL, cols=NULL, na.rm=FALSE, dims=1, ..., useNames=NA)

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

## S4 method for signature 'SVT_SparseArray'
colProds(x, rows=NULL, cols=NULL, na.rm=FALSE, dims=1, ..., useNames=NA)

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

## S4 method for signature 'SVT_SparseArray'
colVars(x, rows=NULL, cols=NULL, na.rm=FALSE, center=NULL, dims=1,
```

```
            ..., useNames=NA)

## S4 method for signature 'SVT_SparseArray'
colSds(x, rows=NULL, cols=NULL, na.rm=FALSE, center=NULL, dims=1,
          ..., useNames=NA)

## S4 method for signature 'SVT_SparseArray'
colMedians(x, rows=NULL, cols=NULL, na.rm=FALSE, ..., useNames=NA)
```

Arguments

`x`	An SVT_SparseMatrix or SVT_SparseArray object. Note that the `colMedians()` and `rowMedians()` methods only support 2D objects (i.e. SVT_SparseMatrix objects) at the moment.
`rows`, `cols`, `...`	Not supported.
`na.rm`, `useNames`, `center`	See man pages for the corresponding generics in the **MatrixGenerics** package (e.g. `?MatrixGenerics::rowVars`) for a description of these arguments. Note that, unlike the methods for ordinary matrices defined in the **matrixStats** package, the `center` argument of the `colVars()`, `rowVars()`, `colSds()`, and `rowSds()` methods for SVT_SparseArray objects can only be a *single value* (or a `NULL`). In particular, if `x` has more than one column, then `center` cannot be a vector with one value per column in `x`.
`dims`	See `?base::colSums` for a description of this argument. Note that all the methods above support it, except `colMedians()` and `rowMedians()`.

Details

All these methods operate *natively* on the SVT_SparseArray representation, for maximum efficiency.

Note that more col/row summarization methods might be added in the future.

Value

See man pages for the corresponding generics in the **MatrixGenerics** package (e.g. `?MatrixGenerics::colRanges`) for the value returned by these methods.

Note

The matrixStats method for SVT_SparseMatrix objects are multithreaded. See `set_SparseArray_nthread` for how to control the number of threads.

See Also

- SVT_SparseArray objects.
- The man pages for the various generic functions defined in the **MatrixGenerics** package e.g. `MatrixGenerics::colVars` etc...

Examples

```
## ----------------------------------------------------------------------
## 2D CASE
## ----------------------------------------------------------------------
m0 <- matrix(0L, nrow=6, ncol=4, dimnames=list(letters[1:6], LETTERS[1:4]))
m0[c(1:2, 8, 10, 15:17, 24)] <- (1:8)*10L
m0["e", "B"] <- NA
svt0 <- SparseArray(m0)
svt0

colSums(svt0)
colSums(svt0, na.rm=TRUE)

rowSums(svt0)
rowSums(svt0, na.rm=TRUE)

colMeans(svt0)
colMeans(svt0, na.rm=TRUE)

colRanges(svt0)
colRanges(svt0, useNames=FALSE)
colRanges(svt0, na.rm=TRUE)
colRanges(svt0, na.rm=TRUE, useNames=FALSE)

colVars(svt0)
colVars(svt0, useNames=FALSE)

## Sanity checks:
stopifnot(
  identical(colSums(svt0), colSums(m0)),
  identical(colSums(svt0, na.rm=TRUE), colSums(m0, na.rm=TRUE)),
  identical(rowSums(svt0), rowSums(m0)),
  identical(rowSums(svt0, na.rm=TRUE), rowSums(m0, na.rm=TRUE)),
  identical(colMeans(svt0), colMeans(m0)),
  identical(colMeans(svt0, na.rm=TRUE), colMeans(m0, na.rm=TRUE)),
  identical(colRanges(svt0), colRanges(m0, useNames=TRUE)),
  identical(colRanges(svt0, useNames=FALSE), colRanges(m0, useNames=FALSE)),
  identical(colRanges(svt0, na.rm=TRUE),
            colRanges(m0, na.rm=TRUE, useNames=TRUE)),
  identical(colVars(svt0), colVars(m0, useNames=TRUE)),
  identical(colVars(svt0, na.rm=TRUE),
            colVars(m0, na.rm=TRUE, useNames=TRUE))
)

## ----------------------------------------------------------------------
## 3D CASE (AND ARBITRARY NUMBER OF DIMENSIONS)
## ----------------------------------------------------------------------
set.seed(2009)
svt <- 6L * (poissonSparseArray(5:3, density=0.35) -
             poissonSparseArray(5:3, density=0.35))
dimnames(svt) <- list(NULL, letters[1:4], LETTERS[1:3])
```

```
cs1 <- colSums(svt)
cs1  # cs1[j , k] is equal to sum(svt[ , j, k])

cs2 <- colSums(svt, dims=2)
cs2  # cv2[k] is equal to sum(svt[ , , k])

cv1 <- colVars(svt)
cv1  # cv1[j , k] is equal to var(svt[ , j, k])

cv2 <- colVars(svt, dims=2)
cv2  # cv2[k] is equal to var(svt[ , , k])

## Sanity checks:
k_idx <- setNames(seq_len(dim(svt)[3]), dimnames(svt)[[3]])
j_idx <- setNames(seq_len(dim(svt)[2]), dimnames(svt)[[2]])
cv1b <- sapply(k_idx, function(k)
                      sapply(j_idx, function(j) var(svt[ , j, k, drop=FALSE])))
cv2b <- sapply(k_idx, function(k) var(svt[ , , k]))
stopifnot(
  identical(colSums(svt), colSums(as.array(svt))),
  identical(colSums(svt, dims=2), colSums(as.array(svt), dims=2)),
  identical(cv1, cv1b),
  identical(cv2, cv2b)
)
```

---

randomSparseArray *Random SparseArray object*

Description

randomSparseArray() and poissonSparseArray() can be used to generate a random SparseArray object efficiently.

Usage

```
randomSparseArray(dim, density=0.05)
poissonSparseArray(dim, lambda=-log(0.95), density=NA)

