Package 'HDF5Array'

August 7, 2022

Title HDF5 backend for DelayedArray objects

Description Implement the HDF5Array, H5SparseMatrix, H5ADMatrix, and TENxMatrix classes, 4 convenient and memory-efficient array-like containers for representing and manipulating either: (1) a conventional (a.k.a. dense) HDF5 dataset, (2) an HDF5 sparse matrix (stored in CSR/CSC/Yale format), (3) the central matrix of an h5ad file (or any matrix in the /layers group), or (4) a 10x Genomics sparse matrix. All these containers are DelayedArray extensions and thus support all operations (delayed or block-processed) supported by DelayedArray objects.

biocViews Infrastructure, DataRepresentation, DataImport, Sequencing, RNASeq, Coverage, Annotation, GenomeAnnotation, SingleCell, ImmunoOncology

URL https://bioconductor.org/packages/HDF5Array

BugReports https://github.com/Bioconductor/HDF5Array/issues

Version 1.24.2

License Artistic-2.0

Encoding UTF-8

Author Hervé Pagès

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

Depends R (>= 3.4), methods, DelayedArray (>= 0.15.16), rhdf5 (>= 2.31.6)

Imports utils, stats, tools, Matrix, rhdf5filters, BiocGenerics (>= 0.31.5), S4Vectors, IRanges

LinkingTo S4Vectors (>= 0.27.13), Rhdf5lib

SystemRequirements GNU make

Suggests BiocParallel, GenomicRanges, SummarizedExperiment (>= 1.15.1), h5vcData, ExperimentHub, TENxBrainData, zellkonverter, GenomicFeatures, RUnit, SingleCellExperiment

Collate utils.R H5File-class.R h5ls.R H5DSetDescriptor-class.R
h5utils.R h5dimscales.R uaselection.R h5mread.R
h5mread_from_reshaped.R h5writeDimnames.R h5summarize.R
HDF5ArraySeed-class.R HDF5Array-class.R
ReshapedHDF5ArraySeed-class.R ReshapedHDF5Array-class.R
dump-management.R writeHDF5Array.R
saveHDF5SummarizedExperiment.R H5SparseMatrixSeed-class.R
H5SparseMatrix-class.R H5ADMatrixSeed-class.R
H5ADMatrix-class.R TENxMatrixSeed-class.R TENxMatrix-class.R
writeTENxMatrix.R zzz.R

git_url https://git.bioconductor.org/packages/HDF5Array

git_branch RELEASE_3_15

git_last_commit fb213ba

git_last_commit_date 2022-08-01

Date/Publication 2022-08-07

R topics documented:

H5ADMatrix-class *h5ad central matrices (or matrices in the /layers group) as DelayedMatrix objects*

Description

h5ad files are HDF5 files used for on-disk representation of AnnData Python objects. At the very minimum, they contain a central data matrix, named X, of shape #observations x #variables, and possibly additional data matrices (stored in the HDF5 group /layers) that share the shape and dimnames of X. See https://anndata.readthedocs.io/ for more information.

The H5ADMatrix class is a DelayedMatrix subclass for representing and operating on the central matrix of an h5ad file, or any matrix in its /layers group.

All the operations available for DelayedMatrix objects work on H5ADMatrix objects.

Usage

```
## Constructor function:
H5ADMatrix(filepath, layer=NULL)
```

Arguments

filepath	The path (as a single string) to the h5ad file.
layer	NULL (the default) or the name of a matrix in the /layers group. By default (i.e. when layer is not specified) H5ADMatrix() returns the central matrix (X).

Value

H5ADMatrix() returns an H5ADMatrix object of shape #variables x #observations. Note that in Python and HDF5 the shape of this matrix is considered to be #observations x #variables, but in R it is transposed. This follows the widely adopted convention of transposing HDF5 matrices when they get loaded into R.

References

https://anndata.readthedocs.io/ for AnnData Python objects and the h5ad format.

See Also

- HDF5Array objects for representing conventional (a.k.a. dense) HDF5 datasets as DelayedArray objects.
- H5SparseMatrix objects for representing HDF5 sparse matrices as DelayedMatrix objects.
- DelayedMatrix objects in the **DelayedArray** package.
- The H5ADMatrixSeed helper class.
- readH5AD and writeH5AD in the **zellkonverter** package for importing/exporting an h5ad file as/from a SingleCellExperiment object.

Examples

```
library(zellkonverter)
h5ad_file <- system.file("extdata", "krumsiek11.h5ad",
                         package="zellkonverter")
X <- H5ADMatrix(h5ad_file)
X
```

`H5ADMatrixSeed-class` *H5ADMatrixSeed objects*

Description

H5ADMatrixSeed is a low-level helper class used to represent a pointer to the central matrix stored of an `h5ad` file, or to one of the matrices in the `/layers` group.

It is a virtual class with three concrete subclasses: Dense_H5ADMatrixSeed, CSC_H5ADMatrixSeed, and CSR_H5ADMatrixSeed:

- The Dense_H5ADMatrixSeed class is used when the matrix is stored as a conventional HDF5 dataset in the `h5ad` file. It is a direct entension of the HDF5ArraySeed class.
- The CSC_H5ADMatrixSeed or CSR_H5ADMatrixSeed classes is used when the matrix is stored in the *Compressed Sparse Column* or *Compressed Sparse Row* format in the `h5ad` file. CSC_H5ADMatrixSeed is a direct entension of CSC_H5SparseMatrixSeed, and CSR_H5ADMatrixSeed a direct entension of CSR_H5SparseMatrixSeed.

Note that an H5ADMatrixSeed derivative is not intended to be used directly. Most end users will typically create and manipulate a higher-level H5ADMatrix object instead. See ?`H5ADMatrix` for more information.

Usage

```
## Constructor function:
H5ADMatrixSeed(filepath, layer=NULL)
```

Arguments

`filepath, layer`
: See ?`H5ADMatrix` for a description of these arguments.

Details

Dense_H5ADMatrixSeed objects support the same limited set of methods as HDF5ArraySeed objects, and CSC_H5ADMatrixSeed and CSR_H5ADMatrixSeed objects support the same limited set of methods as H5SparseMatrixSeed objects. See ?`HDF5ArraySeed` and ?`H5SparseMatrixSeed` for the details.

Value

`H5ADMatrixSeed()` returns an H5ADMatrixSeed derivative (Dense_H5ADMatrixSeed or CSC_H5ADMatrixSeed or CSR_H5ADMatrixSeed) of shape #variables x #observations.

H5ADMatrixSeed vs H5ADMatrix objects

In order to have access to the full set of operations that are available for DelayedMatrix objects, an H5ADMatrixSeed derivative first needs to be wrapped in a DelayedMatrix object, typically by calling the `DelayedArray()` constructor on it.

This is what the `H5ADMatrix()` constructor function does.

Note that the result of this wrapping is an H5ADMatrix object, which is just an H5ADMatrixSeed derivative wrapped in a DelayedMatrix object.

References

https://anndata.readthedocs.io/ for AnnData Python objects and the h5ad format.

See Also

- H5ADMatrix objects.
- HDF5ArraySeed and H5SparseMatrixSeed objects.
- `readH5AD` and `writeH5AD` in the **zellkonverter** package for importing/exporting an h5ad file as/from a SingleCellExperiment object.

Examples

```
library(zellkonverter)
h5ad_file <- system.file("extdata", "krumsiek11.h5ad",
                         package="zellkonverter")
seed <- H5ADMatrixSeed(h5ad_file)
seed
path(seed)
dim(seed)
is_sparse(seed)

DelayedArray(seed)
stopifnot(class(DelayedArray(seed)) == "H5ADMatrix")
```

H5File-class *H5File objects*

Description

The H5File class provides a formal representation of an HDF5 file (local or remote).

Usage

```
## Constructor function:
H5File(filepath, s3=FALSE, s3credentials=NULL, .no_rhdf5_h5id=FALSE)
```

Arguments

`filepath`	A single string specifying the path or URL to an HDF5 file.
`s3`	`TRUE` or `FALSE`. Should the `filepath` argument be treated as the URL to a file stored in an Amazon S3 bucket, rather than the path to a local file?
`s3credentials`	A list of length 3, providing the credentials for accessing files stored in a private Amazon S3 bucket. See `?H5Pset_fapl_ros3` in the **rhdf5** package for more information.
`.no_rhdf5_h5id`	For internal use only. Don't use.

Details

IMPORTANT NOTE ABOUT H5File OBJECTS AND PARALLEL EVALUATION

The short story is that H5File objects cannot be used in the context of parallel evaluation at the moment.

Here is why:

H5File objects contain an identifier to an open connection to the HDF5 file. This identifier becomes invalid in the 2 following situations:

- After serialization/deserialization, that is, after loading a serialized H5File object with `readRDS()` or `load()`.
- In the context of parallel evaluation, when using the SnowParam parallelization backend. This is because, unlike the MulticoreParam backend which used a system `fork`, the SnowParam backend uses serialization/deserialization to transmit the object to the workers.

In both cases, the connection to the file is lost and any attempt to read data from the H5File object will fail. Note that the above also happens to any H5File object that got serialized indirectly i.e. as part of a bigger object. For example, if an HDF5Array object was constructed from an H5File object, then it contains the H5File object and therefore `blockApply(..., BPPARAM=SnowParam(4))` cannot be used on it.

Furthermore, even if sometimes an H5File object *seems* to work fine with the MulticoreParam parallelization backend, this is highly unreliable and must be avoided.

Value

An H5File object.

See Also

- H5Pset_fapl_ros3 in the **rhdf5** package for detailed information about how to pass your S3 credentials to the `s3credentials` argument.
- The HDF5Array class for representing and operating on a conventional (a.k.a. dense) HDF5 dataset.
- The H5SparseMatrix class for representing and operating on an HDF5 sparse matrix.
- The H5ADMatrix class for representing and operating on the central matrix of an `h5ad` file, or any matrix in its `/layers` group.
- The TENxMatrix class for representing and operating on a 10x Genomics dataset.

- The h5mread function in this package (**HDF5Array**) that is used internally by HDF5Array, TENxMatrix, and H5ADMatrix objects, for (almost) all their data reading needs.
- h5ls to list the content of an HDF5 file.
- bplapply, MulticoreParam, and SnowParam, in the **BiocParallel** package.

