Package ‘ff’

February 19, 2015

Version 2.2-13
Date 2012-03-29
Title memory-efficient storage of large data on disk and fast access functions
Author Daniel Adler <dadler@uni-goettingen.de>,
    Christian Gläser <christian_glaeser@gmx.de>,
    Oleg Nenadic <onenadi@uni-goettingen.de>,
    Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>,
    Walter Zucchini <wzucchi@uni-goettingen.de>
Maintainer Jens Oehlschlägel <Jens.Oehlschlaegel@truecluster.com>
Depends R (>= 2.10.1), bit (>= 1.1-12), utils
Suggests biglm
Description The ff package provides data structures that are stored on disk but behave (almost) as if they were in RAM by transparently mapping only a section (pagesize) in main memory - the effective virtual memory consumption per ff object. ff supports R's standard atomic data types 'double', 'logical', 'raw' and 'integer' and non-standard atomic types boolean (1 bit), quad (2 bit unsigned), nibble (4 bit unsigned), byte (1 byte signed with NAs), ubyte (1 byte unsigned), short (2 byte signed with NAs), ushort (2 byte unsigned), single (4 byte float with NAs). For example 'quad' allows efficient storage of genomic data as an 'A','T','G','C' factor. The unsigned types support 'circular' arithmetic. There is also support for close-to-atomic types 'factor', 'ordered', 'POSIXct', 'Date' and custom close-to-atomic types.
ff not only has native C-support for vectors, matrices and arrays with flexible dimorder (major column-order, major row-order and generalizations for arrays). There is also a ffdf class not unlike data.frames and import/export filters for csv files.
ff objects store raw data in binary flat files in native encoding, and complement this with metadata stored in R as physical and virtual attributes. ff objects have well-defined hybrid copying semantics, which gives rise to certain performance improvements through virtualization. ff objects can be stored and reopened across R
sessions. ff files can be shared by multiple ff R objects (using different data en/de-coding schemes) in the same process or from multiple R processes to exploit parallelism. A wide choice of finalizer options allows to work with 'permanent' files as well as creating/removing 'temporary' ff files completely transparent to the user. On certain OS/Filesystem combinations, creating the ff files works without notable delay thanks to using sparse file allocation. Several access optimization techniques such as Hybrid Index Preprocessing and Virtualization are implemented to achieve good performance even with large datasets, for example virtual matrix transpose without touching a single byte on disk. Further, to reduce disk I/O, 'logicals' and non-standard data types get stored native and compact on binary flat files i.e. logicals take up exactly 2 bits to represent TRUE, FALSE and NA. Beyond basic access functions, the ff package also provides compatibility functions that facilitate writing code for ff and ram objects and support for batch processing on ff objects (e.g. as.ram, as.ff, ffapply). ff interfaces closely with functionality from package 'bit': chunked looping, fast bit operations and coercions between different objects that can store subscript information ('bit', 'bitwhich', ff 'boolean', ri range index, hi hybrid index). This allows to work interactively with selections of large datasets and quickly modify selection criteria. Further high-performance enhancements can be made available upon request.

License  GPL-2 | file LICENSE
LazyLoad  yes
ByteCompile  yes
Encoding  latin1
URL  http://ff.r-forge.r-project.org/
NeedsCompilation  yes
Repository  CRAN
Date/Publication  2014-04-09 09:54:20

R topics documented:

add .................................................. 5
array2vector .................................. 6
arrayIndex2vectorIndex ..................... 7
as.ff .............................................. 8
as.ff.bit ........................................ 10
as.ffdf ......................................... 11
as.hi ............................................ 12
as.integer.hi .................................. 15
as.vmode ....................................... 17
bigsample ...................................... 18
R topics documented:

- CFUN .................................................. 20
- chunk.bit .............................................. 21
- chunk.ffdf ............................................ 22
- clone .................................................... 24
- clone.ffdf ............................................. 26
- close.ff ................................................ 27
- delete ................................................... 28
- dim.ff ................................................... 30
- dimnames.ffdf ........................................ 32
- dimnames.ff_array ..................................... 33
- dimorderCompatible ................................. 34
- dummy.dimnames ...................................... 35
- Extract.ff ............................................. 36
- Extract.ffdf .......................................... 39
- ff ....................................................... 41
- ffapply .................................................. 49
- ffconform .............................................. 53
- ffdf ..................................................... 55
- ffdfindexget .......................................... 58
- ffdfsort ................................................ 59
- ffdrop ................................................... 61
- ffindexget ............................................. 61
- ffindexorder .......................................... 63
- ffinfo ................................................... 64
- ffload ................................................... 65
- fforder .................................................. 66
- ffreturn ............................................... 68
- ffsave ................................................... 69
- ffsort ................................................... 71
- ffsuitable .............................................. 73
- ffxtensions ............................................ 74
- file.resize ............................................ 75
- filename ............................................... 76
- finalize ............................................... 78
- finalizer .............................................. 79
- fixdiag ............................................... 81
- geterror.ff ........................................... 82
- getpagesize .......................................... 83
- getset.ff .............................................. 84
- hi ......................................................... 85
- hiparse .................................................. 87
- is.ff .................................................... 88
- is.ffdf .................................................. 88
- is.open ............................................... 89
- is.readonly .......................................... 90
- is.sorted ............................................. 91
- length.ff ............................................. 92
- length.ffdf .......................................... 93
R topics documented:

length.hi .......................................................... 94
levels.ff .......................................................... 96
LimWarn ............................................................ 98
matcomb ............................................................ 100
matprint ........................................................... 101
maxffmode .......................................................... 102
maxlength ........................................................... 103
mismatch ............................................................ 104
na.count ............................................................ 105
names.ff ............................................................ 106
nrowAssign .......................................................... 107
open.ff .............................................................. 108
pagesize ............................................................ 109
physical.ff .......................................................... 110
physical.ffdf ......................................................... 111
print.ff ............................................................. 113
ram2ffcode .......................................................... 114
ramattribs ............................................................ 115
ramorder.default ..................................................... 116
ramsort.default ....................................................... 119
read.table.ffdf ....................................................... 121
readwrite.ff ........................................................ 126
regtest.fforder ....................................................... 127
repmam .............................................................. 133
sortLevels ............................................................ 134
splitPathFile .......................................................... 136
swap ................................................................. 139
symmetric ............................................................ 141
symmIndex2vectorIndex .............................................. 142
unclass_- ............................................................. 143
undim ................................................................. 143
unsort ................................................................. 144
update.ff ............................................................. 145
vecprint ............................................................... 147
vector.vmode .......................................................... 148
vector2array .......................................................... 149
vectorIndex2arrayIndex ............................................ 150
vmode ................................................................. 151
vmode.ffdf ............................................................ 153
vt ................................................................. 154
vw ................................................................. 155
write.table.ffdf ......................................................... 157

Index 160
add

Incrementing an ff or ram object

Description

Yet another assignment interface in order to allow to formulate `x[index,...,add=TRUE]<-value` in a way which works transparently, not only for ff, but also for ram objects: `add(x, value, index,...)`.

Usage

```R
add(x, ...)
## S3 method for class 'ff'
add(x, value, ...)
## Default S3 method:
add(x, value, ...)
```

Arguments

- `x`: an ff or ram object
- `value`: the amount to increment, possibly recycled
- `...`: further arguments – especially index information – passed to `[<-` or `[<-.ff`

Value

`invisible()`

Note

Note that `add.default` changes the object in its parent frame and thus violates R’s usual functional programming logic. Duplicated index positions should be avoided, because ff and ram objects behave differently:

```R
add.ff(x, 1, c(3,3))
# will increment x at position 3 TWICE by 1, while
add.default(x, 1, c(3,3))
# will increment x at position 3 just ONCE by 1
```

Author(s)

Jens Oehlschlägel

See Also

`swap, [.ff, LimWarn`
Examples

message("incrementing parts of a vector")
  x <- ff(0, length=12)
  y <- rep(0, 12)
  add(x, 1, 1:6)
  add(y, 1, 1:6)
  x
  y

message("incrementing parts of a matrix")
  x <- ff(0, dim=3:4)
  y <- array(0, dim=3:4)
  add(x, 1, 1:2, 1:2)
  add(y, 1, 1:2, 1:2)
  x
  y

message("BEWARE that ff and ram methods differ in treatment of duplicated index positions")
  add(x, 1, c(3,3))
  add(y, 1, c(3,3))
  x
  y
  rm(x); gc()

array2vector

Array: make vector from array

Description

Makes a vector from an array respecting ‘dim’ and ‘dimorder’

Usage

array2vector(x, dim = NULL, dimorder = NULL)

Arguments

x an array

dim

dimorder

dimorder

Details

This is the inverse function of vector2array. It extracts the vector from the array by first moving through the fastest rotating dimension dim[dimorder[1]], then dim[dimorder[2]], and so forth
**arrayIndex2vectorIndex**

**Value**

a vector

**Author(s)**

Jens Oehlschlägel

**See Also**

`vector2array, arrayIndex2vectorIndex`

**Examples**

```r
array2vector(matrix(1:12, 3, 4))
array2vector(matrix(1:12, 3, 4, byrow=TRUE), dimorder=2:1)
```

---

**arrayIndex2vectorIndex**

*Array: make vector positions from array index*

**Description**

Make vector positions from a (non-symmetric) array index respecting ‘dim’ and ‘dimorder’

**Usage**

```r
arrayIndex2vectorIndex(x, dim = NULL, dimorder = NULL, vw = NULL)
```

**Arguments**

- `x` an n by m matrix with n m-dimensional array indices
- `dim` NULL or `dim`
- `dimorder` NULL or `dimorder`
- `vw` NULL or integer vector[3] or integer matrix[3,m], see details

**Details**

The fastest rotating dimension is `dim[dimorder[1]]`, then `dim[dimorder[2]]`, and so forth. The parameters ‘x’ and ‘dim’ may refer to a subarray of a larger array, in this case, the array indices ‘x’ are interpreted as ‘vw[1,] + x’ within the larger array ‘as.integer(colSums(vw))’.

**Value**

a vector of indices in 1:prod(dim) (or 1:prod(colSums(vw)))

**Author(s)**

Jens Oehlschlägel
Description

Coercing ram to ff and ff to ram objects while optionally modifying object features.

Usage

```
as.ff(x, ...)  
as.ff(x, filename = NULL, overwrite = FALSE, ...)  
## S3 method for class 'ff'
as.ff(x, filename = NULL, overwrite = FALSE, ...)  
## Default S3 method:
as.ff(x, ...)  
## S3 method for class 'ff'
as.ff(x, ...)  
## S3 method for class 'ff'
as.ff(x, ...)  
## S3 method for class 'ff'
as.ff(x, ...)  
## S3 method for class 'ff'
as.ff(x, ...)  
## S3 method for class 'ff'
as.ff(x, ...)  
```

Arguments

- `x`: any object to be coerced
- `filename`: path and filename
- `overwrite`: TRUE to overwrite the old filename
- `...`:...

See Also

array2vector, vectorIndex2arrayIndex

Examples

```
x <- matrix(1:12, 3, 4)
x
arrayIndex2vectorIndex(cbind(as.vector(row(x)), as.vector(col(x)))
, dim=dim(x))
arrayIndex2vectorIndex(cbind(as.vector(row(x)), as.vector(col(x)))
, dim=dim(x), dimorder=2:1)
matrix(1:30, 5, 6)
arrayIndex2vectorIndex(cbind(as.vector(row(x)), as.vector(col(x)))
, vw=rbind(c(0,1), c(3,4), c(2,1)))
arrayIndex2vectorIndex(cbind(as.vector(row(x)), as.vector(col(x)))
, vw=rbind(c(0,1), c(3,4), c(2,1)), dimorder=2:1)
```

Coercing ram to ff and ff to ram objects
Details

If `as.ff.ff` is called on an 'ff' object or `as.ram.default` is called on a non-ff object AND no changes are required, the input object 'x' is returned unchanged. Otherwise the workhorse `clone.ff` is called. If no change of features are requested, the filename attached to the object remains unchanged, otherwise a new filename is requested (or can be set by the user).

Value

A ram or ff object.

Note

If you use `ram <- as.ram(ff)` for caching, please note that you must `close.ff` before you can write back `as.ff(ram, overwrite=TRUE)` (see examples).

Author(s)

Jens Oehlschlägel

See Also

`as.ff.bit, ff, clone, as.vmode, vmode, as.hi`

Examples

```r
message("create ff")
myintff <- ff(1:12)
message("coerce (=clone) integer ff to double ff")
mydoubleff <- as.ff(myintff, vmode="double")
message("cache (=clone) integer ff to integer ram AND close original ff")
myintram <- as.ram(myintff) # filename is retained
  close(myintff)
message("modify ram cache and write back (=clone) to ff")
myintram[1] <- -1L
myintff <- as.ff(myintram, overwrite=TRUE)
message("coerce (=clone) integer ram to double ram")
mydoubleram <- as.ram(myintram, vmode="double")
message("coerce (inplace) integer ram to double ram")
myintram <- as.ram(myintram, vmode="double")
message("more classic: coerce (inplace) double ram to integer ram")
vmode(myintram) <- "integer"
rm(myintff, myintram, mydoubleff, mydoubleram); gc()
```
Description

Function \texttt{as.ff.bit} converts a \texttt{bit} vector to a boolean \texttt{ff} vector. Function \texttt{as.bit.ff} converts a boolean \texttt{ff} vector to a \texttt{ff} vector.

Usage

\begin{verbatim}
## S3 method for class 'bit'
as.ff(x, filename = NULL, overwrite = FALSE, ...)
## S3 method for class 'ff'
as.bit(x, ...)
\end{verbatim}

Arguments

- \texttt{x} the source of conversion
- \texttt{filename} optionally a desired filename
- \texttt{overwrite} logical indicating whether we allow overwriting the target file
- \texttt{...} further arguments passed to \texttt{ff} in case \texttt{as.ff.bit}, ignored in case of \texttt{as.bit.ff}

Details

The data are copied bit-wise but integerwise, therefore these conversions are very fast. \texttt{as.bit.ff} will attach the \texttt{ff} filename to the bit vector, and \texttt{as.ff.bit} will - if attached - use \texttt{THIS} filename and SILENTLY overwrite this file.

Value

A vector of the converted type

Note

NAs are mapped to TRUE in 'bit' and to FALSE in 'ff' booleans. Might be aligned in a future release. Don't use bit if you have NAs - or map NAs explicitly.

Author(s)

Jens Oehlschlägel

See Also

\texttt{bit, ff, as.ff, as.hi.bit}
as.ffdf

Examples

```r
1 <- as.boolean(sample(c(FALSE,TRUE), 1000, TRUE))

b <- as.bit(1)
stopifnot(identical(1,b[]))
b
f <- as.ff(b)
stopifnot(identical(1,f[]))
f
b2 <- as.bit(f)
stopifnot(identical(1,b2[]))
b2
f2 <- as.ff(b2)
stopifnot(identical(filename(f),filename(f2)))
stopifnot(identical(1,f2[]))
f
rm(f,f2); gc()
```

---

as.ffdf

Coercing to ffdf and data.frame

Description

Functions for coercing to ffdf and data.frame

Usage

```r
as.ffdf(x, ...)
## S3 method for class 'ff_vector'
as.ffdf(x, ...)
## S3 method for class 'ff_matrix'
as.ffdf(x, ...)
## S3 method for class 'data.frame'
as.ffdf(x, vmode=NULL, col_args = list(), ...)
## S3 method for class 'ffdf'
as.data.frame(x, ...)
```

Arguments

- `x` - the object to be coerced
- `vmode` - optional specification of the vmodes of columns of the data.frame. Either a character vector of vmodes (named with column names of the data.frame or recycled if not named) or a list named with vmodes where each element identifies those columns of the data.frame that should get the vmode encoded in the name of the element
- `col_args` - further arguments; passed to `ff`
- `...` - further arguments; passed to `ffdf` for `.ff_vector`, `.ff_matrix` and `.data.frame` methods, ignored for `.ffdf identity method`
Value

'as.ffdf' returns an object of class \texttt{ffdf}, 'as.data.frame' returns an object of class \texttt{data.frame}

Author(s)

Jens Oehlenschlägel

See Also

\texttt{is.ffdf,ffdf,\texttt{data.frame}}

Examples

```r

d <- data.frame(x=1:26, y=letters, z=Sys.time()+1:26)
ffd <- as.ffdf(d)
stopifnot(identical(d, as.data.frame(ffd)))
rm(ffd); gc()
```

Description

The generic \texttt{as.hi} and its methods are the main (internal) means for preprocessing index information into the hybrid index class \texttt{hi}. Usually \texttt{as.hi} is called transparently from \texttt{\ldots}. However, you can explicitly do the index-preprocessing, store the Hybrid Index \texttt{hi}, and use the \texttt{hi} for subscripting.

Usage

```r

\texttt{as.hi(x, \ldots)}

## S3 method for class 'hi'
as.hi(x, \ldots)

## S3 method for class 'ri'
as.hi(x, maxindex = length(x), \ldots)

## S3 method for class 'bit'
as.hi(x, range = NULL, maxindex = length(x), vw = NULL, dim = NULL, dimorder = NULL, pack = TRUE, \ldots)

## S3 method for class 'bitwhich'
as.hi(x, maxindex = length(x), pack = FALSE, \ldots)

## S3 method for class 'call'
as.hi(x, maxindex = length(x), pack = FALSE, \ldots)

## S3 method for class 'name'
as.hi(x, envir = parent.frame(), \ldots)

## S3 method for class 'integer'
```

\texttt{Hybrid Index, coercion to}
as.hi(x, maxindex = NA, dim = NULL, dimorder = NULL
, symmetric = FALSE, fixdiag = NULL, vw = NULL, vw.convert = TRUE
, dimorder.convert = TRUE, pack = TRUE, NAs = NULL, ...)
## S3 method for class 'which'
as.hi(x, ...)
## S3 method for class 'double'
as.hi(x, ...)
## S3 method for class 'logical'
as.hi(x, maxindex = NA, dim = NULL, vw = NULL, pack = TRUE, ...)
## S3 method for class 'character'
as.hi(x, names, vw = NULL, vw.convert = TRUE, ...)
## S3 method for class 'matrix'
as.hi(x, dim, dimorder = NULL, symmetric = FALSE, fixdiag = NULL
, vw = NULL, pack = TRUE, ...)

Arguments

x       an appropriate object of the class for which we dispatched
envir   the environment in which to evaluate components of the index expression
maxindex maximum positive index position maxindex, is needed with negative indices, if
          vw or dim is given, maxindex is calculated automatically
names   the names of the indexed vector for character indexing
dim     the dim of the indexed matrix to be stored within the hi object
dimorder the dimorder of the indexed matrix to be stored within the hi object, may con-
          vert interpretation of x
symmetric the symmetric of the indexed matrix to be stored within the hi object
fixdiag  the fixdiag of the indexed matrix to be stored within the hi object
vw      the virtual window vw of the indexed vector or matrix to be stored within the hi
          object, see details
vw.convert FALSE to prevent doubly virtual window conversion, this is needed for some
          internal calls that have done the virtual window conversion already, see details
dimorder.convert FALSE to prevent doubly dimorder conversion, this is needed for some internal
calls that have done the dimorder conversion already, see details
NAs     a vector of NA positions to be stored rlepacked, not fully supported yet
pack    FALSE to prevent rlepacking, note that this is a hint rather than a guarantee,
as.hi.bit might ignore this
range   NULL or a vector with two elements indicating first and last position to be con-
          verted from ‘bit’ to ‘hi’
...     further argument passed from generic to method or from wrapper method to
          as.hi.integer
Details

The generic dispatches appropriately, as.hi'hui returns an hi object unchanged, as.hi'hui call tries to hiparse instead of evaluate its input in order to save RAM. If parsing is successfull as.hi'hui call will ignore its argument pack and always pack unless the subscript is too small to do so. If parsing fails it evaluates the index expression and dispatches again to one of the other methods. as.hi'hui.name and as.hi'hui.( are wrappers to as.hi'hui.call. as.hi'hui.integer is the workhorse for coercing evaluated expressions and as.hi'hui.which is a wrapper removing the which class attribute. as.hi'hui.double, as.hi'hui.logical and as.hi'hui.character are also wrappers to as.hi'hui.integer, but note that as.hi'hui.logical is not memory efficient because it expands all positions and then applies logical subscripting.
as.hi'hui.matrix calls arrayIndex2vectorIndex and then as.hi'hui.integer to interpret and preprocess matrix indices.

If the dim and dimorder parameter indicate a non-standard dimorder (dimorderStandard), the index information in x is converted from a standard dimorder interpretation to the requested dimorder.
If the vw parameter is used, the index information in x is interpreted relative to the virtual window but stored relative to the absolute origin. Back-coercion via as.integer.hi and friends will again return the index information relative to the virtual window, thus retaining symmetry and transparency of the virtual window to the user.
You can use length to query the index length (possibly length of negative subscripts), poslength to query the number of selected elements (even with negative subscripts), and maxindex to query the largest possible index position (within virtual window, if present)
Duplicated negative indices are removed and will not be recovered by as.integer.hi.

Value

an object of class hi

Note

Avoid changing the Hybrid Index representation, this might crash the [.ff subscripting.

Author(s)

Jens Oehlschlägel

See Also

hi for the Hybrid Index class, hiparse for parsing details, as.integer.hi for back-coercion, [.ff for ff subscripting

Examples

message("integer indexing with and without rel-packing")
as.hi(1:12)
as.hi(1:12, pack=FALSE)
message("if index is double, the wrapper method just converts to integer")
as.hi(as.double(1:12))
message("if index is character, the wrapper method just converts to integer")
as.hi(c("a","b","c"), names=letters)
message("negative index must use maxindex (or vw)")
as.integer.hi

Functions that (back-)convert an hi object to the respective subscripting information.

Usage

## S3 method for class 'hi'
as.which(x, ...)

## S3 method for class 'hi'
as.bitwhich(x, ...)

Description

Hybrid Index, coercing from
as.integer.hi

as.integer.hi

Arguments

- **x** an object of class **hi**
- **maxindex** the length of the subscripted object (needed for logical output)
- **names** the names vector of the subscripted object
- **dim** the dim of the subscripted object
- **dimorder** the dimorder of the subscripted object
- **vw** the virtual window vw of the subscripted object
- **vw.convert** vw.convert
- **symmetric** TRUE if the subscripted matrix is symmetric
- **fixdiag** TRUE if the subscripted matrix has fixdiag
- **...** further arguments passed

Value

as.integer.hi returns an integer vector, see as.hi.integer. as.logical.hi returns a logical vector, see as.hi.logical. as.character.hi returns a character vector, see as.hi.character. as.matrix.hi returns a matrix index, see as.hi.matrix.

Author(s)

Jens Oehlschlägel

See Also

hi, as.hi

Examples

```r
x <- 1:6
names(x) <- letters[1:6]
as.integer(as.hi(c(1:3)))
as.logical(as.hi(c(TRUE, TRUE, TRUE, FALSE, FALSE, FALSE)))
as.character(as.hi(letters[1:3]), names=names(x), names=names(x))
x <- matrix(1:12, 6)
as.matrix(as.hi(rbind(c(1,1), c(1,2), c(2,1)), dim=dim(x), dim=dim(x))
```
Description

as.vmode is a generic that converts some R ram object to the desired vmode.

Usage

```r
as.vmode(x, ...)  
as.boolean(x, ...)  
as.quad(x, ...)  
as.nibble(x, ...)  
as.byte(x, ...)  
as.ubyte(x, ...)  
as.short(x, ...)  
as.ushort(x, ...)  
## Default S3 method:
as.vmode(x, vmode, ...)  
## S3 method for class 'ff'  
as.vmode(x, ...)  
## Default S3 method:
as.boolean(x, ...)  
## Default S3 method:
as.quad(x, ...)  
## Default S3 method:
as.nibble(x, ...)  
## Default S3 method:
as.byte(x, ...)  
## Default S3 method:
as.ubyte(x, ...)  
## Default S3 method:
as.short(x, ...)  
## Default S3 method:
as.ushort(x, ...)  
```

Arguments

- **x**: any object
- **vmode**: virtual mode
- **...**: The ... don’t have a function yet, they are only defined to keep the generic flexible.

Details

Function as.vmode actually coerces to one of the usual storagemodes (see .rammode) but flags them with an additional attribute 'vmode' if necessary. The coercion generics can also be called
directly:

- `as.boolean`: 1 bit logical without NA
- `as.logical`: 2 bit logical with NA
- `as.quad`: 2 bit unsigned integer without NA
- `as.nibble`: 4 bit unsigned integer without NA
- `as.byte`: 8 bit signed integer with NA
- `as.ubyte`: 8 bit unsigned integer without NA
- `as.short`: 16 bit signed integer with NA
- `as.ushort`: 16 bit unsigned integer without NA
- `as.integer`: 32 bit signed integer with NA
- `as.single`: 32 bit float
- `as.double`: 64 bit float
- `as.complex`: 2x64 bit float
- `as.raw`: 8 bit unsigned char
- `as.character`: character

**Value**

A vector of the desired vmode containing the input data

**Author(s)**

Jens Oehlschlägel

**See Also**

`vmode`, `vector.vmode`

**Examples**

```r
as.vmode(1:3, "double")
as.vmode(1:3, "byte")
as.double(1:3)
as.byte(1:3)
```

**bigsample**

Sampling from large pools

**Description**

`bigsample` samples quicker from large pools than `sample` does.
Usage

bigsample(x, ...)
## Default S3 method:
bigsample(x, size, replace = FALSE, prob = NULL, negative = FALSE, ...)
## S3 method for class 'ff'
bigsample(x, size, replace = FALSE, prob = NULL, ...)

Arguments

- **x**: the pool to sample from
- **size**: the number of elements to sample
- **replace**: TRUE to use sampling with replacement
- **prob**: optional vector of sampling probabilities (recycled to pool length)
- **negative**: negative
- **...**: ...

Details

For small pools sample is called.

Value

- a vector of elements sampled from the pool (argument 'x')

Note

Note that bigsample and sample do not necessarily return the same sequence of elements when set.seed is set before.

Author(s)

Daniel Adler, Jens Oehlschlägel, Walter Zucchini

See Also

sample, ff

Examples

message("Specify pool size")
bigsample(1e8, 10)
message("Sample ff elements (same as x[bigsample(length(ff(1:100 / 10)), 10)])")
bigsample(ff(1:100 / 10), 10)
## Not run:
message("Speed factor")
(system.time(for(i in 1:10)sample(1e8, 10))[3]/10)
/ (system.time(for(i in 1:1000)bigsample(1e8, 10))[3]/10000)

## End(Not run)
Collapsing functions for batch processing

Description

These are used in aggregating the chunks resulting from batch processing. They are usually called via `do.call`

Usage

```r
cbind(...)
crbind(...)
cfun(..., FUN, FUNARGS = list())
cquantile(..., probs = seq(0, 1, 0.25), na.rm = FALSE, names = TRUE, type = 7)
csummary(..., na.rm = "ignored")
cmedian(..., na.rm = FALSE)
clength(..., na.rm = FALSE)
csum(..., na.rm = FALSE)
cmean(..., na.rm = FALSE)
```

Arguments

... ... 
FUN a aggregating function
FUNARGS further arguments to the aggregating function
na.rm TRUE to remove NAs
probs see `quantile`
names see `quantile`
type see `quantile`

Details

<table>
<thead>
<tr>
<th>CFUN</th>
<th>FUN</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>cbind</td>
<td>cbind</td>
<td>like <code>cbind</code> but respecting names</td>
</tr>
<tr>
<td>crbind</td>
<td>rbind</td>
<td>like <code>rbind</code> but respecting names</td>
</tr>
<tr>
<td>cfun</td>
<td>crbind</td>
<td>crbind the input chunks and then apply 'FUN' to each column</td>
</tr>
<tr>
<td>cquantile</td>
<td>quantile</td>
<td>crbind the input chunks and then apply 'quantile' to each column</td>
</tr>
<tr>
<td>csummary</td>
<td>summary</td>
<td>crbind the input chunks and then apply 'summary' to each column</td>
</tr>
<tr>
<td>cmedian</td>
<td>median</td>
<td>crbind the input chunks and then apply 'median' to each column</td>
</tr>
<tr>
<td>clength</td>
<td>length</td>
<td>crbind the input chunks and then determine the number of values in each column</td>
</tr>
<tr>
<td>csum</td>
<td>sum</td>
<td>crbind the input chunks and then determine the sum values in each column</td>
</tr>
<tr>
<td>cmean</td>
<td>mean</td>
<td>crbind the input chunks and then determine the (unweighted) mean in each column</td>
</tr>
</tbody>
</table>
In order to use CFUNs on the result of `lapply` or `ffapply` use `do.call`.

**Value**

depends on the CFUN used

**ff options**

xx TODO: extend this for weighted means, weighted median etc.,

**Note**

Currently - for command line convenience - we map the elements of a single list argument to `...`, but this may change in the future.

