Package: RaggedExperiment (via r-universe)

November 24, 2026

Title Representation of Sparse Experiments and Assays Across Samples

Version 1.38.0

Description This package provides a flexible representation of copy number, mutation, and other data that fit into the ragged array schema for genomic location data. The basic representation of such data provides a rectangular flat table interface to the user with range information in the rows and samples/specimen in the columns. The RaggedExperiment class derives from a GRangesList representation and provides a semblance of a rectangular dataset.

License Artistic-2.0

biocViews Infrastructure, DataRepresentation

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

URL https://bioconductor.github.io/RaggedExperiment, https://bioconductor.org/packages/RaggedExperiment

VignetteBuilder knitr

Depends R (>= 4.5.0), GenomicRanges (>= 1.61.1)

Imports BiocBaseUtils, BiocGenerics, Seqinfo, IRanges, Matrix, MatrixGenerics, methods, S4Vectors, stats, SummarizedExperiment (>= 1.39.1), utils

Suggests BiocStyle, knitr, rmarkdown, testthat, GenomeInfoDb, MultiAssayExperiment

RoxygenNote 7.3.2

Encoding UTF-8

Date 2024-10-17

Config/pak/sysreqs zlib1g-dev

Repository Bioconductor 3.24

Date/Publication 2026-11-23 22:38:54 UTC

RemoteUrl https://github.com/bioc/RaggedExperiment

RemoteRef RELEASE_3_24

RemoteSha 2aad9cfe439dd4b343ca5f2e967e9abf6a025fc5

Contents

RaggedExperiment-package 2
assay-functions 3
RaggedExperiment-class 6
sparseSummarizedExperiment 15

Index **18**

`RaggedExperiment-package`

RaggedExperiment: Range-based data representation package

Description

RaggedExperiment allows the user to represent, copy number, mutation, and other types of range-based data formats where optional information about samples can be provided. At the backbone of this package is the GRangesList class. The RaggedExperiment class uses this representation and presents the data in a couple of different ways:

- rowRanges
- colData

The rowRanges method will return the internal `GRangesList` representation of the dataset. A distinction between the SummarizedExperiment and the `RaggedExperiment` classes is that the RaggedExperiment class allows for ragged ranges, meaning that there may be a different number of ranges or rows per sample.

Author(s)

Maintainer: Marcel Ramos <marcel.ramos@sph.cuny.edu> (ORCID)

Authors:

- Martin Morgan <martin.morgan@roswellpark.org>

Other contributors:

- Lydia King <L.King18@nuigalway.ie> [contributor]

See Also

Useful links:

- https://bioconductor.github.io/RaggedExperiment
- https://bioconductor.org/packages/RaggedExperiment
- Report bugs at https://github.com/Bioconductor/RaggedExperiment/issues

assay-functions *Create simplified representation of ragged assay data.*

Description

These methods transform `assay()` from the default (i.e., `sparseAssay()`) representation to various forms of more dense representation. `compactAssay()` collapses identical ranges across samples into a single row. `disjoinAssay()` creates disjoint (non-overlapping) regions, simplifies values within each sample in a user-specified manner, and returns a matrix of disjoint regions x samples.

This method transforms `assay()` from the default (i.e., `sparseAssay()`) representation to a reduced representation summarizing each original range overlapping ranges in `query`. Reduction in each cell can be tailored to indivdual needs using the `simplifyReduce` functional argument.

Usage

```
sparseAssay(
  x,
  i = 1,
  withDimnames = TRUE,
  background = NA_integer_,
  sparse = FALSE
)

compactAssay(
  x,
  i = 1,
  withDimnames = TRUE,
  background = NA_integer_,
  sparse = FALSE
)

disjoinAssay(
  x,
  simplifyDisjoin,
  i = 1,
  withDimnames = TRUE,
  background = NA_integer_
)

qreduceAssay(
  x,
  query,
  simplifyReduce,
  i = 1,
  withDimnames = TRUE,
  background = NA_integer_
)
```

