Package 'HDF5Array'

April 16, 2019

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 1.10.1

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 (>= 0.7.41), rhdf5 (>= 2.25.6)

Imports utils, tools, BiocGenerics (>= 0.25.1), S4Vectors, IRanges

Suggests h5vcData, SummarizedExperiment (>= 1.9.6), GenomicRanges, ExperimentHub, BiocParallel, BiocStyle

License Artistic-2.0

Collate utils.R HDF5Array-class.R dump-management.R writeHDF5Array.R saveHDF5SummarizedExperiment.R TENxMatrix-class.R writeTENxMatrix.R zzz.R

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

git_branch RELEASE_3_8

git_last_commit 0b8ae1d

git_last_commit_date 2018-12-05

Date/Publication 2019-04-15

R topics documented:

`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`

OPTIONAL Call `getHDF5DumpChunkDim(dim)` to get reasonable chunk dimensions to use for writing the dataset to disk. Or choose your own chunk dimensions.

2. 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

We provide 2 classes for representing a conventional (i.e. dense) HDF5 dataset as an array-like object in R:

- HDF5Array: A high-level class that extends DelayedArray. All the operations available for DelayedArray objects work on HDF5Array objects.
- HDF5ArraySeed: A low-level class for pointing to an HDF5 dataset. No operation can be performed directly on an HDF5ArraySeed object. It first needs to be wrapped in a DelayedArray object. An HDF5Array object is just an HDF5ArraySeed object wrapped in a DelayedArray object.

Usage

```
## Constructor functions:
HDF5Array(filepath, name, type=NA)
HDF5ArraySeed(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 for HDF5Array().

An HDF5ArraySeed object for HDF5ArraySeed().

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 DelayedArray objects.
- 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.
- h5ls in the **rhdf5** package.
- The **rhdf5** package on top of which HDF5Array and HDF5ArraySeed objects are implemented.

Examples

```
## ---------------------------------------------------------------------
## CONSTRUCTION
## ---------------------------------------------------------------------
library(rhdf5)
library(h5vcData)

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

## Pick up "Coverages" dataset for Human chromosome 16:
cov0 <- HDF5Array(tally_file, "/ExampleStudy/16/Coverages")
cov0

is(cov0, "DelayedArray")  # TRUE

## ---------------------------------------------------------------------
## dim/dimnames
```

```
## ---------------------------------------------------------------------
dim(cov0)

dimnames(cov0)
dimnames(cov0) <- list(paste0("s", 1:6), c("+", "-"), NULL)
dimnames(cov0)

## ---------------------------------------------------------------------
## SLICING (A.K.A. SUBSETTING)
## ---------------------------------------------------------------------
cov1 <- cov0[ , , 29000001:29000007]
cov1

dim(cov1)
as.array(cov1)
stopifnot(identical(dim(as.array(cov1)), dim(cov1)))
stopifnot(identical(dimnames(as.array(cov1)), dimnames(cov1)))

cov2 <- cov0[ , "+", 29000001:29000007]
cov2
as.matrix(cov2)

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

pcov <- cov0[ , 1, ]  # coverage on plus strand
mcov <- cov0[ , 2, ]  # coverage on minus strand

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

## The convention for a SummarizedExperiment object is to have 1 column
## per sample so first we need to transpose 'pcov' and 'mcov':
pcov <- t(pcov)
mcov <- t(mcov)
se <- SummarizedExperiment(list(pcov=pcov, mcov=mcov))
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)$pcov
assays(se)$mcov
```

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.

Usage

```
saveHDF5SummarizedExperiment(x, dir="my_h5_se", replace=FALSE,
                             chunkdim=NULL, level=NULL, verbose=FALSE)
loadHDF5SummarizedExperiment(dir="my_h5_se")
```

Arguments

x	A SummarizedExperiment object.
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 so should not already exist, unless replace is set to TRUE.
replace	If directory dir already exists, should it be replaced with a new one? The content of the 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

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

saveHDF5SummarizedExperiment creates the directory specified thru the dir argument and then 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. It 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 is generally 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.

Value

saveHDF5SummarizedExperiment returns an invisible SummarizedExperiment object where all the assays are HDF5Array objects pointing to the HDF5 datasets saved in dir. It's in fact the same obect as the object that would be returned by calling loadHDF5SummarizedExperiment on dir.

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.

Examples

```
library(SummarizedExperiment)
nrows <- 200; ncols <- 6
counts <- matrix(runif(nrows * ncols, 1, 1e4), nrows)
colData <- DataFrame(Treatment=rep(c("ChIP", "Input"), 3),
                     row.names=LETTERS[1:6])
se0 <- SummarizedExperiment(assays=SimpleList(counts=counts),
                            colData=colData)
se0

## Save 'se0' as an HDF5-based SummarizedExperiment object:
dir <- sub("file", "h5_se0_", tempfile())
h5_se0 <- saveHDF5SummarizedExperiment(se0, 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(all(DelayedArray(assay(se0)) == assay(h5_se0)))
stopifnot(is(assay(h5_se0b, withDimnames=FALSE), "HDF5Matrix"))
stopifnot(all(DelayedArray(assay(se0)) == assay(h5_se0b)))

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

## Make a copy of directory 'dir':
somedir <- sub("file", "somedir", tempfile())
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(all(DelayedArray(assay(se0)) == assay(h5_se0c)))
```

`TENxMatrix-class` *10x Genomics datasets as DelayedArray 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.

We provide 2 classes for representing a 10x Genomics dataset as a matrix-like object in R:

- TENxMatrix: A high-level class that extends DelayedArray. All the operations available for DelayedArray objects work on TENxMatrix objects.
- TENxMatrixSeed: A low-level class for pointing to a 10x Genomics dataset. No operation can be performed directly on a TENxMatrixSeed object. It first needs to be wrapped in a DelayedArray object. A TENxMatrix object is just a TENxMatrixSeed object wrapped in a DelayedArray object.

Usage

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

`TENxMatrixSeed`: A TENxMatrixSeed 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.
- DelayedArray objects in the **DelayedArray** package.
- `writeTENxMatrix` for writing a matrix-like object as an HDF5-based sparse matrix.
- `detectCores` from the **parallel** package.
- `setAutoBPPARAM` and `setAutoBlockSize` in the **DelayedArray** package.
- `colGrid` and `blockApply` in the **DelayedArray** package.
- `h5ls` in the **rhdf5** package.
- The **rhdf5** package on top of which TENxMatrix and TENxMatrixSeed objects are implemented.
- NumericList and IntegerList objects in the **IRanges** package.

Examples

```
## ---------------------------------------------------------------------
## THE "1.3 Million Brain Cell Dataset" AS A DelayedArray 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 the 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
sparsity(oneM)

is(oneM, "DelayedArray")  # TRUE

## 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:
setAutoBPPARAM(MulticoreParam(workers=6))
setAutoBlockSize(5e8)
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))
```

`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, `getHDF5DumpChunkDim(dim(x))` will be used. See ?`getHDF5DumpChunkDim` for more information.
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)

cov0 <- HDF5Array(tally_file, "/ExampleStudy/16/Coverages")

cov1 <- cov0[ , , 29000001:29000007]

writeHDF5Array(cov1, out_file, "cov1")
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 DelayedArray 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	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.
- 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