**Author(s)**

Jens Oehlschlägel

**See Also**

`ffapply, do.call, na.count`

**Examples**

```r
X <- lapply(split(rnorm(10000), 1:10), summary)
do.call("cbind", X)
do.call("csummary", X)
do.call("cmean", X)
do.call("cfun", c(X, list(FUN=mean, FUNARGS=list(na.rm=TRUE))))
rm(X)
```

---

**chunk.bit**

**Chunk bit vectors**

**Description**

chunking method for ffdf objects automatically considering RAM requirements from recordsize as calculated from `sum(.rambytes[vmode])`

**Usage**

```r
## S3 method for class 'bit'
chunk(x, RECORDBYTES = .rambytes["logical"], BATCHBYTES = getOption("ffbatchbytes"), ...)
```
Arguments

- **x**: bit
- **RECORDBYTES**: optional integer scalar representing the bytes needed to process a single element of the bit vector
- **BATCHBYTES**: integer scalar limiting the number of bytes to be processed in one chunk, default from `getOption("ffbatchbytes")`, see also `.rambytes`
- ... further arguments passed to `chunk`

Value

A list with `ri` indexes each representing one chunk

Author(s)

Jens Oehlschlägel

See Also

`chunk`, `bit`

Examples

```r
n <- 1000
x <- bit(n)
ceiling(n / (300 %/% sum(.rambytes["logical"])))
chunk(x, BATCHBYTES=300)
ceiling((n/2) / (100 %/% sum(.rambytes["logical"])))
chunk(x, from=1, to = n/2, BATCHBYTES=100)
rm(x, n)
```

---

**chunk.ffdf**

Chunk ff_vector and ffdf

Description

Chunking method for ff_vector and ffdf objects (row-wise) automatically considering RAM requirements from recordsize as calculated from `sum(.rambytes[vmode])`

Usage

```r
## S3 method for class 'ff_vector'
chunk(x
, RECORDBYTES = .rambytes[vmode(x)], BATCHBYTES = getOption("ffbatchbytes"), ...)  
## S3 method for class 'ffdf'
chunk(x
, RECORDBYTES = sum(.rambytes[vmode(x)]), BATCHBYTES = getOption("ffbatchbytes"), ...)  
```
Arguments

\textbf{x} \quad \text{ff or ffdf}

\textbf{RECORDBYTES} \quad \text{optional integer scalar representing the bytes needed to process an element of the \texttt{ff}\_vector a single row of the ffdf}

\textbf{BATCHBYTES} \quad \text{integer scalar limiting the number of bytes to be processed in one chunk, default from \texttt{getOption("ffbatchbytes")}, see also \texttt{.rambytes}}

\ldots \quad \text{further arguments passed to \texttt{chunk}}

Value

A list with \texttt{ri} indexes each representing one chunk

Author(s)

Jens Oehlschlägel

See Also

\texttt{chunk, \texttt{ffdf}}

Examples

\begin{verbatim}
x <- data.frame(x=as.double(1:26), y= factor(letters), z= ordered(LETTERS))
a <- as.ffdf(x)
ceiling(26 / (300 \%\% sum(.rambytes[vmode(a)])))
chunk(a, BATCHBYTES=300)
ceiling(13 / (100 \%\% sum(.rambytes[vmode(a)])))
chunk(a, from=1, to = 13, BATCHBYTES=100)
rm(a); gc()

message("dummy example for linear regression with \texttt{biglm} on \texttt{ffdf}")
library(biglm)

message("NOTE that \ . in formula requires calculating terms manually
because . as a data-dependant term is not allowed in \texttt{biglm}\")
form <- Sepal.Length - Sepal.Width + Petal.Length + Petal.Width + Species

lmfit <- lm(form, data=iris)
firis <- as.ffdf(iris)
for (i in chunk(firis, by=50)){
  if (i[1]==1){
    message("first chunk is: ", i[[1]], ":",i[[2]])
    biglmfit <- biglm(form, data=firis[i,,drop=FALSE])
  }else{
    message("next chunk is: ", i[[1]], ":",i[[2]])
    biglmfit <- update(biglmfit, firis[i,,drop=FALSE])
  }
}
\end{verbatim}
summary(lmfit)
summary(glmfit)
stopifnot(all.equal(coef(lmfit), coef(glmfit)))

---

**clone**  
*Cloning ff and ram objects*

**Description**

clone physically duplicates ff (and ram) objects and can additionally change some features, e.g. length.

**Usage**

```r
clone(x, ...)  
## S3 method for class 'ff'
clone(x)
  , initsdata = x
  , length = NULL
  , levels = NULL
  , ordered = NULL
  , dim = NULL
  , dimorder = NULL
  , bydim = NULL
  , symmetric = NULL
  , fixdiag = NULL
  , names = NULL
  , dimnames = NULL
  , ramclass = NULL
  , ramattrs = NULL
  , vmode = NULL
  , update = NULL
  , pattern = NULL
  , filename = NULL
  , overwrite = FALSE
  , pagesize = NULL
  , caching = NULL
  , finalizer = NULL
  , finonexit = NULL
  , FF_RETURN = NULL
  , BATCHSIZE = .Machine$integer.max
  , BATCHBYTES =getOption("ffbatchbytes")
  , VERBOSE = FALSE
  , ...)
## S3 method for class 'list'
clone(x, ...)
## Default S3 method:
clone(x, ...)
```
clone

Arguments

x x

initdata scalar or vector of the implemented vmodes, recycled if needed, default 0, see also as.vmode and vector.vmode

length optional vector length of the object (default: derive from 'initdata' or 'dim'), see length.ff

levels optional character vector of levels if (in this case initdata must be composed of these) (default: derive from initdata)

ordered indicate whether the levels are ordered (TRUE) or non-ordered factor (FALSE, default)

dim optional array dim, see dim.ff and array

dimorder physical layout (default 1:length(dim)), see dimorder and aperm

bydim dimorder by which to interpret the 'initdata', generalization of the 'byrow' parameter in matrix

symmetric extended feature: TRUE creates symmetric matrix (default FALSE)

fixdiag extended feature: non-NULL scalar requires fixed diagonal for symmetric matrix (default NULL is free diagonal)

names see names
dimnames NOT taken from initdata, see dimnames
ramclass class attribute attached when moving all or parts of this ff into ram, see ramclass
ramattribs additional attributes attached when moving all or parts of this ff into ram, see ramattribs

vmode virtual storage mode (default: derive from 'initdata'), see vmode and as.vmode

update set to FALSE to avoid updating with 'initdata' (default TRUE) (used by ffdf)

pattern root pattern for automatic ff filename creation (default "ff"), see also physical

filename ff filename (default tmpfile with 'pattern' prefix), see also physical

overwrite set to TRUE to allow overwriting existing files (default FALSE)

pagesize pagesize in bytes for the memory mapping (default from getOptions("ffpagesize") initialized by getdefaultpagesize), see also physical

caching caching scheme for the backend, currently ‘mmnoflush’ or ‘mmeachflush’ (flush mmpages at each swap, default from getOptions("ffcaching") initialized with ‘memorymap’), see also physical

finalizer name of finalizer function called when ff object is removed, (default "deleteIfOpen" from getOptions("fffinalizer")), standard finalizers are close.ff, delete.ff and deleteIfOpen.ff, see also reg.finalizer

finonexit logical scalar determining whether finalizer is also called when R is closed via q, (default TRUE from getOptions("fffinonexit"))

FF_RETURN logical scalar or ff object to be used. The default NULL creates a ff or ram clone, TRUE returns a ff clone, FALSE returns a ram clone. Handing over an ff object here uses this or stops if not ffsuitable
clone is generic. clone.ffdf is the workhorse behind as.ram and as.ff. For creating the desired object it calls ff which calls update for initialization.

Value

an ff or ram object

Author(s)

Jens Oehlschlägel

See Also

ff, update, as.ram, as.ff

Examples

x <- ff(letters, levels=letters)
y <- clone(x, length=52)
rm(x,y); gc()

clone.ffdf

Cloning ffdf objects

Description

close physically duplicates ffdf objects

Usage

## S3 method for class 'ffdf'
clone(x, nrow=NULL, ...)

Arguments

x

an ffdf

nrow

optionally the desired number of rows in the new object. Currently this works only together with initdata=NULL

... further arguments passed to clone (usually not useful)
Details

Creates a deep copy of an ffdf object by cloning all **physical** components including the `row.names`

Value

An object of type `ffdf`

Author(s)

Jens Oehlschlägel

See Also

*clone*, *ffdf*

Examples

```r
x <- as.ffdf(data.frame(a=1:26, b=letters))

message("Here we change the content of both x and y by reference")
y <- x
x$a[1] <- -1
y$a[1]

message("Here we change the content only of x because y is a deep copy")
y <- clone(x)
y$a[2] <- -2
y$a[2]
rm(x, y); gc()
```

---

**close.ff**

**Closing ff files**

Description

Close frees the Memory Mapping resources and closes the ff file without deleting the file data.

Usage

```r
## S3 method for class 'ff'
close(con, ...)
## S3 method for class 'ffdf'
close(con, ...)
## S3 method for class 'ff_pointer'
close(con, ...)
```
Arguments

con an open ff object

Details

The ff_pointer method is not intended for manual use, it is used at finalizer dispatch time. Closing
ffdf objects will close all of their physical components including their row.names if they are is.ff

Value

TRUE if the file could be closed, FALSE if it was closed already (or NA if not all components of
an ffdf returned FALSE or TRUE on closing)

Author(s)

Jens Oehlschlägel

See Also

ff, open.ff, delete, deleteIfOpen

Examples

x <- ff(1:12)
close(x)
x
open(x)
x
rm(x); gc()

---

delete Deleting the file behind an ff object

Description

The generic delete deletes the content of an object without removing the object itself. The generic
deleteIfOpen does the same, but only if is.open returns TRUE.

Usage

delete(x, ...)
deleteIfOpen(x, ...)
## S3 method for class 'ff'
delete(x, ...)
## S3 method for class 'ffdf'
delete(x, ...)
## S3 method for class 'ff_pointer'
delete(x, ...)  
## Default S3 method:  
delete(x, ...)  
## S3 method for class 'ff'  
deleteIfOpen(x, ...)  
## S3 method for class 'ff_pointer'  
deleteIfOpen(x, ...)

Arguments

x                  an ff or ram object

...                further arguments (not used)

Details

The proper sequence to fully delete an ff object is: delete(x); rm(x), where delete.ff frees the Memory Mapping resources and deletes the ff file, leaving intact the R-side object including its class, physical and virtual attributes. The default method is a compatibility function doing something similar with ram objects: by assigning an empty list to the name of the ram object to the parent frame we destroy the content of the object, leaving an empty stub that prevents raising an error if the parent frame calls the delete(x); rm(x) sequence.  
The deleteIfOpen does the same as delete but protects closed ff objects from deletion, it is mainly intended for use through a finalizer, as are the ff_pointer methods.

Value

dclete returns TRUE if the/all ff files could be removed and FALSE otherwise.
dcleteIfOpen returns TRUE if the/all ff files could be removed, FALSE if not and NA if the ff object was open.

Note

Deletion of ff files can be triggerd automatically via three routes:

1. if an ff object with a 'delete' finalizer is removed  
2. if an ff object was created with ffnonexit=TRUE the finalizer is also called when R shuts down.  
3. if an ff object was created in getOption("fftempdir"), it will be unlinked together with the fftempdir.onUnload

Thus in order to retain an ff file, one has to create it elsewhere than in ffftempdir with a finalizer that does not destroy the file (by default files outside ffftempdir get a 'close' finalizer) i.e. one of the following:

1. name the file AND use fffinalizer="close"  
2. name the file AND use fffinalizer="deleteIfOpen" AND close the ff object before leaving R  
3. name the file AND use fffinalizer="delete" AND use ffnonexit=FALSE
Author(s)
Jens Oehlschlägel

See Also
ff, close_ff, open_ff, reg.finalizer

Examples
message('create the ff file outside getOption("fftempir"),
  it will have default finalizer "close", so you need to delete it explicitly')
x <- ff(1:12, pattern="/ffexample")
delete(x)
rm(x)

Description
Assigning dim to an ff_vector changes it to an ff_array. Beyond that dimorder can be assigned to change from column-major order to row-major order or generalizations for higher order ff_array.

Usage

```r
## S3 method for class 'ff'
dim(x)
## S3 method for class 'ffdf'
dim(x)
## S3 replacement method for class 'ff'
dim(x) <- value
## S3 replacement method for class 'ffdf'
dim(x) <- value
dimorder(x, ...)
dimorder(x, ...) <- value
## Default S3 method:
dimorder(x, ...)
## S3 method for class 'ff_array'
dimorder(x, ...)
## S3 method for class 'ffdf'
dimorder(x, ...)
## S3 replacement method for class 'ff_array'
dimorder(x, ...) <- value
## S3 replacement method for class 'ffdf'
dimorder(x, ...) <- value # just here to catch forbidden assignments
```
Arguments

- `x`: a ff object
- `value`: an appropriate integer vector
- `...`: further arguments (not used)

Details

dim and dimorder are virtual attributes. Thus two copies of an R ff object can point to the same file but interpret it differently. dim has the usual meaning, dimorder defines the dimension order of storage, i.e. `c(1,2)` corresponds to R’s standard column-major order, `c(1,2)` corresponds to row-major order, and for higher dimensional arrays dimorder can also be used. Standard dimorder is `1:length(dim(x))`. For `ffdf` dim returns the number of rows and virtual columns. With `dim<-`, `ffdf` only the number of rows can be changed. For convenience you can assign NA to the number of columns. For `ffdf` the dimorder returns non-standard dimorder if any of its columns contains a ff object with non-standard dimorder (see `dimorderStandard`) An even higher level of virtualization is available using virtual windows, see `vw`.

Value

- `names`: returns a character vector (or NULL)

Note

- `x[]` returns a matrix like `x[,]` and thus respects dimorder, while `x[i:j]` returns a vector and simply returns elements in the stored order. Check the corresponding example twice, in order to make sure you understand that for non-standard dimorder `x[1:length(x)]` is not the same as `as.vector(x[])`.

Author(s)

Jens Oehlschlägel

See Also

dim, dimnames.ff_array, dimorderStandard, vw, virtual

Examples

```r
x <- ff(1:12, dim=c(3,4), dimorder=c(2:1))
y <- x
dim(y) <- c(4,3)
dimorder(y) <- c(1:2)
x y
x[] y[]
x[,bydim=c(2,1)] y[,bydim=c(2,1)]
```
message("NOTE that x[,] like x[,] returns a matrix (respects dimorder),")
message("while x[1:12] returns a vector IN STORAGE ORDER")
message("check the following examples twice to make sure you understand this")
x[,]
x[]
as.vector(x[])
x[1:12]
rm(x,y); gc()

## Not run:
message("some performance comparison between different dimorders")
n <- 100
m <- 100000
a <- ff(1L,dim=c(n,m))
b <- ff(1L, dim=c(n,m), dimorder=2:1)
system.time(lapply(1:n, function(i)sum(a[i,])))
system.time(lapply(1:n, function(i)sum(b[i,])))
system.time(lapply(1:n, function(i){i<-(-i-1)*m/n+1; sum(a[,i:(i+m/n-1)])}))
system.time(lapply(1:n, function(i){i<-(-i-1)*m/n+1; sum(b[,i:(i+m/n-1)])}))
rm(a,b); gc()

## End(Not run)

dimnames.ffdf

### Getting and setting dimnames of ffdf

**Description**

Getting and setting dimnames, columnnames or rownames

**Usage**

```r
## S3 method for class 'ffdf'
dimnames(x)
## S3 replacement method for class 'ffdf'
dimnames(x) <- value
## S3 method for class 'ffdf'
names(x)
## S3 replacement method for class 'ffdf'
names(x) <- value
## S3 method for class 'ffdf'
row.names(x)
## S3 replacement method for class 'ffdf'
row.names(x) <- value
```

**Arguments**

- `x`: a `ffdf` object
- `value`: a character vector, or, for dimnames a list with two character vectors
Details

It is recommended not to assign row.names to a large ffdf object.

Value

The assignment function return the changed ffdf object. The other functions return the expected.

Author(s)

Jens Oehlschlägel

See Also

ffdf, dimnames.ff, rownames, colnames

Examples

```r
ffd <- as.ffdf(data.frame(a=1:26, b=letters))
dimnames(ffd)
row.names(ffd) <- letters
dimnames(ffd)
ffd
rm(ffd); gc()
```

---

**dimnames.ff_array**

**Getting and setting dimnames**

Description

For ff_arrays you can set dimnames.

Usage

```
## S3 method for class 'ff_array'
dimnames(x)
## S3 replacement method for class 'ff_array'
dimnames(x) <- value
```

Arguments

- **x** a ff array (or matrix)
- **value** a list with length(dim(x)) elements (either NULL of character vector of length of dimension

Details

if vw is set, dimnames.ff_array returns the appropriate part of the names, but you can’t set dimnames while vw is set. dimnames returns NULL for ff_vectors and setting dimnames for ff_vector is not allowed, but setting names is.
**Dimension order compatibility**

**Value**

dimnames returns a list, see \texttt{dimnames}

**Author(s)**

Jens Oehlschlägel

**See Also**

\texttt{dimnames}, \texttt{dim.ff}, \texttt{names.ff}, \texttt{vw}, \texttt{virtual}

**Examples**

```r
x <- ff(1:12, dim=c(3,4), dimnames=list(letters[1:3], LETTERS[1:4]))
dimnames(x)
dimnames(x) <- list(LETTERS[1:3], letters[1:4])
dimnames(x)
dimnames(x) <- NULL
dimnames(x)
rm(x); gc()
```

---

**dimorderCompatible**  
Test for dimorder compatibility

**Description**

dimorderStandard returns TRUE if the dimorder is standard (ascending), vectorStandard returns TRUE if the dimorder-bydim combination is compatible with a standard elementwise vector interpretation, dimorderCompatible returns TRUE if two dimorders have a compatible elementwise vector interpretation and vectorCompatible returns TRUE if dimorder-bydim combinations have a compatible elementwise vector interpretation.

**Usage**

dimorderStandard(dimorder)  
vectorStandard(dimorder, bydim = NULL)  
dimorderCompatible(dim, dim2, dimorder, dimorder2)  
vectorCompatible(dim, dim2, dimorder=NULL, dimorder2=NULL, bydim = NULL, bydim2 = NULL)

**Arguments**

- \texttt{dim} a \texttt{dim}
- \texttt{dim2} a \texttt{dim}
- \texttt{dimorder} a \texttt{dimorder}
- \texttt{dimorder2} a \texttt{dimorder}
- \texttt{bydim} a bydim order, see \texttt{[.ff}
- \texttt{bydim2} a bydim order, see argument from \texttt{dim} in \texttt{update.ff}
dummy.dinnames

Value
TRUE if compatibility has been detected, FALSE otherwise

Note
does not yet guarantee to detect all compatible configurations, but the most important ones

Author(s)
Jens Oehlschlägel

See Also
dimorder, ffconform

dummy.dinnames  Array: make dimnames

Description
makes standard dimnames from letters and integers (for testing)

Usage
dummy.dinnames(x)

Arguments
x  an array

Value
a list with character vectors suitable to be assigned as dimnames to x

Author(s)
Jens Oehlschlägel

See Also
dimnames

Examples
dummy.dinnames(matrix(1:12, 3, 4))
Description

These are the main methods for reading and writing data from ff files.

Usage

```r
## S3 method for class 'ff'
x[i, pack = FALSE]
## S3 replacement method for class 'ff'
x[i, add = FALSE, pack = FALSE] <- value
## S3 method for class 'ff_array'
x[..., bydim = NULL, drop = getOption("ffdrop"), pack = FALSE]
## S3 replacement method for class 'ff_array'
x[..., bydim = NULL, add = FALSE, pack = FALSE] <- value
## S3 method for class 'ff'
x[[i]]
## S3 replacement method for class 'ff'
x[[i, add = FALSE]] <- value
```

Arguments

- `x` an ff object
- `i` missing OR a single index expression OR a hi object
- `...` missing OR up to length(dim(x)) index expressions OR hi objects
- `drop` logical scalar indicating whether array dimensions shall be dropped
- `bydim` the dimorder which shall be used in interpreting vector to/from array data
- `pack` FALSE to prevent re-packing in hybrid index preprocessing. see `as.hi`
- `value` the values to be assigned, possibly recycled
- `add` TRUE if the values should rather increment than overwrite at the target positions, see `readwrite.ff`

Details

The single square bracket operators `[` and `[]<` are the workhorses for accessing the content of an ff object. They support `ff_vector` and `ff_array` access (`dim.ff`), they respect virtual windows (`vw`), `names.ff` and `dimnames.ff` and retain `ramclass` and `ramattrs` and thus support POSIXct and `factor`, see `levels.ff`

The functionality of `[` and `[]<` can be combined into one efficient operation, see `swap`

The double square bracket operator `[[` is a shortcut for `get.ff` resp. `set.ff`, however, you should not rely on this for the future, see `LIMWarn`. For programming please prefer `[.`
Value

The read operators [ and [[ return data from the ff object, possibly decorated with names, dim, dimnames and further attributes and classes (see ramclass, ramattrs).

The write operators <- and [[<- return the 'modified' ff object (like all assignment operators do).

Index expressions

\[ x <- ff(1:12, \text{dim} = c(3,4), \text{dimnames} = \text{list}(\text{letters}[1:3], \text{NULL})) \]

- **allowed expression**
  - positive integers \[ x[1,1] \]
  - negative integers \[ x[-(2:12)] \]
  - logical \[ x[\text{c}(\text{TRUE}, \text{FALSE}, \text{FALSE}),1] \]
  - character \[ x["a",1] \]
  - integer matrices \[ x[\text{rbind}(\text{c}(1,1))] \]
  - hybrid index \[ x[\text{hi},1] \]

- **disallowed expression**
  - zeros \[ x[0] \]
  - NAs \[ x[\text{NA}] \]

Dimorder and bydim

Arrays in R have always standard dimorder \(1:\text{length}(\text{dim}(x))\) while ff allows to store an array in a different dimorder. Using nonstandard dimorder (see dimorderStandard) can speed up certain access operations: while matrix dimorder=c(1,2) – column-major order – allows fast extraction of columns, dimorder=c(2,1) allows fast extraction of rows.

While the dimorder – being an attribute of an ff_array – controls how the vector in an ff file is interpreted, the bydim argument to the extractor functions controls, how assignment vector values in [<- are translated to the array and how the array is translated to a vector in [.subscripting. Note that bydim=c(2,1) corresponds to matrix(..., byrow=TRUE).

Multiple vector interpretation in arrays

In case of non-standard dimorder (see dimorderStandard) the vector sequence of array elements in R and in the ff file differs. To access array elements in file order, you can use getset.ff, readwrite.ff or copy the ff object and set \text{dim}(ff)<=NULL to get a vector view into the ff object (using [ dispatches the vector method [.ff]. To access the array elements in R standard dimorder you simply use [ which dispatches to [.ff_array. Note that in this case as.hi will unpack the complete index, see next section.

RAM expansion of index expressions

Some index expressions do not consume RAM due to the hi representation, for example 1:n will almost consume no RAM hoewever large n. However, some index expressions are expanded and require to \text{maxindex}(i) \times \text{.rambytes["integer"]} \text{bytes, either because the sorted sequence of index positions cannot be rle-packed efficiently or because hiparse cannot yet parse such expression and falls back to evaluating/expanding the index expression. If the index positions are not sorted,
the index will be expanded and a second vector is needed to store the information for re-ordering, thus the index requires \( 2 \times \maxindex(i) \times \text{rambytes["integer"]} \) bytes.

**RAM expansion when recycling assignment values**

Some assignment expressions do not consume RAM for recycling, for example \( x[1:n] \leftarrow 1:k \) will not consume RAM however large \( n \) compared to \( k \), when \( x \) has standard \texttt{dimorder}. However, if \( \text{length(value)} > 1 \), assignment expressions with non-ascending index positions trigger recycling the value R-side to the full index length. This will happen if \texttt{dimorder} does not match parameter \texttt{bydim} or if the index is not sorted ascending.

**Author(s)**

Jens Oehlschlägel

**See Also**

\texttt{ff, swap, add, readwrite.ff, LimWarn}

**Examples**

```r
message("look at different dimorders")
x <- ff(1:12, dim=c(3,4), dimorder=c(1,2))
x[]
as.vector(x[])
x[1:12]
x <- ff(1:12, dim=c(3,4), dimorder=c(2,1))
x[]
as.vector(x[])
message("Beware (might be changed")
x[1:12]

message("look at different bydim")
matrix(1:12, nrow=3, ncol=4, byrow=FALSE)
x <- ff(1:12, dim=c(3,4), bydim=c(1,2))
x
matrix(1:12, nrow=3, ncol=4, byrow=TRUE)
x <- ff(1:12, dim=c(3,4), bydim=c(2,1))
x
x[,, bydim=c(2,1)]
as.vector(x[,, bydim=c(2,1)])
message("even consistent interpretation of vectors in assignments")
x[,, bydim=c(1,2)] <- x[,, bydim=c(1,2)]
x
x[,, bydim=c(2,1)] <- x[,, bydim=c(2,1)]
x
rm(x); gc()
```

## Not run:

message("some performance implications of different dimorders")
n <- 100
m <- 100000
Extract.ffdf

Reading and writing data.frames (ffdf)

Description

These are the main methods for reading and writing data from ffdf objects.

Usage

```r
# S3 method for class 'ffdf'
x[i, j, drop = ncol == 1]
# S3 replacement method for class 'ffdf'
x[i, j] <- value
# S3 method for class 'ffdf'
x[i, j, exact = TRUE]
# S3 replacement method for class 'ffdf'
x[i, j] <- value
```
## S3 method for class 'ffdf'
x$i

## S3 replacement method for class 'ffdf'
x$i <- value

### Arguments

- **x**: an ff object
- **i**: a row subscript or a matrix subscript or a list subscript
- **j**: a column subscript
- **drop**: logical. If TRUE the result is coerced to the lowest possible dimension. The default is to drop if only one column is left, but not to drop if only one row is left.
- **value**: A suitable replacement value: it will be repeated a whole number of times if necessary and it may be coerced: see the Coercion section. If NULL, deletes the column if a single column is selected with [[<- or $<-.
- **exact**: logical: see [], and applies to column names.

### Details

The subscript methods [, [[ and $, behave symmetrical to the assignment functions [<-, [[<- and $<-. What the former return is the assignment value to the latter. A notable exception is assigning NULL in [[<- and $<- which removes the virtual column from the ffdf (and the physical component if it is no longer needed by any virtual column). Creating new columns via [[<- and $<- requires giving a name to the new column (character subscripting). [<- does not allow to create new columns, only to replace existing ones.

### Subscript expressions and return values

<table>
<thead>
<tr>
<th>allowed expression</th>
<th>example</th>
<th>returnvalue</th>
</tr>
</thead>
<tbody>
<tr>
<td>row selection</td>
<td>x[i, ]</td>
<td>data.frame or single row as list if drop=TRUE, like from data.frame</td>
</tr>
<tr>
<td>column selection</td>
<td>x[,i]</td>
<td>data.frame or single column as vector unless drop=TRUE, like from data.frame</td>
</tr>
<tr>
<td>matrix selection</td>
<td>x[cbind(i,j)]</td>
<td>vector of the integer-matrix indexed cells (if the column types are compatible)</td>
</tr>
<tr>
<td>virtual selection</td>
<td>x[i]</td>
<td>ffdf with the selected columns only</td>
</tr>
<tr>
<td>physical selection</td>
<td>x[[i]]</td>
<td>the selected ff</td>
</tr>
<tr>
<td>physical selection</td>
<td>x$i</td>
<td>the selected ff</td>
</tr>
</tbody>
</table>

### Author(s)

Jens Oehlschlägel

### See Also

ffdf, Extract.data.frame, Extract.ff
Examples

d <- data.frame(a=letters, b=rev(letters), c=1:26)
x <- as.ffdf(d)

d[1,]
x[1,]

d[1:2,]
x[1:2,]

d[,]1]
x[,]1]

d[,]1:2]
x[,]1:2]

d[cbind(1:2,2:1)]
x[cbind(1:2,2:1)]

d[1]
x[1]

d[,]1]
x[,]1]

d$a
x$a

d$a[1:2]
x$a[1:2]

rm(x); gc()

---

**ff**

**ff classes for representing (large) atomic data**

Description

The ff package provides atomic data structures that are stored on disk but behave (almost) as if they were in RAM by mapping only a section (pagesize) into main memory (the effective main memory consumption per ff object). Several access optimization techniques such as Hyrid Index Preprocessing (**as.hi, update.ff**) and Virtualization (**virtual, vt, vw**) are implemented to achieve good performance even with large datasets. In addition to the basic access functions, the ff package also provides compatibility functions that facilitate writing code for ff and ram objects (**clone, as.ff, as.ram**) and very basic support for operating on ff objects (**ffapply**). While the (possibly packed) raw data is stored on a flat file, meta informations about the atomic data structure such as its dimension, virtual storage mode (**vmode**), factor level encoding, internal length etc. are stored as an ordinary R object (external pointer plus attributes) and can be saved in the workspace. The raw flat file data encoding is always in native machine format for optimal performance and provides
several packing schemes for different data types such as logical, raw, integer and double (in an
extended version support for more tightly packed virtual data types is supported). flatfile data files
can be shared among ff objects in the same R process or even from different R processes due to
Memory-Mapping, although the caching effects have not been tested extensively.
Please do read and understand the limitations and warnings in LimWarn before you do anything
serious with package ff.