Examples

```
## ---------------------------------------------------------------------
## A. BASIC USAGE
## ---------------------------------------------------------------------

## With a local file:
toy_h5 <- system.file("extdata", "toy.h5", package="HDF5Array")
h5file1 <- H5File(toy_h5)
h5ls(h5file1)
path(h5file1)

h5mread(h5file1, "M2", list(1:10, 1:6))
get_h5mread_returned_type(h5file1, "M2")

## With a file stored in an Amazon S3 bucket:
if (Sys.info()[["sysname"]] != "Darwin") {
  public_S3_url <-
   "https://rhdf5-public.s3.eu-central-1.amazonaws.com/rhdf5ex_t_float_3d.h5"
  h5file2 <- H5File(public_S3_url, s3=TRUE)
  h5ls(h5file2)

  h5mread(h5file2, "a1")
  get_h5mread_returned_type(h5file2, "a1")
}

## ---------------------------------------------------------------------
## B. H5File OBJECTS AND PARALLEL EVALUATION
## ---------------------------------------------------------------------
## H5File objects cannot be used in the context of parallel evaluation
## at the moment!

library(BiocParallel)

FUN1 <- function(i, h5file, name)
    sum(HDF5Array::h5mread(h5file, name, list(i, NULL)))

FUN2 <- function(i, h5file, name)
    sum(HDF5Array::h5mread(h5file, name, list(i, NULL, NULL)))

## With the SnowParam parallelization backend, the H5File object
## does NOT work on the workers:
## Not run:
## ERROR!
res1 <- bplapply(1:150, FUN1, h5file1, "M2", BPPARAM=SnowParam(3))
## ERROR!
res2 <- bplapply(1:5, FUN2, h5file2, "a1", BPPARAM=SnowParam(3))
```

```
## End(Not run)

## With the MulticoreParam parallelization backend, the H5File object
## might seem to work on the workers. However this is highly unreliable
## and must be avoided:
## Not run:
if (.Platform$OS.type != "windows") {
  ## UNRELIABLE!
  res1 <- bplapply(1:150, FUN1, h5file1, "M2", BPPARAM=MulticoreParam(3))
  ## UNRELIABLE!
  res2 <- bplapply(1:5, FUN2, h5file2, "a1", BPPARAM=MulticoreParam(3))
}

## End(Not run)
```

`h5ls` *A wrapper to rhdf5::h5ls() that works on H5File objects*

Description

Like `rhdf5::h5ls()`, but works on an H5File object.

Usage

```
h5ls(file, recursive=TRUE, all=FALSE, datasetinfo=TRUE,
           index_type=h5default("H5_INDEX"), order=h5default("H5_ITER"),
           s3=FALSE, s3credentials=NULL, native=FALSE)
```

Arguments

`file`, `recursive`, `all`, `datasetinfo`, `index_type`, `order`, `s3`, `s3credentials`, `native`

See `?rhdf5::h5ls` in the **rhdf5** package for a description of these arguments.

Note that the only difference with `rhdf5::h5ls()` is that, with `HDF5Array::h5ls()`, `file` can be an H5File object.

Value

See `?rhdf5::h5ls` in the **rhdf5** package.

See Also

- `h5ls` in the **rhdf5** package.
- H5File objects.

Examples

```
toy_h5 <- system.file("extdata", "toy.h5", package="HDF5Array")
h5ls(toy_h5)

h5file <- H5File(toy_h5)
h5ls(h5file)

## See '?H5File' for more examples.
```

---

h5mread *An alternative to* rhdf5::h5read

Description

h5mread is the result of experimenting with alternative rhdf5::h5read implementations.

It should still be considered experimental!

Usage

```
h5mread(filepath, name, starts=NULL, counts=NULL, noreduce=FALSE,
        as.integer=FALSE, as.sparse=FALSE,
        method=0L, use.H5Dread_chunk=FALSE)

get_h5mread_returned_type(filepath, name, as.integer=FALSE)
```

Arguments

filepath	The path (as a single string) to the HDF5 file where the dataset to read from is located, or an H5File object. Note that you must create and use an H5File object if the HDF5 file to access is stored in an Amazon S3 bucket. See ?H5File for how to do this. Also please note that H5File objects must NOT be used in the context of parallel evaluation at the moment.
name	The name of the dataset in the HDF5 file.
starts, counts	starts and counts are used to specify the *array selection*. Each argument can be either NULL or a list with one list element per dimension in the dataset. If starts and counts are both NULL, then the entire dataset is read. If starts is a list, each list element in it must be a vector of valid positive indices along the corresponding dimension in the dataset. An empty vector (integer(0)) is accepted and indicates an empty selection along that dimension. A NULL is accepted and indicates a *full* selection along the dimension so has the same meaning as a missing subscript when subsetting an array-like object with [. (Note that for [a NULL subscript indicates an empty selection.) Each list element in counts must be NULL or a vector of non-negative integers of the same length as the corresponding list element in starts. Each value in the vector indicates how many positions to select starting from the associated start

value. A NULL indicates that a single position is selected for each value along the corresponding dimension.

If counts is NULL, then each index in each starts list element indicates a single position selection along the corresponding dimension. Note that in this case the starts argument is equivalent to the index argument of h5read and extract_array (with the caveat that h5read doesn't accept empty selections).

Finally note that when counts is not NULL then the selection described by starts and counts must be *strictly ascending* along each dimension.

noreduce TODO

as.integer TODO

as.sparse TODO

method TODO

use.H5Dread_chunk TODO

Details

COMING SOON...

Value

An array for h5mread.

The type of the array that will be returned by h5mread for get_h5mread_returned_type. Equivalent to:

```
typeof(h5mread(filepath, name, rep(list(integer(0)), ndim)))
```

where ndim is the number of dimensions (a.k.a. the *rank* in HDF5 jargon) of the dataset. get_h5mread_returned_type is provided for convenience.

See Also

- H5File objects.
- h5read in the **rhdf5** package.
- extract_array in the **DelayedArray** package.
- The TENxBrainData dataset (in the **TENxBrainData** package).
- h5mread_from_reshaped to read data from a virtually reshaped HDF5 dataset.

Examples

```
## ---------------------------------------------------------------------
## BASIC USAGE
## ---------------------------------------------------------------------
m0 <- matrix((runif(600) - 0.5) * 10, ncol=12)
M0 <- writeHDF5Array(m0, name="M0")
```

```
m <- h5mread(path(M0), "M0")
stopifnot(identical(m0, m))

m <- h5mread(path(M0), "M0", starts=list(NULL, c(3, 12:8)))
stopifnot(identical(m0[ , c(3, 12:8)], m))

m <- h5mread(path(M0), "M0", starts=list(integer(0), c(3, 12:8)))
stopifnot(identical(m0[NULL , c(3, 12:8)], m))

m <- h5mread(path(M0), "M0", starts=list(1:5, NULL), as.integer=TRUE)
storage.mode(m0) <- "integer"
stopifnot(identical(m0[1:5, ], m))

a0 <- array(1:350, c(10, 5, 7))
A0 <- writeHDF5Array(a0, filepath=path(M0), name="A0")
h5ls(path(A0))

a <- h5mread(path(A0), "A0", starts=list(c(2, 7), NULL, 6),
                             counts=list(c(4, 2), NULL, NULL))
stopifnot(identical(a0[c(2:5, 7:8), , 6, drop=FALSE], a))

## Load the data in a sparse array representation:

m1 <- matrix(c(5:-2, rep.int(c(0L, 99L), 11)), ncol=6)
M1 <- writeHDF5Array(m1, name="M1", chunkdim=c(3L, 2L))

index <- list(5:3, NULL)
m <- h5mread(path(M1), "M1", starts=index)
sas <- h5mread(path(M1), "M1", starts=index, as.sparse=TRUE)
class(sas)  # SparseArraySeed object (see ?SparseArraySeed)
as(sas, "dgCMatrix")
stopifnot(identical(m, sparse2dense(sas)))

## ---------------------------------------------------------------------
## PERFORMANCE
## ---------------------------------------------------------------------
library(ExperimentHub)
hub <- ExperimentHub()

## With the "sparse" TENxBrainData dataset
## ---------------------------------------
fname0 <- hub[["EH1039"]]
h5ls(fname0)  # all datasets are 1D datasets

index <- list(77 * sample(34088679, 5000, replace=TRUE))
## h5mread() is about 4x faster than h5read():
system.time(a <- h5mread(fname0, "mm10/data", index))
system.time(b <- h5read(fname0, "mm10/data", index=index))
stopifnot(identical(a, b))

index <- list(sample(1306127, 7500, replace=TRUE))
## h5mread() is about 20x faster than h5read():
system.time(a <- h5mread(fname0, "mm10/barcodes", index))
```

```
system.time(b <- h5read(fname0, "mm10/barcodes", index=index))
stopifnot(identical(a, b))

## With the "dense" TENxBrainData dataset
## ---------------------------------------
fname1 <- hub[["EH1040"]]
h5ls(fname1)  # "counts" is a 2D dataset

set.seed(33)
index <- list(sample(27998, 300), sample(1306127, 450))
## h5mread() is about 2x faster than h5read():
system.time(a <- h5mread(fname1, "counts", index))
system.time(b <- h5read(fname1, "counts", index=index))
stopifnot(identical(a, b))

## Alternatively 'as.sparse=TRUE' can be used to reduce memory usage:
system.time(sas <- h5mread(fname1, "counts", index, as.sparse=TRUE))
stopifnot(identical(a, sparse2dense(sas)))

## The bigger the selection, the greater the speedup between
## h5read() and h5mread():
## Not run:
  index <- list(sample(27998, 1000), sample(1306127, 1000))
  ## h5mread() about 8x faster than h5read() (20s vs 2m30s):
  system.time(a <- h5mread(fname1, "counts", index))
  system.time(b <- h5read(fname1, "counts", index=index))
  stopifnot(identical(a, b))

  ## With 'as.sparse=TRUE' (about the same speed as with 'as.sparse=FALSE'):
  system.time(sas <- h5mread(fname1, "counts", index, as.sparse=TRUE))
  stopifnot(identical(a, sparse2dense(sas)))

## End(Not run)
```

`h5mread_from_reshaped` *Read data from a virtually reshaped HDF5 dataset*

Description

An `h5mread` wrapper that reads data from a virtually reshaped HDF5 dataset.

Usage

```
h5mread_from_reshaped(filepath, name, dim, starts, noreduce=FALSE,
                      as.integer=FALSE, method=0L)
```

Arguments

`filepath`	The path (as a single string) to the HDF5 file where the dataset to read from is located, or an H5File object.

Note that you must create and use an H5File object if the HDF5 file to access is stored in an Amazon S3 bucket. See ?H5File for how to do this.

Also please note that H5File objects must NOT be used in the context of parallel evaluation at the moment.

`name` The name of the dataset in the HDF5 file.

`dim` A vector of dimensions that describes the virtual reshaping i.e. the reshaping that is virtually applied upfront to the HDF5 dataset to read from.

Note that the HDF5 dataset is treated as read-only so never gets *effectively* reshaped, that is, the dataset dimensions encoded in the HDF5 file are not mmodified.

Also please note that arbitrary reshapings are not supported. Only reshapings that reduce the number of dimensions by collapsing a group of consecutive dimensions into a single dimension are supported. For example, reshaping a 10 x 3 x 5 x 1000 array as a 10 x 15 x 1000 array or as a 150 x 1000 matrix is supported.

`starts` A multidimensional subsetting index *with respect to the reshaped dataset*, that is, a list with one list element per dimension in the reshaped dataset.

Each list element in `starts` must be a vector of valid positive indices along the corresponding dimension in the reshaped dataset. An empty vector (`integer(0)`) is accepted and indicates an empty selection along that dimension. A `NULL` is accepted and indicates a *full* selection along the dimension so has the same meaning as a missing subscript when subsetting an array-like object with `[`. (Note that for `[` a `NULL` subscript indicates an empty selection.)

`noreduce, as.integer, method`

See ?h5mread for a description of these arguments.

Value

An array.

See Also

- H5File objects.
- h5mread.