## Convenience wrappers for the 2D case:
randomSparseMatrix(nrow, ncol, density=0.05)
poissonSparseMatrix(nrow, ncol, lambda=-log(0.95), density=NA)
```

Arguments

dim	The dimensions (specified as an integer vector) of the SparseArray object to generate.
density	The desired density (specified as a number >= 0 and <= 1) of the SparseArray object to generate, that is, the ratio between its number of nonzero elements and its total number of elements. This is nzcount(x)/length(x) or 1 - sparsity(x).

Note that for `poissonSparseArray()` and `poissonSparseMatrix()` `density` must be < 1 and the *actual* density of the returned object won't be exactly as requested but will typically be very close.

`lambda` The mean of the Poisson distribution. Passed internally to the calls to `rpois()`.

Only one of `lambda` and `density` can be specified.

When `density` is requested, `rpois()` is called internally with `lambda` set to `-log(1 - density)`. This is expected to generate Poisson data with the requested density.

Finally note that the default value for `lambda` corresponds to a requested density of 0.05.

`nrow, ncol` Number of rows and columns of the SparseMatrix object to generate.

Details

`randomSparseArray()` mimics the `rsparsematrix()` function from the **Matrix** package but returns a SparseArray object instead of a dgCMatrix object.

`poissonSparseArray()` populates a SparseArray object with Poisson data i.e. it's equivalent to:

```
a <- array(rpois(prod(dim), lambda), dim)
as(a, "SparseArray")
```

but is faster and more memory efficient because intermediate dense array `a` is never generated.

Value

A SparseArray derivative (of class SVT_SparseArray or SVT_SparseMatrix) with the requested dimensions and density.

The type of the returned object is `"double"` for `randomSparseArray()` and `randomSparseMatrix()`, and `"integer"` for `poissonSparseArray()` and `poissonSparseMatrix()`.

Note

Unlike with `Matrix::rsparsematrix()` there's no limit on the number of nonzero elements that can be contained in the returned SparseArray object.

For example `Matrix::rsparsematrix(3e5, 2e4, density=0.5)` will fail with an error but `randomSparseMatrix(3e5, 2e4, density=0.5)` should work (even though it will take some time and the memory footprint of the resulting object will be about 18 Gb).

See Also

- The `Matrix::rsparsematrix` function in the **Matrix** package.
- The `stats::rpois` function in the **stats** package.
- SVT_SparseArray objects.

Examples

```
## ----------------------------------------------------------------------
## randomSparseArray() / randomSparseMatrix()
## ----------------------------------------------------------------------
set.seed(123)
dgcm1 <- rsparsematrix(2500, 950, density=0.1)
set.seed(123)
svt1 <- randomSparseMatrix(2500, 950, density=0.1)
svt1
type(svt1)  # "double"

stopifnot(identical(as(svt1, "dgCMatrix"), dgcm1))

## ----------------------------------------------------------------------
## poissonSparseArray() / poissonSparseMatrix()
## ----------------------------------------------------------------------
svt2 <- poissonSparseMatrix(2500, 950, density=0.1)
svt2
type(svt2)  # "integer"
1 - sparsity(svt2)  # very close to the requested density

set.seed(123)
svt3 <- poissonSparseArray(c(600, 1700, 80), lambda=0.01)
set.seed(123)
a3 <- array(rpois(length(svt3), lambda=0.01), dim(svt3))
stopifnot(identical(svt3, SparseArray(a3)))

## The memory footprint of 'svt3' is 10x smaller than that of 'a3':
object.size(svt3)
object.size(a3)
as.double(object.size(a3) / object.size(svt3))
```

---

readSparseCSV *Read/write a sparse matrix from/to a CSV file*

Description

Read/write a sparse matrix from/to a CSV (comma-separated values) file.

Usage

```
writeSparseCSV(x, filepath, sep=",", transpose=FALSE, write.zeros=FALSE,
                  chunknrow=250)

readSparseCSV(filepath, sep=",", transpose=FALSE)
```

Arguments

x	A matrix-like object, typically sparse. IMPORTANT: The object must have rownames and colnames! These will be written to the file. Another requirement is that the object must be subsettable. More precisely: it must support 2D-style subsetting of the kind `x[i, ]` and `x[ , j]` where `i` and `j` are integer vectors of valid row and column indices.
filepath	The path (as a single string) to the file where to write the matrix-like object or to read it from. Compressed files are supported. If `""`, `writeSparseCSV()` will write the data to the standard output connection. Note that `filepath` can also be a connection.
sep	The field separator character. Values on each line of the file are separated by this character.
transpose	`TRUE` or `FALSE`. By default, rows in the matrix-like object correspond to lines in the CSV file. Set `transpose` to `TRUE` to transpose the matrix-like object on-the-fly, that is, to have its columns written to or read from the lines in the CSV file. Note that using `transpose=TRUE` is semantically equivalent to calling `t()` on the object before writing it or after reading it, but it will tend to be more efficient. Also it will work even if `x` does not support `t()` (not all matrix-like objects are guaranteed to be transposable).
write.zeros	`TRUE` or `FALSE`. By default, the zero values in `x` are not written to the file. Set `write.zeros` to `TRUE` to write them.
chunknrow	`writeSparseCSV()` uses a block-processing strategy to try to speed up things. By default blocks of 250 rows (or columns if `transpose=TRUE`) are used. In our experience trying to increase this (e.g. to 500 or more) will generally not produce significant benefits while it will increase memory usage, so use carefully.

Value

`writeSparseCSV` returns an invisible `NULL`.

`readSparseCSV` returns a SparseMatrix object of class SVT_SparseMatrix.

See Also

- SparseArray objects.
- dgCMatrix objects implemented in the **Matrix** package.

Examples

```
## ---------------------------------------------------------------------
## writeSparseCSV()
## ---------------------------------------------------------------------

## Prepare toy matrix 'm0':
rownames0 <- LETTERS[1:6]
colnames0 <- letters[1:4]
```

```
m0 <- matrix(0L, nrow=length(rownames0), ncol=length(colnames0),
             dimnames=list(rownames0, colnames0))
m0[c(1:2, 8, 10, 15:17, 24)] <- (1:8)*10L
m0

## writeSparseCSV():
writeSparseCSV(m0, filepath="", sep="\t")
writeSparseCSV(m0, filepath="", sep="\t", write.zeros=TRUE)
writeSparseCSV(m0, filepath="", sep="\t", transpose=TRUE)

## Note that writeSparseCSV() will automatically (and silently) coerce
## non-integer values to integer by passing them thru as.integer().