Arguments

x	A RaggedExperiment object
i	integer(1) or character(1) name of assay to be transformed.
withDimnames	logical(1) include dimnames on the returned matrix. When there are no explict rownames, these are manufactured with as.character(rowRanges(x)); rownames are always manufactured for compactAssay() and disjoinAssay().
background	A value (default NA) for the returned matrix after *Assay operations
sparse	logical(1) whether to return a sparseMatrix representation
simplifyDisjoin	A function / functional operating on a *List, where the elements of the list are all within-sample assay values from ranges overlapping each disjoint range. For instance, to use the simplifyDisjoin=mean of overlapping ranges, where ranges are characterized by integer-valued scores, the entries are calculated as

```
                  a
original: |-----------|
                     b
                |----------|

             a    a, b   b
disjoint: |----|------|---|

values <- IntegerList(a, c(a, b), b)
simplifyDisjoin(values)
```

query	GRanges providing regions over which reduction is to occur.
simplifyReduce	A function / functional accepting arguments score, range, and qrange:

- score A *List, where each list element corresponds to a cell in the matrix to be returned by qreduceAssay. Vector elements correspond to ranges overlapping query. The *List objects support many vectorized mathematical operations, so simplifyReduce can be implemented efficiently.
- range A GRangesList instance, 'parallel' to score. Each element of the list corresponds to a cell in the matrix to be returned by qreduceAssay. Each range in the element corresponds to the range for which the score element applies.
- qrange A GRanges instance with the same length as unlist(score), providing the query range window to which the corresponding scores apply.

Value

sparseAssay(): A matrix() with dimensions dim(x). Elements contain the assay value for the *i*th range and *j*th sample. Use 'sparse=TRUE' to obtain a sparseMatrix assay representation.

compactAssay(): Samples with identical range are placed in the same row. Non-disjoint ranges are NOT collapsed. Use 'sparse=TRUE' to obtain a sparseMatrix assay representation.

`disjoinAssay()`: A matrix with number of rows equal to number of disjoint ranges across all samples. Elements of the matrix are summarized by applying `simplifyDisjoin()` to assay values of overlapping ranges

`qreduceAssay()`: A matrix() with dimensions `length(query)` x `ncol(x)`. Elements contain assay values for the ith query range and jth sample, summarized according to the function `simplifyReduce`.

Examples

```
re4 <- RaggedExperiment(GRangesList(
    GRanges(c(A = "chr1:1-10:-", B = "chr1:8-14:-", C = "chr2:15-18:+"),
        score = 3:5),
    GRanges(c(D = "chr1:1-10:-", E = "chr2:11-18:+"), score = 1:2)
), colData = DataFrame(id = 1:2))

query <- GRanges(c("chr1:1-14:-", "chr2:11-18:+"))

weightedmean <- function(scores, ranges, qranges)
{
    ## weighted average score per query range
    ## the weight corresponds to the size of the overlap of each
    ## overlapping subject range with the corresponding query range
    isects <- pintersect(ranges, qranges)
    sum(scores * width(isects)) / sum(width(isects))
}

qreduceAssay(re4, query, weightedmean)

## Not run:
    ## Extended example: non-silent mutations, summarized by genic
    ## region
    suppressPackageStartupMessages({
        library(TxDb.Hsapiens.UCSC.hg19.knownGene)
        library(org.Hs.eg.db)
        library(GenomeInfoDb)
        library(MultiAssayExperiment)
        library(curatedTCGAData)
        library(TCGAutils)
    })

    ## TCGA MultiAssayExperiment with RaggedExperiment data
    mae <- curatedTCGAData("ACC", c("RNASeq2GeneNorm", "CNASNP", "Mutation"),
        version = "1.1.38", dry.run = FALSE)

    ## genomic coordinates
    gn <- genes(TxDb.Hsapiens.UCSC.hg19.knownGene)
    gn <- keepStandardChromosomes(granges(gn), pruning.mode="coarse")
    seqlevelsStyle(gn) <- "NCBI"
    genome(gn)
    gn <- unstrand(gn)

    ## reduce mutations, marking any genomic range with non-silent
    ## mutation as FALSE
```

```
    nonsilent <- function(scores, ranges, qranges)
        any(scores != "Silent")
    mre <- mae[["ACC_Mutation-20160128"]]
    seqlevelsStyle(rowRanges(mre)) <- "NCBI"
    ## hack to make genomes match
    genome(mre) <- paste0(correctBuild(unique(genome(mre)), "NCBI"), ".p13")
    mutations <- qreduceAssay(mre, gn, nonsilent, "Variant_Classification")
    genome(mre) <- correctBuild(unique(genome(mre)), "NCBI")

    ## reduce copy number
    re <- mae[["ACC_CNASNP-20160128"]]
    class(re)
    ## [1] "RaggedExperiment"
    seqlevelsStyle(re) <- "NCBI"
    genome(re) <- "GRCh37.p13"
    cn <- qreduceAssay(re, gn, weightedmean, "Segment_Mean")
    genome(re) <- "GRCh37"

    ## ALTERNATIVE
    ##
    ## TCGAutils helper function to convert RaggedExperiment objects to
    ## RangedSummarizedExperiment based on annotated gene ranges
    mae2 <- mae
    mae2[[1L]] <- re
    mae2[[2L]] <- mre
    qreduceTCGA(mae2)

## End(Not run)
```

RaggedExperiment-class *RaggedExperiment objects*

Description

The RaggedExperiment class is a container for storing range-based data, including but not limited to copy number data, and mutation data. It can store a collection of GRanges objects, as it is derived from the GenomicRangesList.