Examples

```
## ---------------------------------------------------------------------
## BASIC USAGE
## ---------------------------------------------------------------------
a1 <- array(1:350, c(10, 5, 7))
A1 <- writeHDF5Array(a1, name="A1")

## Collapse the first 2 dimensions:
h5mread_from_reshaped(path(A1), "A1", dim=c(50, 7),
                      starts=list(8:11, NULL))
h5mread_from_reshaped(path(A1), "A1", dim=c(50, 7),
                      starts=list(8:11, NULL))
```

```
## Collapse the last 2 dimensions:
h5mread_from_reshaped(path(A1), "A1", dim=c(10, 35),
                      starts=list(NULL, 3:11))

a2 <- array(1:150000 + 0.1*runif(150000), c(10, 3, 5, 1000))
A2 <- writeHDF5Array(a2, name="A2")

## Collapse the 2nd and 3rd dimensions:
h5mread_from_reshaped(path(A2), "A2", dim=c(10, 15, 1000),
                      starts=list(NULL, 8:11, 999:1000))

## Collapse the first 3 dimensions:
h5mread_from_reshaped(path(A2), "A2", dim=c(150, 1000),
                      starts=list(71:110, 999:1000))
```

`H5SparseMatrix-class` *HDF5 sparse matrices as DelayedMatrix objects*

Description

The H5SparseMatrix class is a DelayedMatrix subclass for representing and operating on an HDF5 sparse matrix stored in CSR/CSC/Yale format.

All the operations available for DelayedMatrix objects work on H5SparseMatrix objects.

Usage

```
## Constructor function:
H5SparseMatrix(filepath, group)
```

Arguments

`filepath`	The path (as a single string) to the HDF5 file (`.h5` or `.h5ad`) where the sparse matrix is located.
`group`	The name of the group in the HDF5 file where the sparse matrix is stored.

Value

An H5SparseMatrix object.

See Also

- HDF5Array objects for representing conventional (a.k.a. dense) HDF5 datasets as DelayedArray objects.
- H5ADMatrix objects for representing h5ad central matrices (or matrices in the `/layers` group) as DelayedMatrix objects.
- TENxMatrix objects for representing 10x Genomics datasets as DelayedMatrix objects.

- DelayedMatrix objects in the **DelayedArray** package.
- The H5SparseMatrixSeed helper class.
- `h5ls` to list the content of an HDF5 file (`.h5` or `.h5ad`).

Examples

```
library(zellkonverter)
h5ad_file <- system.file("extdata", "example_anndata.h5ad",
                         package="zellkonverter")
h5ls(h5ad_file)

M <- H5SparseMatrix(h5ad_file, "/obsp/connectivities")
M

is(M, "DelayedMatrix")  # TRUE
dim(M)
seed(M)
path(M)
is_sparse(M)  # TRUE
```

H5SparseMatrixSeed-class

H5SparseMatrixSeed objects

Description

H5SparseMatrixSeed is a low-level helper class for representing a pointer to an HDF5 sparse matrix stored in CSR/CSC/Yale format.

It is a virtual class with two concrete subclasses: CSC_H5SparseMatrixSeed for the *Compressed Sparse Column* format and CSR_H5SparseMatrixSeed for the *Compressed Sparse Row* format. The former is used by 10x Genomics (e.g. "1.3 Million Brain Cell Dataset"). `h5ad` files can use one or the other format to store a sparse matrix.

Note that an H5SparseMatrixSeed derivative is not intended to be used directly. Most end users will typically create and manipulate a higher-level H5SparseMatrix object instead. See `?H5SparseMatrix` for more information.

Usage

```
## --- Constructor function ---

H5SparseMatrixSeed(filepath, group, subdata=NULL)

## --- Accessors --------------

## S4 method for signature 'H5SparseMatrixSeed'
path(object)
```

```
## S4 method for signature 'H5SparseMatrixSeed'
dim(x)

## S4 method for signature 'H5SparseMatrixSeed'
dimnames(x)

## S4 method for signature 'CSC_H5SparseMatrixSeed'
chunkdim(x)
## S4 method for signature 'CSR_H5SparseMatrixSeed'
chunkdim(x)

## --- Data extraction --------

## S4 method for signature 'H5SparseMatrixSeed'
extract_array(x, index)

## S4 method for signature 'H5SparseMatrixSeed'
extract_sparse_array(x, index)

## S4 method for signature 'H5SparseMatrixSeed'
read_sparse_block(x, viewport)

## S4 method for signature 'CSC_H5SparseMatrixSeed'
extractNonzeroDataByCol(x, j)
## S4 method for signature 'CSR_H5SparseMatrixSeed'
extractNonzeroDataByRow(x, i)

## --- Other methods ----------

## S4 method for signature 'H5SparseMatrixSeed'
is_sparse(x)

## S4 method for signature 'H5SparseMatrixSeed'
sparsity(x)
```

Arguments

filepath, group	See ?H5SparseMatrix for a description of these arguments.
subdata	Experimental. Don't use!
object, x	An H5SparseMatrixSeed derivative.
index	See ?extract_array in the **DelayedArray** package.
viewport	See ?read_block in the **DelayedArray** package.
j	An integer vector containing valid column indices.
i	An integer vector containing valid row indices.

Details

Note that the implementation of CSC_H5SparseMatrixSeed and CSR_H5SparseMatrixSeed objects follows the usual convention of transposing the HDF5 matrix when loading it into R. This means that a CSC_H5SparseMatrixSeed object represents a sparse matrix stored in the CSR format (Compressed Sparse Row) at the HDF5 level, and a CSR_H5SparseMatrixSeed object represents a sparse matrix stored in the CSC format (Compressed Sparse Column) at the HDF5 level.

Finally note that H5SparseMatrixSeed derivatives support a very limited set of methods:

- `path()`: Returns the path to the HDF5 file where the sparse matrix is located.
- `dim()`, `dimnames()`.
- `chunkdim()`, `extract_array()`, `extract_sparse_array()`, `read_sparse_block()`, `is_sparse()`: These generics are defined and documented in the **DelayedArrray** package.
- `sparsity()`: Returns the number of zero-valued matrix elements in the object divided by its total number of elements (a.k.a. its length).
- `extractNonzeroDataByCol()`: Works on CSC_H5SparseMatrixSeed objects only. Returns a NumericList or IntegerList object *parallel* to `j`, that is, with one list element per column index in `j`. The row indices of the values are not returned. Furthermore, the values within a given list element can be returned in **any order**. In particular, do NOT assume that they are ordered by ascending row index.
- `extractNonzeroDataByRow()`: Works on CSR_H5SparseMatrixSeed objects only. Returns a NumericList or IntegerList object *parallel* to `i`, that is, with one list element per row index in `i`. The column indices of the values are not returned. Furthermore, the values within a given list element can be returned in **any order**. In particular, do NOT assume that they are ordered by ascending column index.

In order to have access to the full set of operations that are available for DelayedMatrix objects, an H5SparseMatrixSeed derivative would first need to be wrapped in a DelayedMatrix object, typically by calling the `DelayedArray()` constructor on it.

Value

`H5SparseMatrixSeed()` returns an H5SparseMatrixSeed derivative (CSC_H5SparseMatrixSeed or CSR_H5SparseMatrixSeed object).

References

https://en.wikipedia.org/wiki/Sparse_matrix for a description of the CSR/CSC/Yale format (section "Compressed sparse row (CSR, CRS or Yale format)").

See Also

- H5SparseMatrix objects.
- `h5ls` to list the content of an HDF5 file (`.h5` or `.h5ad`).

Examples

```
showClass("H5SparseMatrixSeed")
```

h5writeDimnames *Write/read the dimnames of an HDF5 dataset*

Description

h5writeDimnames and h5readDimnames can be used to write/read the dimnames of an HDF5 dataset to/from the HDF5 file.

Note that h5writeDimnames is used internally by writeHDF5Array(x, ..., with.dimnames=TRUE) to write the dimnames of x to the HDF5 file together with the array data.

set_h5dimnames and get_h5dimnames are low-level utilities that can be used to attach existing HDF5 datasets along the dimensions of a given HDF5 dataset, or to retrieve the names of the HDF5 datasets that are attached along the dimensions of a given HDF5 dataset.

Usage

```
h5writeDimnames(dimnames, filepath, name, group=NA, h5dimnames=NULL)
h5readDimnames(filepath, name, as.character=FALSE)

set_h5dimnames(filepath, name, h5dimnames, dry.run=FALSE)
get_h5dimnames(filepath, name)
```

Arguments

dimnames	The dimnames to write to the HDF5 file. Must be supplied as a list (possibly named) with one list element per dimension in the HDF5 dataset specified via the name argument. Each list element in dimnames must be an atomic vector or a NULL. When not a NULL, its length must equal the extent of the corresponding dimension in the HDF5 dataset.
filepath	For h5writeDimnames and h5readDimnames: The path (as a single string) to the HDF5 file where the dimnames should be written to or read from. For set_h5dimnames and get_h5dimnames: The path (as a single string) to the HDF5 file where to set or get the *h5dimnames*.
name	For h5writeDimnames and h5readDimnames: The name of the dataset in the HDF5 file for which the dimnames should be written or read. For set_h5dimnames and get_h5dimnames: The name of the dataset in the HDF5 file for which to set or get the *h5dimnames*.
group	NA (the default) or the name of the HDF5 group where to write the dimnames. If set to NA then the group name is automatically generated from name. If set to the empty string ("") then no group will be used. Except when group is set to the empty string, the names in h5dimnames (see below) must be relative to the group.
h5dimnames	For h5writeDimnames: NULL (the default) or a character vector containing the names of the HDF5 datasets (one per list element in dimnames) where to write the dimnames. Names associated with NULL list elements in dimnames are ignored and should typically be NAs.

If set to NULL then the names are automatically set to numbers indicating the associated dimensions ("1" for the first dimension, "2" for the second, etc...)

For set_h5dimnames: A character vector containing the names of the HDF5 datasets to attach as dimnames of the dataset specified in name. The vector must have one element per dimension in dataset name. NAs are allowed and indicate dimensions along which nothing should be attached.

as.character — Even though the dimnames of an HDF5 dataset are usually stored as datasets of type "character" (H5 datatype "H5T_STRING") in the HDF5 file, this is not a requirement. By default h5readDimnames will return them *as-is*. Set as.character to TRUE to make sure that they are returned as character vectors. See example below.

dry.run — When set to TRUE, set_h5dimnames doesn't make any change to the HDF5 file but will still raise errors if the operation cannot be done.

Value

h5writeDimnames and set_h5dimnames return nothing.

h5readDimnames returns a list (possibly named) with one list element per dimension in HDF5 dataset name and containing its dimnames retrieved from the file.

get_h5dimnames returns a character vector containing the names of the HDF5 datasets that are currently set as the dimnames of the dataset specified in name. The vector has one element per dimension in dataset name. NAs in the vector indicate dimensions along which nothing is set.

See Also

- writeHDF5Array for a high-level function to write an array-like object and its dimnames to an HDF5 file.
- h5write in the **rhdf5** package that h5writeDimnames uses internally to write the dimnames to the HDF5 file.
- h5mread in this package (**HDF5Array**) that h5readDimnames uses internally to read the dimnames from the HDF5 file.
- h5ls to list the content of an HDF5 file.
- HDF5Array objects.