Usage

```r
ff( initdata = NULL,
    length = NULL,
    levels = NULL,
    ordered = NULL,
    dim = NULL,
    dimorder = NULL,
    bydim = NULL,
    symmetric = FALSE,
    fixdiag = NULL,
    names = NULL,
    dimnames = NULL,
    ramclass = NULL,
    ramattrs = NULL,
    vmode = NULL,
    update = NULL,
    pattern = NULL,
    filename = NULL,
    overwrite = FALSE,
    readonly = FALSE,
    pagesize = NULL, # getOption("ffpagesize")
    caching = NULL, # getOption("ffcaching")
    finalizer = NULL,
    finonexit = NULL, # getOption("fffinonexit")
    FF_RETURN = TRUE,
    BATCHSIZE = .Machine$integer.max,
    BATCHBYTES =getOption("ffbatchbytes")
    VERBOSE = FALSE
)
```

Arguments

- **initdata**: scalar or vector of the .vimplemented vmodes, recycled if needed, default 0, see also as.vmode and vector.vmode
- **length**: optional vector length of the object (default: derive from 'initdata' or 'dim'), see length.ff
- **levels**: optional character vector of levels if (in this case initdata must be composed of these) (default: derive from initdata)
- **ordered**: indicate whether the levels are ordered (TRUE) or non-ordered factor (FALSE, default)
optional array `dim`, see `dim_ff` and `array`  

physical layout (default `1:length(dim)`), see `dimorder` and `aperm`  

dimorder by which to interpret the `initdata`, generalization of the ‘byrow’ parameter in `matrix`  

extended feature: TRUE creates symmetric matrix (default FALSE)  

extended feature: non-NULL scalar requires fixed diagonal for symmetric matrix (default NULL is free diagonal)  

NOT taken from initdata, see `names`  

NOT taken from initdata, see `dimnames`  

class attribute attached when moving all or parts of this `ff` into ram, see `ramclass`  

additional attributes attached when moving all or parts of this `ff` into ram, see `ramattrs`  

virtual storage mode (default: derive from `initdata`), see `vmode` and `as.vmode`  

set to FALSE to avoid updating with `initdata` (default `TRUE`) (used by `ffdf`)  

root pattern with or without path for automatic `ff` filename creation (default `NULL` translates to "ff"), see also argument ‘filename’  

`ff` filename with or without path (default `tmppfile` with ‘pattern’ prefix); without path the file is created in `getOption("fftempdir")`, with path ‘.’ the file is created in `getwd`. Note that files created in `getOption("fftempdir")` have default finalizer "delete" while other files have default finalizer "close". See also arguments ‘pattern’ and ‘finalizer’ and `physical`  

set to TRUE to allow overwriting existing files (default `FALSE`)  

set to `TRUE` to forbid writing to existing files  

pagesize in bytes for the memory mapping (default from `getOption("ffpagesize")`) initialized by `getDefaultpagesize`), see also `physical`  

caching scheme for the backend, currently ‘mminoflush’ or ‘mmeachflush’ (flush mmpages at each swap, default from `getOption("ffcaching")` initialized with ‘mmeachflush’), see also `physical`  

name of finalizer function called when `ff` object is removed (default: `ff` files created in `getOption("fftempdir")` are considered temporary and have default finalizer `delete`, files created in other locations have default finalizer `close`); available finalizer generics are "close", "delete" and "deleteIfOpen", available methods are `close_ff`, `delete_ff` and `deleteIfOpen_ff`, see also argument ‘finonexit’ and `finalizer`  

logical scalar determining whether and `finalize` is also called when R is closed via `q`, (default `TRUE` from `getOption("fffinonexit")`)  

logical scalar or `ff` object to be used. The default `TRUE` creates a new `ff` file. `FALSE` returns a ram object. Handing over an `ff` object here uses this or stops if not `ffsuitable`  

integer scalar limiting the number of elements to be processed in `update_ff` when `length(initdata)>1`, default from `Machine$integer_max`  

integer scalar limiting the number of bytes to be processed in `update_ff` when `length(initdata)>1`, default from `getOption("ffbatchbytes")`, see also `rambytes`  

set to `TRUE` for verbosing in `update_ff` when `length(initdata)>1`, default `FALSE`
Details

The atomic data is stored in `filename` as a native encoded raw flat file on disk, OS specific limitations of the file system apply. The number of elements per `ff` object is limited to the integer indexing, i.e. `.Machine$integer.max`. Atomic objects created with `ff` are `is.open`, a C++ object is ready to access the file via memory-mapping. Currently the C++ backend provides two caching schemes: ‘mmnofflush’ let the OS decide when to flush memory mapped pages and ‘mmeachflush’ will flush memory mapped pages at each page swap per `ff` file. These minimal memory resources can be released by `closeing` or `deleteing` the `ff` file. `ff` objects can be `saved` and `loaded` across R sessions. If the `ff` file still exists in the same location, it will be `opened` automatically at the first attempt to access its data. If the `ff` object is `removed`, at the next garbage collection (see `gc`) the `ff` object’s `finalizer` is invoked. Raw data files can be made accessible as an `ff` object by explicitly given the filename and `vmode` but no size information (length or dim). The `ff` object will open the file and handle the data with respect to the given `vmode`. The `close` finalizer will close the `ff` file, the `delete` finalizer will delete the `ff` file. The default finalizer `deleteIfOpen` will delete open files and do nothing for closed files. If the default finalizer is used, two actions are needed to protect the `ff` file against deletion: create the file outside the standard ‘fftempdir’ and close the `ff` object before removing it or before quitting R. When R is exited through `q`, the finalizer will be invoked depending on the ‘fffinonexit’ option, furthermore the ‘fftempdir’ is `unlinked`.

Value

If (!FF\_RETURN) then a ram object like those generated by `vector`, `matrix`, `array` but with attributes ‘`vmode`’, ‘physical’ and ‘virtual’ accessible via `vmode`, `physical` and `virtual`
If (FF\_RETURN) an object of class ‘ff’ which is a a list with two components:

- **physical** an external pointer of class ‘ff\_pointer’ which carries attributes with copy by reference semantics: changing a physical attribute of a copy changes the original
- **virtual** an empty list which carries attributes with copy by value semantics: changing a virtual attribute of a copy does not change the original

Physical object component

The ‘ff\_pointer’ carries the following ‘physical’ or readonly attributes, which are accessible via `physical`:

- `vmode` see `vmode`
- `maxlength` see `maxlength`
- `pattern` see parameter ‘pattern’
- `filename` see `filename`
- `pagesize` see parameter ‘pagesize’
- `caching` see parameter ‘caching’
- `finalizer` see parameter ‘finalizer’
- `finonexit` see parameter ‘finonexit’
- `readonly` see `is.readonly`
- `class` The external pointer needs class ‘ff\_pointer’ to allow method dispatch of finalizers
Virtual object component

The ‘virtual’ component carries the following attributes (some of which might be NULL):

- Length   see length.ff
- Levels   see levels.ff
- Names    see names.ff
- VW       see vw.ff
- Dim      see dim.ff
- Dimorder see dimorder
- Symmetric see symmetric.ff
- Fixdiag  see fixdiag.ff
- ramclass see ramclass
- ramattrs see ramattrs

Class

You should not rely on the internal structure of ff objects or their ram versions. Instead use the accessor functions like vmode, physical and virtual. Still it would be wise to avoid attributes AND classes ’vmode’, ’physical’ and ’virtual’ in any other packages. Note that the ‘ff’ object’s class attribute also has copy-by-value semantics (’virtual’). For the ’ff’ object the following class attributes are known:

- vector c("ff_vector","ff")
- matrix c("ff_matrix","ff_array","ff")
- array c("ff_array","ff")
- symmetric matrix c("ff_symm","ff")
- distance matrix c("ff_dist","ff_symm","ff")
- reserved for future use c("ff_mixed","ff")

Methods

The following methods and functions are available for ff objects:

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Assign</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>function</td>
<td>ff</td>
<td></td>
<td>constructor for ff and ram objects</td>
</tr>
<tr>
<td>generic</td>
<td>update</td>
<td></td>
<td>updates one ff object with the content of another</td>
</tr>
<tr>
<td>generic</td>
<td>clone</td>
<td></td>
<td>clones an ff object optionally changing some of its features</td>
</tr>
<tr>
<td>method</td>
<td>print</td>
<td></td>
<td>print ff</td>
</tr>
<tr>
<td>method</td>
<td>str</td>
<td></td>
<td>ff object structure</td>
</tr>
<tr>
<td>function</td>
<td>isNff</td>
<td></td>
<td>check if inherits from ff</td>
</tr>
<tr>
<td>generic</td>
<td>asNff</td>
<td></td>
<td>coerce to ff, if not yet</td>
</tr>
<tr>
<td>generic</td>
<td>asNram</td>
<td></td>
<td>coerce to ram retaining some of the ff information</td>
</tr>
<tr>
<td>generic</td>
<td>asNbit</td>
<td></td>
<td>coerce to bit</td>
</tr>
<tr>
<td>generic</td>
<td>vmode</td>
<td>&lt;-</td>
<td>get and set virtual mode (setting only for ram, not for ff objects)</td>
</tr>
</tbody>
</table>
generic as.vmode <- coerce to vmode (only for ram, not for ff objects)

**Physical attributes**

function physical <- set and get physical attributes
generic filename <- get and set filename
generic pattern <- get pattern and set filename path and prefix via pattern
generic maxlength <- get maxlength
generic is.sorted <- set and get if is marked as sorted
generic na.count <- set and get NA count, if set to non-NA only swap methods can change and na.count is maintained automatically
generic is.readonly <- get if is readonly

**Virtual attributes**

function virtual <- set and get virtual attributes
method length <- set and get length
method dim <- set and get dim
generic dimorder <- set and get the order of dimension interpretation
generic vt <- virtually transpose ff_array
method t <- create transposed clone of ff_array
generic vw <- set and get virtual windows
method names <- set and get names
generic dimnames <- set and get dimnames
generic symmetric <- get if is symmetric
generic fixdiag <- set and get fixed diagonal of symmetric matrix
method levels <- levels of factor
generic recodeLevels <- recode a factor to different levels
generic sortLevels <- sort the levels and recode a factor
method is.factor <- if is factor
method is.ordered <- if is ordered (factor)
generic ramclass <- get ramclass
generic ramattrs <- get ramattrs

**Access functions**

function get.ff <- get single ff element (currently [ ] is a shortcut)
function set.ff <- set single ff element (currently [ ]<- is a shortcut)
function getset.ff <- set single ff element and get old value in one access operation
function read.ff <- get vector of contiguous elements
function write.ff <- set vector of contiguous elements
function readwrite.ff <- set vector of contiguous elements and get old values in one access operation
method [ <- get vector of indexed elements, uses HIP, see hi
method [<- <- set vector of indexed elements, uses HIP, see hi
generic swap <- set vector of indexed elements and get old values in one access operation
(galmost) unifies ‘+=’ operation for ff and ram objects
generic bigsample <- sample from ff object

**Opening/Closing/Deleting**

generic is.open <- check if ff is open
method open <- open ff object (is done automatically on access)
method close <- close ff object (releases C++ memory and protects against file deletion if deleteIfOpen)
generic delete <- deletes ff file (unconditionally)
generic deleteIfOpen <- deletes ff file if ff object is open (finalization method)
generic finalizer <- get and set finalizer
generic finalize <- force finalization
function `geterror.ff` get error code
function `geterrstr.ff` get error message

**ff options**

Through `options` or `getOption` one can change and query global features of the `ff` package:

<table>
<thead>
<tr>
<th>option</th>
<th>description</th>
<th>default</th>
</tr>
</thead>
<tbody>
<tr>
<td>fftempdir</td>
<td>default directory for creating <code>ff</code> files</td>
<td><code>tempdir</code></td>
</tr>
<tr>
<td>fffinalizer</td>
<td>name of default finalizer</td>
<td><code>deleteOnOpen</code></td>
</tr>
<tr>
<td>fffinonexit</td>
<td>default for invoking finalizer on exit of R</td>
<td><code>TRUE</code></td>
</tr>
<tr>
<td>ffpagesize</td>
<td>default pagesize</td>
<td><code>getdefaulPageSize</code></td>
</tr>
<tr>
<td>ff caching</td>
<td>caching scheme for the C++ backend</td>
<td><code>&quot;mmapflush&quot;</code></td>
</tr>
<tr>
<td>ffdrop</td>
<td>default for the ‘drop’ parameter in the <code>ff</code> subscript methods</td>
<td><code>TRUE</code></td>
</tr>
<tr>
<td>ffbatchbytes</td>
<td>default for the byte limit in batched/chunked processing</td>
<td><code>memory.limit() %/% 100</code></td>
</tr>
</tbody>
</table>

**OS specific**


<table>
<thead>
<tr>
<th>File System</th>
<th>File size limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>FAT16</td>
<td>2GB</td>
</tr>
<tr>
<td>FAT32</td>
<td>4GB</td>
</tr>
<tr>
<td>NTFS</td>
<td>16GB</td>
</tr>
<tr>
<td>ext2/3/4</td>
<td>16GB to 2TB</td>
</tr>
<tr>
<td>ReiserFS</td>
<td>4GB (up to version 3.4) / 8TB (from version 3.5)</td>
</tr>
<tr>
<td>XFS</td>
<td>8EB</td>
</tr>
<tr>
<td>JFS</td>
<td>4PB</td>
</tr>
<tr>
<td>HFS</td>
<td>2GB</td>
</tr>
<tr>
<td>HFS Plus</td>
<td>16GB</td>
</tr>
<tr>
<td>USF1</td>
<td>4GB to 256TB</td>
</tr>
<tr>
<td>USF2</td>
<td>512GB to 32PB</td>
</tr>
<tr>
<td>UDF</td>
<td>16EB</td>
</tr>
</tbody>
</table>

**Credits**

Package Version 1.0

Daniel Adler     <dadler@uni-goettingen.de>
R package design, C++ generic file vectors, Memory-Mapping, 64-bit Multi-Indexing adapter and Documentation

Oleg Nenadic     <onenadi@uni-goettingen.de>
Index sequence packing, Documentation

Walter Zucchini  <wzucchi@uni-goettingen.de>
Array Indexing, Sampling, Documentation
Christian Gläser  <christian\_glaeser@gmx.de>
Wrapper for biglm package

Package Version 2.0

Jens Oehlschlägel  <Jens.Oehlschlaegel@truecluster.com>
R package redesign; Hybrid Index Preprocessing; transparent object creation and finalization; vmode design

Daniel Adler  <dadler@uni-goettingen.de>
C++ generic file vectors, vmode implementation and low-level bit-packing/unpacking, arithmetic operations

Licence

Package under GPL-2, included C++ code released by Daniel Adler under the less restrictive ISCL

Note

Note that the standard finalizers are generic functions, their dispatch to the 'ff_pointer' method happens at finalization time, their 'ff' methods exist for direct calling.

See Also

vector, matrix, array, as.ff, as.ram

Examples

message("make sure you understand the following ff options
before you start using the ff package!")
oldoptions <- options(fffinalizer="deleteIfOpen", ffnonexit="TRUE", fftempdir=tempdir())
message("an integer vector")
  ff(1:12)
message("a double vector of length 12")
  ff(0, 12)
message("a 2-bit logical vector of length 12 (vmode='boolean' has 1 bit")
  ff(vmode="logical", length=12)
message("an integer matrix 3x4 (standard colwise physical layout")
  ff(1:12, dim=c(3,4))
message("an integer matrix 3x4 (rowwise physical layout, but filled in standard colwise order")
  ff(1:12, dim=c(3,4), dimorder=c(2,1))
message("an integer matrix 3x4 (standard colwise physical layout, but filled in rowwise order aka matrix(byrow=TRUE))")
  ff(1:12, dim=c(3,4), bydim=c(2,1))
gc()
options(oldoptions)

if (ffxtensions()){
  message("a 26-dimensional boolean array using 1-bit representation
  (File size 8 MB compared to 256 MB int in ram")
  a <- ff(vmode="boolean", dim=rep(2, 26))
  dimnames(a) <- dummy.dimnames(a)
### Not run:

message("This 2GB biglm example can take long, you might want to change
the size in order to define a size appropriate for your computer")
require(bigrm)

b <- 1000
n <- 100000
k <- 3
memory.size(max = TRUE)
system.time(
x <- ff(vmode="double", dim=c(b*n,k), dimnames=list(NULL, LETTERS[1:k]))
)
memory.size(max = TRUE)

```{r}
memory.size(max = TRUE)
```

```{r}
form <- A ~ B + C
first <- TRUE
system.time(

```{r}
ffrowapply(

```{r}
if (first){

```{r}
} else

```{r}
memory.size(max = TRUE)
```

```{r}
first
```{r}
fit
```{r}
summary(fit)
```{r}
rm(x); gc()
```

```
# End(Not run)
```

---

**ffapply**

*Apply for ff objects*
**Description**

The `ffapply` functions support convenient batched processing of ff objects such that each single batch or chunk will not exhaust RAM and such that batches have sizes as similar as possible, see `bbatch`. Differing from R’s standard `apply` which applies a FUNction, the `ffapply` functions do apply an EXPRESSION and provide two indices FROM="i1" and TO="i2", which mark beginning and end of the batch and can be used in the applied expression. Note that the `ffapply` functions change the two indices in their parent frame, to avoid conflicts you can use different names through FROM="i1" and TO="i2". For support of creating return values see details.

**Usage**

```r
ffvecapply(EXPR, X = NULL, N = NULL, VMODE = NULL, VBYTES = NULL, RETURN = FALSE,
            CFUN = NULL, USE_NAMES = TRUE, FF_RETURN = TRUE, BREAK = ".break",
            FROM = "i1", TO = "i2",
            BATCHSIZE = .Machine$integer.max, BATCHBYTES = getOption("ffbatchbytes")
            , VERBOSE = FALSE)
ffrowapply(EXPR, X = NULL, N = NULL, NCOL = NULL, VMODE = NULL, VBYTES = NULL,
            RETURN = FALSE, RETCOL = NCOL, CFUN = NULL, USE.NAME = TRUE, FF_RETURN = TRUE,
            FROM = "i1", TO = "i2",
            BATCHSIZE = .Machine$integer.max, BATCHBYTES = getOption("ffbatchbytes")
            , VERBOSE = FALSE)
ffcolapply(EXPR, X = NULL, N = NULL, NROW = NULL, VMODE = NULL, VBYTES = NULL,
            RETURN = FALSE, RETROW = NROW, CFUN = NULL, USE_NAME = TRUE, FF_RETURN = TRUE,
            FROM = "i1", TO = "i2",
            BATCHSIZE = .Machine$integer.max, BATCHBYTES = getOption("ffbatchbytes")
            , VERBOSE = FALSE)
ffapply(EXPR = NULL, AFUN = NULL, MARGIN = NULL, X = NULL, N = NULL, NCOL = NULL,
            VMODE = NULL, VBYTES = NULL, RETURN = FALSE, CFUN = NULL, USE_NAME = TRUE,
            FF_RETURN = TRUE, IDIM = "idim",
            FROM = "i1", TO = "i2", BREAK = ".break",
            BATCHSIZE = .Machine$integer.max, BATCHBYTES = getOption("ffbatchbytes")
            , VERBOSE = FALSE)
```

**Arguments**

- **EXPR**

  the expression to be applied

- **AFUN**

  `ffapply` only: alternatively to `EXPR` the name of a function to be applied, automatically converted to `EXPR`

- **MARGIN**

  `ffapply` only: the margins along which to loop in `ffapply`

- **X**

  an ff object from which several parameters can be derived, if they are not given directly: N, NCOL, NROW, DIM, VMODE, VBYTES, FF_RETURN

- **N**

  the total number of elements in the loop, e.g. number of elements in `ffvecapply` or number of rows in `ffrowapply`

- **NCOL**

  `ffrowapply` only: the number of columns needed to calculate batch sizes

- **NROW**

  `ffcolapply` only: the number of rows needed to calculate batch sizes

- **DIM**

  `ffapply` only: the dimension of the array needed to calculate batch sizes
VMODE  the \texttt{vmode} needed to prepare the RETURN object and to derive VBYTES if they are not given directly

VBYTES  the bytes per cell – see \texttt{rambytes} – to calculate the RAM requirements per cell

BATCHBYTES  the max number of bytes per batch, default \texttt{getOption("ffbatchbytes")}

BATCHSIZE  an additional restriction on the number of loop elements, default=\texttt{.Machine$integer.max}

FROM  the name of the index that marks the beginning of the batch, default 'i1', change if needed to avoid naming-conflicts in the calling frame

TO  the name of the index that marks the end of the batch, default 'i2', change if needed to avoid naming-conflicts in the calling frame

IDIM  \texttt{ffapply} only: the name of an R variable used for loop-switching, change if needed to avoid naming-conflicts in the calling frame

BREAK  \texttt{ffapply} only: the name of an R object in the calling frame that triggers break out of the batch loop, if 1) it exists 2) is.logical and 3) is TRUE

RETURN  TRUE to prepare a return value (default FALSE)

CFUN  name of a collapsing function, see CFUN

RETCOL  NULL gives return vector\([1:N]\), RETCOL gives return matrix\([1:N, 1:RETCOL]\)

RETROW  NULL gives return vector\([1:N]\), RETROW gives return matrix\([1:RETROW, 1:N]\)

FF\_RETURN  FALSE to return a ram object, TRUE to return an ff object, or an ff object that is \texttt{ffsuitable} to absorb the return data

USE\_NAMES  FALSE to suppress attaching names or dimnames to the result

VERBOSE  TRUE to verbose the batches

\textbf{Details}

\texttt{ffvecapply} is the simplest \texttt{ffapply} method for \texttt{ff\_vectors}. \texttt{ffrowapply} and \texttt{ffcolapply} is for \texttt{ff\_matrix}, and \texttt{ffapply} is the most general method for \texttt{ff\_arrays} and \texttt{ff\_vectors}.

There are many ways to change the return value of the \texttt{ffapply} functions. In its simplest usage – batched looping over an expression – they don’t return anything, see \texttt{invisible}. If you switch \texttt{RETURN=TRUE} in \texttt{ffvecapply} then it is assumed that all looped expressions together return one vector of length \(N\), and via parameter \texttt{FF\_RETURN}, you can decide whether this vector is in ram or an ff object (or even which ff object to use). \texttt{ffrowapply} and \texttt{ffcolapply} additionally have parameter \texttt{RETCOL} resp. \texttt{RETROW} which defaults to returning a matrix of the original size; in order to just return a vector of length \(N\) set this to NULL, or specify a number of columns/rows for the return matrix. It is assumed that the expression will return appropriate pieces for this return structure (see examples). If you specify \texttt{RETURN=TRUE} and a collapsing function name \texttt{CFUN}, then it is assumed that the batched expressions return aggregated information, which is first collected in a list, and finally the collapsing function is called on this list: \texttt{do.call(CFUN, list)}. If you want to return the unmodified list, you have to specify \texttt{CFUN="list"} for obvious reasons.

\texttt{ffapply} allows usages not completly unlike \texttt{apply}: you can specify the name of a function \texttt{AFUN} to be applied over \texttt{MARGIN}. However note that you must specify \texttt{RETURN=TRUE} in order to get a return value. Also note that currently \texttt{ffapply} assumes that your expression returns exactly one value per cell in \texttt{DIM[MARGINS]}. If you want to return something more complicated, you MUST specify a \texttt{CFUN="list"} and your return value will be a list with dim attribute \texttt{DIM[MARGINS]}. This means that for a function \texttt{AFUN} returning a scalar, \texttt{ffapply} behaves very similar to \texttt{apply}, see examples.
Note also that `ffapply` might create an object named `.ffapply.dimexhausted` in its parent frame, and it uses a variable in the parent frame for loop-switching between dimensions, the default name ‘idim’ can be changed using the IDIM parameter. Finally you can break out of the implied loops by assigning TRUE to a variable with the name in BREAK.

**Value**

see details

**Note**

xx The complete generation of the return value is preliminary and the arguments related to defining the return value might still change, especially `ffapply` is work in progress

**Author(s)**

Jens Oehlschlägel

**See Also**

`apply`, `expression`, `bbatch`, `refromto`, `ffsuitable`

**Examples**

```r
message("ffvecapply examples")
x <- ff(vmode="integer", length=1000)
message("loop evaluate expression without returning anything")
ffvecapply(x[i1:i2] <- i1:i2, X=x, VERBOSE=TRUE)
ffvecapply(x[i1:i2] <- i1:i2, X=x, BATCHSIZE=200, VERBOSE=TRUE)
ffvecapply(x[i1:i2] <- i1:i2, X=x, BATCHSIZE=199, VERBOSE=TRUE)
message("lets return the combined expressions as a new ff object")
ffvecapply(i1:i2, N=length(x), VMODE="integer", RETURN=TRUE, BATCHSIZE=200)
message("lets return the combined expressions as a new ram object")
ffvecapply(i1:i2, N=length(x), VMODE="integer", RETURN=TRUE, FF_RETURN=FALSE, BATCHSIZE=200)
message("lets return the combined expressions in existing ff object x")

x[] <- 0L
ffvecapply(i1:i2, N=length(x), VMODE="integer", RETURN=TRUE, FF_RETURN=x, BATCHSIZE=200)

x
message("aggregate and collapse")
ffvecapply(summary(x[i1:i2]), X=x, RETURN=TRUE, CFUN="list", BATCHSIZE=200)
ffvecapply(summary(x[i1:i2]), X=x, RETURN=TRUE, CFUN="cbind", BATCHSIZE=200)
ffvecapply(summary(x[i1:i2]), X=x, RETURN=TRUE, CFUN="cmean", BATCHSIZE=200)

message("how to do colSums with ffrwapply")
x <- ff(1:10000, vmode="integer", dim=c(1000, 10))
ffrrowapply(colSums(x[i1:i2,drop=FALSE]), X=x, RETURN=TRUE, CFUN="list", BATCHSIZE=200)
ffrrowapply(colSums(x[i1:i2,drop=FALSE]), X=x, RETURN=TRUE, CFUN="cbind", BATCHSIZE=200)
ffrrowapply(colSums(x[i1:i2,drop=FALSE]), X=x, RETURN=TRUE, CFUN="cmean", BATCHSIZE=200)

message("further ffrwapply examples")
x <- ff(1:10000, vmode="integer", dim=c(1000, 10))
message("loop evaluate expression without returning anything")
```
Description

`ffconform` returns position of ‘most’ conformable ff argument or zero if the arguments are not conforming.

Usage

`ffconform(..., vmode = NULL, fail = "stop")`

Arguments

`...` two or more ff objects

`vmode` handing over target vmode here supresses searching for a common vmode, see `maxffmode`

`fail` the name of a function to call if not-conforming, default `stop`
Details

A reference argument is defined to be the first argument with a `dim` attribute or the longest vector. The other arguments are then compared to the reference to check for conformity, which is violated if vmodes are not conforming or if the reference has not a multiple length of each other or if the dimensions do not match or if we have a dimorder conflict because not all arguments have the same `dimorderStandard`.

Value

the position of the most conforming argument or 0 (zero) if not conforming.

Note

xx Work in progress for package `R.ff`

Author(s)

Jens Oehlenschlägel

See Also

`ffsuitable`, `maxffmode`, `ymismatch`, `stop`, `warning`, `dimorderStandard`

Examples

```r
a <- ff(1:10)
b <- clone(a)
c <- ff(1:20)
d <- ff(1:21)
ffconform(a, b)
ffconform(c, a)
ffconform(a, c)
ffconform(c, a, b)

d1 <- ff(1:20, dim=c(2,10))
d2 <- ff(1:20, dim=c(10,2))
ffconform(c, d1)
ffconform(c, d2)
ffconform(d1, c)
ffconform(d2, c)
try(ffconform(d1, d2))
ffconform(d1, d1)

rm(a, b, c, d1, d2); gc()
```
Function `ffdf` creates ff data.frames stored on disk very similar to `data.frame`

### Usage

```r
ffdf(...
 , row.names = NULL
 , ff_split = NULL
 , ff_join = NULL
 , ff_args = NULL
 , update = TRUE
 , BATCHSIZE = .Machine$integer.max
 , BATCHBYTES = getOption("ffbatchbytes")
 , VERBOSE = FALSE)
```

### Arguments

- `...`: `ff` vectors or matrices (optionally wrapped in `I()` that shall be bound together to an `ffdf` object.
- `row.names`: A character vector. Not recommended for large objects with many rows.
- `ff_split`: A vector of character names or integer positions identifying input components to physically split into single `ff` vectors. If vector elements have names, these are used as root name for the new `ff` files.
- `ff_join`: A list of vectors with character names or integer positions identifying input components to physically join in the same `ff` matrix. If list elements have names, these are used to name the new `ff` files.
- `update`: By default (TRUE) new `ff` files are updated with content of input `ff` objects. Setting to FALSE prevents this update.
- `ff_args`: a list with further arguments passed to `ff` in case that new `ff` objects are created via `ff_split` or `ff_join`.
- `BATCHSIZE`: passed to `update.ff`
- `BATCHBYTES`: passed to `update.ff`
- `VERBOSE`: passed to `update.ff`

### Details

By default, creating an ‘`ffdf`’ object will NOT create new `ff` files, instead existing files are referenced. This differs from `data.frame`, which always creates copies of the input objects, most notably in `data.frame(matrix())`, where an input matrix is converted to single columns. `ffdf` by contrast, will store an input matrix physically as the same matrix and virtually map it to columns. Physically copying a large `ff` matrix to single `ff` vectors can be expensive. More generally, `ffdf`
objects have a **physical** and a **virtual** component, which allows very flexible dataframe designs: 
a physically stored matrix can be virtually mapped to single columns, a couple of physically stored vectors can be virtually mapped to a single matrix. The means to configure these are \texttt{I} for the virtual representation and the \texttt{`ff_split'} and \texttt{`ff_join'} arguments for the physical representation. An \texttt{ff} matrix wrapped into \texttt{`I()'} will return the input matrix as a single object, using \texttt{`ff_split'} will store this matrix as single vectors - and thus create new \texttt{ff} files. \texttt{`ff_join'} will copy a couple of input vectors into a unified new \texttt{ff} matrix with \texttt{dimorder=c(2,1)}, but virtually they will remain single columns. The returned \texttt{ffdf} object has also a \texttt{dimorder} attribute, which indicates whether the \texttt{ffdf} object contains a matrix with non-standard \texttt{dimorder c(2,1)}, see \texttt{dimorderStandard}.
Currently, virtual windows are not supported for \texttt{ffdf}.

**Value**

A list with components

- **physical** the underlying \texttt{ff} vectors and matrices, to be accessed via \texttt{physical}
- **virtual** the virtual features of the \texttt{ffdf} including the virtual-to-physical mapping, to be accessed via \texttt{virtual}
- **row.names** the optional row.names, see argument row.names

and class \texttt{`ffdf'} (NOTE that \texttt{ffdf} does not inherit from \texttt{ff})

**Methods**

The following methods and functions are available for \texttt{ffdf} objects:

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Assign</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>function</td>
<td>\texttt{ffdf}</td>
<td></td>
<td><strong>Basic functions</strong></td>
</tr>
<tr>
<td>generic</td>
<td>\texttt{update}</td>
<td></td>
<td>constructor for \texttt{ffdf} objects</td>
</tr>
<tr>
<td>generic</td>
<td>\texttt{clone}</td>
<td></td>
<td>updates one \texttt{ffdf} object with the content of another</td>
</tr>
<tr>
<td>method</td>
<td>\texttt{print}</td>
<td></td>
<td>\texttt{print} \texttt{ffdf}</td>
</tr>
<tr>
<td>method</td>
<td>\texttt{str}</td>
<td></td>
<td>\texttt{ffdf} object structure</td>
</tr>
<tr>
<td>function</td>
<td>\texttt{is.ffdf}</td>
<td></td>
<td><strong>Class test and coercion</strong></td>
</tr>
<tr>
<td>generic</td>
<td>\texttt{as.ffdf}</td>
<td></td>
<td>check if inherits from \texttt{ff}</td>
</tr>
<tr>
<td>generic</td>
<td>\texttt{as.data.frame}</td>
<td></td>
<td>coerce to \texttt{ff}, if not yet</td>
</tr>
<tr>
<td>generic</td>
<td>\texttt{vmode}</td>
<td></td>
<td>coerce to \texttt{ram.data.frame}</td>
</tr>
<tr>
<td>function</td>
<td>\texttt{physical}</td>
<td></td>
<td><strong>Virtual attributes</strong></td>
</tr>
<tr>
<td>method</td>
<td>\texttt{virtual}</td>
<td></td>
<td>get virtual attributes</td>
</tr>
<tr>
<td>method</td>
<td>\texttt{length}</td>
<td></td>
<td>get length</td>
</tr>
<tr>
<td>method</td>
<td>\texttt{dim} &lt;-</td>
<td></td>
<td>get \texttt{dim} and set \texttt{nrow}</td>
</tr>
<tr>
<td>generic</td>
<td>\texttt{dimorder}</td>
<td></td>
<td>get the \texttt{dimorder} (non-standard if any component is non-standard)</td>
</tr>
<tr>
<td>method</td>
<td>\texttt{names} &lt;-</td>
<td></td>
<td>set and get \texttt{names}</td>
</tr>
<tr>
<td>method</td>
<td>\texttt{row.names} &lt;-</td>
<td></td>
<td>set and get \texttt{row.names}</td>
</tr>
<tr>
<td>method</td>
<td>\texttt{dimnames} &lt;-</td>
<td></td>
<td>set and get \texttt{dimnames}</td>
</tr>
</tbody>
</table>
method pattern <- set pattern (rename/move files)

Access functions
method [ <- set and get data.frame content ([,]) or get ffdf with less columns ([])
method [[ <- set and get single column ff object
method $ <- set and get single column ff object

Opening/Closing/Deleting
generic is.open tri-bool is.open status of the physical ff components
method open open all physical ff objects (is done automatically on access)
method close close all physical ff objects
method delete deletes all physical ff files
method finalize call finalizer

processing
method chunk create chunked index
method sortLevels sort and recode levels

Other

Note
Note that in theory, accessing a chunk of rows from a matrix with dimorder=c(2,1) should be faster than accessing across a bunch of vectors. However, at least under windows, the OS has difficulties filecaching parts from very large files, therefore - until we have partitioning - the recommended physical storage is in single vectors.

Author(s)
Jens Oehlschlägel

See Also
data.frame, ff, for more example see physical

Examples
m <- matrix(1:12, 3, 4, dimnames=list(c("r1","r2","r3"), c("m1","m2","m3","m4")))
v <- 1:3
ffm <- as.ff(m)
ffv <- as.ff(v)

d <- data.frame(m, v)
ffd <- ffdf(ffm, v=ffv, row.names=row.names(ffm))
all.equal(d, ffd[,])
ffd
physical(ffd)

d <- data.frame(m, v)
ffd <- ffdf(ffm, v=ffv, row.names=row.names(ffm), ff_split=1)
all.equal(d, ffd[,])
ffd
physical(ffd)
Function `ffdfindexget` allows to extract rows from an `ffdf` data.frame according to positive integer subscript stored in an `ff` vector.

Function `ffdfindexset` allows the inverse operation: assigning to rows of an `ffdf` data.frame according to positive integer subscripts stored in an `ff` vector. These functions allow more control than the method dispatch of `[` and `[-` if an `ff` integer subscript is used.

**Usage**