Usage

```
RaggedExperiment(..., colData = DataFrame(), metadata = list())

## S4 method for signature 'RaggedExperiment'
seqinfo(x)

## S4 replacement method for signature 'RaggedExperiment'
seqinfo(x, new2old = NULL, pruning.mode = c("error", "coarse", "fine", "tidy")) <- value
```

```
## S4 method for signature 'RaggedExperiment'
rowRanges(x, ...)

## S4 replacement method for signature 'RaggedExperiment,GRanges'
rowRanges(x, ...) <- value

## S4 method for signature 'RaggedExperiment'
mcols(x, use.names = FALSE, ...)

## S4 replacement method for signature 'RaggedExperiment'
mcols(x, ...) <- value

## S4 method for signature 'RaggedExperiment'
rowData(x, use.names = TRUE, ...)

## S4 replacement method for signature 'RaggedExperiment'
rowData(x, ...) <- value

## S4 method for signature 'RaggedExperiment'
dim(x)

## S4 method for signature 'RaggedExperiment'
dimnames(x)

## S4 replacement method for signature 'RaggedExperiment,list'
dimnames(x) <- value

## S4 replacement method for signature 'RaggedExperiment,ANY'
dimnames(x) <- value

## S4 method for signature 'RaggedExperiment'
length(x)

## S4 method for signature 'RaggedExperiment'
colData(x, ...)

## S4 replacement method for signature 'RaggedExperiment,DataFrame'
colData(x) <- value

## S4 method for signature 'RaggedExperiment,missing'
assay(x, i, withDimnames = TRUE, ...)

## S4 method for signature 'RaggedExperiment,ANY'
assay(x, i, withDimnames = TRUE, ...)

## S4 method for signature 'RaggedExperiment'
assays(x, withDimnames = TRUE, ...)
```

```
## S4 method for signature 'RaggedExperiment'
assayNames(x, ...)

## S4 method for signature 'RaggedExperiment'
show(object)

## S4 method for signature 'RaggedExperiment'
as.list(x, ...)

## S4 method for signature 'RaggedExperiment'
as.data.frame(x, row.names = NULL, optional = FALSE, ...)

## S4 method for signature 'RaggedExperiment'
x$name

## S4 method for signature 'RaggedExperiment,ANY,ANY,ANY'
x[i, j, ..., drop = TRUE]

## S4 method for signature 'RaggedExperiment,Vector'
overlapsAny(
  query,
  subject,
  maxgap = 0L,
  minoverlap = 1L,
  type = c("any", "start", "end", "within", "equal"),
  ...
)

## S4 method for signature 'RaggedExperiment,Vector'
subsetByOverlaps(
  x,
  ranges,
  maxgap = -1L,
  minoverlap = 0L,
  type = c("any", "start", "end", "within", "equal"),
  invert = FALSE,
  ...
)

## S4 method for signature 'RaggedExperiment'
subset(x, subset, select, ...)
```

Arguments

...	Constructor: GRanges, list of GRanges, or GRangesList OR assay: Additional arguments for assay. See details for more information.
colData	A DataFrame describing samples. Length of rowRanges must equal the number of rows in colData

`metadata` A `list` to include in the metadata. Any metadata included in the input objects are lost.

`x` A RaggedExperiment object.

`new2old` The `new2old` argument allows the user to rename, drop, add and/or reorder the "sequence levels" in `x`.

`new2old` can be `NULL` or an integer vector with one element per entry in Seqinfo object `value` (i.e. `new2old` and `value` must have the same length) describing how the "new" sequence levels should be mapped to the "old" sequence levels, that is, how the entries in `value` should be mapped to the entries in `seqinfo(x)`. The values in `new2old` must be >= 1 and <= `length(seqinfo(x))`. `NA`s are allowed and indicate sequence levels that are being added. Old sequence levels that are not represented in `new2old` will be dropped, but this will fail if those levels are in use (e.g. if `x` is a GRanges object with ranges defined on those sequence levels) unless a pruning mode is specified via the `pruning.mode` argument (see below).