Examples

```
## ---------------------------------------------------------------------
## BASIC EXAMPLE
## ---------------------------------------------------------------------
library(rhdf5)  # for h5write()

m0 <- matrix(1:60, ncol=5)
colnames(m0) <- LETTERS[1:5]

h5file <- tempfile(fileext=".h5")
h5write(m0, h5file, "M0")  # h5write() ignores the dimnames
h5ls(h5file)
```

```
h5writeDimnames(dimnames(m0), h5file, "M0")
h5ls(h5file)

get_h5dimnames(h5file, "M0")
h5readDimnames(h5file, "M0")

## Reconstruct 'm0' from HDF5 file:
m1 <- h5mread(h5file, "M0")
dimnames(m1) <- h5readDimnames(h5file, "M0")
stopifnot(identical(m0, m1))

## Create an HDF5Array object that points to HDF5 dataset M0:
HDF5Array(h5file, "M0")

## Sanity checks:
stopifnot(identical(dimnames(m0), h5readDimnames(h5file, "M0")))
stopifnot(identical(dimnames(m0), dimnames(HDF5Array(h5file, "M0"))))

## ---------------------------------------------------------------------
## SHARED DIMNAMES
## ---------------------------------------------------------------------
## If a collection of HDF5 datasets share the same dimnames, the
## dimnames only need to be written once in the HDF5 file. Then they
## can be attached to the individual datasets with set_h5dimnames():

h5write(array(runif(240), c(12, 5:4)), h5file, "A1")
set_h5dimnames(h5file, "A1", get_h5dimnames(h5file, "M0"))
get_h5dimnames(h5file, "A1")
h5readDimnames(h5file, "A1")
HDF5Array(h5file, "A1")

h5write(matrix(sample(letters, 60, replace=TRUE), ncol=5), h5file, "A2")
set_h5dimnames(h5file, "A2", get_h5dimnames(h5file, "M0"))
get_h5dimnames(h5file, "A2")
h5readDimnames(h5file, "A2")
HDF5Array(h5file, "A2")

## Sanity checks:
stopifnot(identical(dimnames(m0), h5readDimnames(h5file, "A1")[1:2]))
stopifnot(identical(dimnames(m0), h5readDimnames(h5file, "A2")))

## ---------------------------------------------------------------------
## USE h5writeDimnames() AFTER A CALL TO writeHDF5Array()
## ---------------------------------------------------------------------
## After calling writeHDF5Array(x, ..., with.dimnames=FALSE) the
## dimnames on 'x' can still be written to the HDF5 file by doing the
## following:

## 1. Write 'm0' to the HDF5 file and ignore the dimnames (for now):
writeHDF5Array(m0, h5file, "M2")

## 2. Use h5writeDimnames() to write 'dimnames(m0)' to the file and
##    associate them with the "M2" dataset:
```

```
h5writeDimnames(dimnames(m0), h5file, "M2")

## 3. Use the HDF5Array() constructor to make an HDF5Array object that
##    points to the "M2" dataset:
HDF5Array(h5file, "M2")

## Note that at step 2. you can use the extra arguments of
## h5writeDimnames() to take full control of where the dimnames
## should be stored in the file:
writeHDF5Array(m0, h5file, "M3")
h5writeDimnames(dimnames(m0), h5file, "M3",
                group="a_secret_place", h5dimnames=c("NA", "M3_dim2"))
h5ls(h5file)
## h5readDimnames() and HDF5Array() still "finds" the dimnames:
h5readDimnames(h5file, "M3")
HDF5Array(h5file, "M3")

## Sanity checks:
stopifnot(identical(dimnames(m0), h5readDimnames(h5file, "M3")))
stopifnot(identical(dimnames(m0), dimnames(HDF5Array(h5file, "M3"))))

## ---------------------------------------------------------------------
## STORE THE DIMNAMES AS NON-CHARACTER TYPES
## ---------------------------------------------------------------------
writeHDF5Array(m0, h5file, "M4")
dimnames <- list(1001:1012, as.raw(11:15))
h5writeDimnames(dimnames, h5file, "M4")
h5ls(h5file)

h5readDimnames(h5file, "M4")
h5readDimnames(h5file, "M4", as.character=TRUE)

## Sanity checks:
stopifnot(identical(dimnames, h5readDimnames(h5file, "M4")))
dimnames(m0) <- dimnames
stopifnot(identical(
    dimnames(m0),
    h5readDimnames(h5file, "M4", as.character=TRUE)
))
```

HDF5-dump-management *HDF5 dump management*

Description

A set of utilities to control the location and physical properties of automatically created HDF5 datasets.

Usage

```
setHDF5DumpDir(dir)
setHDF5DumpFile(filepath)
setHDF5DumpName(name)
setHDF5DumpChunkLength(length=1000000L)
setHDF5DumpChunkShape(shape="scale")
setHDF5DumpCompressionLevel(level=6L)

getHDF5DumpDir()
getHDF5DumpFile(for.use=FALSE)
getHDF5DumpName(for.use=FALSE)
getHDF5DumpChunkLength()
getHDF5DumpChunkShape()
getHDF5DumpCompressionLevel()

lsHDF5DumpFile()

showHDF5DumpLog()

## For developers:
getHDF5DumpChunkDim(dim)
appendDatasetCreationToHDF5DumpLog(filepath, name, dim, type,
                                   chunkdim, level)
```

Arguments

`dir`	The path (as a single string) to the current *HDF5 dump directory*, that is, to the (new or existing) directory where *HDF5 dump files* with automatic names will be created. This is ignored if the user specified an *HDF5 dump file* with `setHDF5DumpFile`. If `dir` is missing, then the *HDF5 dump directory* is set back to its default value i.e. to some directory under `tempdir()` (call `getHDF5DumpDir()` to get the exact path).
`filepath`	For `setHDF5DumpFile`: The path (as a single string) to the current *HDF5 dump file*, that is, to the (new or existing) HDF5 file where the *next automatic HDF5 datasets* will be written. If `filepath` is missing, then a new file with an automatic name will be created (in `getHDF5DumpDir()`) and used for each new dataset. For `appendDatasetCreationToHDF5DumpLog`: See the Note TO DEVELOPERS below.
`name`	For `setHDF5DumpName`: The name of the *next automatic HDF5 dataset* to be written to the current *HDF5 dump file*. For `appendDatasetCreationToHDF5DumpLog`: See the Note TO DEVELOPERS below.
`length`	The maximum length of the physical chunks of the *next automatic HDF5 dataset* to be written to the current *HDF5 dump file*.
`shape`	A string specifying the shape of the physical chunks of the *next automatic HDF5 dataset* to be written to the current *HDF5 dump file*. See `makeCappedVolumeBox`

in the **DelayedArray** package for a description of the supported shapes.

`level` For `setHDF5DumpCompressionLevel`: The compression level to use for writing *automatic HDF5 datasets* to disk. See the `level` argument in `?rhdf5::h5createDataset` (in the **rhdf5** package) for more information about this.

For `appendDatasetCreationToHDF5DumpLog`: See the Note TO DEVELOPERS below.

`for.use` Whether the returned file or dataset name is for use by the caller or not. See below for the details.

`dim` The dimensions of the HDF5 dataset to be written to disk, that is, an integer vector of length one or more giving the maximal indices in each dimension. See the `dims` argument in `?rhdf5::h5createDataset` (in the **rhdf5** package) for more information about this.

`type` The type (a.k.a. storage mode) of the data to be written to disk. Can be obtained with `type()` on an array-like object (which is equivalent to `storage.mode()` or `typeof()` on an ordinary array). This is typically what an application writing datasets to the *HDF5 dump* should pass to the `storage.mode` argument of its call to `rhdf5::h5createDataset`. See the Note TO DEVELOPERS below for more information.

`chunkdim` The dimensions of the chunks.

Details

Calling `getHDF5DumpFile()` and `getHDF5DumpName()` with no argument should be *informative* only i.e. it's a mean for the user to know where the *next automatic HDF5 dataset* will be written. Since a given file/name combination can be used only once, the user should be careful to not use that combination to explicitely create an HDF5 dataset because that would get in the way of the creation of the *next automatic HDF5 dataset*. See the Note TO DEVELOPERS below if you actually need to use this file/name combination.

`lsHDF5DumpFile()` is a just convenience wrapper for `h5ls(getHDF5DumpFile())`.

Value

`getHDF5DumpDir` returns the absolute path to the directory where *HDF5 dump files* with automatic names will be created. Only meaningful if the user did NOT specify an *HDF5 dump file* with `setHDF5DumpFile`.

`getHDF5DumpFile` returns the absolute path to the HDF5 file where the *next automatic HDF5 dataset* will be written.

`getHDF5DumpName` returns the name of the *next automatic HDF5 dataset*.

`getHDF5DumpCompressionLevel` returns the compression level currently used for writing *automatic HDF5 datasets* to disk.

`showHDF5DumpLog` returns the dump log in an invisible data frame.

`getHDF5DumpChunkDim` returns the dimensions of the physical chunks that will be used to write the dataset to disk.

Note

TO DEVELOPERS:

If your application needs to write its own dataset to the *HDF5 dump* then it should:

1. Get a file/name combination by calling `getHDF5DumpFile(for.use=TRUE)` and `getHDF5DumpName(for.use=TRUE)`.
2. [OPTIONAL] Call `getHDF5DumpChunkDim(dim)` to get reasonable chunk dimensions to use for writing the dataset to disk. Or choose your own chunk dimensions.
3. Add an entry to the dump log by calling `appendDatasetCreationToHDF5DumpLog`. Typically, this should be done right after creating the dataset (e.g. with `rhdf5::h5createDataset`) and before starting to write the dataset to disk. The values passed to `appendDatasetCreationToHDF5DumpLog` via the `filepath`, `name`, `dim`, `type`, `chunkdim`, and `level` arguments should be those that were passed to `rhdf5::h5createDataset` via the `file`, `dataset`, `dims`, `storage.mode`, `chunk`, and `level` arguments, respectively. Note that `appendDatasetCreationToHDF5DumpLog` uses a lock mechanism so is safe to use in the context of parallel execution.

This is actually what the coercion method to HDF5Array does internally.

See Also

- `writeHDF5Array` for writing an array-like object to an HDF5 file.
- HDF5Array objects.
- The `h5ls` function on which `lsHDF5DumpFile` is based.
- `makeCappedVolumeBox` in the **DelayedArray** package.
- `type` in the **DelayedArray** package.

Examples

```
getHDF5DumpDir()
getHDF5DumpFile()

## Use setHDF5DumpFile() to change the current HDF5 dump file.
## If the specified file exists, then it must be in HDF5 format or
## an error will be raised. If it doesn't exist, then it will be
## created.
#setHDF5DumpFile("path/to/some/HDF5/file")

lsHDF5DumpFile()

a <- array(1:600, c(150, 4))
A <- as(a, "HDF5Array")
lsHDF5DumpFile()
A

b <- array(runif(6000), c(4, 2, 150))
B <- as(b, "HDF5Array")
lsHDF5DumpFile()
B

C <- (log(2 * A + 0.88) - 5)^3 * t(B[ , 1, ])
```

```
as(C, "HDF5Array")  # realize C on disk
lsHDF5DumpFile()

## Matrix multiplication is not delayed: the output matrix is realized
## block by block. The current "realization backend" controls where
## realization happens e.g. in memory if set to NULL or in an HDF5 file
## if set to "HDF5Array". See '?realize' in the DelayedArray package for
## more information about "realization backends".
setAutoRealizationBackend("HDF5Array")
m <- matrix(runif(20), nrow=4)
P <- C %*% m
lsHDF5DumpFile()

## See all the HDF5 datasets created in the current session so far:
showHDF5DumpLog()

## Wrap the call in suppressMessages() if you are only interested in the
## data frame version of the dump log:
dump_log <- suppressMessages(showHDF5DumpLog())
dump_log
```

`HDF5Array-class` *HDF5 datasets as DelayedArray objects*

Description

The HDF5Array class is a DelayedArray subclass for representing and operating on a conventional (a.k.a. dense) HDF5 dataset.