```r
ffdfindexget(x, index, indexorder = NULL, autoindexorder = 3, FF_RETURN = NULL, BATCHSIZE = NULL, BATCHBYTES = getOption("ffmaxbytes"), VERBOSE = FALSE)
ffdfindexset(x, index, value, indexorder = NULL, autoindexorder = 3, BATCHSIZE = NULL, BATCHBYTES = getOption("ffmaxbytes"), VERBOSE = FALSE)
```

**Arguments**

- `x` A `ffdf` data.frame containing the elements  
- `index` A `ff` integer vector with integer subscripts in the range from 1 to `length(x)`.  
- `value` A `ffdf` data.frame like `x` with the rows to be assigned  
- `indexorder` Optionally the return value of `ffindexorder`, see details  
- `autoindexorder` The minimum number of columns (which need chunked indexordering) for which we switch from on-the-fly ordering to stored `ffindexorder`  
- `FF_RETURN` Optionally an `ffdf` data.frame of the same type as `x` in which the returned values shall be stored, see details.  
- `BATCHSIZE` Optimal limit for the batchsize (see details)  
- `BATCHBYTES` Limit for the number of bytes per batch  
- `VERBOSE` Logical scalar for verbosing
ffdfsort

Details

Accessing rows of an ffdf data.frame identified by integer positions in an ff vector is a non-trivial task, because it could easily lead to random-access to disk files. We avoid random access by loading batches of the subscript values into RAM, order them ascending, and only then access the ff values on disk. Such ordering is done on-the-fly for up to autoindexorder-1 columns that need ordering. For autoindexorder o more columns we do the batched ordering upfront with ffindexorder and then re-use it in each call to ffindexget resp. ffindexset.

Value

Function ffdfindexget returns a ffdf data.frame with those rows selected by the ff index vector. Function ffdfindexset returns x with those rows replaced that had been requested by index and value.

Author(s)

Jens Oehlschlägel

See Also

Extract.ff, ffindexget, ffindexorder

Examples

message("ff integer subscripts with ffdf return/assign values")
x <- ff(factor(letters))
y <- ff(1:26)
d <- ffdf(x,y)
i <- ff(2:9)
di <- d[i,]
di
d[i,] <- di
message("ff integer subscripts: more control with ffindexget/ffindexset")
di <- ffdfindexget(d, i, FF_RETURN=di)
d <- ffdfindexset(d, i, di)
rm(x, y, i, di)
gc()

---

ffdfsort

Sorting: convenience wrappers for data.frames

Description

These functions allow convenient sorting and ordering of collections of (ff) vectors organized in (ffdf) data.frames
Usage

\texttt{dforder(x, \ldots)}
\texttt{dfsor\texttt{t}(x, \ldots)}
\texttt{ramdforder(x, \ldots)}
\texttt{ramdfsor\texttt{t}(x, \ldots)}
\texttt{ffdforder(x, \ldots)}
\texttt{ffdfsor\texttt{t}(x, \ldots)}

Arguments

\texttt{x} \hspace{1cm} \texttt{a data.frame} (for \texttt{dforder}, \texttt{dfsor\texttt{t}}, \texttt{ramdforder}, \texttt{ramdfsor\texttt{t}}) or \texttt{an fdf} object (for \texttt{ffdforder}, \texttt{ffdfsor\texttt{t}})
\texttt{\ldots} \hspace{1cm} \texttt{further arguments passed to sort, ramsor\texttt{t} or ffsor\texttt{t}} (for objects with one column) or passed to \texttt{order, ramorder or fforder} (for objects with multiple columns)

Value

the order functions return an (ff) vector of integer order positions, the sort functions return a sorted clone of the (ffdf) input data.frame

Author(s)

Jens Oehlschlägel

See Also

\texttt{sort, ramsor\texttt{t} or ffsor\texttt{t}}
\texttt{order, ramorder or fforder}

Examples

\begin{verbatim}
x <- ff(sample(1e5, 1e6, TRUE))
y <- ff(sample(1e5, 1e6, TRUE))
z <- ff(sample(1e5, 1e6, TRUE))
d <- ffdf(x, y, z)
d2 <- ffdfsor\texttt{t}(d)
d2
d
d2 <- d[1:2]
i <- ffdforder(d2)
d[i,]
rm(x, y, z, i, d, d2)
gc()
\end{verbatim}
**ffdrop**

*Delete an ffarchive*

**Description**

Delete the `<file>`.Rdata and `<file>.ffData` files behind an ffarchive.

**Usage**

`ffdrop(file)`

**Arguments**

- `file` vector of archive filenames (without extensions)

**Value**

A list with components

- `RData` vector with results of `file.remove` on RData files
- `ffData` Description of 'comp2'

**Note**

This deletes file on disk without warning.

**Author(s)**

Jens Oehlschlägel

**See Also**

`ffsave, ffinfo, ffload`

---

**ffindexget**

*Reading and writing ff vectors using ff subscripts*

**Description**

Function `ffindexget` allows to extract elements from an ff vector according to positive integer subscripts stored in an ff vector. Function `ffindexset` allows the inverse operation: assigning to elements of an ff vector according to positive integer subscripts stored in an ff vector. These functions allow more control than the method dispatch of `[` and `<-` if an ff integer subscript is used.
Usage

```R
ffindexget(x, index, indexorder = NULL, FF_RETURN = NULL
  , BATCHSIZE = NULL, BATCHBYTES = getOption("ffmaxbytes"), VERBOSE = FALSE)
ffindexset(x, index, value, indexorder = NULL
  , BATCHSIZE = NULL, BATCHBYTES = getOption("ffmaxbytes"), VERBOSE = FALSE)
```

Arguments

- `x`: A `ff` vector containing the elements.
- `index`: A `ff` integer vector with integer subscripts in the range from 1 to `length(x)`.  
- `value`: An `ff` vector of the same `vmode` as `x` containing the values to be assigned.  
- `indexorder`: Optionally the return value of `ffindexorder`, see details.  
- `FF_RETURN`: Optionally an `ff` vector of the same `vmode` as `x` in which the returned values shall be stored, see details.  
- `BATCHSIZE`: Optional limit for the batchsize (see details).  
- `BATCHBYTES`: Limit for the number of bytes per batch.  
- `VERBOSE`: Logical scalar for verbosing.

Details

Accessing integer positions in an `ff` vector is a non-trivial task, because it could easily lead to random-access to a disk file. We avoid random access by loading batches of the subscript values into RAM, order them ascending, and only then access the `ff` values on disk. Since ordering is expensive, it may pay to do the batched ordering once upfront and then re-use it with `ffindexorder`, similar to storing and using hybrid index information with `as.hi`.

Value

- Function `ffindexget` returns an `ff` vector with the extracted elements.  
- Function `ffindexset` returns the `ff` vector in which we have updated values.

Author(s)

Jens Oehlenschlägel

See Also

- `Extract.ff`, `ffdfindexget`, `ffindexorder`  

Examples

```R
message("ff integer subscripts with ff return/assign values")
x <- ff(factor(letters))
i <- ff(2:9)
xi <- x[1]
xi
xi[] <- NA
xi
```
ffindexorder

Description

Function `ffindexorder` will calculate chunkwise the order positions to sort all positions in a chunk ascending.

Function `ffindexordersize` does the calculation of the chunksize for `ffindexorder`.

Usage

```r
ffindexordersize(length, vmode, BATCHBYTES = getOption("ffmaxbytes"))
ffindexorder(index, BATCHSIZE, FF_RETURN = NULL, VERBOSE = FALSE)
```

Arguments

- `index` A *ff* integer vector with integer subscripts.
- `BATCHSIZE` Limit for the chunksize (see details)
- `BATCHBYTES` Limit for the number of bytes per batch
- `FF_RETURN` Optionally an *ff* integer vector in which the chunkwise order positions are stored.
- `VERBOSE` Logical scalar for activating verbosing.
- `length` Number of elements in the index
- `vmode` The `vmode` of the *ff* vector to which the index shall be applied with `ffindexget` or `ffindexset`

Details

Accessing integer positions in an *ff* vector is a non-trivial task, because it could easily lead to random-access to a disk file. We avoid random access by loading batches of the subscript values into RAM, order them ascending, and only then access the *ff* values on disk. Such an ordering can be done on-the-fly by `ffindexget` or it can be created upfront with `ffindexorder`, stored and re-used, similar to storing and using hybrid index information with `as.hi`.

Value

Function `ffindexorder` returns an *ff* integer vector with an attribute `BATCHSIZE` (the chunksize finally used, not the one given with argument `BATCHSIZE`).

Function `ffindexordersize` returns a balanced batchsize as returned from `bbatch`. 

```r
x[i] <- xi
x
message("ff integer subscripts: more control with ffindexget/ffindexset")
xi <- ffindexget(x, i, FF_RETURN=xi)
x <- ffindexset(x, i, xi)
rm(x, i, xi)
gc()
```
**Author(s)**

Jens Oehlschlägel

**See Also**

`ffindexget`, `as.hi`, `bbatch`

**Examples**

```r
x <- ff(sample(40))
messagex("fforder requires sorting")
1 <- fforder(x)
messagex("applying this order 1 is done by ffindexget")
x[i]
messagex("applying this order 1 requires random access,
therefore ffindexget does chunkwise sorting")
ffindexget(x, i)
messagex("if we want to apply the order i multiple times,
we can do the chunkwise sorting once and store it")
s <- ffindexordersize(length(i), vmode(i), BATCHBYTES = 100)
o <- ffindexorder(i, s$#)
messagex("this is how the stored chunkwise sorting is used")
ffindexget(x, i, o)
messagx("")
rm(x, i, o)
gc()
```

---

**ffinfo**

*Inspect content of ff saves*

**Description**

Find out which objects and ff files are in a pair of files saved with `ffsave`

**Usage**

`ffinfo(file)`

**Arguments**

- `file` a character string giving the name (without extension) of the `.RData` and `.ffData` files to load
### Value

a list with components

- **RData**: a list, one element for each object (named like the object): a character vector with the names of the ff files
- **ffData**: a list, one element for each path (names like the path): a character vector with the names of the ff files
- **rootpath**: the root path relative to which the files are stored in the .ffData zip

### Note

For large files and the zip64 format use `zip 3.0` and `unzip 6.0` from [http://www.info-zip.org/](http://www.info-zip.org/).

### Author(s)

Jens Oehlschlägel

### See Also

`ffsave`, `ffload`, `ffdrop`

---

**ffload**

* Reload ffSaved Datasets

### Description

Reload datasets written with the function `ffsave` or `ffsave.image`.

### Usage

```r
ffload(file, list = character(0L), envir = parent.frame(),
       rootpath = NULL, overwrite = FALSE)
```

### Arguments

- **file**: a character string giving the name (without extension) of the `.RData` and `.ffData` files to load
- **list**: An optional vector of names selecting those objects to be restored (default `NULL` restores all)
- **envir**: the environment where the data should be loaded.
- **rootpath**: an optional rootpath where to restore the ff files (default `NULL` restores in the original location)
- **overwrite**: logical indicating whether possibly existing ff files shall be overwritten
Details

`ffinfo` can be used to inspect the contents an `ffsave`ed pair of `.RData` and `.ffData` files. Argument list can then be used to restore only part of the `ffsave`.

Value

A character vector with the names of the restored ff files

Note

The ff files are not platform-independent with regard to byte order. For large files and the zip64 format use `zip 3.0` and `unzip 6.0` from http://www.info-zip.org/.

Author(s)

Jens Oehlschlägel

See Also

`load`, `ffsave`, `ffinfo`, `ffdrop`

---

**fforder** Sorting: order from ff vectors

**Description**

Returns order with regard to one or more ff vectors

**Usage**

```r
fforder(...
  , index = NULL
  , use.index = NULL
  , aux = NULL
  , auxindex = NULL
  , has.na = TRUE
  , na.last = TRUE
  , decreasing = FALSE
  , BATCHBYTES =getOption("ffmaxbytes")
  , VERBOSE = FALSE
)
```
fforder

Arguments

- ... one of more ff vectors which define the order
- index an optional ff integer vector used to store the order output
- use.index A boolean flag telling fforder whether to use the positions in 'index' as input. If you do this, it is your responsibility to assure legal positions - otherwise you risk a crash.
- aux An optional named list of ff vectors that can be used for temporary copying – the names of the list identify the vmodes for which the respective ff vector is suitable.
- auxindex An optional ff integer vector for temporary storage of integer positions.
- has.na boolean scalar telling fforder whether the vector might contain NAs. Note that you risk a crash if there are unexpected NAs with has.na=FALSE
- na.last boolean scalar telling fforder whether to order NAs last or first. Note that 'boolean' means that there is no third option NA as in order
- decreasing boolean scalar telling fforder whether to order increasing or decreasing
- BATCHBYTES maximum number of RAM bytes fforder should try not to exceed
- VERBOSE cat some info about the ordering

Details

fforder tries to order the vector in-RAM, if not possible it uses (a yet simple) out-of-memory algorithm. Like ramorder the in-RAM ordering method is chosen depending on context information.

Value

An ff vector with the positions that are required to sort the input as specified – with an attribute na.count with as many values as columns in ...

Author(s)

Jens Oehlschlälé

See Also

ramorder, ffsort, ffdorder, ffindexget

Examples

```r
x <- ff(sample(1e5, 1e6, TRUE))
y <- ff(sample(1e5, 1e6, TRUE))
d <- ffdf(x, y)

i <- fforder(y)
y[i]
i <- fforder(x, index=i)
x[i]
d[i,]
```
i <- fforder(x, y)
d[i,]

i <- ffdforder(d)
d[i,]

rm(x, y, d, i)
gc()

---

**ffreturn**

*Return suitable ff object*

**Description**

*ffreturn* returns *FF_RETURN* if it is *ffsuitable* otherwise creates a suitable *ffsuitable* object

**Usage**

`ffreturn(FF_RETURN = NULL, FF_PROTO = NULL, FF_ATTR = NULL)`

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FF_RETURN</strong></td>
<td>the object to be tested for suitability</td>
</tr>
<tr>
<td><strong>FF_PROTO</strong></td>
<td>the prototype object which <strong>FF_RETURN</strong> should match</td>
</tr>
<tr>
<td><strong>FF_ATTR</strong></td>
<td>a list of additional attributes dominating those from <strong>FF_PROTO</strong></td>
</tr>
</tbody>
</table>

**Value**

a suitable *ffsuitable* object

**Note**

xx Work in progress for package *R.ff*

**Author(s)**

Jens Oehlschlägel

**See Also**

*ffconform, ffsuitable*
ffsave

Save R and ff objects

Description

ffsave writes an external representation of R and ff objects to an ffarchive. The objects can be read back from the file at a later date by using the function ffload.

Usage

ffsave(...
  , list = character(0L)
  , file = stop("'file' must be specified")
  , envir = parent.frame()
  , rootpath = NULL
  , add = FALSE
  , move = FALSE
  , compress = !move
  , compression_level = 6
  , precheck=TRUE
)
ffsave.image(file = stop("'file' must be specified"), safe = TRUE, ...)

Arguments

...        For ffsave the names of the objects to be saved (as symbols or character strings),
list       for ffsave.image further arguments passed to ffsave
file       A character vector containing the names of objects to be saved.
envir      A name for the the ffarchive, i.e. the two files <file>.RData and <file>.ffData
add        environment to search for objects to be saved.
move       logical indicating whether the objects shall be added to the ffarchive (in this
compress    case rootpath is taken from an existing archive)
compression_level logical indicating whether ff files shall be moved instead of copied into the
rootpath    <file>.ffData
precheck    logical specifying whether saving to a named file is to use compression.
safe        compression level passed to zip, default 6
            optional path component that all all ff files share and that can be dropped/replaced
            when calling ffload
            logical: should the existence of the objects be checked before starting to save
            (and in particular before opening the file/connection)?
safe        logical. If TRUE, a temporary file is used for creating the saved workspace. The
            temporary file is renamed to <file>.ffData if the save succeeds. This preserves an existing
            workspace <file>.ffData if the save fails, but at the cost of using extra disk space during the save.
Details

`ffsave` stores objects and ff files in an ffarchive named `<file>`: i.e. it saves all specified objects via `save` in a file named `<file>.RData` and saves all ff files related to these objects in a zipfile named `<file>.ffData` using an external zip utility.

By default files are stored relative to the rootpath="" and will be restored relative to `\code{"" (in its original location). By providing a partial path prefix via argument `rootpath` the files are stored relative to this rootpath. The rootpath is stored in the `<file>.RData` with the name `.ff.rootpath`. I.e. even if the ff objects were saved with argument `rootpath` to `ffsave`, `ffload` by default restores in the original location. By using argument `rootpath` to `ffload` you can restore relative to a different rootpath (and using argument `rootpath` to `ffsave` gave you shorter relative paths).

By using argument `add` in `ffsave` you can add more objects to an existing ffarchive and by using argument `list` in `ffload` you can selectively restore objects.

The content of the ffarchive can be inspected using `ffinfo` before actually loading any of the objects.

The ffarchive can be deleted from disk using `ffdrop`.

Value

a character vector with messages returned from the zip utility (one for each ff file zipped)

Note

The ff files are not platform-independent with regard to byte order. For large files and the zip64 format use `zip 3.0` and `unzip 6.0` from `http://www.info-zip.org/`.

Author(s)

Jens Oehlschlägel

See Also

`ffinfo` for inspecting the content of the ffarchive

`ffload` for loading all or some of the ffarchive

`ffdrop` for deleting one or more ffarchives

Examples

```r
## Not run:
message("let's create some ff objects")
n <- 8e3
a <- ff(sample(n, n, TRUE), vmode="integer", length=n, filename="d:/tmp/a.ff")
b <- ff(sample(255, n, TRUE), vmode="ubyte", length=n, filename="d:/tmp/b.ff")
x <- ff(sample(255, n, TRUE), vmode="ubyte", length=n, filename="d:/tmp/x.ff")
y <- ff(sample(255, n, TRUE), vmode="ubyte", length=n, filename="d:/tmp/y.ff")
z <- ff(sample(255, n, TRUE), vmode="ubyte", length=n, filename="d:/tmp/z.ff")
df <- ffdf(x=x, y=y, z=z)
rm(x,y,z)
message("save all of them")
```
ffsave.image("d:/tmp/x")  
str(ffinfo("d:/tmp/x"))

message("save some of them with shorter relative pathnames ...")  
ffsave(a, b, file="d:/tmp/y", rootpath="d:/tmp")  
str(ffinfo("d:/tmp/y"))

message("... and add others later")  
ffsave(df, add=TRUE, file="d:/tmp/y", rootpath="d:/tmp")  
str(ffinfo("d:/tmp/y"))

message("... and add others later")  
system.time(ffsave(a, file="d:/tmp/z", move=TRUE))  
ffinfo("d:/tmp/z")

message("let’s delete/close/remove all objects")  
close(a)  # no file anymore, since we moved a into the ffarchive  
delete(b, df)  
rm(df, a, b, n)  
message("prove it")  
ls()

message("restore all but ff files in a different directory")  
ls()  
system.time(ffload("d:/tmp/x", rootpath="d:/tmp2"))  
lapply(ls(), function(i)filename(get(i)))

delete(a, b, df)  
rm(df, a, b)

ffdrop(c("d:/tmp/x", "d:/tmp/y", "d:/tmp/z"))

## End(Not run)

### ffsort

**Sorting of ff vectors**

**Description**

Sorting: sort an ff vector – optionally in-place

**Usage**

```r
ffsort(x  
  , aux = NULL  
  , has.na = TRUE  
  , na.last = TRUE  
  , decreasing = FALSE  
  , inplace = FALSE  
  , decorate = FALSE  
  , BATCHBYTES = getOption("ffmaxbytes")
```
Arguments

- **x**: an ff vector
- **aux**: NULL or an ff vector of the same type for temporary storage
- **has.na**: boolean scalar telling ffsort whether the vector might contain NAs. *Note* that you risk a crash if there are unexpected NAs with has.na=FALSE
- **na.last**: boolean scalar telling ffsort whether to sort NAs last or first. *Note* that 'boolean' means that there is no third option NA as in `sort`
- **decreasing**: boolean scalar telling ffsort whether to sort increasing or decreasing
- **inplace**: boolean scalar telling ffsort whether to sort the original ff vector (TRUE) or to create a sorted copy (FALSE, the default)
- **decorate**: boolean scalar telling ffsort whether to decorate the returned ff vector with `is.sorted` and `na.count` attributes.
- **BATCHBYTES**: maximum number of RAM bytes ffsort should try not to exceed
- **VERBOSE**: cat some info about the sorting

Details

ffsort tries to sort the vector in-RAM respecting the BATCHBYTES limit. If a fast sort is not possible, it uses a slower in-place sort (shellsort). If in-RAM is not possible, it uses (a yet simple) out-of-memory algorithm. Like `ramsort` the in-RAM sorting method is chosen depending on context information. If a key-index sort can be used, ffsort completely avoids merging disk based subsorts. If argument `decorate`=TRUE is used, then `na.count(x)` will return the number of NAs and `is.sorted(x)` will return TRUE if the sort was done with `na.last=TRUE` and `decreasing=FALSE`.

Value

An ff vector – optionally decorated with `is.sorted` and `na.count`, see argument 'decorate'

Note

the ff vector may not have a names attribute

Author(s)

Jens Oehlschlägel

See Also

`ramsort`, `fforder`, `ffdfsord`
ffsuitable  

Examples

n <- 1e6
x <- ff(c(NA, 999999:1), vmode="double", length=n)
x <- ffsort(x)
x
is.sorted(x)
na.count(x)
x <- ffsort(x, decorate=TRUE)
is.sorted(x)
na.count(x)
x <- ffsort(x, BATCHBYTES=n, VERBOSE=TRUE)

ffsuitable  Test ff object for suitability

Description

ffsuitable tests whether FF_RETURN is an ff object like FFPROTO and having attributes FF_ATTR.

Usage

ffsuitable(FF_RETURN, FF_PROTO = NULL, FF_ATTR = list()
, strict.dimorder = TRUE, fail = "warning")
ffsuitable_attribs(x)

Arguments

x an object from which to extract attributes for comparison
FF_RETURN the object to be tested for suitability
FF_PROTO the prototype object which FF_RETURN should match
FF_ATTR a list of additional attributes dominating those from FF_PROTO
strict.dimorder if TRUE ffsuitability requires that the dimorders are standard (ascending)
fail name of a function to be called if not ffsuitable (default warning)

Value

TRUE if FF_RETURN object is suitable, FALSE otherwise

Note

xx Work in progress for package R.ff

Author(s)

Jens Oehlschlägel
ffxtensions

Test for availability of ff extensions

Description

checks if this version of package ff supports ff extensions.

Usage

ffxtensions()
ffsymmxtensions()

Details

ff extensions are needed for certain bitcompressed vmodes and ff symm extensions for symmetric matrices.

Value

logical scalar

Author(s)

Jens Oehlschlägel

See Also

ffconform, ffreturn

ffmode

Examples

ffxtensions()
ffsymmxtensions()
file.resize  

Change size of an existing file

Description

Change size of an existing file (on some platforms sparse files are used) or move file to other name and/or location.

Usage

```plaintext
file.resize(path, size)
file.move(from, to)
```

Arguments

- path: file path (on windows it uses a 'windows' backslash path!)
- size: new filesize in bytes as double
- from: old file path
- to: new file path

Details

file.resize can enlarge or shrink the file. When enlarged, the file is filled up with zeros. Some platform implementations feature sparse files, so that this operation is very fast. We have tested:

- Ubuntu Linux 8, i386
- FreeBSD 7, i386
- Gentoo Linux Virtual-Server, i386
- Gentoo Linux, x86_64
- Windows XP

The following work but do not support sparse files

- Mac OS X 10.5, i386
- Mac OS X 10.4, PPC

file.move tries to file.rename, if this fails (e.g. across file systems) the file is copied to the new location and the old file is removed, see file.copy and file.remove.

Value

logical scalar representing the success of this operation

Author(s)

Daniel Adler
See Also

`file.create, file.rename, file.info, file.copy, file.remove`

Examples

```r
x <- tempfile()
newsize <- 23 # resize and size to 23 bytes.
file.resize(x, newsize)
file.info(x)$size == newsize
## Not run:
newsize <- 8*(2^30) # create new file and size to 8 GB.
file.resize(x, newsize)
file.info(x)$size == newsize

## End(Not run)
y <- tempfile()
file.move(x, y)
file.remove(y)
```

### filename

**Get or set filename**

Get or set filename from ram or `ff` object via the `filename` and `filename<-` generics or rename all files behind a `ffdf` using the `pattern<-` generic.

#### Usage

```r
filename(x, ...)
filename(x, ...) <- value
## Default S3 method:
filename(x, ...)
## S3 method for class 'ff_pointer'
filename(x, ...)
## S3 method for class 'ffdf'
filename(x, ...)
## S3 replacement method for class 'ff'
filename(x, ...) <- value
pattern(x, ...)
pattern(x, ...) <- value
## S3 method for class 'ff'
pattern(x, ...)
## S3 replacement method for class 'ff'
pattern(x, ...) <- value
## S3 replacement method for class 'ffdf'
pattern(x, ...) <- value
```
Arguments

- `x`: a ram or ff object, or for pattern assignment only - a ffdf object
- `value`: a new filename
- `...`: dummy to keep R CMD CHECK quiet

Details

Assigning a filename<= means renaming the corresponding file on disk - even for ram objects. If that fails, the assignment fails. If a file is moved in or out of `getOption("fftempdir")` the `finalizer` is changed accordingly to 'delete' in `getOption("fftempdir")` and 'close' otherwise.

A pattern is an incomplete filename (optional path and optional filename-prefix) that is turned to filenames by adding a random string using and optionally an extension from optionally an extension from `getOption("ffextension")` (see `ff tempfile`). filename<= exhibits R's standard behaviour of considering "filename" and "/filename" both to be located in `getwd`. By constrast pattern<= will create "filename" without path in `getOption("fftempdir")` and only "/filename" in `getwd`.

Value

filename and pattern return a character filename or pattern. For `ff df` returns a list with one filename element for each physical component. The assignment functions return the changed object, which will keep the change even without re-assigning the return-value

Author(s)

Jens Oehlschlägel

See Also

`ff tempfile`, `finalizer`, `ff`, `as.ff`, `as.ram`, `update.ff`, `file.move`

Examples

```r
## Not run:
message("Neither giving pattern nor filename gives a random filename
with extension ff extension in ff tempdir")
x <- ff(1:12)
finalizer(x)
filename(x)
message("Giving a pattern with just a prefix moves to a random filename
beginning with the prefix in ff tempdir")

pattern(x) <- "myprefix_"
filename(x)
message("Giving a pattern with a path and prefix moves to a random filename
beginning with prefix in path (use . for getwd) ")

pattern(x) <- "./myprefix"
filename(x)
message("Giving a filename moves to exactly this filename and extension
in the R-expected place")

if (!file.exists("./myfilename.myextension")){
```
finalize

Call finalize

Description

This calls the currently assigned finalizer, either via R’s finalization mechanism or manually.

Usage

finalize(x, ...)
## S3 method for class 'ff_pointer'
finalize(x, ...)
## S3 method for class 'ff'
finalize(x, ...)
## S3 method for class 'ffdf'
finalize(x, ...)

Arguments

x either an ff or ffdf object or an ff_pointer, see details
...
currently ignored

Details

The finalize.ff_pointer method is called from R after it had been passed to reg.finalizer. It will set the finalizer name to NULL and call the finalizer.

The finalize generic can be called manually on ff or ffdf objects. It will call the finalizer but not touch the finalizer name.

For more details see finalize

Value

returns whatever the called finalizer returns, for ffdf a list with the finalization returns of each physical component is returned.
finalizer

Note

finalize.ff_pointer MUST NEVER be called manually - neither directly nor by calling the generic on an ff_pointer (could erroneously signal that there is no pending finalization lurking around)

Author(s)

Jens Oehlschlägel

See Also

finalizer

Examples