## Example where type(x) is "double":
m1 <- m0 * runif(length(m0))
m1
type(m1)
writeSparseCSV(m1, filepath="", sep="\t")

## Example where type(x) is "logical":
writeSparseCSV(m0 != 0, filepath="", sep="\t")

## Example where type(x) is "raw":
m2 <- m0
type(m2) <- "raw"
m2
writeSparseCSV(m2, filepath="", sep="\t")

## ------------------------------------------------------------------------
## readSparseCSV()
## ------------------------------------------------------------------------

csv_file <- tempfile()
writeSparseCSV(m0, csv_file)

svt1 <- readSparseCSV(csv_file)
svt1

svt2 <- readSparseCSV(csv_file, transpose=TRUE)
svt2

## If you need the sparse data as a dgCMatrix object, just coerce the
## returned object:
as(svt1, "dgCMatrix")
as(svt2, "dgCMatrix")

## Sanity checks:
stopifnot(identical(m0, as.matrix(svt1)))
stopifnot(identical(t(m0), as.matrix(svt2)))
```

---

read_block_as_sparse *read_block_as_sparse*

Description

`read_block_as_sparse()` is an internal generic function used by `S4Arrays::read_block()` when `is_sparse(x)` is `TRUE`.

Usage

```
read_block_as_sparse(x, viewport)

## S4 method for signature 'ANY'
read_block_as_sparse(x, viewport)
```

Arguments

`x`	An array-like object for which `is_sparse(x)` is `TRUE`.
`viewport`	An ArrayViewport object compatible with `x`, that is, such that `refdim(viewport)` is identical to `dim(x)`.

Details

Like `read_block_as_dense()` in the **S4Arrays** package, `read_block_as_sparse()` is 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. See `?read_block` in the **S4Arrays** package for more information.

Also, like `extract_sparse_array()`, `read_block_as_sparse()` should *always* be called on an array-like object `x` for which `is_sparse(x)` is `TRUE`.

For maximum efficiency, `read_block_as_sparse()` methods should:

1. NOT check that `is_sparse(x)` is `TRUE`.
2. NOT try to do anything with the dimnames on `x` (`read_block()` takes care of that).
3. always operate natively on the sparse representation of the data in `x`, that is, they should never *expand* it into a dense representation (e.g. with *as.array()*).

Value

A block of data as a SparseArray derivative (COO_SparseArray or SVT_SparseArray) of the same `type()` as `x`.

See Also

- `read_block` in the **S4Arrays** package for the higher-level user-facing function for reading array blocks.
- ArrayGrid in the **S4Arrays** package for ArrayGrid and ArrayViewport objects.
- `is_sparse` in the **S4Arrays** package to check whether an object uses a sparse representation of the data or not.
- SparseArray objects.
- `S4Arrays::type` in the **S4Arrays** package to get the type of the elements of an array-like object.

- extract_sparse_array for the workhorse behind the default read_block_as_sparse() method.
- dgCMatrix objects implemented in the **Matrix** package.

---

rowsum-methods *rowsum() methods for sparse matrices*

Description

The **SparseArray** package provides memory-efficient rowsum() methods for SparseMatrix and dgCMatrix objects.

Note that colsum() also works on these objects via the default method defined in the **S4Arrays** package.

Usage

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

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

Arguments

x	An SVT_SparseMatrix or dgCMatrix object.
group, reorder	See ?base::rowsum for a description of these arguments.
...	Like the default S3 rowsum() method defined in the **base** package, the methods documented in this man page support additional argument na.rm, set to FALSE by default. If TRUE, missing values (NA or NaN) are omitted from the calculations.

Value

An *ordinary* matrix, like the default rowsum() method. See ?base::rowsum for how the matrix returned by the default rowsum() method is obtained.

See Also

- rowsum in base R.
- S4Arrays::rowsum in the **S4Arrays** package for the rowsum() and colsum() S4 generic functions.
- SVT_SparseMatrix objects.
- dgCMatrix objects implemented in the **Matrix** package.

Examples

```
svt0 <- randomSparseMatrix(7e5, 100, density=0.15)
dgcm0 <- as(svt0, "dgCMatrix")
m0 <- as.matrix(svt0)

group <- sample(10, nrow(m0), replace=TRUE)

## Calling rowsum() on the sparse representations is usually faster
## than on the dense representation:
rs1 <- rowsum(m0, group)
rs2 <- rowsum(svt0, group)   # about 3x faster
rs3 <- rowsum(dgcm0, group)  # also about 3x faster

## Sanity checks:
stopifnot(identical(rs1, rs2))
stopifnot(identical(rs1, rs3))
```

SparseArray *SparseArray objects*

Description

The **SparseArray** package defines the SparseArray virtual class whose purpose is to be extended by other S4 classes that aim at representing in-memory multidimensional sparse arrays.

It has currently two concrete subclasses, COO_SparseArray and SVT_SparseArray, both also defined in this package. Each subclass uses its own internal representation for the nonzero multidimensional data, the *COO layout* for COO_SparseArray, and the *SVT layout* for SVT_SparseArray. The two layouts are described in the COO_SparseArray and SVT_SparseArray man pages, respectively.

Finally, the package also defines the SparseMatrix virtual class, as a subclass of the SparseArray class, for the specific 2D case.

Usage

