Package ‘HDF5Array’

April 15, 2020

Title HDF5 backend for DelayedArray objects

Description Implements the HDF5Array and TENxMatrix classes, 2 convenient and memory-efficient array-like containers for on-disk representation of HDF5 datasets. HDF5Array is for datasets that use the conventional (i.e. dense) HDF5 representation. TENxMatrix is for datasets that use the HDF5-based sparse matrix representation from 10x Genomics (e.g. the 1.3 Million Brain Cell Dataset). Both containers being DelayedArray extensions, they support all operations supported by DelayedArray objects. These operations can be either delayed or block-processed.

Version 1.14.4

Encoding UTF-8

Author Hervé Pagès

Maintainer Hervé Pagès <hpages@fredhutch.org>

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

Depends R (>= 3.4), methods, DelayedArray (>= 0.12.3), rhdf5 (>= 2.30.1)

Imports utils, tools, Matrix, BiocGenerics (>= 0.32.0), S4Vectors, IRanges

LinkingTo S4Vectors (>= 0.24.4), Rhdf5lib

SystemRequirements GNU make

Suggests h5vcData, SummarizedExperiment (>= 1.15.1), GenomicRanges, ExperimentHub, TENxBrainData, BiocParallel, GenomicFeatures, BiocStyle

License Artistic-2.0

Collate DSetHandle-class.R h5mread.R h5mread_from_reshaped.R utils.R HDF5ArraySeed-class.R HDF5Array-class.R ReshapedHDF5ArraySeed-class.R ReshapedHDF5Array-class.R dump-management.R writeHDF5Array.R saveHDF5SummarizedExperiment.R TENxMatrixSeed-class.R TENxMatrix-class.R writeTENxMatrix.R zzz.R

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

git_branch RELEASE_3_10

git_last_commit 5e6ab9a

git_last_commit_date 2020-04-04

Date/Publication 2020-04-14

R topics documented:

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, method=0L)

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

Arguments

filepath — The path (as a single character string) to the HDF5 file where the dataset to read from is located.

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

`method` 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

- `h5read` in the **rhdf5** package.
- `type` in the **DelayedArray** 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))

## ---------------------------------------------------------------------
## 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() about 3x 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() 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

index <- list(sample(  27998, 250, replace=TRUE),
              sample(1306127, 250, replace=TRUE))
## h5mread() about 2x faster than h5read():
system.time(a <- h5mread(fname1, "counts", index))
system.time(b <- h5read(fname1, "counts", index=index))
stopifnot(identical(a, b))

## The bigger the selection, the greater the speedup between
## h5read() and h5mread():
## Not run:
  index <- list(sample(  27998, 1000, replace=TRUE),
                sample(1306127, 1000, replace=TRUE))
  ## h5mread() about 8x faster than h5read() (22s vs 3min):
  system.time(a <- h5mread(fname1, "counts", index))
  system.time(b <- h5read(fname1, "counts", index=index))
  stopifnot(identical(a, b))

## 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 character string) to the HDF5 file where the dataset to read from is located.

`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

- 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))
```

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 `rhdf5::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 in the **rhdf5** package, 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".
setRealizationBackend("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 a conventional (i.e. dense) HDF5 dataset.

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

Usage

```
## Constructor function:
HDF5Array(filepath, name, type=NA)
```

Arguments

`filepath`	The path (as a single character string) to the HDF5 file where the dataset is located.
`name`	The name of the dataset in the HDF5 file.
`type`	`NA` or the *R atomic type* (specified as a single string) corresponding to the type of the HDF5 dataset.

Value

An HDF5Array object.

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 (i.e. dense) HDF5 representation.

If your dataset uses the conventional (i.e. dense) HDF5 representation, use the `HDF5Array()` constructor.

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

See Also

- 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` in the **rhdf5** package.

Examples

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

library(rhdf5)  # for h5ls()
```

```
h5ls(tally_file)

## Pick up "Coverages" dataset for Human chromosome 16:
cvg0 <- HDF5Array(tally_file, "/ExampleStudy/16/Coverages")
cvg0

is(cvg0, "DelayedArray")  # TRUE
seed(cvg0)
path(cvg0)
chunkdim(cvg0)

## ---------------------------------------------------------------------
## dim/dimnames
## ---------------------------------------------------------------------
dim(cvg0)

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

## ---------------------------------------------------------------------
## 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)

## ---------------------------------------------------------------------
## 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. HDF5ArraySeed objects are not intended to be used directly. Most end users should create and manipulate HDF5Array objects instead. See ?HDF5Array for more information.

Usage