```r
x <- ff(1:12, pattern="./finalizerdemo")
fnam <- filename(x)
finalizer(x)
is.open(x)
file.exists(fnam)

finalize(x)

finalizer(x)
is.open(x)
file.exists(fnam)

delete(x)
finalizer(x)
is.open(x)
file.exists(fnam)

rm(x)
gc()
```

**finalizer**

Get and set finalizer (name)

Description

The generic finalizer allows to get the current finalizer. The generic finalizer<- allows to set the current finalizer or to change an existing finalizer (but not to remove a finalizer).

Usage

```r
finalizer(x, ...)
finalizer(x, ...) <- value
## S3 method for class 'ff'
finalizer(x, ...)
```
## S3 replacement method for class 'ff'

```
finalizer(x, ...) <- value
```

### Arguments

- **x**
  - an `ff` object
- **value**
  - the name of the new finalizer
- **...**
  - ignored

### Details

If an `ff` object is created a finalizer is assigned, it has the task to free resources no longer needed, for example, remove the `ff` file or free the C++ RAM associated with an open `ff` file. The assigned finalizer depends on the location of the `ff` file: if the file is created in `getOption(fftempdir)` it is considered temporary and has default finalizer `delete`, files created in other locations have default finalizer `close`. The user can override this either by setting options("fffinalizer") or by using argument `finalizer` when creating single `ff` objects. Available finalizer generics are "close", "delete" and "deleteIfOpen", available methods are `close.ff`, `delete.ff` and `deleteIfOpen.ff`.

In order to be able to change the finalizer before finalization, the finalizer is NOT directly passed to R's finalization mechanism `reg.finalizer` (an active finalizer can never be changed other than be executed). Instead the NAME of the desired finalizer is stored in the `ff` object and `finalize.ff_pointer` is passed to `reg.finalizer`. `finalize.ff_pointer` will at finalization-time determine the desired finalizer and call it.

There are two possible triggers for execution `finalize.ff_pointer`:

1. the garbage collection `gc` following removal `rm` of the `ff` object
2. closing R if `finonexit` was `TRUE` at `ff` creation-time, determined by options("fffinonexit") and `ff` argument `finonexit`

Furthermore there are two possible triggers for calling the finalizer:

1. an explicit call to `finalize`
2. an explicit call to one of the finalizers `close`, `delete` and `deleteIfOpen`

The user can define custom finalizers by creating a generic function like `delete`, a `ff_pointer` method like `delete.ff_pointer` and a `ff` method for manual calls like `delete.ff`. The user then is responsible to take care of two things:

1. adequate freeing of resources
2. proper maintenance of the finalizer name in the `ff` object via `physical$finalizer`

`is.null(finalizer(ff))` indicates NO active finalizer, i.e. no pending execution of `finalize.ff_pointer` lurking around after call of `reg.finalizer`. This requires that:

1. the `ff_pointer` method sets the finalizer name to `NULL`
2. the `ff` may change a non-`NULL` finalizer name to a different name but not change it to `NULL`
Value

`finalizer` returns the name of the active finalizer or NULL if no finalizer is active.
`finalizer<-` returns the changed ff object (reassignment of this return value not needed to keep the change). If there was no pending call to `finalize.ff_pointer(is.null(finalizer(ff)))`, `finalizer<-` will create one by calling `reg.finalizer` with the current setting of `physical$finonexit`.

Note

You can not assign NULL to an active finalizer using `finalizer<-` because this would not stop R’s finalization mechanism and would carry the risk of assigning MULTIPLE finalization tasks.

Author(s)

Jens Oehlschlägel

See Also

`ff, finalize, reg.finalizer`

Examples

```r
x <- ff(1:12, pattern="./finalizerdemo")
fnam <- filename(x)
finalizer(x)
finalizer(x) <- "delete"
finalizer(x)
rm(x)
file.exists(fnam)
gc()
file.exists(fnam)
```

Description

Check if an object has fixed diagonal

Usage

```r
fixdiag(x, ...)
```

```r
fixdiag(x, ...) <- value
```

## S3 method for class 'ff'
```r
fixdiag(x, ...)
```

## Default S3 method:
```r
fixdiag(x, ...)
```

## S3 method for class 'dist'
```r
fixdiag(x, ...)
```
Arguments

- **x**: an ff or ram object
- **value**: assignment value
- **...**: further arguments (not used)

Details

FF symmetric matrices can be declared to have fixed diagonal at creation time. Compatibility function `fixdiag.default` returns NULL, `fixdiag.dist` returns 0.

Value

NULL or the scalar representing the fixed diagonal

Author(s)

Jens Oehlschlägel

See Also

- `fixdiag`, `ff`, `dist`

Examples

```r
fixdiag(matrix(1:16, 4, 4))
fixdiag(dist(rnorm(1:4)))
```

---

`geterror.ff`  Get error and error string

Description

Get last error code and error string that occurred on an ff object.

Usage

```r
geterror.ff(x)
geterrstr.ff(x)
```

Arguments

- **x**: an ff object

Value

`geterror.ff` returns an error integer code (no error = 0) and `geterrstr.ff` returns the error message (no error = "no error").
getpagesize

Author(s)
Jens Oehlschlägel, Daniel Adler (C++ back-end)

See Also
ff

Examples
x <- ff(1:12)
geterror.ff(x)
geterrstr.ff(x)
rm(x); gc()

getpagesize  Get page size information

Description
The function is used for obtaining the natural OS-specific page size in Bytes. getpagesize returns
the OS-specific page size in Bytes for memory mapped files, while getdefaultpagesize returns
a suggested page size. getalignedpagesize returns the pagesize as a multiple of the OS-specific
page size in Bytes, which is the correct way to specify pagesize in ff.

Usage
getpagesize()
getdefaultpagesize()
getalignedpagesize(pagesize)

Arguments
pagesize  a desired pagesize in bytes

Value
An integer giving the page size in Bytes.

Author(s)
Daniel Adler, Jens Oehlschlägel

Examples
getpagesize()
getdefaultpagesize()
getalignedpagesize(2000000)
getset.ff  

Reading and writing vectors of values (low-level)

Description

The three functions get.ff, set.ff and getset.ff provide the simplest interface to access an ff file: getting and setting vector of values identified by positive subscripts

Usage

get.ff(x, i)
set.ff(x, i, value, add = FALSE)
getset.ff(x, i, value, add = FALSE)

Arguments

x  an ff object
i  an index position within the ff file
value the value to write to position i
add  TRUE if the value should rather increment than overwrite at the index position

Details

getset.ff combines the effects of get.ff and set.ff in a single operation: it retrieves the old value at position i before changing it. getset.ff will maintain na.count.

Value

get.ff returns a vector, set.ff returns the 'changed' ff object (like all assignment functions do) and getset.ff returns the value at the subscript positions. More precisely getset.ff(x, i, value, add=FALSE) returns the old values at the subscript positions i while getset.ff(x, i, value, add=TRUE) returns the incremented values at the subscript positions.

Note

get.ff, set.ff and getset.ff are low level functions that do not support ramclass and ramattrs and thus will not give the expected result with factor and POSIXct

Author(s)

Jens Oehlschlägel

See Also

readwrite.ff for low-level access to contiguous chunks and [.ff for high-level access
**Examples**

```r
x <- ff(0, length=12)
get.ff(x, 3L)
set.ff(x, 3L, 1)
x
set.ff(x, 3L, 1, add=TRUE)
x
getset.ff(x, 3L, 1, add=TRUE)
getset.ff(x, 3L, 1)
x
rm(x); gc()
```

---

**hi Hybrid index class**

**Description**

Class for hybrid index representation, plain and rle-packed

**Usage**

```r
hi(from, to, by = 1L, maxindex = NA, vw = NULL, pack = TRUE, NAs = NULL)
## S3 method for class 'hi'
print(x, ...)
## S3 method for class 'hi'
str(object, nest.lev=0, ...)
```

**Arguments**

- **from**: integer vector of lower sequence bounds
- **to**: integer vector of upper sequence bounds
- **by**: integer of stepsizes
- **maxindex**: maximum indep position (needed for negative indices)
- **vw**: virtual window information, see `vw`
- **pack**: FALSE to suppress rle-packing
- **NAs**: a vector of NA positions (not yet used)
- **x**: an object of class ‘hi’ to be printed
- **object**: an object of class ‘hi’ to be str’ed
- **nest.lev**: current nesting level in the recursive calls to str
- **...**: further arguments passed to the next method
Details

Class `hi` will represent index data either as a plain positive or negative index vector or as an rle-packed version thereof. The current implementation switches from plain index positions `i` to rle-packed storage of `diff(i)` as soon as the compression ratio is 3 or higher. Note that sequences shorter than 2 must never be packed (could cause C-side crash). Furthermore hybrid indices are guaranteed to be sorted ascending, which helps `ffs` access method avoiding to swap repeatedly over the same memory pages (or file positions).

Value

A list of class 'hi' with components

- `x` directly accessed by the C-code: the sorted index as returned by `rlepack`
- `ix` NULL or positions to restore original order
- `re` logical scalar indicating if sequence was reversed from descending to ascending (in this case `is.null(ix)`)
- `minindex` directly accessed by the C-code: represents the lowest positive subscript to be enumerated in case of negative subscripts
- `maxindex` directly accessed by the C-code: represents the highest positive subscript to be enumerated in case of negative subscripts
- `length` number of subscripts, whether negative or positive, not the number of selected elements
- `dim` NULL or dim – used by `as.matrix.hi`
- `dimorder` NULL or `dimorder`
- `symmetric` logical scalar indicating whether we have a symmetric matrix
- `fixdiag` logical scalar indicating whether we have a fixed diagonal (can only be true for symmetric matrices)
- `vw` virtual window information `vw`
- `NAs` NULL or NA positions as returned by `rlepack`

Note

`hi` defines the class structure, however usually `as.hi` is used to actually Hybrid Index Preprocessing for `ff`

Author(s)

Jens Oehlschlägel

See Also

`as.hi` for coercion, `rlepack`, `intrle`, `maxindex`, `poslength`

Examples

```r
hi(c(1, 11, 29), c(9, 19, 21), c(1,1,-2))
as.integer(hi(c(1, 11, 29), c(9, 19, 21), c(1,1,-2)))
```
**hiparse**  

*Hybrid Index, parsing*

---

**Description**

*hiparse* implements the parsing done in Hybrid Index Preprocessing in order to avoid RAM for expanding index expressions. *Not to be called directly*

**Usage**

```r
hiparse(x, envir, first = as.integer(NA), last = as.integer(NA))
```

**Arguments**

- `x`: an index expression, precisely: *call*
- `envir`: the environment in which to evaluate components of the index expression
- `first`: first index position found so far
- `last`: last index position found so far

**Details**

This primitive parser recognizes the following tokens: numbers like 1, symbols like x, the colon sequence operator `:` and the concat operator `c`. *hiparse* will *Recall* until the index expression is parsed or an unknown token is found. If an unknown token is found, *hiparse* evaluates it, inspects it and either accepts it or throws an error, caught by *as.hi.call*, which falls back to evaluating the index expression and dispatching (again) an appropriate *as.hi* method. Reasons for suspending the parsing: if the inspected token is of class 'hi', 'ri', 'bit', 'bitwhich', 'is.logical', 'is.character', 'is.matrix' or has length>16.

**Value**

undefined (and redefined as needed by *as.hi.call*)

**Author(s)**

Jens Oehlschlägel

**See Also**

*hi*, *as.hi.call*
**is.ff**  
*Test for class ff*

**Description**  
checks if x inherits from class "ff"

**Usage**  
is.ff(x)

**Arguments**  
x any object

**Value**  
logical scalar

**Author(s)**  
Jens Oehlschlägel

**See Also**  
inherits, as.ff, is.ffdf

**Examples**  
is.ff(integer())
**is.open**

**Value**

logical scalar

**Author(s)**

Jens Oehlschlägel

**See Also**

`inherits`, `as.ffdf`, `is.ff`

**Examples**

```r
is.ffdf(integer())
```

---

**is.open**  
*Test if object is opened*

**Description**

Test whether an ff or ffdf object or a ff_pointer is opened.

**Usage**

```r
is.open(x, ...)
## S3 method for class 'ff'
is.open(x, ...)
## S3 method for class 'ffdf'
is.open(x, ...)
## S3 method for class 'ff_pointer'
is.open(x, ...)
```

**Arguments**

- `x`  
an *ff* or *ffdf* object
- `...`  
further arguments (not used)

**Details**

ff objects open automatically if accessed while closed. For ffdf objects we test all of their physical components including their *row.names* if they are *is.ff*

**Value**

TRUE or FALSE (or NA if not all components of an ffdf object are opened or closed)
Author(s)

Jens Oehlschlägel

See Also

is.readonly, open.ff, close.ff

Examples

```r
x <- ff(1:12)
is.open(x)
close(x)
is.open(x)
rm(x); gc()
```

is.readonly Get readonly status

Description

Get readonly status of an ff object

Usage

```r
is.readonly(x, ...)
## S3 method for class 'ff'
is.readonly(x, ...)
```

Arguments

```
x x
... ...
```

Details

ff objects can be created/opened with `readonly=TRUE`. After each opening of the ff file readonly status is stored in the `physical` attributes and serves as the default for the next opening. Thus querying a closed ff object gives the last readonly status.

Value

logical scalar

Author(s)

Jens Oehlschlägel
is.sorted

See Also

open.ff, physical

Examples

```r
x <- ff(1:12)
is.readonly(x)
close(x)
open(x, readonly=TRUE)
is.readonly(x)
close(x)
is.readonly(x)
rm(x)
```

Description

Functions to mark an ff or ram object as ‘is.sorted’ and query this. Responsibility to maintain this attribute is with the user.

Usage

```r
## Default S3 method:
is.sorted(x, ...)
## Default S3 replacement method:
is.sorted(x, ...) <- value
```

Arguments

- `x`: an ff or ram object
- `...`: ignored
- `value`: NULL (to remove the ‘is.sorted’ attribute) or TRUE or FALSE

Details

Sorting is slow, see `sort`. Checking whether an object is sorted can avoid unnessary sorting – see `is.unsorted, intisasc` – but still takes too much time with large objects stored on disk. Thus it makes sense to maintain an attribute, that tells us whether sorting can be skipped. Note that – though you change it yourself – `is.sorted` is a `physical` attribute of an object, because it represents an attribute of the `data`, which is shared between different `virtual` views of the object.

Value

TRUE (if set to TRUE) or FALSE (if set to NULL or FALSE)
Note

*ff* will set \texttt{is.sorted(x) <- FALSE} if \texttt{clone} or \texttt{length<-.ff} have increased length.

Author(s)

Jens Oehlschlägel

See Also

\texttt{is.ordered.ff} for testing factor levels, \texttt{is.unsorted} for testing the data, \texttt{intisasc} for a quick version thereof, \texttt{na.count} for yet another \texttt{physical} attribute

Examples

```r
x <- 1:12
is.sorted(x) <- !(is.na(is.unsorted(x)) || is.unsorted(x))
is.sorted(x)
x[1] <- 100L
message("don't forget to maintain once it's no longer TRUE")
is.sorted(x) <- FALSE
message("check whether as 'is.sorted' attribute is maintained")
!is.null(physical(x)$is.sorted)
message("remove the 'is.sorted' attribute")
is.sorted(x) <- NULL
message("NOTE that querying 'is.sorted' still returns FALSE")
is.sorted(x)
```

Description

Gets and sets length of \texttt{ff} objects.

Usage

```r
## S3 method for class 'ff'
length(x)
## S3 replacement method for class 'ff'
length(x) <- value
```

Arguments

- \texttt{x} : object to query
- \texttt{value} : new object length
**Details**

Changing the length of ff objects is only allowed if no `vw` is used. Changing the length of ff objects will remove any `dim.ff` and `dimnames.ff` attribute. Changing the length of ff objects will remove any `na.count` or `is.sorted` attribute and warn about this. New elements are usually zero, but it may depend on OS and filesystem what they really are. If you want standard R behaviour: filling with NA, you need to do this yourself. As an exception to this rule, ff objects with `names.ff` will be filled with NA's automatically, and the length of the names will be adjusted (filled with position numbers where needed, which can easily consume a lot of RAM, therefore removing 'names' will help to faster increase length without RAM problems).

**Value**

Integer scalar

**Note**

Special care needs to be taken with regard ff objects that represent factors. For ff factors based on UNSIGNED `vmodes`, new values of zero are silently interpreted as the first factor level. For ff factors based on SIGNED `vmodes`, new values of zero result in illegal factor levels. See `nrow<-`.

**Author(s)**

Jens Oehlschlägel

**See Also**

`length, maxlength, file.resize, dim, virtual`

**Examples**

```r
x <- ff(1:12)
maxlength(x)
length(x)
length(x) <- 10
maxlength(x)
length(x)
length(x) <- 16
maxlength(x)
length(x)
rm(x); gc()
```

---

**Description**

Getting "length" (number of columns) of a ffdf dataframe
Usage

```r
## S3 method for class 'ffdf'
length(x)
```

Arguments

- `x` an `ffdf` object

Value

integer number of columns

Author(s)

Jens Oehlschlägel

See Also

dim.ffdf, length.ff, ffdf

Examples

```r
length(as.ffdf(data.frame(a=1:26, b=letters)))
gc()
```

---

**length.hi**  
*Hybrid Index, querying*

Description

Functions to query some index attributes

Usage

```r
## S3 method for class 'hi'
length(x)
maxindex(x, ...)
poslength(x, ...)
## S3 method for class 'hi'
maxindex(x, ...)
## S3 method for class 'ri'
maxindex(x, ...)
## S3 method for class 'bit'
maxindex(x, ...)
## S3 method for class 'bitwhich'
maxindex(x, ...)
## S3 method for class 'logical'
maxindex(x, ...)
```
## length.hi

```r
## Default S3 method:
machine(x, ...)
## S3 method for class 'hi'
poslength(x, ...)
## S3 method for class 'ri'
poslength(x, ...)
## S3 method for class 'bit'
poslength(x, ...)
## S3 method for class 'bitwhich'
poslength(x, ...)
## S3 method for class 'logical'
poslength(x, ...)
## Default S3 method:
poslength(x, ...)
```

### Arguments

- `x`: an object of class `hi`
- `...`: further arguments (not used)

### Details

`length.hi` returns the number of the subscript elements in the index (even if they are negative). By contrast the generic `poslength` returns the number of selected elements (which for negative indices is `maxindex(x) - length(unique(x))`). The generic `maxindex` returns the highest possible index position.

### Value

An integer scalar

### Note

Duplicated negative indices are removed

### Author(s)

Jens Oehlschlägel

### See Also

`hi, as.hi, length.ff, length`

### Examples

```r
length(as.hi(-1, maxindex=12))
poslength(as.hi(-1, maxindex=12))
maxindex(as.hi(-1, maxindex=12))
message("note that")
length(as.hi(c(-1, -1), maxindex=12))
length(as.hi(c(1,1), maxindex=12))
```
levels.ff

Getting and setting factor levels

Description

levels.ff<- sets factor levels, levels.ff gets factor levels

Usage

## S3 method for class 'ff'
levels(x)

## S3 replacement method for class 'ff'
levels(x) <- value

is.factor(x)

is.ordered(x)

## S3 method for class 'ff'
is.factor(x)

## S3 method for class 'ff'
is.ordered(x)

## Default S3 method:

is.factor(x)

## Default S3 method:

is.ordered(x)

Arguments

x
an ff object

value
the new factor levels, if NA is an allowed level it needs to be given explicitly, nothing is excluded

Details

The ff object must have an integer vmode, see .rammode. If the mode is unsigned – see .vunsigned – the first factor level is coded with 0L instead of 1L in order to maximize the number of codable levels. Usually the internal ff coding – see ram2ffcode – is invisible to the user: when subscripting from an ff factor, unsigned codings are automatically converted to R’s standard factor codes starting at 1L. However, you need to be aware of the internal ff coding in two situations.

1. If you convert an ff integer object to an ff factor object and vice versa by assigning levels and is.null(oldlevels)!=is.null(newlevels).
2. Assigning data that does not match any level usually results in NA, however, in unsigned types there is no NA and all unknown data are mapped to the first level.

Value

levels returns a character vector of levels (possibly including as.character(NA)).
Note

When levels are assigned to an ff object that formerly had no levels, we assign automatically
\texttt{ramclass} == \texttt{"factor"}. If you want to change to an ordered factor, use \texttt{virtual$ramclass} \leftarrow \texttt{c("ordered", "factor")}

Author(s)

Jens Oehlschl"a gel

See Also

\texttt{ramclass}, \texttt{factor}, \texttt{virtual}

Examples

message("--- create an ff factor including NA as last level")
x \leftarrow \texttt{ff("a", levels=c(letters, NA), length=99)}
message(" we expect a warning because "A" is an unknown level")
x[] \leftarrow \texttt{c("a", NA,"A")}
x
levels(x)

message("--- create an ff ordered factor")
x \leftarrow \texttt{ff(letters, levels=letters, ramclass=c("ordered","factor"), length=260)}
x
levels(x)

message(" make it a non-ordered factor")
\texttt{virtual(x)$ramclass} \leftarrow \texttt{"factor"}
x
\texttt{rm(x); gc()}

## Not run:
message("--- create an unsigned quad factor")
x \leftarrow \texttt{ff(c("A","T","G","C"), levels=c("A","T","G","C"), vmode="quad", length=100)}
x
message(" 0:3 coding usually invisible to the user")
\texttt{unclass(x[1:4])}
message(" after removing levels, the 0:3 coding becomes visible to the user")
message(" we expect a warning here")
levels(x) \leftarrow \texttt{NULL}
x[1:4]
\texttt{rm(x); gc()}

## End(Not run)
Description

This help page lists the currently known limitations of package ff, as well as differences between ff and ram methods.

Automatic file removal

Remind that not giving parameter `ff(filename=)` will result in a temporary file in `fftempdir` with 'delete' finalizer, while giving parameter `ff(filename=)` will result in a permanent file with 'close' finalizer. Do avoid setting `setwd(getOption("fftempdir"))`! Make sure you really understand the implications of automatic unlinking of `getOption("fftempdir")` .onUnload, of finalizer choice and of finalizing behaviour at the end of R sessions as defaulted in `getOption("fffinonexit")`. Otherwise you might experience 'unexpected' losses of files and data.

Size of objects

Currently ff objects cannot have length zero and are limited to `.Machine$integer.max` elements. We have not yet ported the R code to support 64bit double indices (in essence 52 bits integer) although the C++ back-end has been prepared for this. Furthermore filesize limitations of the OS apply, see `ff`.

Side effects

In contrast to standard R expressions, ff expressions violate the functional programming logic and are called for their side effects. This is also true for ram compatibility functions `swap.default`, and `add.default`.

Hybrid copying semantics

If you modify a copy of an ff object, changes of data ([<-) and of physical attributes will be shared, but changes in virtual and class attributes will not.

Limits of compatibility between ff and ram objects

If it’s not too big, you can move an ff object completely into R’s RAM through `as.ram`. However, you should watch out for three limitations:

1. Ram objects don’t have hybrid copying semantics; changes to a copy of a ram object will never change the original ram object
2. Assigning values to a ram object can easily upgrade to a higher storage.mode. This will create conflicts with the vmode of the ram object, which goes undetected until you try to write back to disk through `as.ff`.
3. Writing back to disk with `as.ff` under the same filename requires that the original ff object has been deleted (or at least closed if you specify parameter overwrite=TRUE).
Index expressions

ff index expressions do not allow zeros and NAs, see see [.ff] and see as.hi

Availability of bydim parameter

Parameter bydim is only available in ff access methods, see [.ff]

Availability of add parameter

Parameter add is only available in ff access methods, see [.ff]

Compatibility of swap and add

If index expressions contain duplicated positions, the ff and ram methods for swap and add will behave differently, see swap.

Definition of [] and [[]<-

You should consider the behaviour of [.ff] and [[<-.ff] as undefined and not use them in programming. Currently they are shortcuts to get.ff and set.ff, which unlike [.ff] and [[<-.ff] do not support factor and POSIXct, nor dimorder or virtual windows vw. In contrast to the standard methods, [.ff] and [[<-.ff] only accepts positive integer index positions. The definition of [.ff] and [[<-.ff] may be changed in the future.

Multiple vector interpretation in arrays

R objects have always standard dimorder 1:length(dim). In case of non-standard dimorder (see dimorderStandard) the vector sequence of array elements in R and in the ff file differs. To access array elements in file order, you can use getset.ff, readwrite.ff or copy the ff object and set dim(ff)<-NULL to get a vector view into the ff object (using [ dispatches the vector method [.ff]). To access the array elements in R standard dimorder you simply use [ which dispatches to [.ff_array. Note that in this case as.hi will unpack the complete index, see next section.

RAM expansion of index expressions

Some index expressions do not consume RAM due to the hi representation. For example 1:n will almost consume no RAM however large n. However, some index expressions are expanded and require to maxindex(i) * .rambytes["integer"] bytes, either because the sorted sequence of index positions cannot be rle-packed efficiently or because hiparse cannot yet parse such expression and falls back to evaluating/expanding the index expression. If the index positions are not sorted, the index will be expanded and a second vector is needed to store the information for re-ordering, thus the index requires 2 * maxindex(i) * .rambytes["integer"] bytes.

RAM expansion when recycling assignment values

Some assignment expressions do not consume RAM for recycling. For example x[1:n] <- 1:k will not consume RAM however large is n compared to k, when x has standard dimorder. However, if length(value)>1, assignment expressions with non-ascending index positions trigger recycling the value R-side to the full index length. This will happen if dimorder does not match parameter bydim or if the index is not sorted in ascending order.
Byteorder incompatibility

Note that ff files cannot been transferred between systems with different byteorder.

Description

create matrix indices from row and columns positions

Usage

matcomb(r, c)

Arguments

r    integer vector of row positions

c    integer vector of column positions

Details

rows rotate faster than columns

Value

a k by 2 matrix of matrix indices where  k = length(r) * length(c)

Author(s)

Jens Oehlschlägel

See Also

row, col, expand.grid

Examples

matcomb(1:3, 1:4)
matcomb(2:3, 2:4)
**Description**

Print beginning and end of big matrix

**Usage**

```
matprint(x, maxdim = c(16, 16), digits =getOption("digits"))
## S3 method for class 'matprint'
print(x, quote = FALSE, right = TRUE, ...)
```

**Arguments**

- `x` a matrix
- `maxdim` max number of rows and columns for printing
- `digits` see `format`
- `quote` see `print`
- `right` see `print`
- `...` see `print`

**Value**

a list of class 'matprint' with components

- `subscript` a list with four vectors of subscripts: row begin, column begin, row end, column end
- `example` the extracted example matrix as character including separators
- `rsep` logical scalar indicating whether row separator is included
- `csep` logical scalar indicating whether column separator is included

**Author(s)**

Jens Oehlschlägel

**See Also**

`vecprint`

**Examples**

```
matprint(matrix(1:(300*400), 300, 400))
```
maxffmode

**Description**

maxffmode returns the lowest vmode that can absorb all input vmodes without data loss.

**Usage**

maxffmode(...)

**Arguments**

... one or more vectors of vmodes

**Value**

the smallest fffmode which can absorb the input vmodes without data loss

**Note**

The output can be larger than any of the inputs (if the highest input vmode is an integer type without NA and any other input requires NA).

**Author(s)**

Jens Oehlschlägel

**See Also**

.vcoerceable, fffmode, ffconform

**Examples**

maxffmode(c("quad","logical"), "ushort")
**maxlength**

*Get physical length of an ff or ram object*

**Description**

`maxlength` returns the physical length of an ff or ram object.

**Usage**

```r
maxlength(x, ...)  
## S3 method for class 'ff'
maxlength(x, ...)