```
## Constructor function:
SparseArray(x, type=NA)
```

Arguments

x	An ordinary matrix or array, or a dg[C|R]Matrix object, or an lg[C|R]Matrix object, or any matrix-like or array-like object that supports coercion to SVT_SparseArray.
type	A single string specifying the requested type of the object. Normally, the SparseArray object returned by the constructor function has the same `type()` as `x` but the user can use the `type` argument to request a different type. Note that doing: `sa <- SparseArray(x, type=type)`

is equivalent to doing:

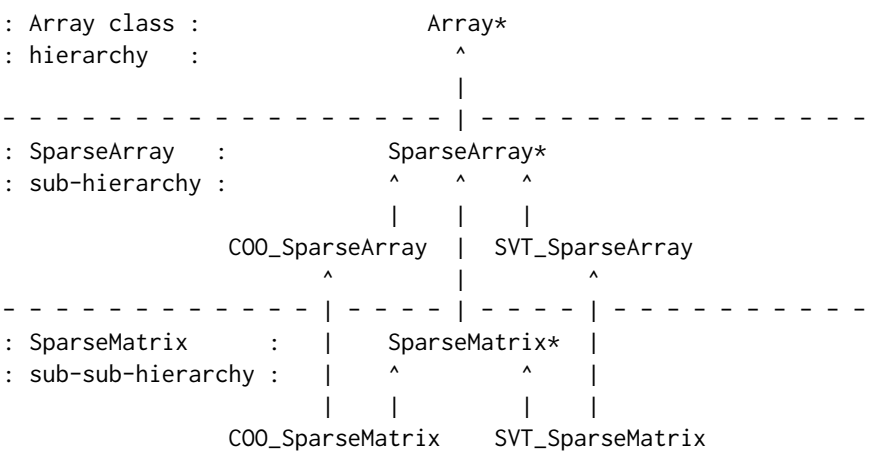
```
sa <- SparseArray(x)
type(sa) <- type
```

but the former is more convenient and will generally be more efficient.

Supported types are all R atomic types plus `"list"`.

Details

The SparseArray class extends the Array virtual class defined in the **S4Arrays** package. Here is the full SparseArray sub-hierarchy as defined in the **SparseArray** package (virtual classes are marked with an asterisk):



Any object that belongs to a class that extends SparseArray e.g. (a SVT_SparseArray or SVT_SparseMatrix object) is called a *SparseArray derivative*.

Most of the *standard matrix and array API* defined in base R should work on SparseArray derivatives, including `dim()`, `length()`, `dimnames()`, `` `dimnames<-`() ``, `[`, `drop()`, `` `[<-` `` (subassignment), `t()`, `rbind()`, `cbind()`, etc...

SparseArray derivatives also support `type()`, `` `type<-`() ``, `is_sparse()`, `nzcount()`, `nzwhich()`, `nzvals()`, `sparsity()`, `arbind()`, and `acbind()`.

`sparsity(x)` returns the ratio between the number of zero-valued elements in array-like object `x` and its total number of elements (`length(x)` or `prod(dim(x))`). More precisely, `sparsity(x)` is `1 - nzcount(x)/length(x)`.

Value

A *SparseArray derivative*, that is a SVT_SparseArray, COO_SparseArray, SVT_SparseMatrix, or COO_SparseMatrix object.

The `type()` of the input object is preserved, except if a different one was requested via the `type` argument.

What is considered a zero depends on the `type()`:

- `"logical"` zero is `FALSE`;

- "integer" zero is 0L;
- "double" zero is 0;
- "complex" zero is 0+0i;
- "raw" zero is raw(1);
- "character" zero is "" (empty string);
- "list" zero is NULL.

See Also

- The COO_SparseArray and SVT_SparseArray classes.
- SparseArray_subsetting for subsetting a SparseArray object.
- SparseArray_subassignment for SparseArray subassignment.
- SparseArray_abind for combining multidimensional SparseArray objects.
- SparseArray_summarization for SparseArray summarization methods.
- SparseArray_Ops for operations from the Ops group on SparseArray objects.
- SparseArray_Math for operations from the Math and Math2 groups on SparseArray objects.
- SparseArray_Complex for operations from the Complex group on SparseArray objects.
- SparseArray_misc for miscellaneous operations on a SparseArray object.
- SparseMatrix_mult for SparseMatrix multiplication and cross-product.
- matrixStats_methods for SparseArray col/row summarization methods.
- rowsum_methods for rowsum() methods for sparse matrices.
- randomSparseArray to generate a random SparseArray object.
- readSparseCSV to read/write a sparse matrix from/to a CSV (comma-separated values) file.
- S4 classes dgCMatrix, dgRMatrix, and lgCMatrix defined in the **Matrix** package, for the de facto standard for sparse matrix representations in the R ecosystem.
- is_sparse in the **S4Arrays** package.
- The Array class defined in the **S4Arrays** package.
- Ordinary array objects in base R.
- base::which in base R.

Examples

```
## ---------------------------------------------------------------------
## Display details of class definition & known subclasses
## ---------------------------------------------------------------------

showClass("SparseArray")

## ---------------------------------------------------------------------
## The SparseArray() constructor
## ---------------------------------------------------------------------

a <- array(rpois(9e6, lambda=0.3), dim=c(500, 3000, 6))
```

```
SparseArray(a)    # an SVT_SparseArray object

m <- matrix(rpois(9e6, lambda=0.3), ncol=500)
SparseArray(m)    # an SVT_SparseMatrix object

dgc <- sparseMatrix(i=c(4:1, 2:4, 9:12, 11:9), j=c(1:7, 1:7),
                    x=runif(14), dims=c(12, 7))
class(dgc)
SparseArray(dgc)  # an SVT_SparseMatrix object

dgr <- as(dgc, "RsparseMatrix")
class(dgr)
SparseArray(dgr)  # a COO_SparseMatrix object

## ---------------------------------------------------------------------
## nzcount(), nzwhich(), nzvals()
## ---------------------------------------------------------------------
x <- SparseArray(a)

## Get the number of nonzero array elements in 'x':
nzcount(x)

## nzwhich() returns the indices of the nonzero array elements in 'x'.
## Either as an integer (or numeric) vector of length 'nzcount(x)'
## containing "linear indices":
nzidx <- nzwhich(x)
length(nzidx)
head(nzidx)

## Or as an integer matrix with 'nzcount(x)' rows and one column per
## dimension where the rows represent "array indices" (a.k.a. "array
## coordinates"):
Mnzidx <- nzwhich(x, arr.ind=TRUE)
dim(Mnzidx)

## Each row in the matrix is an n-tuple representing the "array
## coordinates" of a nonzero element in 'x':
head(Mnzidx)
tail(Mnzidx)

## Extract the values of the nonzero array elements in 'x' and return
## them in a vector "parallel" to 'nzwhich(x)':
#nzvals <- nzvals(x)  # NOT READY YET!
#length(nzvals)       # NOT READY YET!
#head(nzvals)         # NOT READY YET!

## Sanity checks:
stopifnot(identical(nzidx, which(a != 0)))
stopifnot(identical(Mnzidx, which(a != 0, arr.ind=TRUE, useNames=FALSE)))
#stopifnot(identical(nzvals, a[nzidx]))   # NOT READY YET!
#stopifnot(identical(nzvals, a[Mnzidx]))  # NOT READY YET!
```

SparseArray-abind *Combine multidimensional SparseArray objects*

Description

Like ordinary matrices and arrays in base R, SparseMatrix derivatives can be combined by rows or columns, with `rbind()` or `cbind()`, and multidimensional SparseArray derivatives can be bound along any dimension with `abind()`.

Note that `arbind()` can also be used to combine the objects along their first dimension, and `acbind()` can be used to combine them along their second dimension.

See Also

- `cbind` in base R.
- `abind` in the **S4Arrays** package.
- SparseArray objects.
- Ordinary `array` objects in base R.

Examples

```
## ---------------------------------------------------------------------
## COMBINING SparseMatrix OBJECTS
## ---------------------------------------------------------------------

m1a <- matrix(1:15, nrow=3, ncol=5,
              dimnames=list(NULL, paste0("M1y", 1:5)))
m1b <- matrix(101:135, nrow=7, ncol=5,
              dimnames=list(paste0("M2x", 1:7), paste0("M2y", 1:5)))
sm1a <- SparseArray(m1a)
sm1b <- SparseArray(m1b)

rbind(sm1a, sm1b)