If `new2old=NULL`, then sequence levels can only be added to the existing ones, that is, `value` must have at least as many entries as `seqinfo(x)` (i.e. `length(values) >= length(seqinfo(x))`) and also `seqlevels(values)[seq_len(length(seqlevels(x)))]` must be identical to `seqlevels(x)`.

Note that most of the times it's easier to proceed in 2 steps:

1. First align the seqlevels on the left (`seqlevels(x)`) with the seqlevels on the right.
2. Then call `seqinfo(x) <- value`. Because `seqlevels(x)` and `seqlevels(value)` now are identical, there's no need to specify `new2old`.

This 2-step approach will typically look like this:

```
seqlevels(x) <- seqlevels(value)  # align seqlevels
seqinfo(x) <- seqinfo(value)  # guaranteed to work
```

Or, if `x` has seqlevels not in `value`, it will look like this:

```
seqlevels(x, pruning.mode="coarse") <- seqlevels(value)
seqinfo(x) <- seqinfo(value)  # guaranteed to work
```

The `pruning.mode` argument will control what happens to `x` when some of its seqlevels get droppped. See below for more information.

`pruning.mode` When some of the seqlevels to drop from `x` are in use (i.e. have ranges on them), the ranges on these sequences need to be removed before the seqlevels can be dropped. We call this *pruning*. The `pruning.mode` argument controls how to *prune* `x`. Four pruning modes are currently defined: `"error"`, `"coarse"`, `"fine"`, and `"tidy"`. `"error"` is the default. In this mode, no pruning is done and an error is raised. The other pruning modes do the following:

- `"coarse"`: Remove the elements in `x` where the seqlevels to drop are in use. Typically reduces the length of `x`. Note that if `x` is a list-like object (e.g. GRangesList, GAlignmentPairs, or GAlignmentsList), then any list element in `x` where at least one of the sequence levels to drop is in

use is *fully* removed. In other words, when `pruning.mode="coarse"`, the `seqlevels` setter will keep or remove *full list elements* and not try to change their content. This guarantees that the exact ranges (and their order) inside the individual list elements are preserved. This can be a desirable property when the list elements represent compound features like exons grouped by transcript (stored in a GRangesList object as returned by `exonsBy( , by="tx")`), or paired-end or fusion reads, etc...

- `"fine"`: Supported on list-like objects only. Removes the ranges that are on the sequences to drop. This removal is done within each list element of the original object `x` and doesn't affect its length or the order of its list elements. In other words, the pruned object is guaranteed to be *parallel* to the original object.
- `"tidy"`: Like the `"fine"` pruning above but also removes the list elements that become empty as the result of the pruning. Note that this pruning mode is particularly well suited on a GRangesList object that contains transcripts grouped by gene, as returned by `transcriptsBy( , by="gene")`. Finally note that, as a convenience, this pruning mode is supported on non list-like objects (e.g. GRanges or GAlignments objects) and, in this case, is equivalent to the `"coarse"` mode.

See the "B. DROP SEQLEVELS FROM A LIST-LIKE OBJECT" section in the examples below for an extensive illustration of these pruning modes.

`value`
- dimnames: A `list` of dimension names
- mcols: A `DataFrame` representing the assays

`use.names` (logical default FALSE) whether to propagate rownames from the object to rownames of metadata `DataFrame`

`i` logical(1), integer(1), or character(1) indicating the assay to be reported. For `[`, `i` can be any supported `Vector` object, e.g., `GRanges`.

`withDimnames` logical (default TRUE) whether to use dimension names in the resulting object

`object` A RaggedExperiment object.

`row.names` `NULL` or a character vector giving the row names for the data frame. Missing values are not allowed.

`optional` logical. If `TRUE`, setting row names and converting column names (to syntactic names: see `make.names`) is optional. Note that all of R's **base** package `as.data.frame()` methods use `optional` only for column names treatment, basically with the meaning of `data.frame(*, check.names = !optional)`. See also the `make.names` argument of the `matrix` method.

`name` a literal character string or a name (possibly backtick quoted). For extraction, this is normally (see under 'Environments') partially matched to the `names` of the object.

`j` integer(), character(), or logical() index selecting columns from RaggedExperiment

`drop` logical (default TRUE) whether to drop empty samples

`query` A RaggedExperiment instance.