All the operations available for DelayedArray objects work on HDF5Array objects.

Usage

```
## Constructor function:
HDF5Array(filepath, name, as.sparse=FALSE, type=NA)
```

Arguments

`filepath`	The path (as a single string or H5File object) to the HDF5 file (`.h5` or `.h5ad`) where the dataset is located. Note that you must create and use an H5File object if the HDF5 file to access is stored in an Amazon S3 bucket. See `?H5File` for how to do this. Also please note that H5File objects must NOT be used in the context of parallel evaluation at the moment.
`name`	The name of the dataset in the HDF5 file.
`as.sparse`	Whether the HDF5 dataset should be flagged as sparse or not, that is, whether it should be considered sparse (and treated as such) or not. Note that HDF5 doesn't natively support sparse storage at the moment so HDF5 datasets cannot

be stored in a sparse format, only in a dense one. However a dataset stored in a dense format can still contain a lot of zeros. Using `as.sparse=TRUE` on such dataset will enable some optimizations that can lead to a lower memory footprint (and possibly better performance) when operating on the HDF5Array.

IMPORTANT NOTE: If the dataset is in the 10x Genomics format (i.e. if it uses the HDF5-based sparse matrix representation from 10x Genomics), you should use the `TENxMatrix()` constructor instead of the `HDF5Array()` constructor.

`type` By default the `type` of the returned object is inferred from the H5 datatype of the HDF5 dataset. This can be overridden by specifying the `type` argument. The specified type must be an *R atomic type* (e.g. `"integer"`) or `"list"`.

Value

An HDF5Array (or HDF5Matrix) object. (Note that HDF5Matrix extends HDF5Array.)

Note

The "1.3 Million Brain Cell Dataset" and other datasets published by 10x Genomics use an HDF5-based sparse matrix representation instead of the conventional (a.k.a. dense) HDF5 representation.

If your dataset uses the conventional (a.k.a. dense) HDF5 representation, use the `HDF5Array()` constructor documented here.

But if your dataset uses the HDF5 sparse matrix representation from 10x Genomics, use the `TENxMatrix()` constructor instead.

See Also

- H5File objects.
- H5SparseMatrix objects for representing HDF5 sparse matrices as DelayedMatrix objects.
- H5ADMatrix objects for representing h5ad central matrices (or matrices in the `/layers` group) as DelayedMatrix objects.
- TENxMatrix objects for representing 10x Genomics datasets as DelayedMatrix objects.
- ReshapedHDF5Array objects for representing HDF5 datasets as DelayedArray objects with a user-supplied upfront virtual reshaping.
- DelayedArray objects in the **DelayedArray** package.
- `writeHDF5Array` for writing an array-like object to an HDF5 file.
- HDF5-dump-management for controlling the location and physical properties of automatically created HDF5 datasets.
- `saveHDF5SummarizedExperiment` and `loadHDF5SummarizedExperiment` in this package (the **HDF5Array** package) for saving/loading an HDF5-based SummarizedExperiment object to/from disk.
- The HDF5ArraySeed helper class.
- `h5ls` to list the content of an HDF5 file (`.h5` or `.h5ad`).

Examples

```
## ---------------------------------------------------------------------
## A. CONSTRUCTION
## ---------------------------------------------------------------------

## With a local file:
toy_h5 <- system.file("extdata", "toy.h5", package="HDF5Array")
h5ls(toy_h5)

HDF5Array(toy_h5, "M2")
HDF5Array(toy_h5, "M2", type="integer")
HDF5Array(toy_h5, "M2", type="complex")

## With a file stored in an Amazon S3 bucket:
if (Sys.info()[["sysname"]] != "Darwin") {
    public_S3_url <-
     "https://rhdf5-public.s3.eu-central-1.amazonaws.com/rhdf5ex_t_float_3d.h5"
    h5file <- H5File(public_S3_url, s3=TRUE)
    h5ls(h5file)

    HDF5Array(h5file, "a1")
}

## ---------------------------------------------------------------------
## B. BASIC MANIPULATION
## ---------------------------------------------------------------------

library(h5vcData)
tally_file <- system.file("extdata", "example.tally.hfs5",
                          package="h5vcData")
h5ls(tally_file)

## Pick up "Coverages" dataset for Human chromosome 16:
name <- "/ExampleStudy/16/Coverages"
cvg <- HDF5Array(tally_file, name)
cvg

is(cvg, "DelayedArray")  # TRUE
seed(cvg)
path(cvg)
chunkdim(cvg)

## The data in the dataset looks sparse. In this case it is recommended
## to set 'as.sparse' to TRUE when constructing the HDF5Array object.
## This will make block processing (used in operations like sum()) more
## memory efficient and likely faster:
cvg0 <- HDF5Array(tally_file, name, as.sparse=TRUE)
is_sparse(cvg0)  # TRUE

## Note that we can also flag the HDF5Array object as sparse after
## creation:
is_sparse(cvg) <- TRUE
```

```
cvg  # same as 'cvg0'

## dim/dimnames:

dim(cvg0)

dimnames(cvg0)
dimnames(cvg0) <- list(paste0("s", 1:6), c("+", "-"), NULL)
dimnames(cvg0)

## ---------------------------------------------------------------------
## C. SLICING (A.K.A. SUBSETTING)
## ---------------------------------------------------------------------

cvg1 <- cvg0[ , , 29000001:29000007]
cvg1

dim(cvg1)
as.array(cvg1)
stopifnot(identical(dim(as.array(cvg1)), dim(cvg1)))
stopifnot(identical(dimnames(as.array(cvg1)), dimnames(cvg1)))

cvg2 <- cvg0[ , "+", 29000001:29000007]
cvg2
as.matrix(cvg2)

## ---------------------------------------------------------------------
## D. SummarizedExperiment OBJECTS WITH DELAYED ASSAYS
## ---------------------------------------------------------------------

## DelayedArray objects can be used inside a SummarizedExperiment object
## to hold the assay data and to delay operations on them.

library(SummarizedExperiment)

pcvg <- cvg0[ , 1, ]  # coverage on plus strand
mcvg <- cvg0[ , 2, ]  # coverage on minus strand

nrow(pcvg)  # nb of samples
ncol(pcvg)  # length of Human chromosome 16

## The convention for a SummarizedExperiment object is to have 1 column
## per sample so first we need to transpose 'pcvg' and 'mcvg':
pcvg <- t(pcvg)
mcvg <- t(mcvg)
se <- SummarizedExperiment(list(pcvg=pcvg, mcvg=mcvg))
se
stopifnot(validObject(se, complete=TRUE))

## A GPos object can be used to represent the genomic positions along
## the dataset:
gpos <- GPos(GRanges("16", IRanges(1, nrow(se))))
gpos
```

```
rowRanges(se) <- gpos
se
stopifnot(validObject(se))
assays(se)$pcvg
assays(se)$mcvg
```

HDF5ArraySeed-class *HDF5ArraySeed objects*

Description

HDF5ArraySeed is a low-level helper class for representing a pointer to an HDF5 dataset.

Note that an HDF5ArraySeed object is not intended to be used directly. Most end users will typically create and manipulate a higher-level HDF5Array object instead. See ?HDF5Array for more information.

Usage

```
## --- Constructor function ---

HDF5ArraySeed(filepath, name, as.sparse=FALSE, type=NA)

## --- Accessors --------------

## S4 method for signature 'HDF5ArraySeed'
path(object)

## S4 replacement method for signature 'HDF5ArraySeed'
path(object) <- value

## S4 method for signature 'HDF5ArraySeed'
dim(x)

## S4 method for signature 'HDF5ArraySeed'
dimnames(x)

## S4 method for signature 'HDF5ArraySeed'
type(x)

## S4 method for signature 'HDF5ArraySeed'
is_sparse(x)

## S4 replacement method for signature 'HDF5ArraySeed'
is_sparse(x) <- value

## S4 method for signature 'HDF5ArraySeed'
chunkdim(x)
```

```
## --- Data extraction --------

## S4 method for signature 'HDF5ArraySeed'
extract_array(x, index)

## S4 method for signature 'HDF5ArraySeed'
extract_sparse_array(x, index)
```

Arguments

`filepath, name, as.sparse, type`	See ?HDF5Array for a description of these arguments.
`object, x`	An HDF5ArraySeed object or derivative.
`value`	For the `path()` setter: The new path (as a single string) to the HDF5 file where the dataset is located. For the `is_sparse()` setter: `TRUE` or `FALSE`.
`index`	See ?`extract_array` in the **DelayedArray** package.

Details

The HDF5ArraySeed class has one direct subclass: Dense_H5ADMatrixSeed. See ?`Dense_H5ADMatrixSeed` for more information.

Note that the implementation of HDF5ArraySeed objects follows the widely adopted convention of transposing HDF5 matrices when they get loaded into R.

Finally note that an HDF5ArraySeed object supports a very limited set of methods:

- `path()`: Returns the path to the HDF5 file where the dataset is located.
- `dim()`, `dimnames()`.
- `type()`, `is_sparse()`, `chunkdim()`, `extract_array()`, `extract_sparse_array()`: These generics are defined and documented in the **DelayedArrray** package.

Value

`HDF5ArraySeed()` returns an HDF5ArraySeed object.

HDF5ArraySeed vs HDF5Array objects

In order to have access to the full set of operations that are available for DelayedArray objects, an HDF5ArraySeed object first needs to be wrapped in a DelayedArray object, typically by calling the `DelayedArray()` constructor on it.

This is what the `HDF5Array()` constructor function does.

Note that the result of this wrapping is an HDF5Array object, which is just an HDF5ArraySeed object wrapped in a DelayedArray object.

See Also

- HDF5Array objects.
- h5ls to list the content of an HDF5 file.

Examples

```
library(h5vcData)
tally_file <- system.file("extdata", "example.tally.hfs5",
                          package="h5vcData")
h5ls(tally_file)

name <- "/ExampleStudy/16/Coverages"  # name of the dataset of interest
seed1 <- HDF5ArraySeed(tally_file, name)
seed1
path(seed1)
dim(seed1)
chunkdim(seed1)

seed2 <- HDF5ArraySeed(tally_file, name, as.sparse=TRUE)
seed2

## Alternatively:
is_sparse(seed1) <- TRUE
seed1  # same as 'seed2'

DelayedArray(seed1)
stopifnot(class(DelayedArray(seed1)) == "HDF5Array")
```

ReshapedHDF5Array-class

Virtually reshaped HDF5 datasets as DelayedArray objects

Description

The ReshapedHDF5Array class is a DelayedArray subclass for representing an HDF5 dataset with a user-supplied upfront virtual reshaping.

All the operations available for DelayedArray objects work on ReshapedHDF5Array objects.

Usage

```
## Constructor function:
ReshapedHDF5Array(filepath, name, dim, type=NA)
```

Arguments

`filepath, name, type`
: See ?HDF5Array for a description of these arguments.

`dim` A vector of dimensions that describes the virtual reshaping i.e. the reshaping that is virtually applied upfront to the HDF5 dataset when the ReshapedHDF5Array object gets constructed.