## ---------------------------------------------------------------------
## COMBINING SparseArray OBJECTS WITH 3 DIMENSIONS
## ---------------------------------------------------------------------

a2a <- array(1:105, dim=c(5, 7, 3),
             dimnames=list(NULL, paste0("A1y", 1:7), NULL))
a2b <- array(1001:1105, dim=c(5, 7, 3),
             dimnames=list(paste0("A2x", 1:5), paste0("A2y", 1:7), NULL))
sa2a <- SparseArray(a2a)
sa2b <- SparseArray(a2b)

abind(sa2a, sa2b)               # same as 'abind(sa2a, sa2b, along=3)'
abind(sa2a, sa2b, rev.along=0)  # same as 'abind(sa2a, sa2b, along=4)'

a3a <- array(1:60, dim=c(3, 5, 4),
```

```
            dimnames=list(NULL, paste0("A1y", 1:5), NULL))
a3b <- array(101:240, dim=c(7, 5, 4),
            dimnames=list(paste0("A2x", 1:7), paste0("A2y", 1:5), NULL))
sa3a <- SparseArray(a3a)
sa3b <- SparseArray(a3b)

arbind(sa3a, sa3b)  # same as 'abind(sa3a, sa3b, along=1)'

## ---------------------------------------------------------------------
## Sanity checks
## ---------------------------------------------------------------------

sm1 <- rbind(sm1a, sm1b)
m1 <- rbind(m1a, m1b)
stopifnot(identical(as.array(sm1), m1), identical(sm1, SparseArray(m1)))

sa2 <- abind(sa2a, sa2b)
stopifnot(identical(sa2, abind(sa2a, sa2b, along=3)))
a2 <- abind(a2a, a2b, along=3)
stopifnot(identical(as.array(sa2), a2), identical(sa2, SparseArray(a2)))

sa2 <- abind(sa2a, sa2b, rev.along=0)
stopifnot(identical(sa2, abind(sa2a, sa2b, along=4)))
a2 <- abind(a2a, a2b, along=4)
stopifnot(identical(as.array(sa2), a2), identical(sa2, SparseArray(a2)))

sa3 <- arbind(sa3a, sa3b)
a3 <- arbind(a3a, a3b)
stopifnot(identical(as.array(sa3), a3), identical(sa3, SparseArray(a3)))
```

`SparseArray-aperm` *SparseArray transposition*

Description

Transpose a SparseArray object by permuting its dimensions.

WORK-IN-PROGRESS

Value

COMING SOON...

See Also

- `aperm()` in base R.
- SparseArray objects.
- Ordinary array objects in base R.

Examples

```
## COMING SOON...
```

`SparseArray-Complex-methods`
'Complex' methods for SparseArray objects

Description

WORK-IN-PROGRESS

Value

COMING SOON...

See Also

- `S4groupGeneric` in the **methods** package.
- SparseArray objects.
- Ordinary array objects in base R.

Examples

```
## COMING SOON...
```

`SparseArray-dim-tuning`
Add/drop ineffective dims to/from a SparseArray object

Description

The *ineffective dimensions* of an array-like object are its dimensions that have an extent of 1.

Drop all *ineffective dimensions* from SparseArray object `x` with `drop(x)`.

Add and/or drop arbitrary *ineffective dimensions* to/from SparseArray object `x` with the `dim()` setter.

Details

The *ineffective dimensions* of an array-like object are its dimensions that have an extent of 1. For example, for a 1 x 1 x 15 x 1 x 6 array, the *ineffective dimensions* are its 1st, 2nd, and 4th dimensions.

Note that *ineffective dimensions* can be dropped or added from/to an array-like object `x` without changing its length (`prod(dim(x))`) or altering its content.

`drop(x)`**:** Drop all *ineffective dimensions* from SparseArray derivative `x`. If `x` has at most one effective dimension, then the result is returned as an ordinary vector. Otherwise it's returned as a SparseArray derivative.

`dim(x) <- value`**:** Add and/or drop arbitrary *ineffective dimensions* to/from SparseArray derivative `x`. `value` must be a vector of dimensions compatible with `dim(x)`, that is, it must preserve all the effective dimenions in their original order.

See Also

- `drop()` in base R.
- The `dim()` getter and setter in base R.
- SparseArray objects.
- Ordinary array objects in base R.

Examples

```
## An array with ineffective 1st and 4th dimensions:
a <- array(0L, dim=c(1, 1, 5, 4, 1, 3))
dimnames(a) <- list(NULL, NULL, letters[1:5], NULL, NULL, LETTERS[1:3])
a[c(1:2, 8, 10, 15:17, 20, 24, 40, 56:60)] <- (1:15)*10L
svt <- SparseArray(a)
dim(svt)

## Drop ineffective dims:
dim(drop(svt))  # the 1st, 2nd, and 5th dimensions were dropped

## Drop some ineffective dims and adds new ones:
svt2 <- svt
dim(svt2) <- c(1, 5, 4, 1, 1, 3, 1)
dim(svt2)

## Sanity check:
stopifnot(identical(as.array(drop(svt)), drop(a)))
a2 <- `dim<-`(a, c(1, 5, 4, 1, 1, 3, 1))
dimnames(a2)[c(2, 6)] <- dimnames(a)[c(3, 6)]
stopifnot(identical(as.array(svt2), a2))
```

SparseArray-Math-methods *'Math' and 'Math2' methods for SparseArray objects*

Description

SparseArray derivatives support a *subset* of operations from the `Math` and `Math2` groups. See `?S4groupGeneric` in the **methods** package for more information about the `Math` and `Math2` group generics.

IMPORTANT NOTES:

- Only operations from these groups that preserve sparsity are supported. For example, `sqrt()`, `trunc()`, `log1p()`, and `sin()` are supported, but `cumsum()`, `log()`, `cos()`, or `gamma()` are not.
- Only SVT_SparseArray objects are supported at the moment. Support for COO_SparseArray objects might be added in the future.
- `Math` and `Math2` operations only support SVT_SparseArray objects of `type()` `"double"` at the moment.

Value

A SparseArray derivative of the same dimensions as the input object.

See Also

- `S4groupGeneric` in the **methods** package.
- SparseArray objects.
- Ordinary array objects in base R.

Examples