## Default S3 method:
maxlength(x, ...)
```

**Arguments**

- `x` : ff or ram object
- `...` : additional arguments (not used)

**Value**

integer scalar

**Author(s)**

Jens Oehlschlägel

**See Also**

`length.ff`, `maxindex`

**Examples**

```r
x <- ff(1:12)
length(x) <- 10
length(x)
maxlength(x)
x
rm(x); gc()
```
mismatch  Test for recycle mismatch

Description

mismatch will return TRUE if the larger of n, ny is not a multiple of the other and the other is >0 (see arithmetic.c). ymismatch will return TRUE if nx is not a multiple of ny and ny >0.

Usage

mismatch(nx, ny)
ymismatch(nx, ny)

Arguments

nx  x length
ny  y length

Value

logical scalar

Author(s)

Jens Oehlschlägel

See Also

ffconform

Examples

ymismatch(4,0)
ymismatch(4,2)
ymismatch(4,3)
ymismatch(2,4)
mismatch(4,0)
mismatch(4,2)
mismatch(4,3)
mismatch(2,4)
**Description**

The 'na.count' physical attribute gives the current number of NAs if properly initialized and properly maintained, see details.

**Usage**

```r
## S3 method for class 'ff'
na.count(x, ...)
## Default S3 method:
na.count(x, ...)
## S3 replacement method for class 'ff'
na.count(x, ...) <- value
## Default S3 replacement method:
na.count(x, ...) <- value
```

**Arguments**

- `x`: an ff or ram object
- `...`: further arguments (not used)
- `value`: NULL (to remove the 'na.count' attribute) or TRUE to activate or an integer value

**Details**

The 'na.count' feature is activated by assigning the current number of NAs to `na.count(x) <- currentNA` and deactivated by assigning NULL. The 'na.count' feature is maintained by the, `getset.ff`, `readwrite.ff` and `swap`, other ff methods for writing – `set.ff`, `[[.ff`, `write.ff`, `[[.ff` – will stop if 'na.count' is activated. The functions `na.count` and `na.count<-` are generic. For ram objects, the default method for `na.count` calculates the number of NAs on the fly, thus no maintenance restrictions apply.

**Value**

NA (if set to NULL or NA) or an integer value otherwise

**Author(s)**

Jens Oehlschlägel, Daniel Adler (C++ back-end)

**See Also**

`getset.ff`, `readwrite.ff` and `swap` for methods that support maintenance of 'na.count', NA, `is.sorted` for yet another physical attribute
Examples

```r
message("--- ff examples ---")
x <- ff(1:12)
na.count(x)
message("activate the 'na.count' physical attribute and set the current na.count manually")
na.count(x) <- 0L
message("add one NA with a method that maintains na.count")
swap(x, NA, 1)
na.count(x)
message("remove the 'na.count' physical attribute (and stop automatic maintenance)")
na.count(x) <- NULL
message("activate the 'na.count' physical attribute and have ff automatically calculate the current na.count")
na.count(x) <- TRUE
na.count(x)
message("--- ram examples ---")
x <- 1:12
na.count(x)
x[1] <- NA
message("activate the 'na.count' physical attribute and have R automatically calculate the current na.count")
na.count(x) <- TRUE
na.count(x)
message("remove the 'na.count' physical attribute (and stop automatic maintenance)")
na.count(x) <- NULL
na.count(x)
rm(x); gc()
```

---

names.ff  

Description

For `ff_vector`s you can set names, though this is not recommended for large objects.

Usage

```r
## S3 method for class 'ff'
names(x)
## S3 replacement method for class 'ff'
names(x) <- value
## S3 method for class 'ff_array'
names(x)
## S3 replacement method for class 'ff_array'
names(x) <- value
```

Arguments

- `x`  
  a `ff` vector

- `value`  
  a character vector
Details

If `vw` is set, `names.ff` returns the appropriate part of the names, but you can't set names while `vw` is set. `names.ff_array` returns NULL and setting names for `ff_arrays` is not allowed, but setting `dimnames` is.

Value

`names` returns a character vector (or NULL)

Author(s)

Jens Oehlschlægel

See Also

`names`, `dimnames.ff_array`, `vw`, `virtual`

Examples

```r
x <- ff(1:26, names=letters)
names(x)
names(x) <- LETTERS
names(x)
names(x) <- NULL
names(x)
rm(x); gc()
```

---

**nrowAssign**

*Assigning the number of rows or columns*

Description

Function `nrow<-` assigns `dim` with a new number of rows. Function `ncol<-` assigns `dim` with a new number of columns.

Usage

```r
nrow(x) <- value
ncol(x) <- value
```

Arguments

- `x` : a object that has `dim` AND can be assigned ONE new dimension
- `value` : the new size of the assigned dimension

Details

Currently only assigning new rows to `ffdf` is supported. The new ffdf rows are not initialized (usually become zero). NOTE that
Value

The object with a modified dimension

Author(s)

Jens Oehlschlägel

See Also

ffdf, dim.ffdf

Examples

a <- as.ff(1:26)
b <- as.ff(factor(letters)) # vmode="integer"
c <- as.ff(factor(letters), vmode="ubyte")
df <- ffdf(a,b,c)
nrow(df) <- 2*26
df
message("NOTE that the new rows have silently the first level 'a' for UNSIGNED vmodes")
message("NOTE that the new rows have an illegal factor level <0> for SIGNED vmodes")
message("It is your responsibility to put meaningful content here")
message("As an example we replace the illegal zeros by NA")
df$b[27:52] <- NA
df
rm(a,b,c,df); gc()

open.ff opens an ff file, optionally marking it readonly and optionally specifying a caching scheme.

Usage

## S3 method for class 'ff'
open(con, readonly = FALSE, pagesize = NULL, caching = NULL, assert = FALSE, ...)
## S3 method for class 'ffdf'
open(con, readonly = FALSE, pagesize = NULL, caching = NULL, assert = FALSE, ...)

Arguments

con
an ff or ffdf object
readonly
readonly
pagesize
number of bytes to use as pagesize or NULL to take the pagesize stored in the physical attribute of the ff object, see getalignedpagesize
caching          one of 'mmnoflush' or 'mmeachflush', see `ff`
assert           setting this to TRUE will give a message if the ff was not open already
...              further arguments (not used)

Details

`ff` objects will be opened automatically when accessing their content and the file is still closed. Opening `ffdf` objects will open all of their `physical` components including their `row.names` if they are `is.ff`

Value

TRUE if object could be opened, FALSE if it was opened already (or NA if not all components of an `ffdf` returned FALSE or TRUE on opening)

Author(s)

Jens Oehlschlägel

See Also

`ff, close.ff, delete, deleteIfOpen, getalignedpagesize`

Examples

```r
x <- ff(1:12)
close(x)
is.open(x)
open(x)
is.open(x)
close(x)
is.open(x)
x[]
is.open(x)
y <- x
close(y)
is.open(x)
rm(x,y); gc()
```
Usage

```r
class(x, ...)    # S3 method for class 'ff'
```

Arguments

- `x`  
  An `ff` object
- `...`  
  Further arguments (not used)

Value

An integer number of bytes

Author(s)

Jens Oehlschlägel

See Also

getpagesize

Examples

```r
x <- ff(1:12)
pagesize(x)
```

---

### physical.ff

* Getting and setting physical and virtual attributes of `ff` objects *

Description

Functions for getting and setting physical and virtual attributes of `ff` objects.

Usage

```r
## S3 method for class 'ff'
physical(x)
## S3 method for class 'ff'
virtual(x)
# S3 replacement method for class 'ff'
physical(x) <- value
# S3 replacement method for class 'ff'
virtual(x) <- value
```
Arguments

- `x`: an ff object
- `value`: a list with named elements

Details

ff objects have physical and virtual attributes, which have different copying semantics: physical attributes are shared between copies of ff objects while virtual attributes might differ between copies. `as.ram` will retain some physical and virtual attributes in the ram clone, such that `as.ff` can restore an ff object with the same attributes.

Value

`physical` and `virtual` returns a list with named elements

Author(s)

Jens Oehlschlägel

See Also

`physical`, `physical.fddf`, `ff`, `as.ram`; `is.sorted` and `na.count` for applications of physical attributes; `levels.ff` and `ramattrs` for applications of virtual attributes

Examples

```r
x <- ff(1:12)
x
physical(x)
virtual(x)
y <- as.ram(x)
physical(y)
virtual(y)
rm(x,y); gc()
```

Description

Functions for getting physical and virtual attributes of fddf objects.

Usage

```r
# S3 method for class 'fddf'
physical(x)
# S3 method for class 'fddf'
virtual(x)
```
Arguments

x an ffdf object

Details

ffdf objects enjoy a complete decoupling of virtual behaviour from physical storage. The physical component is simply a (potentially named) list where each element represents an atomic ff vector or matrix. The virtual component is itself a dataframe, each row of which defines a column of the ffdf through a mapping to the physical component.

Value

'physical.ffdf' returns a list with atomic ff objects.
'virtual.ffdf' returns a dataframe with the following columns:

<table>
<thead>
<tr>
<th>VirtualVmode</th>
<th>the vmode of this row (=ffdf column)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AsIs</td>
<td>logical defining the AsIs status of this row (=ffdf column)</td>
</tr>
<tr>
<td>VirtualIsMatrix</td>
<td>logical defining whether this row (=ffdf column) represents a matrix</td>
</tr>
<tr>
<td>PhysicalIsMatrix</td>
<td>logical reporting whether the corresponding physical element is a matrix</td>
</tr>
<tr>
<td>PhysicalElementNo</td>
<td>integer identifying the corresponding physical element</td>
</tr>
<tr>
<td>PhysicalFirstCol</td>
<td>integer identifying the first column of the corresponding physical element (1 if it is not a matrix)</td>
</tr>
<tr>
<td>PhysicalLastCol</td>
<td>integer identifying the last column of the corresponding physical element (1 if it is not a matrix)</td>
</tr>
</tbody>
</table>

Author(s)

Jens Oehlschlägel

See Also

ffdf,physical,virtual,vmode

Examples

x <- 1:2
y <- matrix(1:4, 2, 2)
z <- matrix(1:4, 2, 2)

message("Here the y matrix is first converted to single columns by data.frame, then those columns become ff")
d <- as.ffdf(data.frame(x=x, y=y, z=I(z)))
physical(d)
virtual(d)

message("Here the y matrix is first converted to ff, and then stored still as matrix in the ffdf object (although virtually treated as columns of ffdf")
d <- ffdf(x=as.ff(x), y=as.ff(y), z=I(as.ff(z))
physical(d)
virtual(d)

message("Apply the usual methods extracting physical attributes")
lapply(physical(d), filename)
lapply(physical(d), vmode)
message("And don't confuse with virtual vmode")
vmode(d)
rm(d); gc()

---

**print.ff**  
*Print and str methods*

**Description**
- Printing ff objects and compactly showing their structure

**Usage**
```r
## S3 method for class 'ff'
print(x, ...)
## S3 method for class 'ff_vector'
print(x, maxlength = 16, ...)
## S3 method for class 'ff_matrix'
print(x, maxdim = c(16, 16), ...)
## S3 method for class 'ff'
str(object, nest.lev=0, ...)
## S3 method for class 'ffdf'
str(object, nest.lev=0, ...)
```

**Arguments**
- `x`  
  - a ff object
- `object`  
  - a ff object
- `nest.lev`  
  - current nesting level in the recursive calls to str
- `maxlength`  
  - max number of elements to print from an ff_vector
- `maxdim`  
  - max number of elements to print from each dimension from an ff_array
- `...`  
  - further arguments to print

**Details**
- The print methods just print a few exemplary elements from the beginning and end of the dimensions.
**Value**

`invisible()`

**Author(s)**

Jens Oehlschlägel

**See Also**

`ff`, `print`, `str`

**Examples**

```r
x <- ff(1:10000)
x
print(x, maxlength=30)
dim(x) <- c(100,100)
x
rm(x); gc()
```

---

**ram2ffcode**

**Factor codings**

**Description**

Function `ram2ffcode` creates the *internal* factor codes used by `ff` to store factor levels. Function `ram2ramcode` is a compatibility function used instead if `RETURN_FF` is `FALSE`.

**Usage**

```r
ram2ffcode(value, levels, vmode)
ram2ramcode(value, levels)
```

**Arguments**

- `value`: factor or character vector of values
- `levels`: character vector of factor levels
- `vmode`: one of the integer vmodes in `.rammode`

**Details**

Factors stored in unsigned vmodes `.unsigned` have their first level represented as `0L` instead of `1L`.

**Value**

A vector of integer values representing the corresponding factor levels.
ramattribs

Author(s)

Jens Oehlschlägel

See Also

factor, levels.ff, vmode

Examples

ram2ffcode(letters, letters, vmode="byte")
ram2ffcode(letters, letters, vmode="ubyte")
ram2ffcode(letters, letters, vmode="nibble")
message('note that ram2ffcode() does NOT warn that vmode="nibble" cannot store 26 levels')

ramattribs

Get ramclass and ramattribs

Description

Functions ramclass and ramattribs return the respective virtual attributes, that determine which
class (and attributes) an ff object receives when subscripted (or coerced) to ram.

Usage

ramclass(x, ...)
## S3 method for class 'ff'
ramclass(x, ...)
## Default S3 method:
ramclass(x, ...)
ramattribs(x, ...)
## S3 method for class 'ff'
ramattribs(x, ...)
## Default S3 method:
ramattribs(x, ...)

Arguments

x x
...

further arguments (not used)

Details

ramclass and ramattribs provide a general mechanism to store atomic classes in ff objects, for
example factor – see levels.ff – and POSIXct, see the example.
ramorder.default

Value

ramclass returns a character vector with classnames and ramattrs returns a list with names elemens just like attributes. The vectors ramclass_excludes and ramattrs_excludes name those attributes, which are not exported from ff to ram objects when using as.ram.

Author(s)

Jens Oehlschlägel

See Also

ff, virtual, as.ram, levels.ff, attributes, DateTimeClasses

Examples

x <- ff(as.POSIXct(as.POSIXlt(Sys.time(), "GMT")), length=12)
x
ramclass(x)
ramattrs(x)
class(x[])
attributes(x[])
virtual(x)$ramattrs$tzone = NULL
attributes(x[])
rm(x); gc()

ramorder.default Sorting: order R vector in-RAM and in-place

Description

Function ramorder will order the input vector in-place (without making a copy) and return the number of NAs found

Usage

## Default S3 method:
ramorder(x, i, has.na = TRUE, na.last = TRUE, decreasing = FALSE,
       stable = TRUE, optimize = c("time", "memory"), VERBOSE = FALSE, ...)
## Default S3 method:
mergeorder(x, i, has.na = TRUE, na.last = TRUE, decreasing = FALSE, ...)
## Default S3 method:
radixorder(x, i, has.na = TRUE, na.last = TRUE, decreasing = FALSE, ...)
## Default S3 method:
keyorder(x, i, keyrange=range(x, na.rm=has.na), has.na = TRUE, na.last = TRUE
       , decreasing = FALSE, ...)
## Default S3 method:
shellorder(x, i, has.na = TRUE, na.last = TRUE, decreasing = FALSE, stabilize=FALSE, ...)
Arguments

- **x**: an atomic R vector
- **i**: a integer vector with a permutation of positions in x (you risk a crash if you violate this)
- **keyrange**: an integer vector with two values giving the smallest and largest possible value in x, note that you should give this explicitly for best performance, relying on the default needs one pass over the data to determine the range
- **has.na**: boolean scalar telling ramorder whether the vector might contain NAs. Note that you risk a crash if there are unexpected NAs with has.na=FALSE
- **na.last**: boolean scalar telling ramorder whether to order NAs last or first. Note that 'boolean' means that there is no third option NA as in order
- **decreasing**: boolean scalar telling ramorder whether to order increasing or decreasing
- **stable**: set to false if stable ordering is not needed (may enlarge the set of ordering methods considered)
- **optimize**: by default ramorder optimizes for 'time' which requires more RAM, set to 'memory' to minimize RAM requirements and sacrifice speed
- **VERBOSE**: cat some info about chosen method
- **stabilize**: Set to TRUE for stabilizing the result of shellorder (for equal keys the order values will be sorted, this only works if i=1:n) to minimize RAM requirements and sacrifice speed
- **...**: ignored

Details

Function ramorder is a front-end to a couple of single-threaded ordering algorithms that have been carefully implemented to be fast with and without NAs.

The default is a mergeorder algorithm without copying (Sedgewick 8.4) for integer and double data which requires 2x the RAM of its input vector (character or complex data are not supported). Mergeorder is fast, stable with a reliable runtime.

For integer data longer than a certain length we improve on mergeorder by using a faster LSD radixorder algorithm (Sedgewick 10.5) that uses 2x the RAM of its input vector plus 65536+1 integers.

For booleans, logicals, integers at or below the resolution of smallint and for factors below a certain number of levels we use a key-index order instead of mergeorder or radix order (note that R has a (slower) key-index order in sort.list available with confusingly named method='radix' but the standard order does not leverage it for factors (2-11.1). If you call keyorder directly, you should provide a known 'keyrange' directly to obtain the full speed.

Finally the user can request a order method that minimizes memory use at the price of longer computation time with optimize='memory' – currently a shellorder.

Value

integer scalar with the number of NAs. This is always 0 with has.na=FALSE
Note

This function is called for its side-effects and breaks the functional programming paradigm. Use with care.

Author(s)

Jens Oehlschlägel

References


See Also

order, fforder, dforder, ramsort

Examples

n <- 50
x <- sample(c(NA, NA, 1:26), n, TRUE)
order(x)
i <- 1:n
ramorder(x, i)
i
x[i]

## Not run:
message("Note how the datatype influences sorting speed")
n <- 1e7
x <- sample(1:26, n, TRUE)

y <- as.double(x)
i <- 1:n
system.time(ramorder(y, i))

y <- as.integer(x)
i <- 1:n
system.time(ramorder(y, i))

y <- as.short(x)
i <- 1:n
system.time(ramorder(y, i))

y <- factor(letters)[x]
i <- 1:n
system.time(ramorder(y, i))

## End(Not run)
ramsort.default Sorting: Sort R vector in-RAM and in-place

Description

Function `ramsort` will sort the input vector in-place (without making a copy) and return the number of NAs found.

Usage

```r
## Default S3 method:
ramsort(x, has.na = TRUE, na.last = TRUE, decreasing = FALSE,
        optimize = c("time", "memory"), VERBOSE = FALSE, ...)
## Default S3 method:
mergesort(x, has.na = TRUE, na.last = TRUE, decreasing = FALSE, ...)
## Default S3 method:
radixsort(x, has.na = TRUE, na.last = TRUE, decreasing = FALSE, ...)
## Default S3 method:
keysort(x, keyrange = range(x, na.rm = has.na), has.na = TRUE,
        na.last = TRUE, decreasing = FALSE, ...)
## Default S3 method:
shellsort(x, has.na = TRUE, na.last = TRUE, decreasing = FALSE, ...)
```

Arguments

- `x` an atomic R vector
- `keyrange` an integer vector with two values giving the smallest and largest possible value in `x`, note that you should give this explicity for best performance, relying on the default needs one pass over the data to determine the range
- `has.na` boolean scalar telling `ramsort` whether the vector might contain NAs. Note that you risk a crash if there are unexpected NAs with `has.na`=FALSE
- `na.last` boolean scalar telling `ramsort` whether to sort NAs last or first. Note that `boolean` means that there is no third option NA as in `sort`
- `decreasing` boolean scalar telling `ramsort` whether to sort increasing or decreasing
- `optimize` by default `ramsort` optimizes for 'time' which requires more RAM, set to 'memory' to minimize RAM requirements and sacrifice speed
- `VERBOSE` cat some info about chosen method
- `...` ignored

Details

Function `ramsort` is a front-end to a couple of single-threaded sorting algorithms that have been carefully implemented to be fast with and without NAs.

The default is a mergesort algorithm without copying (Sedgewick 8.4) for integer and double data.
which requires 2x the RAM of its input vector (character or complex data are not supported). Merge-sort is fast, stable with a reliable runtime. For integer data longer than a certain length we improve on mergesort by using a faster LSD radix-sort algorithm (Sedgewick 10.5) that uses 2x the RAM of its input vector plus 65536+1 integers. For booleans, logicals, integers at or below the resolution of smallint and for factors below a certain number of levels we use a key-index sort instead of mergesort or radix sort (note that R has a (slower) key-index sort in sort.list available with confusingly named method='radix' but the standard sort does not leverage it for factors (2-11.1). If you call keysort directly, you should provide a known 'keyrange' directly to obtain the full speed. Finally the user can request a sort method that minimizes memory use at the price of longer computation time with optimize='memory' – currently a shellsort.

Value

integer scalar with the number of NAs. This is always 0 with has.na=FALSE

Note

This function is called for its side-effects and breaks the functional programming paradigm. Use with care.

Author(s)

Jens Oehlschlägel

References


See Also

sort, ffsort, dfsort, ramorder

Examples

```r
n <- 50
x <- sample(c(NA, NA, 1:26), n, TRUE)
  sort(x)
ramsort(x)
  x
```

```r
# Not run:
message("Note how the datatype influences sorting speed")
n <- 5e6
x <- sample(1:26, n, TRUE)

y <- as.double(x)
  system.time(ramsort(y))

y <- as.integer(x)
  system.time(ramsort(y))
```
### Description

Function `read.table.ffdf` reads separated flat files into `ffdf` objects, very much like (and using) `read.table`. It can also work with any convenience wrappers like `read.csv` and provides its own convenience wrapper (e.g. `read.csv.ffdf`) for R’s usual wrappers.

### Usage

```r
read.table.ffdf(
  x = NULL
, file, fileEncoding = ""
 , nrows = -1, first.rows = NULL, next.rows = NULL
, levels = NULL, appendLevels = TRUE
, FUN = "read.table", ...
, transFUN = NULL
, asffdf_args = list()
, BATCHBYTES =getOption("ffbatchbytes")
 , VERBOSE = FALSE
)
read.csv.ffdf(...)
read.csv2.ffdf(...)
read.delim.ffdf(...)
read.delim2.ffdf(...)```

### Arguments

- **x** NULL or an optional `ffdf` object to which the read records are appended. If this is provided, it defines crucial features that are otherwise determined during the 'first' chunk of reading: `vmodes`, `colnames`, `colClasses`, sequence of predefined `levels`. In order to also read the first chunk into such predefined `ffdf`, an `x` with 1 row is treated special: instead of appending the first row will be overwritten. This is necessary because we cannot provide `x` with zero rows (we cannot create `ff` vectors with zero elements).

- **file** the name of the file which the data are to be read from. Each row of the table appears as one line of the file. If it does not contain an `absolute` path, the file
name is relative to the current working directory, `getwd()`.
Tilde-expansion is performed where supported.
Alternatively, file can be a readable text-mode connection (which will be
opened for reading if necessary, and if so closed (and hence destroyed) at the
end of the function call).

**fileEncoding** character string: if non-empty declares the encoding used on a file (not a con-
nexion) so the character data can be re-encoded. See file.

**nrows** integer: the maximum number of rows to read in (includes first.rows in case a
'first' chunk is read) Negative and other invalid values are ignored.

**first.rows** integer: number of rows to be read in the first chunk, see details. Default is the
value given at next.rows or 1e3 otherwise. Ignored if x is given.

**next.rows** integer: number of rows to be read in further chunks, see details. By default
calculated as BATCHBYTES \% \% sum(.rambytes[vmode(x)])

**levels** NULL or an optional list, each element named with col.names of factor columns
specifies the levels Ignored if x is given.

**appendLevels** logical. A vector of permissions to expand levels for factor columns. Recycled
as necessary, or if the logical vector is named, unspecified values are taken to be
TRUE. Ignored during processing of the 'first' chunk.

**FUN** character: name of a function that is called for reading each chunk, see read.table,
read.csv, etc.

... further arguments, passed to FUN in read.table.ffdf, or passed to read.table.ffdf
in the convenience wrappers

**transFUN** NULL or a function that is called on each data.frame chunk after reading with
FUN and before further processing (for filtering, transformations etc.)

**asffdf_args** further arguments passed to as.ffdf when converting the data.frame of the
first chunk to ffdf. Ignored if x is given.

**BATCHBYTES** integer: bytes allowed for the size of the data.frame storing the result of read-
ing one chunk. Default getOption("ffbatchbytes").

**VERBOSE** logical: TRUE to verbose timings for each processed chunk (default FALSE)

Details

read.table.ffdf has been designed to read very large (many rows) separated flatfiles in row-
chunks and store the result in a ffdf object on disk, but quickly accessible via ff techniques.
The first chunk is read with a default of 1000 rows, for subsequent chunks the number of rows is
calculated to not require more RAM than getOption("ffbatchbytes"). The following could be
indications to change the parameter first.rows:

1. set first.rows=-1 to read the complete file in one go (requires enough RAM)
2. set first.rows to a smaller number if the pre-allocation of RAM for the first chunk with
   parameter nrows in read.table is too large, i.e. with many columns on machine with little
   RAM.
3. set first.rows to a larger number if you expect better factor level ordering (factor levels are
   sorted in the first chunk, but not at subsequent chunks, however, factor level ordering can be
   fixed later, see below).
By default the `ffdf` object is created on the fly at the end of reading the 'first' chunk, see argument `first.rows`. The creation of the `ffdf` object is done via `as.ffdf` and can be finetuned by passing argument `asffdf_args`. Even more control is possible by passing in a `ffdf` object as argument `x` to which the read records are appended.

`read.table.ffdf` has been designed to behave as much like `read.table` as possible. However, note the following differences:

1. Arguments 'colClasses' and 'col.names' are now enforced also during 'next.rows' chunks. For example giving `colclasses=NA` will force that no colClasses are derived from the `first.rows` respective from the `ffdf` object in parameter `x`.
2. `colClass` 'ordered' is allowed and will create an `ordered` factor
3. Character vector are not supported, character data must be read as one of the following colClasses: 'Date', 'POSIXct', 'factor', 'ordered'. By default character columns are read as factors. Accordingly arguments 'as.is' and 'stringsAsFactors' are not allowed.
4. The sequence of `levels.ff` from chunked reading can depend on chunk size: by default new levels found on a chunk are appended to the levels found in previous chunks, no attempt is made to sort and recode the levels during chunked processing. Levels can be sorted and recoded most efficiently after all records have been read using `sortLevels`.
5. The default for argument 'comment.char' is `''` even for those FUN that have a different default. However, explicit specification of 'comment.char' will have priority.

**Value**

An `ffdf` object. If created during the 'first' chunk pass, it will have one physical component per virtual column.

**Note**

Note that using the 'skip' argument still requires to read the file from beginning in order to count the lines to be skipped. If you first read part of the file in order to understand its structure and then want to continue, a more efficient solution that using 'skip' is opening a `file connection` and pass that to argument 'file'. `read.table.ffdf` does the same in order to skip efficiently over previously read chunks.

**Author(s)**

Jens Oehlschlägel, Christophe Dutang

**See Also**

`write.table.ffdf`, `read.table`, `ffdf`

**Examples**

```r
x <- data.frame(log=rep(c(FALSE, TRUE), length.out=26),
                 int=1:26, dbl=1:26 + 0.1,
                 fac=factor(letters),
                 ord=ordered(LETTERS),
                 dct=Sys.time()+1:26,
                 dat=seq(as.Date("1910/1/1"), length.out=26, by=1))
x <- x[c(13:1, 13:1),]
csvfile <- tempPathFile(path=getOption("fftempdir"), extension="csv")
```
write.csv(x, file=csvfile, row.names=FALSE)

y <- read.csv(file=csvfile, header=TRUE)
y
cat("Read csv with header
"
ffx <- read.csv.ffdf(file=csvfile, header=TRUE)
ffx
sapply(ffx[,], class)

message("NOTE that read.table fails for ordered factors, this is fixed in read.table.ffdf")
try(read.csv(file=csvfile, header=TRUE, colClasses=c(ord="ordered")))

# TODO could fix this with the following two commands (Gabor Grothendieck)
# but does not know what bad side-effects this could have
# setOldClass("ordered")
# setAs("character", "ordered", function(from) ordered(from))
y <- read.csv(file=csvfile, header=TRUE, colClasses=c(dct="POSIXct", dat="Date"))
y
ffx <- read.csv.ffdf(file=csvfile, header=TRUE
, colClasses=c(ord="ordered", dct="POSIXct", dat="Date"))
ffx
sapply(ffx[,], class)

message("NOTE that reading in chunks can change the sequence of levels and thus the coding")
message("(Sorting levels during chunked reading can be too expensive)"
ffx <- read.csv.ffdf(file=csvfile, header=TRUE
, colClasses=c(ord="ordered", dct="POSIXct", dat="Date"), first.rows=6, next.rows=10
, VERbose=TRUE)
y <- ffx$fac[]
print(levels(y))
data.frame(values=as.character(y), codes=as.integer(y))

message("If we don't know the levels we can sort them after reading")
message("(Will rewrite all factor codes)"
message("NOTE that you MUST assign the return value of sortLevels()")
ffx <- sortLevels(ffx)
y <- ffx$fac[]
print(levels(y))
data.frame(values=as.character(y), codes=as.integer(y))

message("If we KNOW the levels we can fix levels upfront")
ffx <- read.csv.ffdf(file=csvfile, header=TRUE
, colClasses=c(ord="ordered", dct="POSIXct", dat="Date"), first.rows=6, next.rows=10, levels=list(fac=letters, ord=LETTERS))
y <- ffx$fac[]
print(levels(y))
data.frame(values=as.character(y), codes=as.integer(y))

message("Or we inspect a sufficiently large chunk of data and use those")
ffx1 <- read.csv.ffdf(file=csvfile, header=FALSE, skip=1+nrow(ffx1), VERbose=TRUE)
message("We can check for unexpected factor levels, say we only allowed a:1")
ffx <- read.csv.ffdf(file=csvfile, header=TRUE,
  colClasses=c("ordered", "POSIXct", "Date"),
  levels=list(fac=letters[1:12], ord=LETTERS[1:12]), appendLevels=FALSE)
sapply(colnames(ffx), function(i)sum(is.na(ffx[[i]][i])))

message("We can fine-tune the creation of the ffdf:")
message("- let's create the ff files outside of fftempdir")
message("- let's reduce required disk space and thus file.system cache RAM")
vmode(ffx)
message("By default we had record size")
sum(.ffbytes[vmode(ffx)])

ffy <- read.csv.ffdf(file=csvfile, header=TRUE,
  colClasses=c("ordered", "POSIXct", "Date"),
  asffdf_args=list(
    vmode = c("boolean", "byte", "single", "nibble",
     "integer", "double", "single"),
    col_args=list(pattern = "/.csv") # create in getwd() with prefix csv
  )
)
vmode(ffy)
message("This recordsize is more than 50% reduced")
sum(.ffbytes[vmode(ffy)])

message("Don't forget to wrap-up files that are not in fftempdir")
delete(ffy); rm(ffy)
message("It's a good habit to also wrap-up temporary stuff (or at least know how this is done)")
rm(ffx); gc()

fwffile <- tempfile()
cat(file=fwffile, "123456", "987654", sep="\n")
x <- read.fwf(fwffile, widths=c(1,2,3))   # 1 23 456 | 9 87 654
y <- read.table.ffdf(file=fwffile, FUN="read.fwf", widths=c(1,2,3))
stopifnot(identical(x, y[1,]))
x <- read.fwf(fwffile, widths=c(1,1,-2,3)) # 1 456 | 9 654
y <- read.table.ffdf(file=fwffile, FUN="read.fwf", widths=c(1,-2,3))
stopifnot(identical(x, y[1,]))
unlink(fwffile)

cat(file=fwffile, "123", "987654", sep="\n")
x <- read.fwf(fwffile, widths=c(1,0,2,3))   # 1 NA 23 NA | 9 NA 87 654
y <- read.table.ffdf(file=fwffile, FUN="read.fwf", widths=c(1,0,2,3))
stopifnot(identical(x, y[1,]))
unlink(fwffile)

cat(file=fwffile, "123456", "987654", sep="\n")
x <- read.fwf(fwffile, widths=list(c(1,0,2,3), c(2,2,2))) # 1 NA 23 456 98 76 54
y <- read.table.ffdf(file=fwffile, FUN="read.fwf", widths=list(c(1,0,2,3), c(2,2,2)))
stopifnot(identical(x, y[1,]))
unlink(fwffile)
Description

Simple low-level interface for reading and writing vectors from ff files.

Usage

read.ff(x, i, n)
write.ff(x, i, value, add = FALSE)
readwrite.ff(x, i, value, add = FALSE)

Arguments

x an ff object
i a start position in the ff file
n number of elements to read
value vector of elements to write
add TRUE if the values should rather increment than overwrite at the target positions

Details

readwrite.ff combines the effects of read.ff and write.ff in a single operation: it retrieves the old values starting from position i before changing them. getset.ff will maintain na.count.