```
## Constructor function:
HDF5ArraySeed(filepath, name, type=NA)
```

Arguments

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

Details

No operation can be performed directly on an HDF5ArraySeed object. It first needs to be wrapped in a DelayedArray object. The result of this wrapping is an HDF5Array object (an HDF5Array object is just an HDF5ArraySeed object wrapped in a DelayedArray object).

Value

An HDF5ArraySeed object.

See Also

- HDF5Array objects.
- h5ls in the **rhdf5** package.

Examples

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

library(rhdf5)  # for h5ls()
h5ls(tally_file)
```

```
seed <- HDF5ArraySeed(tally_file, "/ExampleStudy/16/Coverages")
seed
path(seed)
dim(seed)
chunkdim(seed)
```

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 object.

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` in the **rhdf5** package.

Examples

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

library(rhdf5)  # for h5ls()
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` in the **rhdf5** package.

Examples

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

library(rhdf5)  # for h5ls()
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, verbose=FALSE)

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.

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

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 `quickResaveHDF5SummarizedExperi`
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) <- 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

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 (i.e. dense) HDF5 representation.

The TENxMatrix class is a DelayedMatrix subclass for representing an HDF5-based sparse matrix like one used by 10x Genomics for the 1.3 Million Brain Cell Dataset.

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

Usage

```
## Constructor functions:
TENxMatrix(filepath, group="mm10")

## sparsity() and a convenient data extractor:
sparsity(x)
extractNonzeroDataByCol(x, j)
```

Arguments

filepath The path (as a single character 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.

x A TENxMatrix (or TENxMatrixSeed) object.

j An integer vector containing valid column indices.

Value

`TENxMatrix`: A TENxMatrix object.

`sparsity`: The number of zero-valued matrix elements in the object divided by its total number of elements (a.k.a. its length).

`extractNonzeroDataByCol`: A NumericList or IntegerList object *parallel* to `j` i.e. 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 you should not assume that they are ordered by ascending row index.

Note

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

If your dataset uses the conventional (i.e. dense) HDF5 representation, use the `HDF5Array()` constructor.

See Also

- HDF5Array objects for representing conventional (i.e. 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.
- `colGrid` and `blockApply` in the **DelayedArray** package.
- The TENxMatrixSeed helper class.
- `h5ls` in the **rhdf5** package.
- 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"]]

## The structure of this HDF5 file can be seen using the h5ls() command
## from the rhdf5 package:
library(rhdf5)
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, "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
## 50000 columns of the 1.3 Million Brain Cell Dataset:
oneM50k <- oneM[ , 1:50000]

## 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. On a modern Linux laptop with 8 core (as reported
## by parallel::detectCores()) and 16 Gb of RAM, reasonably good
## performance is achieved by setting the automatic number of workers
## to 6 and automatic block size to 500 Mb:
workers <- 6
block_size <- 5e8  # 500 Mb
if (.Platform$OS.type != "windows") {
    setAutoBPPARAM(MulticoreParam(workers))
} else {
    ## MulticoreParam() is not supported on Windows so we use SnowParam()
    ## on this platform. Also we reduce the block size to 250 Mb on
    ## 32-bit Windows to avoid memory allocation problems (they tend to
    ## be common there because a process cannot use more than 2 Gb of
    ## memory).
    setAutoBPPARAM(SnowParam(workers))
    if (.Platform$r_arch == "i386")
        block_size <- 2.5e8  # 250 Mb
}
setAutoBlockSize(block_size)
DelayedArray:::set_verbose_block_processing(TRUE)