Note that the HDF5 dataset is treated as read-only so is not *effectively* reshaped, that is, the dataset dimensions encoded in the HDF5 file are not mmodified.

Also please note that arbitrary reshapings are not supported. Only reshapings that reduce the number of dimensions by collapsing a group of consecutive dimensions into a single dimension are supported. For example, reshaping a 10 x 3 x 5 x 1000 array as a 10 x 15 x 1000 array or as a 150 x 1000 matrix is supported.

Value

A ReshapedHDF5Array (or ReshapedHDF5Matrix) object. (Note that ReshapedHDF5Matrix extends ReshapedHDF5Array.)

See Also

- HDF5Array objects for representing HDF5 datasets as DelayedArray objects without upfront virtual reshaping.
- DelayedArray objects in the **DelayedArray** package.
- `writeHDF5Array` for writing an array-like object to an HDF5 file.
- `saveHDF5SummarizedExperiment` and `loadHDF5SummarizedExperiment` in this package (the **HDF5Array** package) for saving/loading an HDF5-based SummarizedExperiment object to/from disk.
- The ReshapedHDF5ArraySeed helper class.
- `h5ls` to list the content of an HDF5 file.

Examples

```
library(h5vcData)
tally_file <- system.file("extdata", "example.tally.hfs5",
                          package="h5vcData")
h5ls(tally_file)

## Pick up "Coverages" dataset for Human chromosome 16 and collapse its
## first 2 dimensions:
cvg <- ReshapedHDF5Array(tally_file, "/ExampleStudy/16/Coverages",
                         dim=c(12, 90354753))
cvg

is(cvg, "DelayedArray")  # TRUE
seed(cvg)
path(cvg)
dim(cvg)
chunkdim(cvg)
```

ReshapedHDF5ArraySeed-class
ReshapedHDF5ArraySeed objects

Description

ReshapedHDF5ArraySeed is a low-level helper class for representing a pointer to a virtually reshaped HDF5 dataset.

ReshapedHDF5ArraySeed objects are not intended to be used directly. Most end users should create and manipulate ReshapedHDF5Array objects instead. See ?ReshapedHDF5Array for more information.

Usage

```
## Constructor function:
ReshapedHDF5ArraySeed(filepath, name, dim, type=NA)
```

Arguments

`filepath, name, dim, type`
: See ?ReshapedHDF5Array for a description of these arguments.

Details

No operation can be performed directly on a ReshapedHDF5ArraySeed object. It first needs to be wrapped in a DelayedArray object. The result of this wrapping is a ReshapedHDF5Array object (a ReshapedHDF5Array object is just a ReshapedHDF5ArraySeed object wrapped in a DelayedArray object).

Value

A ReshapedHDF5ArraySeed object.

See Also

- ReshapedHDF5Array objects.
- h5ls to list the content of an HDF5 file.

Examples

```
library(h5vcData)
tally_file <- system.file("extdata", "example.tally.hfs5",
                          package="h5vcData")
h5ls(tally_file)

## Collapse the first 2 dimensions:
seed <- ReshapedHDF5ArraySeed(tally_file, "/ExampleStudy/16/Coverages",
                              dim=c(12, 90354753))
```

```
seed
path(seed)
dim(seed)
chunkdim(seed)
```

`saveHDF5SummarizedExperiment` *Save/load an HDF5-based SummarizedExperiment object*

Description

`saveHDF5SummarizedExperiment` and `loadHDF5SummarizedExperiment` can be used to save/load an HDF5-based SummarizedExperiment object to/from disk.

NOTE: These functions use functionalities from the **SummarizedExperiment** package internally and so require this package to be installed.

Usage

```
saveHDF5SummarizedExperiment(x, dir="my_h5_se", prefix="", replace=FALSE,
                             chunkdim=NULL, level=NULL, as.sparse=NA,
                             verbose=NA)

loadHDF5SummarizedExperiment(dir="my_h5_se", prefix="")

quickResaveHDF5SummarizedExperiment(x, verbose=FALSE)
```

Arguments

`x`	A SummarizedExperiment object or derivative. For `quickResaveHDF5SummarizedExperiment` the object must have been previously saved with `saveHDF5SummarizedExperiment` (and has been possibly modified since then).
`dir`	The path (as a single string) to the directory where to save the HDF5-based SummarizedExperiment object or to load it from. When saving, the directory will be created if it doesn't already exist. If the directory already exists and no prefix is specified and `replace` is set to `TRUE`, then it's replaced with an empty directory.
`prefix`	An optional prefix to add to the names of the files created inside `dir`. Allows saving more than one object in the same directory.
`replace`	When no prefix is specified, should a pre-existing directory be replaced with a new empty one? The content of the pre-existing directory will be lost!
`chunkdim, level`	The dimensions of the chunks and the compression level to use for writing the assay data to disk. Passed to the internal calls to `writeHDF5Array`. See ?`writeHDF5Array` for more information.

as.sparse — Whether the assay data should be flagged as sparse or not. If set to NA (the default), then the specific as.sparse value to use for each assay is determined by calling is_sparse() on them.

Passed to the internal calls to writeHDF5Array. See ?writeHDF5Array for more information and an IMPORTANT NOTE.

verbose — Set to TRUE to make the function display progress.

In the case of saveHDF5SummarizedExperiment(), verbose is set to NA by default, in which case verbosity is controlled by DelayedArray:::get_verbose_block_processing(). Setting verbose to TRUE or FALSE overrides this.

Details

saveHDF5SummarizedExperiment(): Creates the directory specified thru the dir argument and populates it with the HDF5 datasets (one per assay in x) plus a serialized version of x that contains pointers to these datasets. This directory provides a self-contained HDF5-based representation of x that can then be loaded back in R with loadHDF5SummarizedExperiment.

Note that this directory is *relocatable* i.e. it can be moved (or copied) to a different place, on the same or a different computer, before calling loadHDF5SummarizedExperiment on it. For convenient sharing with collaborators, it is suggested to turn it into a tarball (with Unix command tar), or zip file, before the transfer.

Please keep in mind that saveHDF5SummarizedExperiment and loadHDF5SummarizedExperiment don't know how to produce/read tarballs or zip files at the moment, so the process of packaging/extracting the tarball or zip file is entirely the user responsibility. This is typically done from outside R.

Finally please note that, depending on the size of the data to write to disk and the performance of the disk, saveHDF5SummarizedExperiment can take a long time to complete. Use verbose=TRUE to see its progress.

loadHDF5SummarizedExperiment(): Typically very fast, even if the assay data is big, because all the assays in the returned object are HDF5Array objects pointing to the on-disk HDF5 datasets located in dir. HDF5Array objects are typically light-weight in memory.

quickResaveHDF5SummarizedExperiment(): Preserves the HDF5 file and datasets that the assays in x are already pointing to (and which were created by an earlier call to saveHDF5SummarizedExperiment). All it does is re-serialize x on top of the .rds file that is associated with this HDF5 file (and which was created by an earlier call to saveHDF5SummarizedExperiment or quickResaveHDF5SummarizedExperimen Because the delayed operations possibly carried by the assays in x are not realized, this is very fast.

Value

saveHDF5SummarizedExperiment returns an invisible SummarizedExperiment object that is the same as what loadHDF5SummarizedExperiment will return when loading back the object. All the assays in the object are HDF5Array objects pointing to datasets in the HDF5 file saved in dir.

Difference between saveHDF5SummarizedExperiment() and saveRDS()

Roughly speaking, saveRDS() only serializes the part of an object that resides in memory (the reality is a little bit more nuanced, but discussing the full details is not important here, and would only distract us). For most objects in R, that's the whole object, so saveRDS() does the job.

However some objects are pointing to on-disk data. For example: a TxDb object (the TxDb class is implemented and documented in the **GenomicFeatures** package) points to an SQLite db; an HDF5Array object points to a dataset in an HDF5 file; a SummarizedExperiment derivative can have one or more of its assays that point to datasets (one per assay) in an HDF5 file. These objects have 2 parts: one part is in memory, and one part is on disk. The 1st part is sometimes called the *object shell* and is generally thin (i.e. it has a small memory footprint). The 2nd part is the data and is typically big. The object shell and data are linked together via some kind of pointer stored in the shell (e.g. an SQLite connection, or a path to a file, etc...). Note that this is a *one way link* in the sense that the object shell "knows" where to find the on-disk data but the on-disk data knows nothing about the object shell (and is completely agnostic about what kind of object shell could be pointing to it). Furthermore, at any given time on a given system, there could be more than one object shell pointing to the same on-disk data. These object shells could exist in the same R session or in sessions in other languages (e.g. Python). These various sessions could be run by the same or by different users.

Using `saveRDS()` on such object will only serialize the shell part so will produce a small `.rds` file that contains the serialized object shell but not the object data.

This is problematic because:

1. If you later unserialize the object (with `readRDS()`) on the same system where you originally serialized it, it is possible that you will get back an object that is fully functional and semantically equivalent to the original object. But here is the catch: this will be the case ONLY if the data is still at the original location and has not been modified (i.e. nobody wrote or altered the data in the SQLite db or HDF5 file in the mean time), and if the serialization/unserialization cycle didn't break the link between the object shell and the data (this serialization/unserialization cycle is known to break open SQLite connections).
2. After serialization the object shell and data are stored in separate files (in the new `.rds` file for the shell, still in the original SQLite or HDF5 file for the data), typically in very different places on the file system. But these 2 files are not relocatable, that is, moving or copying them to another system or sending them to collaborators will typically break the link between them. Concretely this means that the object obtained by using `readRDS()` on the destination system will be broken.

`saveHDF5SummarizedExperiment()` addresses these issues by saving the object shell and assay data in a folder that is relocatable.

Note that it only works on SummarizedExperiment derivatives. What it does exactly is (1) write all the assay data to an HDF5 file, and (2) serialize the object shell, which in this case is everything in the object that is not the assay data. The 2 files (HDF5 and `.rds`) are written to the directory specified by the user. The resulting directory contains a full representation of the object and is relocatable, that is, it can be moved or copied to another place on the system, or to another system (possibly after making a tarball of it), where `loadHDF5SummarizedExperiment()` can then be used to load the object back in R.

Note

The files created by `saveHDF5SummarizedExperiment` in the user-specified directory `dir` should not be renamed.

The user-specified *directory* created by `saveHDF5SummarizedExperiment` is relocatable i.e. it can be renamed and/or moved around, but not the individual files in it.