```
m <- matrix(0, nrow=15, ncol=6)
m[c(2, 6, 12:17, 22:33, 55, 59:62, 90)] <-
    c(runif(22)*1e4, Inf, -Inf, NA, NaN)
svt <- SparseArray(m)

svt2 <- trunc(sqrt(svt))
svt2

## Sanity check:
m2 <- suppressWarnings(trunc(sqrt(m)))
stopifnot(identical(as.matrix(svt2), m2))
```

SparseArray-misc-methods *Miscellaneous operations on a SparseArray object*

Description

This man page documents various base array operations that are supported by SparseArray derivatives, and that didn't belong to any of the groups of operations documented in the other man pages of the **SparseArray** package.

Note that only COO_SparseArray objects support these operations at the moment.

Usage

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

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

## S4 method for signature 'COO_SparseArray'
is.nan(x)

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

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

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

Arguments

x	An COO_SparseArray object.
type, allowNA, keepNA	See ?base::nchar for a description of these arguments.

Details

More operations will be added in the future. For example SVT_SparseArray objects also need to support `is.na()`, `is.infinite()`, and `is.nan()`.

Value

See man pages for the corresponding default methods in the **base** package (e.g. ?base::is.na, ?base::nchar, etc...) for the value returned by these methods.

See Also

- base::is.na and base::is.infinite in base R.
- base::tolower in base R.
- base::nchar in base R.
- SparseArray objects.
- Ordinary array objects in base R.

Examples

```
## COMING SOON...
#a <- array(FALSE, dim=5:3)
#nzidx <- c(1:2, 8, 10, 15:17, 20, 24, 40, 56:60)
#a[nzidx] <- TRUE
#coo <- as(a, "COO_SparseArray")
```

SparseArray-Ops-methods

'Ops' methods for SparseArray objects

Description

SparseArray derivatives support operations from the Arith, Compare, and Logic groups, with some restrictions. All together, these groups are referred to as the Ops group. See ?S4groupGeneric in the **methods** package for more information about the Ops group generic.

IMPORTANT NOTES:

- Only SVT_SparseArray objects are supported at the moment. Support for COO_SparseArray objects might be added in the future.
- Arith operations don't support SVT_SparseArray objects of type() "complex" at the moment.

Details

Two forms of operations are supported:

1. Between an SVT_SparseArray object svt and a single value y:

   ```
   svt op y
   y op svt
   ```

   The operations from the Arith group that support this form are: *, /, ^, %%,%/%. Note that, except for * (for which both svt * y and y * svt are supported), single value y must be on the right e.g. svt ^ 3.

   All operations from the Compare group support this form, with single value y either on the left or the right. However, there are some operation-dependent restrictions on the value of y.

2. Between two SVT_SparseArray objects svt1 and svt2 of same dimensions (a.k.a. *conformable arrays*):

```
svt1 op svt2
```

The operations from the Arith group that support this form are: +, -, *.
The operations from the Compare group that support this form are: !=, <, >.

Value

A SparseArray derivative of the same dimensions as the input object(s).

See Also

- S4groupGeneric in the **methods** package.
- SparseArray objects.
- Ordinary array objects in base R.

Examples

```
m <- matrix(0L, nrow=15, ncol=6)
m[c(2, 6, 12:17, 22:33, 55, 59:62, 90)] <- 101:126
svt <- SparseArray(m)

## Can be 5x or 10x faster than with a dgCMatrix object on a big
## SVT_SparseMatrix object!
svt2 <- (svt^1.5 + svt)
svt2

## Sanity check:
m2 <- (m^1.5 + m)
stopifnot(identical(as.matrix(svt2), m2))
```

SparseArray-subassignment

SparseArray subassignment

Description

Like ordinary arrays in base R, SparseArray derivatives support subassignment via the [<- operator.

See Also

- [<- in base R.
- SparseArray objects.
- Ordinary array objects in base R.

Examples

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

svt[5:3, c(4,2,4), 2:3] <- -99L

## Sanity checks:
a[5:3, c(4,2,4), 2:3] <- -99L
stopifnot(identical(as.array(svt), a), identical(svt, SparseArray(a)))
```

SparseArray-subsetting

Subsetting a SparseArray object

Description

Like ordinary arrays in base R, SparseArray derivatives support subsetting via the single bracket operator ([).

See Also

- [in base R.
- SparseArray::drop to drop *ineffective dimensions*.
- SparseArray objects.
- Ordinary array objects in base R.

Examples

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

svt[5:3, c(4,2,4), 2:3]

svt[ , c(4,2,4), 2:3]

svt[ , c(4,2,4), -1]

svt[ , c(4,2,4), 1]

svt2 <- svt[ , c(4,2,4), 1, drop=FALSE]
svt2

## Ineffective dimensions can always be dropped as a separate step:
drop(svt2)
```

```
svt[ , c(4,2,4), integer(0)]

dimnames(a) <- list(letters[1:5], NULL, LETTERS[1:3])
svt <- SparseArray(a)

svt[c("d", "a"), c(4,2,4), "C"]

svt2 <- svt["e", c(4,2,4), , drop=FALSE]
svt2

drop(svt2)

## Sanity checks:
svt2 <- svt[5:3, c(4,2,4), 2:3]
a2   <- a  [5:3, c(4,2,4), 2:3]
stopifnot(identical(as.array(svt2), a2), identical(svt2, SparseArray(a2)))
svt2 <- svt[ , c(4,2,4), 2:3]
a2   <- a  [ , c(4,2,4), 2:3]
stopifnot(identical(as.array(svt2), a2), identical(svt2, SparseArray(a2)))
svt2 <- svt[ , c(4,2,4), -1]
a2   <- a  [ , c(4,2,4), -1]
stopifnot(identical(as.array(svt2), a2), identical(svt2, SparseArray(a2)))
svt2 <- svt[ , c(4,2,4), 1]
a2   <- a  [ , c(4,2,4), 1]
stopifnot(identical(as.array(svt2), a2), identical(svt2, SparseArray(a2)))
svt2 <- svt[ , c(4,2,4), 1, drop=FALSE]
a2   <- a  [ , c(4,2,4), 1, drop=FALSE]
stopifnot(identical(as.array(svt2), a2), identical(svt2, SparseArray(a2)))
svt2 <- drop(svt2)
a2 <- drop(a2)
stopifnot(identical(as.array(svt2), a2), identical(svt2, SparseArray(a2)))
svt2 <- svt[ , c(4,2,4), integer(0)]
a2   <- a  [ , c(4,2,4), integer(0)]
stopifnot(identical(as.array(svt2), a2),
          identical(unname(svt2), unname(SparseArray(a2))))
svt2 <- svt[c("d", "a"), c(4,2,4), "C"]
a2   <- a  [c("d", "a"), c(4,2,4), "C"]
stopifnot(identical(as.array(svt2), a2), identical(svt2, SparseArray(a2)))
svt2 <- svt["e", c(4,2,4), , drop=FALSE]
a2   <- a  ["e", c(4,2,4), , drop=FALSE]
stopifnot(identical(as.array(svt2), a2), identical(svt2, SparseArray(a2)))
svt2 <- drop(svt2)
a2 <- drop(a2)
stopifnot(identical(as.array(svt2), a2), identical(svt2, SparseArray(a2)))
```

SparseArray-summarization

SparseArray summarization methods

Description

The **SparseArray** package provides memory-efficient summarization methods for SparseArray objects. The following methods are supported at the moment: `anyNA()`, `any()`, `all()`, `min()`, `max()`, `range()`, `sum()`, `prod()`, `mean()`, `var()`, `sd()`.

More might be added in the future.

Note that these are *S4 generic functions* defined in base R and in the **BiocGenerics** package, with default methods defined in base R. This man page documents the methods defined for SparseArray objects.

Details

All these methods operate *natively* on the COO_SparseArray or SVT_SparseArray representation, for maximum efficiency.

Value

See man pages for the corresponding default methods in the **base** package (e.g. `?base::range`, `?base::mean`, etc...) for the value returned by these methods.

See Also

- SparseArray objects.
- The man pages for the various default methods defined in the **base** package e.g. `base::range`, `base::mean`, `base::anyNA`, etc...

Examples