`subject`, `ranges` Each of them can be an IntegerRanges (e.g. IRanges, Views) or IntegerRangesList (e.g. IRangesList, ViewsList) derivative. In addition, if `subject` or `ranges` is an IntegerRanges object, `query` or `x` can be an integer vector to be converted to length-one ranges.

If `query` (or `x`) is an IntegerRangesList object, then `subject` (or `ranges`) must also be an IntegerRangesList object.

If both arguments are list-like objects with names, each list element from the 2nd argument is paired with the list element from the 1st argument with the matching name, if any. Otherwise, list elements are paired by position. The overlap is then computed between the pairs as described below.

If `subject` is omitted, `query` is queried against itself. In this case, and only this case, the `drop.self` and `drop.redundant` arguments are allowed. By default, the result will contain hits for each range against itself, and if there is a hit from A to B, there is also a hit for B to A. If `drop.self` is `TRUE`, all self matches are dropped. If `drop.redundant` is `TRUE`, only one of A->B and B->A is returned.

`maxgap` A single integer >= -1.

If `type` is set to `"any"`, `maxgap` is interpreted as the maximum *gap* that is allowed between 2 ranges for the ranges to be considered as overlapping. The *gap* between 2 ranges is the number of positions that separate them. The *gap* between 2 adjacent ranges is 0. By convention when one range has its start or end strictly inside the other (i.e. non-disjoint ranges), the *gap* is considered to be -1.

If `type` is set to anything else, `maxgap` has a special meaning that depends on the particular `type`. See `type` below for more information.

`minoverlap` A single non-negative integer.

Only ranges with a minimum of `minoverlap` overlapping positions are considered to be overlapping.

When `type` is `"any"`, at least one of `maxgap` and `minoverlap` must be set to its default value.

`type` By default, any overlap is accepted. By specifying the `type` parameter, one can select for specific types of overlap. The types correspond to operations in Allen's Interval Algebra (see references). If `type` is `start` or `end`, the intervals are required to have matching starts or ends, respectively. Specifying `equal` as the type returns the intersection of the `start` and `end` matches. If `type` is `within`, the query interval must be wholly contained within the subject interval. Note that all matches must additionally satisfy the `minoverlap` constraint described above.

The `maxgap` parameter has special meaning with the special overlap types. For `start`, `end`, and `equal`, it specifies the maximum difference in the starts, ends or both, respectively. For `within`, it is the maximum amount by which the subject may be wider than the query. If `maxgap` is set to -1 (the default), it's replaced internally by 0.

`invert` If `TRUE`, keep only the ranges in `x` that do *not* overlap `ranges`.

`subset` logical expression indicating elements or rows to keep: missing values are taken as false.

`select` If `query` is an IntegerRanges derivative: When `select` is `"all"` (the default), the results are returned as a Hits object. Otherwise the returned value is an integer vector *parallel* to `query` (i.e. same length) containing the first, last, or arbitrary overlapping interval in `subject`, with `NA` indicating intervals that did not overlap any intervals in `subject`.

If `query` is an IntegerRangesList derivative: When `select` is `"all"` (the default), the results are returned as a HitsList object. Otherwise the returned value depends on the `drop` argument. When `select != "all" && !drop`, an IntegerList is returned, where each element of the result corresponds to a space in `query`. When `select != "all" && drop`, an integer vector is returned containing indices that are offset to align with the unlisted `query`.

Value

constructor returns a `RaggedExperiment` object

'rowRanges' returns a `GRanges` object summarizing ranges corresponding to `assay()` rows.

'rowRanges<-' returns a `RaggedExperiment` object with replaced ranges

'mcols' returns a `DataFrame` object of the metadata columns

'assays' returns a `SimpleList`

'overlapsAny' returns a logical vector of length equal to the number of rows in the `query`; `TRUE` when the copy number region overlaps the `subject`.

'subsetByOverlaps' returns a RaggedExperiment containing only copy number regions overlapping `subject`.