Value

read.ff returns a vector of values, write.ff returns the 'changed' ff object (like all assignment functions do) and readwrite.ff returns the values at the target position. More precisely readwrite.ff(x, i, value, add=FALSE) returns the old values at the position i while readwrite.ff(x, i, value, add=TRUE) returns the incremented values of x.

Note

read.ff, write.ff and readwrite.ff are low level functions that do not support ramclass and ramattribs and thus will not give the expected result with factor and POSIXct

Author(s)

Jens Oehlschlägel
See Also

`getset.ff` for low-level scalar access and `[..ff` for high-level access

Examples

```r
x <- ff(0, length=12)
read.ff(x, 3, 6)
write.ff(x, 3, rep(1, 6))
  x
write.ff(x, 3, rep(1, 6), add=TRUE)
  x
readwrite.ff(x, 3, rep(1, 6), add=TRUE)
readwrite.ff(x, 3, rep(1, 6))
  x
rm(x); gc()
```
Examples

regtest.fforder()

## Not run:
n <- 5e6
message("performance comparison at n=" , n, "")

message("sorting doubles")
x <- y <- as.double(runif(n))

x[] <- y
system.time(sort(x))[3]
x[] <- y
system.time(shellsort(x))[3]
x[] <- y
system.time(shellsort(x, has.na=FALSE))[3]
x[] <- y
system.time(mergesort(x))[3]
x[] <- y
system.time(mergesort(x, has.na=FALSE))[3]

x[] <- y
system.time(sort(x, decreasing=TRUE))[3]
x[] <- y
system.time(shellsort(x, decreasing=TRUE))[3]
x[] <- y
system.time(shellsort(x, decreasing=TRUE, has.na=FALSE))[3]
x[] <- y
system.time(mergesort(x, decreasing=TRUE))[3]
x[] <- y
system.time(mergesort(x, decreasing=TRUE, has.na=FALSE))[3]

x <- y <- as.double(sample(c(rep(NA, n/2), runif(n/2))))

x[] <- y
system.time(sort(x))[3]
x[] <- y
system.time(shellsort(x))[3]
x[] <- y
system.time(mergesort(x))[3]

x[] <- y
system.time(sort(x, decreasing=TRUE))[3]
x[] <- y
system.time(shellsort(x, decreasing=TRUE))[3]
x[] <- y
system.time(mergesort(x, decreasing=TRUE))[3]

x <- y <- sort(as.double(runif(n)))
x[] <- y
system.time(sort(x))  # only here R is faster because R checks for being sorted
x[] <- y
system.time(shellsort(x))[3]
x[] <- y
system.time(shellsort(x, has.na=FALSE))[3]
x[] <- y
system.time(mergesort(x))[3]
x[] <- y
system.time(mergesort(x, has.na=FALSE))[3]

x[] <- y
system.time(sort(x, decreasing=TRUE))[3]
x[] <- y
system.time(shellsort(x, decreasing=TRUE))[3]
x[] <- y
system.time(shellsort(x, decreasing=TRUE, has.na=FALSE))[3]
x[] <- y
system.time(mergesort(x, decreasing=TRUE))[3]
x[] <- y
system.time(mergesort(x, decreasing=TRUE, has.na=FALSE))[3]

y <- rev(y)
x[] <- y
system.time(sort(x))[3]
x[] <- y
system.time(shellsort(x))[3]
x[] <- y
system.time(shellsort(x, has.na=FALSE))[3]
x[] <- y
system.time(mergesort(x))[3]
x[] <- y
system.time(mergesort(x, has.na=FALSE))[3]

x[] <- y
system.time(sort(x, decreasing=TRUE))[3]
x[] <- y
system.time(shellsort(x, decreasing=TRUE))[3]
x[] <- y
system.time(shellsort(x, decreasing=TRUE, has.na=FALSE))[3]
x[] <- y
system.time(mergesort(x, decreasing=TRUE))[3]
x[] <- y
system.time(mergesort(x, decreasing=TRUE, has.na=FALSE))[3]

rm(x,y)

message("ordering doubles")

x <- as.double(runif(n))
system.time(order(x))[3]
i <- 1:n
system.time(shellorder(x, i))[3]

i <- 1:n
system.time(shellorder(x, i, stabilize=TRUE))[3]

i <- 1:n
system.time(mergeorder(x, i))[3]

x <- as.double(sample(c(rep(NA, n/2), runif(n/2))))
system.time(order(x))[3]

i <- 1:n
system.time(shellorder(x, i))[3]

i <- 1:n
system.time(shellorder(x, i, stabilize=TRUE))[3]

i <- 1:n
system.time(mergeorder(x, i))[3]

x <- as.double(sort(runif(n)))
system.time(order(x))[3]

i <- 1:n
system.time(shellorder(x, i))[3]

i <- 1:n
system.time(shellorder(x, i, stabilize=TRUE))[3]

i <- 1:n
system.time(mergeorder(x, i))[3]

x <- rev(x)
system.time(order(x))[3]

i <- 1:n
system.time(shellorder(x, i))[3]

i <- 1:n
system.time(shellorder(x, i, stabilize=TRUE))[3]

i <- 1:n
system.time(mergeorder(x, i))[3]

x <- as.double(runif(n))
system.time(order(x, decreasing=TRUE))[3]

i <- 1:n
system.time(shellorder(x, i, decreasing=TRUE))[3]

i <- 1:n
system.time(shellorder(x, i, decreasing=TRUE, stabilize=TRUE))[3]

i <- 1:n
system.time(mergeorder(x, i, decreasing=TRUE))[3]

x <- as.double(sample(c(rep(NA, n/2), runif(n/2))))
system.time(order(x, decreasing=TRUE))[3]

i <- 1:n
system.time(shellorder(x, i, decreasing=TRUE))[3]

i <- 1:n
system.time(shellorder(x, i, decreasing=TRUE, stabilize=TRUE))[3]

i <- 1:n
system.time(mergeorder(x, i, decreasing=TRUE))[3]
x <- as.double(sort(runif(n)))
system.time(order(x, decreasing=TRUE))[3]
i <- 1:n
system.time(shellorder(x, i, decreasing=TRUE))[3]
i <- 1:n
system.time(shellorder(x, i, decreasing=TRUE, stabilize=TRUE))[3]
i <- 1:n
system.time(mergeorder(x, i, decreasing=TRUE))[3]

x <- rev(x)
system.time(order(x, decreasing=TRUE))[3]
i <- 1:n
system.time(shellorder(x, i, decreasing=TRUE))[3]
i <- 1:n
system.time(shellorder(x, i, decreasing=TRUE, stabilize=TRUE))[3]
i <- 1:n
system.time(mergeorder(x, i, decreasing=TRUE))[3]

keys <- c("short","ushort")
for (v in c("integer", keys)){
  if (v %in% keys){
    k <- .vmax[v] -.vmin[v]+1L
    if (is.na(.vNA[v])){
      y <- sample(c(rep(NA, k), .vmin[v]:.vmax[v]), n, TRUE)
    } else{
      y <- sample(.vmin[v]:.vmax[v], n, TRUE)
    }
  } else{
    k <- .Machine$integer.max
    y <- sample(k, n, TRUE)
  }

  message("sorting ",v)
  x <- y
  message("sort(x )", system.time(sort(x))[3])
  x <- y
  message("shellsort(x )", system.time(shellsort(x))[3])
  x <- y
  message("mergesort(x )", system.time(mergesort(x))[3])
  x <- y
  message("radixsort(x )", system.time(radixsort(x))[3])
  if (v %in% keys){
    x <- y
    message("keysort(x )", system.time(keysor)
x <- y
message("shellsort(x, has_na=FALSE ") , system.time(shellsort(x, has_na=FALSE) )[3])
x <- y
message("mergesort(x, has_na=FALSE ") , system.time(mergesort(x, has_na=FALSE) )[3])
x <- y
message("radixsort(x, has_na=FALSE ") , system.time(radixsort(x, has_na=FALSE) )[3])
if (v %in% keys){
x <- y
message("keysort(x, has_na=FALSE ") , system.time(keys(x, has_na=FALSE) )[3])
x <- y
message("keysort(x, has_na=FALSE, keyrange=c(.vmin[v], .vmax[v])) ", system.time(keys(x, has_na=FALSE, keyrange=c(.vmin[v], .vmax[v])) ))[3]
}

message("ordering", v)
x[] <- y
i <- 1:n
message("order(x) ", system.time(order(x) )[3])
x[] <- y
i <- 1:n
message("shellsorder(x, i) ", system.time(shellsorder(x, i) )[3])
x[] <- y
i <- 1:n
message("mergeorder(x, i) ", system.time(mergeorder(x, i) )[3])
x[] <- y
i <- 1:n
message("radixorder(x, i) ", system.time(radixorder(x, i) )[3])
if (v %in% keys){
x[] <- y
i <- 1:n
message("keyorder(x, i) ", system.time(keyorder(x, i) )[3])
x[] <- y
i <- 1:n
message("keyorder(x, i, keyrange=c(.vmin[v], .vmax[v])) ", system.time(keyorder(x, i, keyrange=c(.vmin[v], .vmax[v])) ))[3]
}

if (!is.na(.vNA[v])){
x[] <- y
i <- 1:n
message("shellsorder(x, i, has_na=FALSE ", system.time(shellsorder(x, i, has_na=FALSE) )[3])
x[] <- y
i <- 1:n
message("mergeorder(x, i, has_na=FALSE ", system.time(mergeorder(x, i, has_na=FALSE) )[3])
x[] <- y
i <- 1:n
message("radixorder(x, i, has_na=FALSE ", system.time(radixorder(x, i, has_na=FALSE) )[3])
if (v %in% keys){
x[] <- y
i <- 1:n
message("keyorder(x, i, keyrange=c(.vmin[v], .vmax[v])) ", system.time(keyorder(x, i, keyrange=c(.vmin[v], .vmax[v])) ))[3]
}
repnam

Replicate with names

Description

Function repnam replicates its argument to the desired length, either by simply replicating or - if it has names - by replicating the default and matching the argument by its names.

Usage

repnam(argument, names = NULL, len=length(names), default = list(NULL))

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>argument</td>
<td>a named or non-named vector or list to be replicated</td>
</tr>
<tr>
<td>names</td>
<td>NULL or a character vector of names to which the argument names are matched</td>
</tr>
<tr>
<td>len</td>
<td>the desired length (required if names is not given)</td>
</tr>
<tr>
<td>default</td>
<td>the desired default which is replicated in case names are used (the default list(NULL) is suitable for a list argument)</td>
</tr>
</tbody>
</table>

Value

an object like argument or default having length len

Note

This is for internal use, e.g. to handle argument colClasses in read.table.ffdf

Author(s)

Jens Oehlschlägel

See Also

rep, vector, repfromto
Examples

message("a list example")
repnam(list(y=c(1,2), z=3), letters)
repnam(list(c(1,2), 3), letters)

message("a vector example")
repnam(c(y=1, z=3), letters, default=NA)
repnam(c(1, 3), letters, default=NA)

sortLevels

Factor level manipulation

Description

appendLevels combines levels without sorting such that levels of the first argument will not require re-coding.
recodeLevels is a generic for recoding a factor to a desired set of levels - also has a method for large ff objects
sortLevels is a generic for level sorting and recoding of single factors or of all factors of a ffdf dataframe.

Usage

appendLevels(...)
recodeLevels(x, lev)
## S3 method for class 'factor'
recodeLevels(x, lev)
## S3 method for class 'ff'
recodeLevels(x, lev)
sortLevels(x)
## S3 method for class 'factor'
sortLevels(x)
## S3 method for class 'ff'
sortLevels(x)
## S3 method for class 'ffdf'
sortLevels(x)

Arguments

... character vector of levels or is.factor objects from which the level attribute is taken
x a factor or ff factor or a ffdf dataframe (sortLevels only)
lev a character vector of levels
Details

When reading a long file with categorical columns the final set of factor levels is only known once the complete file has been read. When a file is so large that we read it in chunks, the new levels need to be added incrementally. `bind.data.frame` sorts combined levels, which requires recoding. For `ff` factors this would require recoding of all previous chunks at the next chunk - potentially on disk, which is too expensive. Therefore `read.table.ffdf` will simply appendLevels without sorting, and the `recodeLevels` and `sortLevels` generics provide a convenient means for sorting and recoding levels after all chunks have been read.

Value

`appendLevels` returns a vector of combined levels, `recodeLevels` and `sortLevels` return the input object with changed levels. Do read the note!

Note

You need to re-assign the return value not only for ram- but also for ff-objects. Remember ff’s hybrid copying semantics: `LmWarn`. If you forget to re-assign the returned object, you will end up with ff objects that have their integer codes re-coded to the new levels but still carry the old levels as a virtual attribute.

Author(s)

Jens Oehlschlägel

See Also

`read.table.ffdf`, `levels.ff`

Examples

```r
message("Let's create a factor with little levels")
x <- ff(letters[4:6], levels=letters[4:6])
message("Let's interpret the same ff file without levels in order to see the codes")
y <- x
levels(y) <- NULL

levels(x)
data.frame(factor=x[], codes=y[])

levels(x) <- appendLevels(levels(x), letters)
levels(x)
data.frame(factor=x[], codes=y[])

x <- sortLevels(x) # implicit recoding is chunked were necessary
levels(x)
data.frame(factor=x[], codes=y[])

message("NEVER forget to reassign the result of recodeLevels or sortLevels, look at the following mess")
recodeLevels(x, rev(levels(x)))
```
message("NOW the codings have changed, but not the levels, the result is wrong data")
levels(x)
data.frame(factor=x[], codes=y[])

rm(x); gc()

## Not run:
n <- 5e7

message("reading a factor from a file ist as fast ...")
system.time(
  fx <- ff(factor(letters[1:25]), length=n)
)
system.time(x <- fx[])
str(x)
rm(x); gc()

message("... as creating it in-RAM (R-2.11.1) which is theoretically impossible ...")
system.time(
  x <- integer(n)
x[] <- 1:25
  levels(x) <- letters[1:25]
  class(x) <- "factor"
)
str(x)
rm(x); gc()

message("... but is possible if we avoid some unnecessary copying that is triggered by assignment functions")
system.time(
  x <- integer(n)
x[] <- 1:25
  setattr(x, "levels", letters[1:25])
  setattr(x, "class", "factor")
)
str(x)
rmm(x); gc()

rm(n)

## End(Not run)

---

**splitPathFile**  
*Analyze pathfile-strings*

**Description**

splitPathFile splits a vector of pathfile-strings into path- and file-components without loss of
information. unsplitPathFile restores the original pathfile-string vector. standardPathFile standardizes a vector of pathfile-strings: backslashes are replaced by slashes, except for the first two leading backslashes indicating a network share. tempPathFile returns - similar to tempfile - a vector of filenames given path(s) and file-prefix(es) and an optional extension. fftempFile returns - similar to tempPathFile - a vector of filenames following a vector of pathfile patterns that are interpreted in a ff-specific way.

Usage

```r
splitPathFile(x)
unsplitPathFile(splitted)
standardPathFile(x)
tempPathFile(splitted=NULL, path=splitted$path, prefix=splitted$file, extension=NULL)
fftempFile(x)
```

Arguments

- `x`: a character vector of pathfile strings
- `splitted`: a return value from `splitPathFile`
- `path`: a character vector of path components
- `prefix`: a character vector of file components
- `extension`: optional extension like "csv" (or NULL)

Details

dirname and basename remove trailing file separators and therefore cannot distinguish pathfile string that contains ONLY a path from a pathfile string that contains a path AND file. Therefore `file.path(dirname(pathfile), basename(pathfile))` cannot always restore the original pathfile string.

splitPathFile decomposes each pathfile string into three parts: a path BEFORE the last file separator, the file separator, the filename component AFTER the last file separator. If there is no file separator in the string, splitPathFile tries to guess whether the string is a path or a file component: ".", ".." and "~" are recognized as path components. No tilde expansion is done, see `path.expand`. Backslashes are converted to the current `.Platform$file.sep` using splitPathFile except for the first two leading backslashes indicating a network share.

unsplitPathFile restores the original pathfile-string vector up to translated backslashes.

tempPathFile internally uses tempfile to create its filenames, if an extension is given it repeats filename creation until none of them corresponds to an existing file.

fftempFile takes a path-prefix pattern as input, splits it, will replace an empty path by `getOption("fftempdir")` and will use `getOption("ffextension")` as extension.

Value

A list with components

- `path`: a character vector of path components
- `fsep`: a character vector of file separators or ""
- `file`: a character vector of file components
Note

There is no guarantee that the path and file components contain valid path- or file-names. Like 
`basename`, `splitPathFile` can return ".", ".\" or even "", however, all these make sense as a prefix in `tempPathFile`.

Author(s)

Jens Oehlschlägel

See Also

tempfile, dirname, basename, file.path

Examples

```r
dirname <- dirname(pathfile)
basnam <- basename(pathfile)

db <- file.path(dirnam,basnam)
ident = gsub("\\\\","/",db) == gsub("\\\\","/",pathfile)
sum(ident)

do.call("data.frame", c(list(ident=ident, pathfile=pathfile
, dirnam=dirname, basnam=basename), splitted))

## Not run:
message("show the difference between tempfile and fftempfile")
do.call("data.frame", c(list(ident=ident, pathfile=pathfile, dirnam=dirname, basnam=basename
, splitted, list(filename=tempPathFile(splitted), fftempfile=fftempfile(pathfile))))

message("for a single string splitPathFile is slower,
for vectors of strings it scales much better than dirname+basename")

system.time(for (i in 1:1000){
d <- dirname(pathfile)
b <- basename(pathfile)
})

system.time(for (i in 1:1000){
s <- splitPathFile(pathfile)
})

len <- c(1,10,100,1000)
timings <- matrix(0, 2, length(len), dimnames=list(c("dir.base.name", "splitPathFile"), len))
for (j in seq(along=len)){

```
swap

Reading and writing in one operation (high-level)

Description

The generic swap combines $x[i]$ and $x[i] \leftarrow \text{value}$ in a single operation.

Usage

\[
\text{swap}(x, \text{value}, \ldots)
\]

\[
\text{## S3 method for class 'ff'}
\]

\[
\text{swap}(x, \text{value}, i, \text{add} = \text{FALSE}, \text{pack} = \text{FALSE}, \ldots)
\]

\[
\text{## S3 method for class 'ff_array'}
\]

\[
\text{swap}(x, \text{value}, \ldots, \text{bydim} = \text{NULL}, \text{drop} = \text{getOption("ffdrop")}, \text{add} = \text{FALSE}, \text{pack} = \text{FALSE})
\]

\[
\text{## Default S3 method:}
\]

\[
\text{swap}(x, \text{value}, \ldots, \text{add} = \text{FALSE})
\]

Arguments

- **x**
  - a ff or ram object
- **value**
  - the new values to write, possibly recycled, see \[.ff\]
- **i**
  - index information, see \[.ff\]
- **...**
  - missing OR up to length(dim(x)) index expressions OR (ff only) hi objects
- **drop**
  - logical scalar indicating whether array dimensions shall be dropped
- **bydim**
  - how to interpret vector to array data, see \[.ff\]
- **add**
  - TRUE if the values should rather increment than overwrite at the target positions, see \texttt{readwrite.ff}
- **pack**
  - FALSE to prevent rle-packing in hybrid index preprocessing, see \texttt{as.hi}
Details

\[
y \leftarrow \text{swap}(x, \text{value}, i, \text{add}=\text{FALSE}, ...)\]

is a shorter and more efficient version of

\[
y \leftarrow x[i, \text{add}=\text{FALSE}, ...] \\
x[i, \text{add}=\text{FALSE}, ...] \leftarrow \text{value}
\]

and

\[
y \leftarrow \text{swap}(x, \text{value}, i, \text{add}=\text{TRUE}, ...)
\]

is a shorter and more efficient version of

\[
y \leftarrow x[i, \text{add}=\text{TRUE}, ...] \\
y \leftarrow y + \text{value} \\
x[i, \text{add}=\text{FALSE}, ...] \leftarrow y
\]

Value

Values at the target positions. More precisely \text{swap}(x, \text{value}, i, \text{add}=\text{FALSE}) returns the old values at the position \(i\) while \text{swap}(x, \text{value}, i, \text{add}=\text{TRUE}) returns the incremented values of \(x\).

Note

Note that \text{swap.default} changes the object in its parent frame and thus violates R’s usual functional programming logic. When using \text{add=TRUE}, duplicated index positions should be avoided, because \text{ff} and \text{ram} objects behave differently:

\[
\text{swap.ff}(x, 1, \text{c}(3,3), \text{add}=\text{TRUE}) \\
\text{# will increment } x \text{ at position 3 TWICE by 1, while} \\
\text{swap.default}(x, 1, \text{c}(3,3), \text{add}=\text{TRUE}) \\
\text{# will increment } x \text{ at position 3 just ONCE by 1}
\]

Author(s)

Jens Oehlschlägel

See Also

[.ff, add, readwrite.ff, getset.ff, LimWarn]

Examples

\[
x \leftarrow \text{ff}("a", \text{levels}=\text{letters}, \text{length}=52) \\
y \leftarrow \text{swap}(x, "b", \text{sample}(\text{length}(x), 26))
\]
Description

Check if an object is inherently symmetric (its structure, not its data)

Usage

\[
\text{symmetric}(x, \ldots)
\]

## S3 method for class 'ff'
\[
\text{symmetric}(x, \ldots)
\]

## Default S3 method:
\[
\text{symmetric}(x, \ldots)
\]

## S3 method for class 'dist'
\[
\text{symmetric}(x, \ldots)
\]

Arguments

- \(x\) an \(ff\) or \(ram\) object
- \(\ldots\) further arguments (not used)

Details

\(ff\) matrices can be declared symmetric at creation time. Compatibility function \(\text{symmetric.default}\) returns \(FALSE\), \(\text{symmetric.dist}\) returns \(TRUE\).

Value

TRUE or FALSE

Author(s)

Jens Oehlschlägel

See Also

\textit{symmetric, ff, dist, isSymmetric}

Examples

\[
\text{symmetric(matrix(1:16, 4, 4))}
\]
\[
\text{symmetric(dist(rnorm(1:4)))}
\]
symmIndex2vectorIndex  Array: make vector positions from symmetric array index

Description
make vector positions from (non-symmetric) array index respecting ‘dim’ and ‘fixdiag’

Usage
symmIndex2vectorIndex(x, dim, fixdiag = NULL)

Arguments
- x: a matrix[,1:2] with matrix subscripts
- dim: the dimensions of the symmetric matrix
- fixdiag: NULL assumes free diagonal, any value assumes fixed diagonal

Details
With ‘fixdiag = NULL’

Value
a vector of indices in 1:prod(dim(x))

Author(s)
Jens Oehlschlägel

See Also
arrayIndex2vectorIndex

Examples
symmIndex2vectorIndex(rbind(  c(1,1),  ,c(1,10),  ,c(10,1),  ,c(10,10))), dim=c(10,10))
symmIndex2vectorIndex(rbind(  c(1,1),  ,c(1,10),  ,c(10,1),  ,c(10,10))), dim=c(10,10), fixdiag=1)
**unclass**

*Unclassed assignment*

**Description**

With `unclass` you can circumvent class dispatch on the assignment operator.

**Usage**

```
unclass(x) <- value
```

**Arguments**

- `x`: some object
- `value`: the value to be assigned

**Value**

the modified object

**Author(s)**

Jens Oehlschlägel

**See Also**

`unclass`, `undim`

**Examples**

```
x <- factor(letters)
unclass(x)[1:3] <- 1L
x
```

---

**undim**

*Undim*

**Description**

`undim` returns its input with the dim attribute removed.

**Usage**

```
undim(x)
```
Arguments

\( x \) an object

Value

\( x \) without \texttt{dim} attribute

Author(s)

Jens Oehlschlägel

See Also

\texttt{unclass<-, unclass, unname, dim}

Examples

\begin{verbatim}
x <- matrix(1:12, 3)
x
undim(x)
\end{verbatim}

\begin{verbatim}
unsort
\end{verbatim}

\textit{Hybrid Index, internal utilities}

Description

Non-documented internal utilities that might change

Usage

\begin{verbatim}
unsort(x, ix)
unsort.hi(x, index)
unsort.ahi(x, index, ixre = any(sapply(index, function(i) {
  if (is.null(i$ix)) {
    if (i$re) TRUE else FALSE
  } else {
    TRUE
  }
})
})), ix = lapply(index, function(i) {
  if (is.null(i$ix)) {
    if (i$re)
      orig <- rev(1:poslength(i))
    else orig <- 1:poslength(i)
  } else {
    orig <- i$ix
  }
})
\end{verbatim}
**Arguments**

- `x`  
- `ix`  
- `ixre`  
- `index`  
- `maxindex`  
- `names`  

**Details**

These are utility functions for restoring original order after sorting. For now we 'mimic' the intuitive but wrong argument order of `match()` which should rather have the 'table' argument as its first argument, then one could properly method-dispatch on the type of table. We might change to proper 'unsort' generic, but then we have to change argument order.

**Value**

undefined

**Author(s)**

Jens Oehlschl"agel

**See Also**

`hi, as.hi`

---

**Description**

`update.ff` copies updates one ff object with the content of another object.

**Usage**

```r
## S3 method for class 'ff'
update(object, from, delete = FALSE, bydim = NULL, fromdim = NULL,
       BATCHSIZE = .Machine$integer.max, BATCHBYTES = getOption("ffbatchbytes")
       , VERBOSE = FALSE, ...)
## S3 method for class 'ffdf'
update(object, from, ...)
```
Arguments

- **object**: an ff object to which to update
- **from**: an object from which to update
- **delete**: NA for quick update with file-exchange, TRUE for quick update with deleting the 'from' object after the update, can speed up updating significantly
- **bydim**: how to interpret the content of the object, see `ff`
- **fromdim**: how to interpret the content of the 'from' object, see `ff`
- **BATCHSIZE**: BATCHSIZE
- **BATCHBYTES**: BATCHBYTES
- **VERBOSE**: VERBOSE
- **...**: further arguments

Details

If the source object is `ff` and not `delete=FALSE` then instead of slow copying we - if possible - try to swap and rename the files behind the ff objects. Quick update requires that the two `ff` objects are `vectorCompatible`, that both don’t use `vw`, that they have identical `maxlength` and identical `levels.ff`.

Value

An ff object like the input 'object' updated with the content of the 'from' object.

Note

You don’t have a guarantee that with `delete=TRUE` the 'from' object gets deleted or with `delete=NA` the 'from' objects carries the content of 'object'. Such expectations only turn true if really a quick update was possible.

Author(s)

Jens Oehlschlägel

See Also

`ff`, `clone`, `ffvecapply`, `vectorCompatible`, `filename`

Examples

```r
x <- ff(1:100)
y <- ff(-1:100)
message("You should make it a habit to re-assign the return value of update although this is not needed currently."")
x <- update(x, from=y)
x y x[] <- 1:100
```
vecprint

Print beginning and end of big vector

Description

Print beginning and end of big vector

Usage

vecprint(x, maxlen = 16, digits = getOption("digits"))

## S3 method for class 'vecprint'
print(x, quote = FALSE, ...)

Arguments

x a vector

maxlen max number of elements for printing
digits see format

quote see print

... see print

Value

a list of class 'vecprint' with components

subscript a list with two vectors of subscripts: vector begin and vector end

eexample the extracted example vector as.character including separator

sep the row separator ":"
Author(s)

Jens Oehlschlägel

See Also

matprint

Examples

vecprint(10000:1)

---

vector.vmode Create vector of virtual mode

Description

vector.vmode creates a vector of a given vmode and length

Usage

vector.vmode(vmode = "logical", length = 0)
boolean(length = 0)
quad(length = 0)
nibble(length = 0)
byte(length = 0)
ubyte(length = 0)
short(length = 0)
ushort(length = 0)

Arguments

vmode virtual mode
length desired length

Details

Function vector.vmode creates the vector in one of the usual storage modes (see .rammode) but flags them with an additional attribute 'vmode' if necessary. The creators can also be used directly:

boolean 1 bit logical without NA
logical 2 bit logical with NA
quad 2 bit unsigned integer without NA
nibble 4 bit unsigned integer without NA
byte 8 bit signed integer with NA
ubyte 8 bit unsigned integer without NA
short 16 bit signed integer with NA
ushort 16 bit unsigned integer without NA
**integer** 32 bit signed integer with NA
**single** 32 bit float
**double** 64 bit float
**complex** 2x64 bit float
**raw** 8 bit unsigned char
**character** character

**Value**

a vector of the desired vmode initialized with 0

**Author(s)**

Jens Oehlschlägel

**See Also**

as.vmode, vector

**Examples**

```r
vector.vmode("byte",12)
vector.vmode("double",12)
byte(12)
double(12)
```

---

**vector2array**

Array: make array from vector

**Description**

makes array from vector respecting 'dim' and 'dimorder'

**Usage**

```r
vector2array(x, dim, dimorder = NULL)
```

**Arguments**

- `x` an input vector, recycled if needed
- `dim` `dim`
- `dimorder` `dimorder`
Details

FILLS vector into array of dim where fastest rotating is dim[dimorder[1]], next is dim[dimorder[2]] and so forth. This is a generalization of converting vector to matrix(, byrow=TRUE). NOTE that the result is a ram array always stored in STANDARD dimorder !!! In this usage we sometimes term the dimorder 'bydim' because it does not change the physical layout of the result, rather bydim refers to the dimorder in which to interpret the vector (not the result). In ff, update and clone we have 'bydim' to contrast it from 'dimorder', the latter describing the layout of the file.

Value

a suitable array

Author(s)

Jens Oehlschlägel

See Also

array2vector, vectorIndex2arrayIndex

Examples

vector2array(1:12, dim=c(3, 4))  # matrix(1:12, 3, 4)
vector2array(1:12, dim=c(3, 4), dimorder=2:1)  # matrix(1:12, 3, 4, byrow=TRUE)

vectorIndex2arrayIndex

Array: make array from index vector positions

Description

make array from index vector positions respecting 'dim' and 'dimorder'

Usage

vectorIndex2arrayIndex(x, dim = NULL, dimorder = NULL, vw = NULL)

Arguments

x a vector of indices in 1:prod(dim)
dim NULL or dim
dimorder NULL or dimorder
vw NULL or integer matrix[2,m], see details
vmode

Details

The fastest rotating dimension is dim[dimorder[1]], then dim[dimorder[2]], and so forth. The parameters 'x' and 'dim' may refer to a subarray of a larger array, in this case, the array indices 'x' are interpreted as 'vw[1,] + x' within the larger array 'vw[1,] + x + vw[2,]'.