Author(s)

Hervé Pagès

See Also

- SummarizedExperiment and RangedSummarizedExperiment objects in the **SummarizedExperiment** package.
- The `writeHDF5Array` function which `saveHDF5SummarizedExperiment` uses internally to write the assay data to disk.
- `base::saveRDS`

Examples

```
## ---------------------------------------------------------------------
## saveHDF5SummarizedExperiment() / loadHDF5SummarizedExperiment()
## ---------------------------------------------------------------------
library(SummarizedExperiment)

nrow <- 200
ncol <- 6
counts <- matrix(as.integer(runif(nrow * ncol, 1, 1e4)), nrow)
colData <- DataFrame(Treatment=rep(c("ChIP", "Input"), 3),
                     row.names=LETTERS[1:6])
se0 <- SummarizedExperiment(assays=list(counts=counts), colData=colData)
se0

## Save 'se0' as an HDF5-based SummarizedExperiment object:
dir <- tempfile("h5_se0_")
h5_se0 <- saveHDF5SummarizedExperiment(se0, dir)
list.files(dir)

h5_se0
assay(h5_se0, withDimnames=FALSE)   # HDF5Matrix object

h5_se0b <- loadHDF5SummarizedExperiment(dir)
h5_se0b
assay(h5_se0b, withDimnames=FALSE)  # HDF5Matrix object

## Sanity checks:
stopifnot(is(assay(h5_se0, withDimnames=FALSE), "HDF5Matrix"))
stopifnot(identical(assay(se0), as.matrix(assay(h5_se0))))
stopifnot(is(assay(h5_se0b, withDimnames=FALSE), "HDF5Matrix"))
stopifnot(identical(assay(se0), as.matrix(assay(h5_se0b))))

## ---------------------------------------------------------------------
## More sanity checks
## ---------------------------------------------------------------------

## Make a copy of directory 'dir':
somedir <- tempfile("somedir")
dir.create(somedir)
```

```
file.copy(dir, somedir, recursive=TRUE)
dir2 <- list.files(somedir, full.names=TRUE)

## 'dir2' contains a copy of 'dir'. Call loadHDF5SummarizedExperiment()
## on it.
h5_se0c <- loadHDF5SummarizedExperiment(dir2)

stopifnot(is(assay(h5_se0c, withDimnames=FALSE), "HDF5Matrix"))
stopifnot(identical(assay(se0), as.matrix(assay(h5_se0c))))

## ---------------------------------------------------------------------
## Using a prefix
## ---------------------------------------------------------------------

se1 <- se0[51:100, ]
saveHDF5SummarizedExperiment(se1, dir, prefix="xx_")
list.files(dir)
loadHDF5SummarizedExperiment(dir, prefix="xx_")

## ---------------------------------------------------------------------
## quickResaveHDF5SummarizedExperiment()
## ---------------------------------------------------------------------

se2 <- loadHDF5SummarizedExperiment(dir, prefix="xx_")
se2 <- se2[1:14, ]
assay1 <- assay(se2, withDimnames=FALSE)
assays(se2, withDimnames=FALSE) <- c(assays(se2), list(score=assay1/100))
rowRanges(se2) <- GRanges("chr1", IRanges(1:14, width=5))
rownames(se2) <- letters[1:14]
se2

## This will replace saved 'se1'!
quickResaveHDF5SummarizedExperiment(se2, verbose=TRUE)
list.files(dir)
loadHDF5SummarizedExperiment(dir, prefix="xx_")
```

`TENxMatrix-class` *10x Genomics datasets as DelayedMatrix objects*

Description

A 10x Genomics dataset like the "1.3 Million Brain Cell Dataset" is an HDF5 sparse matrix stored in CSR/CSC/Yale format ("Compressed Sparse Row").

The TENxMatrix class is a DelayedMatrix subclass for representing and operating on this kind of dataset.

All the operations available for DelayedMatrix objects work on TENxMatrix objects.

Usage

```
## Constructor function:
TENxMatrix(filepath, group="matrix")
```

Arguments

filepath	The path (as a single string) to the HDF5 file where the 10x Genomics dataset is located.
group	The name of the group in the HDF5 file containing the 10x Genomics data.

Details

In addition to all the methods defined for DelayedMatrix objects, TENxMatrix objects support the following specialized methods: `sparsity()` and `extractNonzeroDataByCol()`. See `?H5SparseMatrixSeed` for more information about what these methods do.

Value

`TENxMatrix()` returns a TENxMatrix object.

Note

If your dataset uses the HDF5 sparse matrix representation from 10x Genomics, use the `TENxMatrix()` constructor documented here.

But if your dataset uses the conventional (a.k.a. dense) HDF5 representation, use the `HDF5Array()` constructor instead.

See Also

- HDF5Array objects for representing conventional (a.k.a. dense) HDF5 datasets as DelayedArray objects.
- DelayedMatrix objects in the **DelayedArray** package.
- `writeTENxMatrix` for writing a matrix-like object as an HDF5-based sparse matrix.
- The `TENxBrainData` dataset (in the **TENxBrainData** package).
- `detectCores` from the **parallel** package.
- `setAutoBPPARAM` and `setAutoBlockSize` in the **DelayedArray** package.
- `colAutoGrid` and `blockApply` in the **DelayedArray** package.
- The TENxMatrixSeed helper class.
- `h5ls` to list the content of an HDF5 file.
- NumericList and IntegerList objects in the **IRanges** package.

Examples

```
## ---------------------------------------------------------------------
## THE "1.3 Million Brain Cell Dataset" AS A DelayedMatrix OBJECT
## ---------------------------------------------------------------------

## The 1.3 Million Brain Cell Dataset from 10x Genomics is available
## via ExperimentHub:

library(ExperimentHub)
```

```
hub <- ExperimentHub()
query(hub, "TENxBrainData")
fname <- hub[["EH1039"]]

## 'fname' is an HDF5 file. Use h5ls() to list its content:
h5ls(fname)

## The 1.3 Million Brain Cell Dataset is represented by the "mm10"
## group. We point the TENxMatrix() constructor to this group to
## create a TENxMatrix object representing the dataset:
oneM <- TENxMatrix(fname, group="mm10")
oneM

is(oneM, "DelayedMatrix")  # TRUE
seed(oneM)
path(oneM)
sparsity(oneM)

## Some examples of delayed operations:
oneM != 0
oneM^2

## ---------------------------------------------------------------------
## SOME EXAMPLES OF ROW/COL SUMMARIZATION
## ---------------------------------------------------------------------

## In order to reduce computation times, we'll use only the first
## 25000 columns of the 1.3 Million Brain Cell Dataset:
oneM25k <- oneM[ , 1:25000]

## Row/col summarization methods like rowSums() use a block-processing
## mechanism behind the scene that can be controlled via global
## settings. 2 important settings that can have a strong impact on
## performance are the automatic number of workers and automatic block
## size, controlled by setAutoBPPARAM() and setAutoBlockSize()
## respectively.
library(BiocParallel)
if (.Platform$OS.type != "windows") {
    ## On a modern Linux laptop with 8 cores (as reported by
    ## parallel::detectCores()) and 16 Gb of RAM, reasonably good
    ## performance is achieved by setting the automatic number of workers
    ## to 5 or 6 and the automatic block size between 300 Mb and 400 Mb:
    workers <- 5
    block_size <- 3e8  # 300 Mb
    setAutoBPPARAM(MulticoreParam(workers))
} else {
    ## MulticoreParam() is not supported on Windows so we use SnowParam()
    ## on this platform. Also we reduce the block size to 200 Mb on
    ## 32-bit Windows to avoid memory allocation problems (they tend to
    ## be common there because a process cannot use more than 3 Gb of
    ## memory).
    workers <- 4
    setAutoBPPARAM(SnowParam(workers))
```

```
    block_size <- if (.Platform$r_arch == "i386") 2e8 else 3e8
}
setAutoBlockSize(block_size)

## We're ready to compute the library sizes, number of genes expressed
## per cell, and average expression across cells:
system.time(lib_sizes <- colSums(oneM25k))
system.time(n_exprs <- colSums(oneM25k != 0))
system.time(ave_exprs <- rowMeans(oneM25k))

## Note that the 3 computations above load the data in oneM25k 3 times
## in memory. This can be avoided by computing the 3 summarizations in
## a single pass with blockApply(). First we define the function that
## we're going to apply to each block of data:
FUN <- function(block)
  list(colSums(block), colSums(block != 0), rowSums(block))

## Then we call blockApply() to apply FUN() to each block. The blocks
## are defined by the grid passed to the 'grid' argument. In this case
## we supply a grid made with colAutoGrid() to generate blocks of full
## columns (see ?colAutoGrid for more information):
system.time({
  block_results <- blockApply(oneM25k, FUN, grid=colAutoGrid(oneM25k),
                              verbose=TRUE)
})

## 'block_results' is a list with 1 list element per block in
## colAutoGrid(oneM25k). Each list element is the result that was
## obtained by applying FUN() on the block so is itself a list of
## length 3.
## Let's combine the results:
lib_sizes2 <- unlist(lapply(block_results, `[[`, 1L))
n_exprs2 <- unlist(lapply(block_results, `[[`, 2L))
block_rowsums <- unlist(lapply(block_results, `[[`, 3L), use.names=FALSE)
tot_exprs <- rowSums(matrix(block_rowsums, nrow=nrow(oneM25k)))
ave_exprs2 <- setNames(tot_exprs / ncol(oneM25k), rownames(oneM25k))

## Sanity checks:
stopifnot(all.equal(lib_sizes, lib_sizes2))
stopifnot(all.equal(n_exprs, n_exprs2))
stopifnot(all.equal(ave_exprs, ave_exprs2))

## Turn off parallel evaluation and reset automatic block size to factory
## settings:
setAutoBPPARAM()
setAutoBlockSize()

## ---------------------------------------------------------------------
## extractNonzeroDataByCol()
## ---------------------------------------------------------------------

## extractNonzeroDataByCol() provides a convenient and very efficient
## way to extract the nonzero data in a compact form:
```

```
nonzeros <- extractNonzeroDataByCol(oneM, 1:25000)  # takes < 5 sec.

## The data is returned as an IntegerList object with one list element
## per column and no row indices associated to the values in the object.
## Furthermore, the values within a given list element can be returned
## in any order:
nonzeros

names(nonzeros) <- colnames(oneM25k)

## This can be used to compute some simple summaries like the library
## sizes and the number of genes expressed per cell. For these use
## cases, it is a lot more efficient than using colSums(oneM25k) and
## colSums(oneM25k != 0):
lib_sizes3 <- sum(nonzeros)
n_exprs3 <- lengths(nonzeros)

## Sanity checks:
stopifnot(all.equal(lib_sizes, lib_sizes3))
stopifnot(all.equal(n_exprs, n_exprs3))
```

TENxMatrixSeed-class *TENxMatrixSeed objects*

Description

TENxMatrixSeed is a low-level helper class that is a direct extension of the H5SparseMatrixSeed class. It is used to represent a pointer to an HDF5 sparse matrix that is stored in the CSR/CSC/Yale format ("Compressed Sparse Row") and follows the 10x Genomics convention for storing the dimensions of the matrix.

Note that a TENxMatrixSeed object is not intended to be used directly. Most end users will typically create and manipulate a higher-level TENxMatrix object instead. See ?`TENxMatrix` for more information.

Usage

```
## Constructor function:
TENxMatrixSeed(filepath, group="matrix")
```

Arguments

`filepath, group`
: See ?`TENxMatrix` for a description of these arguments.

Details

A TENxMatrixSeed object supports the same limited set of methods as an H5SparseMatrixSeed object. See ?`H5SparseMatrixSeed` for the details.

Value

`TENxMatrixSeed()` returns a TENxMatrixSeed object.

TENxMatrixSeed vs TENxMatrix objects

In order to have access to the full set of operations that are available for DelayedMatrix objects, a TENxMatrixSeed object first needs to be wrapped in a DelayedMatrix object, typically by calling the `DelayedArray()` constructor on it.

This is what the `TENxMatrix()` constructor function does.

Note that the result of this wrapping is a TENxMatrix object, which is just a TENxMatrixSeed object wrapped in a DelayedMatrix object.