Methods (by generic)

- `seqinfo(RaggedExperiment)`: seqinfo accessor
- `seqinfo(RaggedExperiment) <- value`: Replace seqinfo metadata of the ranges
- `rowRanges(RaggedExperiment)`: rowRanges accessor
- `rowRanges(x = RaggedExperiment) <- value`: rowRanges replacement
- `mcols(RaggedExperiment)`: get the metadata columns of the ranges, rectangular representation of the 'assays'
- `mcols(RaggedExperiment) <- value`: set the metadata columns of the ranges corresponding to the assays
- `rowData(RaggedExperiment)`: get the rowData or metadata for the ranges
- `rowData(RaggedExperiment) <- value`: set the rowData or metadata for the ranges
- `dim(RaggedExperiment)`: get dimensions (number of sample-specific row ranges by number of samples)
- `dimnames(RaggedExperiment)`: get row (sample-specific) range names and sample names
- `dimnames(x = RaggedExperiment) <- value`: set row (sample-specific) range names and sample names
- `dimnames(x = RaggedExperiment) <- value`: set row range names and sample names to NULL

- `length(RaggedExperiment)`: get the length of row vectors in the object, similar to SummarizedExperiment
- `colData(RaggedExperiment)`: get column data
- `colData(x = RaggedExperiment) <- value`: change the colData
- `assay(x = RaggedExperiment, i = missing)`: assay missing method uses first metadata column
- `assay(x = RaggedExperiment, i = ANY)`: assay numeric method.
- `assays(RaggedExperiment)`: assays
- `assayNames(RaggedExperiment)`: names in each assay
- `show(RaggedExperiment)`: show method
- `as.list(RaggedExperiment)`: Allow extraction of metadata columns as a plain `list`
- `as.data.frame(RaggedExperiment)`: Allow conversion to plain `data.frame`
- `$`: Easily access the `colData` columns with the dollar sign operator
- `x[i`: Subset a RaggedExperiment object
- `overlapsAny(query = RaggedExperiment, subject = Vector)`: Determine whether copy number ranges defined by `query` overlap ranges of `subject`.
- `subsetByOverlaps(x = RaggedExperiment, ranges = Vector)`: Subset the RaggedExperiment to contain only copy number ranges overlapping ranges of `subject`.
- `subset(RaggedExperiment)`: subset helper function for dividing by rowData and / or colData values

Constructors

`RaggedExperiment(..., colData=DataFrame())`: Creates a RaggedExperiment object using multiple `GRanges` objects or a list of `GRanges` objects. Additional column data may be provided as a `DataFrame` object.

Accessors

In the following, 'x' represents a `RaggedExperiment` object:

`rowRanges(x)`:

Get the ranged data. Value is a `GenomicRanges` object.

`assays(x)`:

Get the assays. Value is a `SimpleList`.

`assay(x, i)`:

An alternative to `assays(x)[[i]]` to get the *i*th (default first) assay element.

`mcols(x)`, `mcols(x) <- value`:

Get or set the metadata columns. For `RaggedExperiment`, the columns correspond to the assay *i*th elements.

`rowData(x)`, `rowData(x) <- value`:

Get or set the row data. Value is a `DataFrame` object. Also corresponds to the `mcols` data.

Note for advanced users and developers. Both `mcols` and `rowData` setters may reduce the size of the internal `RaggedExperiment` data representation. Particularly after subsetting, the internal row index is modified and such setter operations will use the index to subset the data and reduce the "rows" of the internal data representation.

Subsetting

`x[i, j]`: Get ranges or elements (`i` and `j`, respectively) with optional metadata columns where `i` or `j` can be missing, an NA-free logical, numeric, or character vector.

Coercion

In the following, 'object' represents a `RaggedExperiment` object:

`as(object, "GRangesList")`:

Creates a GRangesList object from a `RaggedExperiment`.

`as(from, "RaggedExperiment")`:

Creates a `RaggedExperiment` object from a GRangesList, or GRanges object.

Examples

```
## Create an empty RaggedExperiment instance
re0 <- RaggedExperiment()
re0

## Create a couple of GRanges objects with row ranges names
sample1 <- GRanges(
    c(a = "chr1:1-10:-", b = "chr1:11-18:+"),
    score = 1:2)
sample2 <- GRanges(
    c(c = "chr2:1-10:-", d = "chr2:11-18:+"),
    score = 3:4)

## Include column data
colDat <- DataFrame(id = 1:2)

## Create a RaggedExperiment object from a couple of GRanges
re1 <- RaggedExperiment(sample1=sample1, sample2=sample2, colData = colDat)
re1

## With list of GRanges
lgr <- list(sample1 = sample1, sample2 = sample2)

## Create a RaggedExperiment from a list of GRanges
re2 <- RaggedExperiment(lgr, colData = colDat)

grl <- GRangesList(sample1 = sample1, sample2 = sample2)

## Create a RaggedExperiment from a GRangesList
re3 <- RaggedExperiment(grl, colData = colDat)
```

```
## Subset a RaggedExperiment
assay(re3[c(1, 3),])
subsetByOverlaps(re3, GRanges("chr1:1-5"))  # by ranges
```