## We're ready to compute the library sizes, number of genes expressed
## per cell, and average expression across cells:
system.time(lib_sizes <- colSums(oneM50k))
system.time(n_exprs <- colSums(oneM50k != 0))
system.time(ave_exprs <- rowMeans(oneM50k))

## Note that the 3 computations above load the data in oneM50k 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 colGrid() to generate blocks of full
## columns (see ?colGrid for more information):
system.time({
  block_results <- blockApply(oneM50k, FUN, grid=colGrid(oneM50k))
})

## 'block_results' is a list with 1 list element per block in
## colGrid(oneM50k). 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(oneM50k)))
ave_exprs2 <- setNames(tot_exprs / ncol(oneM50k), rownames(oneM50k))

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

## Reset automatic number of workers and automatic block size to factory
## settings:
setAutoBPPARAM()
setAutoBlockSize()
DelayedArray:::set_verbose_block_processing(FALSE)

## ---------------------------------------------------------------------
## extractNonzeroDataByCol()
## ---------------------------------------------------------------------
## extractNonzeroDataByCol() provides a convenient and very efficient
## way to extract the nonzero data in a compact form:
nonzeroes <- extractNonzeroDataByCol(oneM, 1:50000)  # 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:
nonzeroes

names(nonzeroes) <- colnames(oneM50k)

## 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(oneM50k) and
## colSums(oneM50k != 0):
lib_sizes3 <- sum(nonzeroes)
n_exprs3 <- lengths(nonzeroes)

## 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 for representing a pointer to an HDF5-based sparse matrix like one used by 10x Genomics for the 1.3 Million Brain Cell Dataset. TENxMatrixSeed objects are not intended to be used directly. Most end users should create and manipulate TENxMatrix objects instead. See ?TENxMatrix for more information.

Usage

```
## Constructor function:
TENxMatrixSeed(filepath, group="mm10")
```

Arguments

`filepath, group`

See ?TENxMatrix for a description of these arguments.

Details

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

Value

`TENxMatrixSeed()` returns a TENxMatrixSeed object.

See ?TENxMatrix for the value returned by `sparsity()` and `extractNonzeroDataByCol()`.

See Also

- TENxMatrix objects.
- The **rhdf5** package on top of which TENxMatrixSeed objects are implemented.
- The TENxBrainData dataset (in the **TENxBrainData** package).

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"]]

## The structure of this HDF5 file can be seen using the h5ls() command
## from the rhdf5 package:
library(rhdf5)
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, "mm10")
seed
path(seed)
dim(seed)
sparsity(seed)
```

`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, chunkdim=NULL, level=NULL,
                  verbose=FALSE)
```

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.

`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.

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

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.

See Also

- HDF5Array objects.
- `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` in the **rhdf5** package.

Examples

```
## ---------------------------------------------------------------------
## WRITE AN ORDINARY ARRAY TO AN HDF5 FILE
## ---------------------------------------------------------------------
m <- matrix(runif(350, min=-1), nrow=25)
out_file <- tempfile()

M1 <- writeHDF5Array(m, out_file, name="M1", chunkdim=c(5, 5))
M1
chunkdim(M1)

## ---------------------------------------------------------------------
## WRITE A DelayedArray OBJECT TO AN HDF5 FILE
## ---------------------------------------------------------------------
M2 <- log(t(DelayedArray(m)) + 1)
M2 <- writeHDF5Array(M2, out_file, name="M2", chunkdim=c(5, 5))
M2
chunkdim(M2)

library(rhdf5)
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, out_file, "cvg1")
h5ls(out_file)
```

`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 (i.e. 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=FALSE)
```

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, `getHDF5DumpCompressionLeve` will be used. See `?getHDF5DumpCompressionLevel` for more information.

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

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` in the **rhdf5** package.

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 the h5ls() command from the rhdf5 package to see the structure of
## the file:
library(rhdf5)
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, "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