See Also

- TENxMatrix objects.
- H5SparseMatrixSeed objects.
- The `TENxBrainData` dataset (in the **TENxBrainData** package).
- `h5ls` to list the content of an HDF5 file.

Examples

```
## The 1.3 Million Brain Cell Dataset from 10x Genomics is available
## via ExperimentHub:
library(ExperimentHub)
hub <- ExperimentHub()
query(hub, "TENxBrainData")
fname <- hub[["EH1039"]]

## 'fname' is an HDF5 file. Use h5ls() to list its content:
h5ls(fname)

## The 1.3 Million Brain Cell Dataset is represented by the "mm10"
## group. We point the TENxMatrixSeed() constructor to this group
## to create a TENxMatrixSeed object representing the dataset:
seed <- TENxMatrixSeed(fname, group="mm10")
seed
path(seed)
dim(seed)
is_sparse(seed)
sparsity(seed)

DelayedArray(seed)
stopifnot(class(DelayedArray(seed)) == "TENxMatrix")
```

writeHDF5Array *Write an array-like object to an HDF5 file*

Description

A function for writing an array-like object to an HDF5 file.

Usage

```
writeHDF5Array(x, filepath=NULL, name=NULL,
               H5type=NULL, chunkdim=NULL, level=NULL, as.sparse=NA,
               with.dimnames=FALSE, verbose=NA)
```

Arguments

x	The array-like object to write to an HDF5 file. If x is a DelayedArray object, writeHDF5Array *realizes* it on disk, that is, all the delayed operations carried by the object are executed while the object is written to disk. See "On-disk realization of a DelayedArray object as an HDF5 dataset" section below for more information.
filepath	NULL or the path (as a single string) to the (new or existing) HDF5 file where to write the dataset. If NULL, then the dataset will be written to the current *HDF5 dump file* i.e. to the file whose path is getHDF5DumpFile.
name	NULL or the name of the HDF5 dataset to write. If NULL, then the name returned by getHDF5DumpName will be used.
H5type	The H5 datatype to use for the HDF5 dataset to be written to the HDF5 file is automatically inferred from the type of x (type(x)). Advanced users can override this by specifying the H5 datatype they want via the H5type argument. See rhdf5::h5const("H5T") for a list of available H5 datatypes. See References section below for the link to the HDF Group's Support Portal where H5 predefined datatypes are documented. A typical use case is to use a datatype that is smaller than the automatic one in order to reduce the size of the dataset on disk. For example you could use "H5T_IEEE_F32LE" when type(x) is "double" and you don't care about preserving the precision of 64-bit floating-point numbers (the automatic H5 datatype used for "double" is "H5T_IEEE_F64LE"). Another example is to use "H5T_STD_U16LE" when x contains small non-negative integer values like counts (the automatic H5 datatype used for "integer" is "H5T_STD_I32LE").
chunkdim	The dimensions of the chunks to use for writing the data to disk. By default (i.e. when chunkdim is set to NULL), getHDF5DumpChunkDim(dim(x)) will be used. See ?getHDF5DumpChunkDim for more information. Set chunkdim to 0 to write *unchunked data* (a.k.a. *contiguous data*).
level	The compression level to use for writing the data to disk. By default, getHDF5DumpCompressionLevel() will be used. See ?getHDF5DumpCompressionLevel for more information.

as.sparse — Whether the data in the returned HDF5Array object should be flagged as sparse or not. If set to NA (the default), then is_sparse(x) is used.

IMPORTANT NOTE: This only controls the as.sparse flag of the returned HDF5Array object. See man page of the HDF5Array() constructor for more information. In particular this does NOT affect how the data will be laid out to the HDF5 file in any way (HDF5 doesn't natively support sparse storage at the moment). In other words, the data will always be stored in a dense format, even when as.sparse is set to TRUE.

with.dimnames — By default the dimnames on x are not written to the HDF5 file. Set with.dimnames to TRUE to also have them written.

Note that h5writeDimnames is used internally to write the dimnames to disk. Setting with.dimnames to FALSE and calling h5writeDimnames is another way to write the dimnames on x to disk that gives more control. See ?h5writeDimnames for more information.

verbose — Whether block processing progress should be displayed or not. If set to NA (the default), verbosity is controlled by DelayedArray:::get_verbose_block_processing(). Setting verbose to TRUE or FALSE overrides this.

Details

Please note that, depending on the size of the data to write to disk and the performance of the disk, writeHDF5Array() can take a long time to complete. Use verbose=TRUE to see its progress.

Use setHDF5DumpFile and setHDF5DumpName to control the location of automatically created HDF5 datasets.

Use setHDF5DumpChunkLength, setHDF5DumpChunkShape, and setHDF5DumpCompressionLevel, to control the physical properties of automatically created HDF5 datasets.

Value

An HDF5Array object pointing to the newly written HDF5 dataset on disk.

On-disk realization of a DelayedArray object as an HDF5 dataset

When passed a DelayedArray object, writeHDF5Array *realizes* it on disk, that is, all the delayed operations carried by the object are executed on-the-fly while the object is written to disk. This uses a block-processing strategy so that the full object is not realized at once in memory. Instead the object is processed block by block i.e. the blocks are realized in memory and written to disk one at a time.

In other words, writeHDF5Array(x, ...) is semantically equivalent to writeHDF5Array(as.array(x), ...), except that as.array(x) is not called because this would realize the full object at once in memory.

See ?DelayedArray for general information about DelayedArray objects.

References

Documentation of the H5 predefined datatypes on the HDF Group's Support Portal: https://portal.hdfgroup.org/display/HDF5/Predefined+Datatypes

See Also

- HDF5Array objects.
- h5writeDimnames for writing the dimnames of an HDF5 dataset to disk.
- saveHDF5SummarizedExperiment and loadHDF5SummarizedExperiment in this package (the **HDF5Array** package) for saving/loading an HDF5-based SummarizedExperiment object to/from disk.
- HDF5-dump-management to control the location and physical properties of automatically created HDF5 datasets.
- h5ls to list the content of an HDF5 file.

Examples

```
## ---------------------------------------------------------------------
## WRITE AN ORDINARY ARRAY TO AN HDF5 FILE
## ---------------------------------------------------------------------
m <- matrix(runif(364, min=-1), nrow=26,
            dimnames=list(letters, LETTERS[1:14]))

h5file <- tempfile(fileext=".h5")
M1 <- writeHDF5Array(m, h5file, name="M1", chunkdim=c(5, 5))
M1
chunkdim(M1)

## By default, writeHDF5Array() does not write the dimnames to the HDF5
## file so they are lost:
dimnames(M1)   # no dimnames

## Set 'with.dimnames' to TRUE to write them to the file:
M1b <- writeHDF5Array(m, h5file, name="M1b", with.dimnames=TRUE)
dimnames(M1b)  # dimnames are back

## With sparse data:
sm <- rsparsematrix(20, 8, density=0.1)
M2 <- writeHDF5Array(sm, h5file, name="M2", chunkdim=c(5, 5))
M2
is_sparse(M2)  # TRUE

## ---------------------------------------------------------------------
## WRITE A DelayedArray OBJECT TO AN HDF5 FILE
## ---------------------------------------------------------------------
M3 <- log(t(DelayedArray(m)) + 1)
M3 <- writeHDF5Array(M3, h5file, name="M3", chunkdim=c(5, 5))
M3
chunkdim(M3)

library(h5vcData)
tally_file <- system.file("extdata", "example.tally.hfs5",
                          package="h5vcData")
h5ls(tally_file)
```

```
cvg0 <- HDF5Array(tally_file, "/ExampleStudy/16/Coverages")

cvg1 <- cvg0[ , , 29000001:29000007]

writeHDF5Array(cvg1, h5file, "cvg1")
h5ls(h5file)
```

---

writeTENxMatrix *Write a matrix-like object as an HDF5-based sparse matrix*

Description

The 1.3 Million Brain Cell Dataset and other datasets published by 10x Genomics use an HDF5-based sparse matrix representation instead of the conventional (a.k.a. dense) HDF5 representation.

writeTENxMatrix writes a matrix-like object to this format.

IMPORTANT NOTE: Only use writeTENxMatrix if the matrix-like object to write is sparse, that is, if most of its elements are zero. Using writeTENxMatrix on dense data is very inefficient! In this case, you should use writeHDF5Array instead.

Usage

```
writeTENxMatrix(x, filepath=NULL, group=NULL, level=NULL, verbose=NA)
```

Arguments

x	The matrix-like object to write to an HDF5 file. The object to write should typically be sparse, that is, most of its elements should be zero. If x is a DelayedMatrix object, writeTENxMatrix *realizes* it on disk, that is, all the delayed operations carried by the object are executed while the object is written to disk.
filepath	NULL or the path (as a single string) to the (new or existing) HDF5 file where to write the data. If NULL, then the data will be written to the current *HDF5 dump file* i.e. to the file whose path is getHDF5DumpFile.
group	NULL or the name of the HDF5 group where to write the data. If NULL, then the name returned by getHDF5DumpName will be used.
level	The compression level to use for writing the data to disk. By default, getHDF5DumpCompressionLevel() will be used. See ?getHDF5DumpCompressionLevel for more information.
verbose	Whether block processing progress should be displayed or not. If set to NA (the default), verbosity is controlled by DelayedArray:::get_verbose_block_processing(). Setting verbose to TRUE or FALSE overrides this.

Details

Please note that, depending on the size of the data to write to disk and the performance of the disk, `writeTENxMatrix` can take a long time to complete. Use `verbose=TRUE` to see its progress.

Use `setHDF5DumpFile` and `setHDF5DumpName` to control the location of automatically created HDF5 datasets.

Value

A TENxMatrix object pointing to the newly written HDF5 data on disk.

See Also

- TENxMatrix objects.
- The `TENxBrainData` dataset (in the **TENxBrainData** package).
- HDF5-dump-management to control the location and physical properties of automatically created HDF5 datasets.
- `h5ls` to list the content of an HDF5 file.

Examples

```
## ---------------------------------------------------------------------
## A SIMPLE EXAMPLE
## ---------------------------------------------------------------------
m0 <- matrix(0L, nrow=25, ncol=12,
             dimnames=list(letters[1:25], LETTERS[1:12]))
m0[cbind(2:24, c(12:1, 2:12))] <- 100L + sample(55L, 23, replace=TRUE)
out_file <- tempfile()
M0 <- writeTENxMatrix(m0, out_file, group="m0")
M0
sparsity(M0)

path(M0)  # same as 'out_file'

## Use h5ls() to list the content of this HDF5 file:
h5ls(path(M0))

## ---------------------------------------------------------------------
## USING THE "1.3 Million Brain Cell Dataset"
## ---------------------------------------------------------------------

## The 1.3 Million Brain Cell Dataset from 10x Genomics is available via
## ExperimentHub:
library(ExperimentHub)
hub <- ExperimentHub()
query(hub, "TENxBrainData")
fname <- hub[["EH1039"]]
oneM <- TENxMatrix(fname, group="mm10")  # see ?TENxMatrix for the details
oneM

## Note that the following transformation preserves sparsity:
```

```
M2 <- log(oneM + 1)  # delayed
M2                   # a DelayedMatrix instance

## In order to reduce computation times, we'll write only the first
## 5000 columns of M2 to disk:
out_file <- tempfile()
M3 <- writeTENxMatrix(M2[ , 1:5000], out_file, group="mm10", verbose=TRUE)
M3                   # a TENxMatrix instance
```

Index