`sparseSummarizedExperiment` *Create SummarizedExperiment representations by transforming ragged assays to rectangular form.*

Description

These methods transform `RaggedExperiment` objects to similar `SummarizedExperiment` objects. They do so by transforming assay data to more rectangular representations, following the rules outlined for similarly names transformations `sparseAssay()`, `compactAssay()`, `disjoinAssay()`, and `qreduceAssay()`. Because of the complexity of the transformation, ti usually only makes sense transform `RaggedExperiment` objects with a single assay; this is currently enforced at time of coercion.

Usage

```
sparseSummarizedExperiment(x, i = 1, withDimnames = TRUE, sparse = FALSE)

compactSummarizedExperiment(x, i = 1L, withDimnames = TRUE, sparse = FALSE)

disjoinSummarizedExperiment(x, simplifyDisjoin, i = 1L, withDimnames = TRUE)

qreduceSummarizedExperiment(
  x,
  query,
  simplifyReduce,
  i = 1L,
  withDimnames = TRUE
)
```

Arguments

`x`	`RaggedExperiment`
`i`	`integer(1)`, `character(1)`, or `logical()` selecting the assay to be transformed.
`withDimnames`	`logical(1)` default TRUE. propagate dimnames to SummarizedExperiment.
`sparse`	logical(1) whether to return a `sparseMatrix` representation
`simplifyDisjoin`	`function` of 1 argument, used to transform assays. See `assay-functions`.
`query`	`GRanges` provding regions over which reduction is to occur.
`simplifyReduce`	`function` of 3 arguments used to transform assays. See `assay-functions`.

Value

All functions return RangedSummarizedExperiment.

sparseSummarizedExperiment has rowRanges() identical to the row ranges of x, and assay() data as sparseAssay(). This is very space-inefficient representation of ragged data. Use 'sparse=TRUE' to obtain a sparseMatrix assay representation.

compactSummarizedExperiment has rowRanges() identical to the row ranges of x, and assay() data as compactAssay(). This is space-inefficient representation of ragged data when samples are primarily composed of different ranges. Use 'sparse=TRUE' to obtain a sparseMatrix assay representation.

disjoinSummarizedExperiment has rowRanges() identical to the disjoint row ranges of x, disjoint(rowRanges(x)), and assay() data as disjoinAssay().

qreduceSummarizedExperiment has rowRanges() identical to query, and assay() data as qreduceAssay().

sparseMatrix

Convert a dgCMatrix to a RaggedExperiment given that the rownames are coercible to GRanges.

In the following example, x is a dgCMatrix from the Matrix package.

```
`as(x, "RaggedExperiment")`
```

Examples

```
x <- RaggedExperiment(GRangesList(
    GRanges(c("A:1-5", "A:4-6", "A:10-15"), score=1:3),
    GRanges(c("A:1-5", "B:1-3"), score=4:5)
))

## sparseSummarizedExperiment

sse <- sparseSummarizedExperiment(x)
assay(sse)
rowRanges(sse)

## compactSummarizedExperiment

cse <- compactSummarizedExperiment(x)
assay(cse)
rowRanges(cse)

## disjoinSummarizedExperiment

disjoinAssay(x, lengths)
dse <- disjoinSummarizedExperiment(x, lengths)
assay(dse)
rowRanges(dse)

## qreduceSummarizedExperiment

x <- RaggedExperiment(GRangesList(
```

```
    GRanges(c("A:1-3", "A:4-5", "A:10-15"), score=1:3),
    GRanges(c("A:4-5", "B:1-3"), score=4:5)
))
query <- GRanges(c("A:1-2", "A:4-5", "B:1-5"))

weightedmean <- function(scores, ranges, qranges)
{
    ## weighted average score per query range
    ## the weight corresponds to the size of the overlap of each
    ## overlapping subject range with the corresponding query range
    isects <- pintersect(ranges, qranges)
    sum(scores * width(isects)) / sum(width(isects))
}

qreduceAssay(x, query, weightedmean)
qse <- qreduceSummarizedExperiment(x, query, weightedmean)
assay(qse)
rowRanges(qse)

sm <- Matrix::sparseMatrix(
    i = c(2, 3, 4, 3, 4, 3, 4),
    j = c(1, 1, 1, 3, 3, 4, 4),
    x = c(2L, 4L, 2L, 2L, 2L, 4L, 2L),
    dims = c(4, 4),
    dimnames = list(
        c("chr2:1-10", "chr2:2-10", "chr2:3-10", "chr2:4-10"),
        LETTERS[1:4]
    )
)

as(sm, "RaggedExperiment")
```