```
m0 <- matrix(0L, nrow=6, ncol=4)
m0[c(1:2, 8, 10, 15:17, 24)] <- (1:8)*10L
m0[5, 2] <- NA
svt0 <- as(m0, "SVT_SparseMatrix")
svt0

anyNA(svt0)

range(svt0)

range(svt0, na.rm=TRUE)

sum(svt0, na.rm=TRUE)

sd(svt0, na.rm=TRUE)

## Sanity checks:
stopifnot(
  identical(anyNA(svt0), anyNA(m0)),
  identical(range(svt0), range(m0)),
  identical(range(svt0, na.rm=TRUE), range(m0, na.rm=TRUE)),
  identical(sum(svt0), sum(m0)),
  identical(sum(svt0, na.rm=TRUE), sum(m0, na.rm=TRUE)),
```

```
  all.equal(sd(svt0, na.rm=TRUE), sd(m0, na.rm=TRUE))
)
```

SparseMatrix-mult *SparseMatrix multiplication and cross-product*

Description

Like ordinary matrices in base R, SparseMatrix derivatives can be multiplied with the `%*%` operator. They also support `crossprod()` and `tcrossprod()`.

Value

The `%*%`, `crossprod()` and `tcrossprod()` methods for SparseMatrix objects always return an *ordinary* matrix of `type()` `"double"`.

Note

Matrix multiplication and cross-product of SVT_SparseMatrix objects are multithreaded. See `set_SparseArray_nthread` for how to control the number of threads.

See Also

- `%*%` and `crossprod` in base R.
- SparseMatrix objects.
- `S4Arrays::type` in the **S4Arrays** package to get the type of the elements of an array-like object.
- Ordinary matrix objects in base R.

Examples

```
m1 <- matrix(0L, nrow=15, ncol=6)
m1[c(2, 6, 12:17, 22:33, 55, 59:62, 90)] <- 101:126
svt1 <- as(m1, "SVT_SparseMatrix")

set.seed(333)
svt2 <- poissonSparseMatrix(nrow=6, ncol=7, density=0.2)

svt1 %*% svt2
m1   %*% svt2

## Unary crossprod() and tcrossprod():
crossprod(svt1)              # same as t(svt1) %*% svt1
tcrossprod(svt1)             # same as svt1 %*% t(svt1)

## Binary crossprod() and tcrossprod():
crossprod(svt1[1:6, ], svt2)  # same as t(svt1[1:6, ]) %*% svt2
tcrossprod(svt1, t(svt2))     # same as svt1 %*% svt2
```

```
## Sanity checks:
m12 <- m1 %*% as.matrix(svt2)
stopifnot(
  identical(svt1 %*% svt2, m12),
  identical(m1 %*% svt2, m12),
  identical(crossprod(svt1), t(svt1) %*% svt1),
  identical(tcrossprod(svt1), svt1 %*% t(svt1)),
  identical(crossprod(svt1[1:6, ], svt2), t(svt1[1:6, ]) %*% svt2),
  identical(tcrossprod(svt1, t(svt2)), m12)
)
```

`sparseMatrix-utils` *Internal utilities to handle sparseMatrix derivatives*

Description

The **SparseArray** package defines some utilities to handle sparseMatrix derivatives (e.g. dgCMatrix and lgCMatrix objects) from the **Matrix** package. These are for internal use only.

See Also

- dgCMatrix objects implemented in the **Matrix** package.

`SVT_SparseArray-class` *SVT_SparseArray objects*

Description

The SVT_SparseArray class is a new container for efficient in-memory representation of multidimensional sparse arrays. It uses the *SVT layout* to represent the nonzero multidimensional data internally.

An SVT_SparseMatrix object is an SVT_SparseArray object of 2 dimensions.

Note that SVT_SparseArray and SVT_SparseMatrix objects replace the older and less efficient COO_SparseArray and COO_SparseMatrix objects.

Usage

```
## Constructor function:
SVT_SparseArray(x, type=NA)
```

Arguments

x	An ordinary matrix or array, or a dgCMatrix/lgCMatrix object, or any matrix-like or array-like object that supports coercion to SVT_SparseArray.

type	A single string specifying the requested type of the object.
	Normally, the SVT_SparseArray object returned by the constructor function has the same `type()` as `x` but the user can use the `type` argument to request a different type. Note that doing:
	`svt <- SVT_SparseArray(x, type=type)`
	is equivalent to doing:
	`svt <- SVT_SparseArray(x)`
	`type(svt) <- type`
	but the former is more convenient and will generally be more efficient.
	Supported types are all R atomic types plus `"list"`.

Details

SVT_SparseArray is a concrete subclass of the SparseArray virtual class. This makes SVT_SparseArray objects SparseArray derivatives.

The nonzero data in a SVT_SparseArray object is stored in a *Sparse Vector Tree*. We'll refer to this internal data representation as the *SVT layout*. See the "SVT layout" section below for more information.

The SVT layout is similar to the CSC layout (compressed, sparse, column-oriented format) used by CsparseMatrix derivatives from the **Matrix** package, like dgCMatrix or lgCMatrix objects, but with the following improvements:

- The SVT layout supports sparse arrays of arbitrary dimensions.
- With the SVT layout, the sparse data can be of any type. Whereas CsparseMatrix derivatives only support sparse data of type `"double"` or `"logical"` at the moment.
- The SVT layout imposes no limit on the number of nonzero elements that can be stored. With dgCMatrix/lgCMatrix objects, this number must be < 2^31.
- Overall, the SVT layout allows more efficient operations on SVT_SparseArray objects.

Value

An SVT_SparseArray or SVT_SparseMatrix object.

SVT layout

An SVT (Sparse Vector Tree) is a tree of depth N - 1 where N is the number of dimensions of the sparse array.

The leaves in the tree can only be of two kinds: NULL or *leaf vector*. Leaves that are leaf vectors can only be found at the deepest level in the tree (i.e. at depth N - 1). All leaves found at a lower depth must be NULLs.

A leaf vector represents a sparse vector of length equal to the first dimension of the sparse array. This is done using a set of offset/value pairs sorted by strictly ascending offset. More precisely, a leaf vector is represented by an ordinary list of 2 parallel vectors:

1. an integer vector of offsets (i.e. 0-based positions);
2. a vector (atomic or list) of nonzero values.

The 2nd vector determines the type of the leaf vector i.e. `"double"`, `"integer"`, `"logical"`, etc... All the leaf vectors in the SVT have the type of the sparse array.

Examples:

- An SVT_SparseArray object with 1 dimension has its nonzero data stored in an SVT of depth 0. Such SVT is represented by a single "leaf vector".
- An SVT_SparseArray object with 2 dimensions has its nonzero data stored in an SVT of depth 1. Such SVT is represented by a list of length the extend of the 2nd dimension (number of columns). Each list element is an SVT of depth 0 (as described above), or a NULL if the corresponding column is empty (i.e. has no nonzero data).

  For example, the nonzero data of an 8-column sparse matrix will be stored in an SVT that looks like this:

```
    .------------------list-of-length-8-----------------.
   /       /       /      |       |      \       \       \
  |       |       |       |       |       |       |       |
 leaf    leaf    NULL    leaf    leaf    leaf    leaf    NULL
vector  vector          vector  vector  vector  vector
```

  The NULL leaves represent the empty columns (i.e. the columns with no nonzero elements).
- An SVT_SparseArray object with 3 dimensions has its nonzero data stored in an SVT of depth 2. Such SVT is represented by a list of length the extend of the 3rd dimension. Each list element must be an SVT of depth 1 (as described above) that stores the nonzero data of the corresponding 2D slice, or a NULL if the 2D slice is empty (i.e. has no nonzero data).

See Also

- The SparseArray class for the virtual parent class of COO_SparseArray and SVT_SparseArray.
- S4 classes dgCMatrix and lgCMatrix defined in the **Matrix** package, for the de facto standard of sparse matrix representations in the R ecosystem.
- Virtual class CsparseMatrix defined in the **Matrix** package for the parent class of all classes that use the "CSC layout".
- Ordinary array objects in base R.

Examples

```
## ---------------------------------------------------------------------
## EXAMPLE 1
## ---------------------------------------------------------------------
m0 <- matrix(0L, nrow=6, ncol=4)
m0[c(1:2, 8, 10, 15:17, 24)] <- (1:8)*10L
m0

svt0 <- as(m0, "SVT_SparseMatrix")
svt0
```

```
## CSC (Compressed sparse column) layout vs SVT layout:

dgcm <- as(m0, "dgCMatrix")
dgcm@x
dgcm@i
dgcm@p

str(svt0)

## ---------------------------------------------------------------------
## EXAMPLE 2
## ---------------------------------------------------------------------
m1 <- matrix(rpois(54e6, lambda=0.4), ncol=1200)

## Note that 'SparseArray(m1)' can also be used for this:
svt1 <- SVT_SparseArray(m1)
svt1

dgcm1 <- as(m1, "dgCMatrix")

## Compare type and memory footprint:
type(svt1)
object.size(svt1)
type(dgcm1)
object.size(dgcm1)

## Transpose:
system.time(svt <- t(t(svt1)))
system.time(dgcm <- t(t(dgcm1)))
identical(svt, svt1)
identical(dgcm, dgcm1)

## rbind():
m2 <- matrix(rpois(45e6, lambda=0.4), ncol=1200)
svt2 <- SVT_SparseArray(m2)
dgcm2 <- as(m2, "dgCMatrix")

system.time(rbind(svt1, svt2))
system.time(rbind(dgcm1, dgcm2))
```

`thread-control` *Number of threads used by SparseArray operations*

Description

Use `get_SparseArray_nthread` or `set_SparseArray_nthread` to get or set the number of threads to use by the multithreaded operations implemented in the **SparseArray** package.

Usage

```
get_SparseArray_nthread()
set_SparseArray_nthread(nthread=NULL)
```

Arguments

nthread	The number of threads to use by multithreaded operations implemented in the **SparseArray** package. On systems where OpenMP is available, this must be NULL or an integer value >= 1. When NULL (the default), a "reasonable" value will be used that never exceeds one third of the number of logical cpus available on the machine. On systems where OpenMP is not available, the supplied nthread is ignored and set_SparseArray_nthread() is a no-op.

Details

Multithreaded operations in the **SparseArray** package are implemented in C with OpenMP (https://www.openmp.org/).

Note that OpenMP is not available on all systems. On systems where it's available, get_SparseArray_nthread() is guaranteed to return a value >= 1. On systems where it's not available (e.g. macOS), get_SparseArray_nthread() returns 0 and set_SparseArray_nthread() is a no-op.

IMPORTANT: The portable way to disable multithreading is by calling set_SparseArray_nthread(1), NOT set_SparseArray_nthread(0) (the latter returns an error on systems where OpenMP is available).

Value

get_SparseArray_nthread() returns an integer value >= 1 on systems where OpenMP is available, and 0 on systems where it's not.

set_SparseArray_nthread() returns the *previous* nthread value, that is, the value returned by get_SparseArray_nthread() before the call to set_SparseArray_nthread(). Note that the value is returned invisibly.

See Also

- SparseMatrix_mult for SparseMatrix multiplication and cross-product.
- matrixStats_methods for SparseArray col/row summarization methods.
- SparseArray objects.

Examples

```
get_SparseArray_nthread()

if (get_SparseArray_nthread() != 0) {  # multithreading is available
    svt1 <- poissonSparseMatrix(77000L, 15000L, density=0.01)

    ## 'user' time is typically N x 'elapsed' time where N is roughly the
    ## number of threads that was effectively used:
```

```
    system.time(cv1 <- colVars(svt1))

    svt2 <- poissonSparseMatrix(77000L, 300L, density=0.3) * 0.77
    system.time(cp12 <- crossprod(svt1, svt2))

    prev_nthread <- set_SparseArray_nthread(1)  # disable multithreading
    system.time(cv1 <- colVars(svt1))
    system.time(cp12 <- crossprod(svt1, svt2))

    ## Restore previous 'nthread' value:
    set_SparseArray_nthread(prev_nthread)
}
```

Index