Index

[,RaggedExperiment,ANY,ANY,ANY-method
(RaggedExperiment-class), 6
$,RaggedExperiment-method
(RaggedExperiment-class), 6

as.data.frame,RaggedExperiment-method
(RaggedExperiment-class), 6
as.list,RaggedExperiment-method
(RaggedExperiment-class), 6
assay,RaggedExperiment,ANY-method
(RaggedExperiment-class), 6
assay,RaggedExperiment,missing-method
(RaggedExperiment-class), 6
assay-functions, 3
assayNames,RaggedExperiment-method
(RaggedExperiment-class), 6
assays,RaggedExperiment-method
(RaggedExperiment-class), 6

backtick, *10*

class:RaggedExperiment
(RaggedExperiment-class), 6
coerce,dgCMatrix,RaggedExperiment-method
(sparseSummarizedExperiment), 15
coerce,GRangesList,RaggedExperiment-method
(RaggedExperiment-class), 6
coerce,RaggedExperiment,GRangesList-method
(RaggedExperiment-class), 6
coerce-RaggedExperiment
(sparseSummarizedExperiment), 15
colData,RaggedExperiment-method
(RaggedExperiment-class), 6
colData<-,RaggedExperiment,DataFrame-method
(RaggedExperiment-class), 6
compactAssay (assay-functions), 3
compactSummarizedExperiment
(sparseSummarizedExperiment), 15

data.frame, *10*
DataFrame, *8*, *10*, *12*, *13*
dim,RaggedExperiment-method
(RaggedExperiment-class), 6
dimnames,RaggedExperiment-method
(RaggedExperiment-class), 6
dimnames<-,RaggedExperiment,ANY-method
(RaggedExperiment-class), 6
dimnames<-,RaggedExperiment,list-method
(RaggedExperiment-class), 6
disjoinAssay (assay-functions), 3
disjoinSummarizedExperiment
(sparseSummarizedExperiment), 15

exonsBy, *10*

GAlignmentPairs, *9*
GAlignments, *10*
GAlignmentsList, *9*
GRanges, *9*, *10*, *12*, *14*
GRangesList, *2*, *9*, *10*, *14*

Hits, *12*
HitsList, *12*

IntegerList, *12*
IntegerRanges, *11*, *12*
IntegerRangesList, *11*, *12*
IRanges, *11*
IRangesList, *11*

length,RaggedExperiment-method
(RaggedExperiment-class), 6

make.names, *10*
mcols,RaggedExperiment-method
(RaggedExperiment-class), 6
mcols<-,RaggedExperiment-method
(RaggedExperiment-class), 6

name, *10*
names, *10*

overlapsAny,RaggedExperiment,Vector-method
(RaggedExperiment-class), 6

qreduceAssay (assay-functions), 3
qreduceSummarizedExperiment
(sparseSummarizedExperiment), 15

RaggedExperiment, *2*, *12*
RaggedExperiment
(RaggedExperiment-class), 6
RaggedExperiment-class, 6
RaggedExperiment-package, 2
rowData,RaggedExperiment-method
(RaggedExperiment-class), 6
rowData<-,RaggedExperiment-method
(RaggedExperiment-class), 6
rowRanges, *2*
rowRanges,RaggedExperiment-method
(RaggedExperiment-class), 6
rowRanges<-,RaggedExperiment,GRanges-method
(RaggedExperiment-class), 6

Seqinfo, *9*
seqinfo,RaggedExperiment-method
(RaggedExperiment-class), 6
seqinfo<-,RaggedExperiment-method
(RaggedExperiment-class), 6
show,RaggedExperiment-method
(RaggedExperiment-class), 6
SimpleList, *12*, *13*
sparseAssay (assay-functions), 3
sparseMatrix, *4*, *15*, *16*
sparseSummarizedExperiment, 15
subset,RaggedExperiment-method
(RaggedExperiment-class), 6
subsetByOverlaps,RaggedExperiment,Vector-method
(RaggedExperiment-class), 6
SummarizedExperiment, *2*, *13*

transcriptsBy, *10*

Views, *11*
ViewsList, *11*