Value

an n by m matrix with n m-dimensional array indices

Author(s)

Jens Oehlschlägel

See Also

vector2array, arrayIndex2vectorIndex, symmIndex2vectorIndex

Examples

```r
matrix(1:12, 3, 4)
vectorIndex2arrayIndex(1:12, dim=3:4)
vectorIndex2arrayIndex(1:12, dim=3:4, dimorder=2:1)
matrix(1:30, 5, 6)
vectorIndex2arrayIndex(c(6L, 7L, 8L, 11L, 12L, 13L, 16L, 17L, 18L, 21L, 22L, 23L), vw=rbind(c(0,1), c(3,4), c(2,1)))
vectorIndex2arrayIndex(c(2L, 8L, 14L, 3L, 9L, 15L, 4L, 10L, 16L, 5L, 11L, 17L), vw=rbind(c(0,1), c(3,4), c(2,1)), dimorder=2:1)
```
```r
# S3 replacement method for class 'ff'
vmode(x) <- value
regtest.vmode()
```

**Arguments**

- **x** any object
- **value** a vmode from .vmode
- **...** The ... don't have a function yet, they are only defined to keep the generic flexible.

**Details**

`vmode` is generic with default and `ff` methods. The following meta data vectors can be queried by `.vmode` or `.ffmode`:

- **.vmode** virtual mode
- **.vunsigned** TRUE if unsigned vmode
- **.vvalues** number of possible values (incl. NA)
- **.vimplemented** TRUE if this vmode is available in `ff` (initialized `.onLoad` and stored in `globalenv`)
- **.rammode** storage mode of this vmode
- **.ffmode** integer used to code the vmode in C-code
- **.vvalues** number of possible integers incl. NA in this vmode (or NA for other vmodes)
- **.vmin** min integer in this vmode (or NA for other vmodes)
- **.vmax** max integer in this vmode (or NA for other vmodes)
- **.vNA** NA or 0 if no NA for this vmode
- **.rambytes** bytes needed in ram
- **.ffbytes** bytes needed by `ff` on disk
- **.vcoerceable** list of vectors with those vmodes that can absorb this vmode

The following functions relate to `vmode`:

- `vector.vmode` creating (ram) vector of some vmode
- `as.vmode` generic for coercing to some vmode (dropping other attributes)
- `vmode<-.maxffmode` generic for coercing to some vmode (keeping other attributes)
- `maxffmode` determine lowest `.ffmode` that can absorb all input vmodes without information loss

Some of those call the vmode-specific functions:

<table>
<thead>
<tr>
<th>creation</th>
<th>coercion</th>
<th>vmode description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>as.boolean</td>
<td>1 bit logical without NA</td>
</tr>
<tr>
<td>logical</td>
<td>as.logical</td>
<td>2 bit logical with NA</td>
</tr>
<tr>
<td>quad</td>
<td>as.quad</td>
<td>2 bit unsigned integer without NA</td>
</tr>
<tr>
<td>nibble</td>
<td>as.nibble</td>
<td>4 bit unsigned integer without NA</td>
</tr>
<tr>
<td>byte</td>
<td>as.byte</td>
<td>8 bit signed integer with NA</td>
</tr>
<tr>
<td>ubyte</td>
<td>as.ubyte</td>
<td>8 bit unsigned integer without NA</td>
</tr>
</tbody>
</table>
vmode.ffdf

<table>
<thead>
<tr>
<th>Type</th>
<th>as</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>short</td>
<td>as.short</td>
<td>16 bit signed integer with NA</td>
</tr>
<tr>
<td>ushort</td>
<td>as.ushort</td>
<td>16 bit unsigned integer without NA</td>
</tr>
<tr>
<td>integer</td>
<td>as.integer</td>
<td>32 bit signed integer with NA</td>
</tr>
<tr>
<td>single</td>
<td>as.single</td>
<td>32 bit float</td>
</tr>
<tr>
<td>double</td>
<td>as.double</td>
<td>64 bit float</td>
</tr>
<tr>
<td>complex</td>
<td>as.complex</td>
<td>2x64 bit float</td>
</tr>
<tr>
<td>raw</td>
<td>as.raw</td>
<td>8 bit unsigned char</td>
</tr>
<tr>
<td>character</td>
<td>as.character</td>
<td>character</td>
</tr>
</tbody>
</table>

Value

vmode returns a character scalar from .vmode or "NULL" for NULL
rambytes returns a vector of byte counts required by each of the vmodes

Note

regtest.vmode checks correctness of some vmode features

Author(s)

Jens Oehlschlägel

See Also

ff, storage.mode, mode

Examples

data.frame(.vmode=.vmode, .vimplemented=.vimplemented, .rammode=.rammode, .ffmode=.ffmode
, .vmin=.vmin, .vmax=.vmax, .vNA=.vNA, .rambytes=.rambytes, .ffbytes=.ffbytes)
vmode(1)
vmode(1L)
.vcoerceable["byte"]
.vcoerceable["ubyte"]
Arguments

x    ffdf
... ignored

Value

a character vector with one element for each column

Author(s)

Jens Oehlschlägel

See Also

vmode, ffdf

Examples

vt(x) for class 'ff'
vt(x, ...) for default S3 method:
vt(x) for class 'ff'
t(x)

Arguments

x  an ff or ram object
... further arguments (not used)

Description

The vt generic does a matrix or array transpose by modifying virtual attributes rather than by physically copying matrix elements.

Usage

vt(x, ...)
## S3 method for class 'ff'
vt(x, ...)
## Default S3 method:
vt(x, ...)
## S3 method for class 'ff'
t(x)
Details

The `vt.ff` method does transpose through reversing `dim.ff` and `dimorder`. The `vt.default` method is a wrapper to the standard transpose `t`. The `t.ff` method creates a transposed clone. If `x` has a virtual window `vw` defined, `vt.ff` returns an `ff` object with a transposed virtual window, the `t.ff` method return a transposed clone of the virtual window content only.

Value

an object that behaves like a transposed matrix

Author(s)

Jens Oehlschlägel

See Also

dim.ff, vw, virtual

Examples

```r
x <- ff(1:20, dim=c(4,5))
x
vt(x)
y <- t(x)
y
vw(x) <- cbind(c(1,3,0),c(1,4,0))
x
vt(x)
y <- t(x)
y
rm(x,y); gc()
```

---

vw

*Getting and setting virtual windows*

Description

The virtual window `vw` function allows one to define a virtual window into an `ff_vector` or `ff_array`. The `ff` object will behave like a smaller array and it is mapped into the specified region of the complete array. This allows for example to execute recursive divide and conquer algorithms that work on parts of the full object, without the need to repeatedly create subfiles.
Usage

```
vw(x, ...) <- value
## S3 method for class 'ff'
vw(x, ...)
## Default S3 method:
vw(x, ...)
## S3 replacement method for class 'ff_vector'
vw(x, ...) <- value
## S3 replacement method for class 'ff_array'
vw(x, ...) <- value
```

Arguments

- `x`: an `ff_vector` or `ff_array`
- `...`: further arguments (not used)
- `value`: a vector or matrix with an Offset, Window and Rest component, see details and examples

Details

Each dimension of an `ff` array (or vector) is decomposed into three components, an invisible Offset, a visible Window and an invisible Rest. For each dimension the sum of the `vw` components must match the dimension (or length). For an `ff_vector`, `vw` is simply a vector[1:3], for an array is is a matrix[1:3,1:length(dim(x))]. `vw` is a virtual attribute.

Value

NULL or a `vw` specification, see details

Author(s)

Jens Oehlschlägel

See Also

`length.ff`, `dim.ff`, `virtual`

Examples

```
x <- ff(1:26, names=letters)
y <- x
vw(x) <- c(0, 13, 13)
vw(y) <- c(13, 13, 0)
x
y
x[1] <- -1
y[1] <- -2
```


Function `write.table.ffdf` writes a `ffdf` object to a separated flat file, very much like (and using) `write.table`. It can also work with any convenience wrappers like `write.csv` and provides its own convenience wrapper (e.g. `write.csv.ffdf`) for R’s usual wrappers.

**Usage**

`write.table.ffdf(x = NULL
, file, append = FALSE
, nrows = -1, first.rows = NULL, next.rows = NULL
, FUN = "write.table", ...
, transFUN = NULL
, BATCHBYTES =getOption("ffbatchbytes")
, VERBOSE = FALSE
)`

write.csv.ffdf(...)
write.csv2.ffdf(...)
write.csv(...)
write.csv2(...)

**Arguments**

- **x**
  - a `ffdf` object which to export to the separated file
- **file**
  - either a character string naming a file or a connection open for writing. "" indicates output to the console.
- **append**
  - logical. Only relevant if file is a character string. If TRUE, the output is appended to the file. If FALSE, any existing file of the name is destroyed.
- **nrows**
  - integer: the maximum number of rows to write in (includes first.rows in case a 'first' chunk is read) Negative and other invalid values are ignored.
- **first.rows**
  - the number of rows to write with the first chunk (default: next.rows)
- **next.rows**
  - integer: number of rows to write in further chunks, see details. By default calculated as BATCHBYTES %/% sum(.rambytes[vmode(x)])
FUN character: name of a function that is called for writing each chunk, see `write.table`, `write.csv`, etc.

... further arguments, passed to FUN in `write.table.ffdf`, or passed to `write.table.ffdf` in the convenience wrappers

transFUN NULL or a function that is called on each data.frame chunk before writing with FUN (for filtering, transformations etc.)

BATCHBYTES integer: bytes allowed for the size of the `data.frame` storing the result of reading one chunk. Default `getOption("ffbatchbytes")`.

VERBOOSE logical: TRUE to verbose timings for each processed chunk (default FALSE)

Details

`write.table.ffdf` has been designed to export very large `ffdf` objects to separated flatfiles in chunks. The first chunk is potentially written with col.names. Further chunks are appended. `write.table.ffdf` has been designed to behave as much like `write.table` as possible. However, note the following differences:

1. by default `row.names` are only written if the `ffdf` has row.names.

Value

invisible

Note

`write.csv` and `write.csv2` have been fixed in order to suppress `col.names` if append=TRUE is passed. Note also that `write.table.ffdf` passes `col.names=FALSE` for all chunks following the first chunk - but not so for FUN="write.csv" and FUN="write.csv2".

Author(s)

Jens Oehlschlägel, Christophe Dutang

See Also

`read.table.ffdf, write.table, ffdf`

Examples

```r
x <- data.frame(log=rep(c(FALSE, TRUE), length.out=26), int=1:26, dbl=1:26 + 0.1,
    fac=factor(letters), ord=ordered(LETTERS), dct=Sys.time()+1:26,
    dat=seq(as.Date("1910/1/1"), length.out=26, by=1))
ffx <- as.ffdf(x)

csvfile <- tempPathFile(path=getOption("fftempdir"), extension="csv")
write.csv.ffdf(ffx, file=csvfile)
write.csv.ffdf(ffx, file=csvfile, append=TRUE)

ffy <- read.csv.ffdf(file=csvfile, header=TRUE)
```
write.table.ffdf

, colClasses=c(ord="ordered", dct="POSIXct", dat="Date"))

rm(ffx, ffy); gc()
unlink(csvfile)

## Not run:
# Attention, this takes very long
vmodes <- c(log="boolean", int="byte", dbl="single"
, fac="short", ord="short", dct="single", dat="single")

message("create a ffdf with 7 columns and 78 mio rows")

system.time({
x <- data.frame(log=rep(c(FALSE, TRUE), length.out=26), int=1:26, dbl=1:26 + 0.1
, fac=factor(letters), ord=ordered(LETTERS), dct=Sys.time()+1:26
, dat=seq(as.Date("1910/1/1"), length.out=26, by=1))
x <- do.call("rbind", rep(list(x), 10))
x <- do.call("rbind", rep(list(x), 10))
x <- do.call("rbind", rep(list(x), 10))
x <- do.call("rbind", rep(list(x), 10))
ffx <- as.ffdf(x, vmode = vmodes)
for (i in 2:300){
    message(i, "\n")
    last <- nrow(ffx) + nrow(x)
    first <- last - nrow(x) + 1L
    nrow(ffx) <- last
    ffx[first:last,] <- x
}
})

csvfile <- tempPathFile(path=getOption("fftempdir"), extension="csv")

write.csv.ffdf(ffx, file=csvfile, VERBOSE=TRUE)
ffy <- read.csv.ffdf(file=csvfile, header=TRUE
, colClasses=c(ord="ordered", dct="POSIXct", dat="Date"
, asffdf_args=list(vmode = vmodes), VERBOSE=TRUE)

rm(ffx, ffy); gc()
unlink(csvfile)

## End(Not run)
Index

*Topic IO

add, 5
as.ff, 8
as.ff.bit, 10
as.ffdf, 11
as.hi, 12
as.integer.hi, 15
as.vmode, 17
chunk.bit, 21
chunk.ffdf, 22
clone, 24
clone.ffdf, 26
close.ff, 27
delete, 28
dim.ff, 30
dimnames.ff_array, 33
dimorderCompatible, 34
Extract.ff, 36
Extract.ffdf, 39
ff, 41
ffconform, 53
ffdf, 55
ffdfindexget, 58
ffdfsort, 59
ffdrop, 61
ffindexget, 61
ffindexorder, 63
ffinfo, 64
ffload, 65
fforder, 66
ffreturn, 68
ffsave, 69
ffsort, 71
ffsuitable, 73
ffxextensions, 74
file.resize, 75
filename, 76
finalize, 78
finalizer, 79

fixdiag, 81
ggetError.ff, 82
getpagesize, 83
getset.ff, 84
hi, 85
hiparse, 87
is.ff, 88
is.ffdf, 88
is.open, 89
is.readonly, 90
is.sorted, 91
length.ff, 92
length.ffdf, 93
length.hi, 94
levels.ff, 96
LimWarn, 98
maxffmode, 102
maxlength, 103
mismatch, 104
na.count, 105
names.ff, 106
open.ff, 108
pagesize, 109
physical.ff, 110
physical.ffdf, 111
print.ff, 113
ram2ffcode, 114
ramattribs, 115
read.table.ffdf, 121
readwrite.ff, 126
swap, 139
symmetric, 141
unclass_.-, 143
undim, 143
unsort, 144
update.ff, 145
vector.vmode, 148
vmode, 151
vmode.ffdf, 153
vt, 154
vw, 155
write.table.ffdf, 157

*Topic arith
ffdfsort, 59
fforder, 66
ffsort, 71
is.sorted, 91
ramorder.default, 116
rassort.default, 119
regtest.fforder, 127

*Topic array
array2vector, 6
arrayIndex2vectorIndex, 7
dummy.dimnames, 35
Extract.ff, 36
ff, 41
ffapply, 49
matcomb, 100
matprint, 101
nrowAssign, 107
swap, 139
symmIndex2vectorIndex, 142
vector2array, 149
vectorIndex2arrayIndex, 150
vt, 154
vw, 155

*Topic attribute
as.vmode, 17
ff, 41
length.ff, 92
levels.ff, 96
physical.ff, 110
ramattribs, 115
sortLevels, 134
vector.vmode, 148
vmode, 151

*Topic classes
as.ff.bit, 10
ff, 41
ramattribs, 115

*Topic connection
read.table.ffdf, 121
write.table.ffdf, 157

*Topic data
add, 5
array2vector, 6
arrayIndex2vectorIndex, 7
as.ff, 8
as.ff.bit, 10
as.ffdf, 11
as.hi, 12
as.integer.hi, 15
as.vmode, 17
bigsample, 18
chunk.bit, 21
chunk.ffdf, 22
close, 27
close.ff, 27
delete, 28
dim.ff, 30
dimnames.ff_array, 33
dimorderCompatible, 34
dummy.dimnames, 35
Extract.ff, 36
Extract.ffdf, 39
ffapply, 49
ffconform, 53
ffdf, 55
ffdfindexget, 58
ffdfsort, 59
ffindexget, 61
ffindexorder, 63
fforder, 66
ffreturn, 68
ffsort, 71
ffsuitable, 73
ffxtensions, 74
file.resize, 75
filename, 76
fixdiag, 81
geterror.ff, 82
getset.ff, 84
hi, 85
hiparse, 87
is.ff, 88
is.ffdf, 88
is.open, 89
is.readonly, 90
is.sorted, 91
length.ff, 92
length.ffdf, 93
length.hi, 94
levels.ff, 96
limWarn, 98
matcomb, 100
maxffmode, 102
maxlength, 103
mismatch, 104
na.count, 105
names.ff, 106
open.ff, 108
pagesize, 109
physical.ff, 110
physical.ffdf, 111
print.ff, 113
ram2ffcode, 114
ramattributes, 115
readwrite.ff, 126
swap, 139
symmetric, 141
symmindex2vectorIndex, 142
unclass=_, 143
undim, 143
unsort, 144
update.ff, 145
vector.vmode, 148
vector2array, 149
vectorIndex2arrayIndex, 150
vmode, 151
vmode.ffdf, 153
vt, 154
vw, 155
*Topic distribution
  bigsample, 18
*Topic file
  ffdrop, 61
  ffinfo, 64
  ffload, 65
  ffsave, 69
  read.table.ffdf, 121
  splitPathFile, 136
  write.table.ffdf, 157
*Topic list
  CFUN, 20
*Topic logic
  as.ff.bit, 10
*Topic manip
  CFUN, 20
  ffdfsort, 59
  fforder, 66
  ffsort, 71
  ramorder.default, 116
  ramsort.default, 119
  regtest.fforder, 127
*Topic package
  ff, 41
  LimWarn, 98
*Topic print
  matprint, 101
  print.ff, 113
  vecprint, 147
*Topic univar
  ffdfsort, 59
  fforder, 66
  ffsort, 71
  ramorder.default, 116
  ramsort.default, 119
  regtest.fforder, 127
*Topic utilities
  repnam, 133
  .Machine, 44
  .Platform, 137
  .ffbytes (vmode), 151
  .ffmode, 102
  .ffmode (vmode), 151
  .onLoad, 152
  .onUnload, 29, 98
  .rambytes, 21–23, 26, 43, 51
  .rambytes (vmode), 151
  .rammode, 17, 96, 114, 148
  .rammode (vmode), 151
  .vNA (vmode), 151
  .vcoerceable, 102
  .vcoerceable (vmode), 151
  .vimplemented, 25, 42
  .vimplemented (vmode), 151
  .vmax (vmode), 151
  .vmin (vmode), 151
  .vmode (vmode), 151
  .vunsigned, 96, 114
  .vunsigned (vmode), 151
  .vvalues (vmode), 151
  ::, 87
  [], 40, 46, 57
  [.., 5, 12, 14, 34, 37, 84, 99, 127, 139, 140
  [.. (Extract.ff)], 36
  [.._array, 37, 99
  [.._array (Extract.ff)], 36
  [.._ffdf (Extract.ffdf)], 39
  [<=, 46
[-.ff (Extract.ff), 36
[-.ff_array (Extract.ff), 36
[-.ffdf (Extract.ffdf), 39
[[, 46, 57
[[, 99
[[.ff (Extract.ff), 36
[[.ffdf (Extract.ffdf), 39
[[<.-, 46
[[<-ff (Extract.ff), 36
[[<-ffdf (Extract.ffdf), 39
$. 57
$.ffdf (Extract.ffdf), 39
$<-ffdf (Extract.ffdf), 39

add, 5, 38, 46, 99, 140
add.default, 98
apern, 25, 43
appendLevels (sortLevels), 134
apply, 50–52
array, 6, 25, 35, 43, 44, 48, 150
array2vector, 6, 8, 150
arrayIndex2vectorIndex, 7, 7, 14, 142, 151
as.bit, 45
as.bit.ff (as.ff.bit), 10
as.bit.hi (as.integer.hi), 15
as.bitwhich.hi (as.integer.hi), 15
as.boolean, 152
as.boolean (as.vmode), 17
as.byte, 152
as.byte (as.vmode), 17
as.character, 153
as.character.hi (as.integer.hi), 15
as.complex, 153
as.data.frame, 56
as.data.frame.ffdf (as.ffdf), 11
as.double, 153
as.ff, 8, 10, 26, 41, 45, 48, 77, 88, 98, 111
as.ff.bit, 9, 10
as.ffdf, 11, 56, 89, 122, 123
as.hi, 9, 12, 16, 36, 37, 41, 62–64, 86, 87, 95,
99, 139, 145
as.hi.bit, 10
as.hi.call, 87
as.hi.character, 16
as.hi.integer, 16
as.hi.logical, 16
as.hi.matrix, 16
as.integer, 153
as.integer.hi, 14, 15

as.logical, 152
as.logical.hi (as.integer.hi), 15
as.matrix.hi, 86
as.matrix.hi (as.integer.hi), 15
as.nibble, 152
as.nibble (as.vmode), 17
as.quad, 152
as.quad (as.vmode), 17
as.ram, 26, 41, 45, 48, 77, 98, 111, 116
as.ram (as.ff), 8
as.raw, 153
as.short, 153
as.short (as.vmode), 17
as.single, 153
as.ubyte, 152
as.ubyte (as.vmode), 17
as.ushort, 153
as.ushort (as.vmode), 17
as.vmode, 9, 17, 25, 42, 43, 46, 149, 152
as.which.hi (as.integer.hi), 15
AsIs, 112
attributes, 116

basename, 137, 138
bbatch, 50, 52, 63, 64
bigsample, 18, 46
bit, 10, 22, 45
boolean, 152
boolean (vector.vmode), 148
byte, 152
byte (vector.vmode), 148
c, 87
call, 87
cbind, 20
ccbind (CFUN), 20
CFUN, 20, 51
cfun (CFUN), 20
counter, 55, 153
counter (vector), 55
chunk, 22, 23, 57
chunk.bit, 21
chunk.ff_vector (chunk.ffdf), 22
chunk.ffdf, 22
class, 29
clength (CFUN), 20
close, 9, 24, 26, 27, 41, 45, 56, 92, 146, 155
close, 9
clone, 26
clone.ffdf, 26
close, 43, 44, 46, 57, 80, 122
close.ff, 9, 25, 27, 30, 43, 80, 90, 109
close.ff_pointer (close.ff), 27
close.ffdf (close.ff), 27
cmean (CFUN), 20
cmedian (CFUN), 20
col, 100
colnames, 33, 121
complex, 153
connection, 122, 123
cquantile (CFUN), 20
crbind (CFUN), 20
csum (CFUN), 20
csummary (CFUN), 20
data.frame, 11, 12, 40, 55, 57, 60, 112, 122, 158
DateTimeClasses, 116
delete, 28, 28, 43, 44, 46, 57, 80, 109
delete.ff, 25, 43, 80
delete.ff_pointer, 80
deleteIfOpen, 28, 44, 46, 47, 80, 109
deleteIfOpen (delete), 28
deleteIfOpen.ff, 25, 43, 80
dfforder, 118
dfforder (ffdfsort), 59
dfsort, 120
dfsort (ffdfsort), 59
dim, 6, 7, 13, 16, 25, 31, 34, 37, 43, 46, 54, 93, 107, 144, 149, 150
dim, 56
dim, 25, 30, 34, 36, 43, 45, 93, 155, 156
dim, 94, 108
dim (ffdf), 30
dim<-.ff (dim), 30
dim<-.ffdf (dim), 30
dimnames, 25, 34, 35, 37, 43, 46, 56, 107
dimnames, 33, 36, 93
dimnames (dimnames), 33
dimnames (dimnames), 33, 31, 107
dimnames (fdffdf), 32
dimnames<-.ff_array
(dimnames.ff_array), 33
dimnames<-.ffdf (dimnames.ffdf), 32
dimorder, 6, 7, 13, 14, 16, 25, 34, 35, 37, 38, 43, 45, 46, 56, 86, 99, 149, 150, 155
dimorder (dim), 30
dimorder<-(dim), 30
dimorderCompatible, 34
dimorderStandard, 14, 31, 37, 54, 56, 99
dimorderStandard (dimorderCompatible), 34
dirname, 137, 138
dist, 82, 141
do.call, 20, 21
double, 153
dummy.dimnames, 35
expand.grid, 100
expression, 50, 52
Extract.data.frame, 40
Extract.ff, 36, 40, 59, 62
Extract.ffdf, 39
factor, 36, 97, 99, 115, 134
ff, 9–11, 19, 23, 26, 28, 30, 38, 40, 41, 45, 55, 57, 58, 62, 63, 73, 76–78, 80–83, 86,
ff_pointer (ff), 41
ffapply, 21, 41, 49
ffcolapply (ffapply), 49
ffconform (ffapply), 49
ffdata.frame, 11, 12, 23, 25–27, 31–33, 40, 43, 55, 56,
ffdf, 58, 62
ffdfindexget, 58, 62
ffdfindexset (ffdfindexget), 58
ffdforder, 67
ffdforder (ffdfsort), 59
ffdforder (ffdfsort), 59
ffdforder (ffdfsort), 59
ffdforder (ffdfsort), 59
ffdrop, 61, 65, 66, 70
ffindex, 59, 61, 63, 64, 67
ffindexorder, 59, 61, 63, 62, 63
ffindexordersize (ffindexorder), 63
ffindexset, 59, 63
ffindexset (ffindexset), 61
ffinfo, 61, 64, 66, 70
ffload, 61, 65, 65, 69, 70
fforder, 60, 66, 72, 118
ffreturn, 68, 74
ffrowapply (ffapply), 49
ffsave, 61, 64–66, 69
ffsort, 60, 67, 71, 120
ffsuitable, 25, 43, 51, 52, 54, 68, 73
ffsuitable_attribs (ffsuitable), 73
ffsymmxtensions (ffxtensions), 74
ff tempfile, 77
ff tempfile (splitPathFile), 136
ffvecapply, 146
ffvecapply (ffapply), 49
ffxtensions, 74
file, 122, 123
file.copy, 75, 76
file.create, 76
file.info, 76
file.move, 77
file.move (file.resize), 75
file.path, 137, 138
file.remove, 61, 75, 76
file.rename, 75, 76
file.resize, 75, 93
filename, 25, 43, 44, 46, 76, 146
filename (filename), 76
finalize, 43, 46, 57, 78, 80, 81
finalize.ff_pointer, 80, 81
finalizer, 43, 44, 46, 77–79, 79
finalizer (finalizer), 79
fixdiag, 13, 16, 46, 81, 82
fixdiag.ff, 43
fixdiag (fixdiag), 81
format, 101, 147
gc, 44, 80
gff, 36, 46, 99
gff (getset.fg), 84
get alignment pagesize, 108, 109
get alignment pagesize (getpagesize), 83
get default pagesize, 25, 43, 47
get default pagesize (getpagesize), 83
get error, 47, 82
get error (geterror.ff), 47
get error (geterror.ff), 82
get option, 47
get pagesize, 83, 110
get set ff, 37, 46, 84, 99, 105, 127, 140
get wdd, 43, 77, 122
global env, 152
hi, 12–16, 36, 37, 46, 85, 87, 95, 99, 139, 145
hiparse, 14, 37, 87, 99
I, 56
inherits, 88, 89
integer, 153
intisasc, 91, 92
intrl, 86
invisible, 51, 158
is.factor, 46, 134
is.factor (levels.ff), 96
is ff, 28, 45, 88, 89, 109
is ff df, 12, 56, 88, 88
is open, 28, 44, 46, 57, 89
is ordered, 46
is ordered (levels.ff), 96
is ordered ff, 92
is readonly, 44, 46, 90, 90
is sorted, 46, 72, 91, 93, 105, 111
is sorted <- .default (is sorted), 91
is unsorted, 91, 92
isSymmetric, 141
keyorder.default (ramorder.default), 116
keysort.default (ramsort.default), 119
lapply, 21
length, 14, 16, 20, 25, 42, 46, 56, 93, 95
length.ff, 25, 42, 45, 92, 94, 95, 103, 156
length.ff df, 93
length hi, 94
length <- .ff (length.ff), 92
levels, 46, 121, 122, 134
levels.ff, 36, 45, 96, 111, 115, 116, 123, 135, 146
levels <- .ff (levels.ff), 96
limWarn, 5, 36, 38, 42, 98, 135, 140
list, 112
load, 44, 66
logical, 152
matcomb, 100
matprint, 101, 148
matrix, 25, 43, 44, 48, 101
max.ff mode, 53, 54, 102, 152
max index, 14, 37, 38, 86, 99, 103
max index (length hi), 94
max length, 44, 46, 93, 103, 146
mean, 20
median, 20
memory.limit, 47
merge order.default (ramorder.default), 116
mergesort.default (ramsort.default), 119
mismatch, 104
mode, 153
NA, 105
na.count, 21, 46, 67, 72, 84, 92, 93, 105, 111, 126
na.count<- default (na.count), 105
na.count<- .ff (na.count), 105
names, 13, 16, 25, 33, 37, 43, 46, 56, 107, 133
dimnames, 34, 36, 45, 93, 106
names.ff_array (names.ff), 106
dimnames.ffdf (dimnames.ffdf), 32
names<- .ff (names.ff), 106
names<- .ff_array (names.ff), 106
dimnames<- .ffdf (dimnames.ffdf), 32
nrow<- (nrowAssign), 107
ncol<- (nrowAssign), 107
nrowAssign, 107
open, 44, 46, 57
open.ff, 28, 30, 90, 91, 108
open.ffdf (open.ff), 108
options, 47
order, 60, 67, 117, 118
ordered, 123
páginaize, 109
path.expand, 137
pattern, 46, 57
pattern (filename), 76
pattern<- (filename), 76
physical.ff, 110
physical.ffdf, 111, 111
physical<- .ff (physical.ff), 110
POSIXct, 36, 99, 115
poslength, 14, 86
poslength (length.hi), 94
print, 45, 56, 101, 114, 147
print.ff, 113
print.ff_matrix (print.ff), 113
print.ff_vector (print.ff), 113
print.ffdf (print.ff), 113
print.hi (hi), 85
print.matprint (matprint), 101
print.vecprint (vecprint), 147
q, 25, 43, 44
quad, 152
quantile, 20
radixorder.default (ramorder.default), 116
radixsort.default (ramsort.default), 119
ram2ffcode, 96, 114
ram2ramcode (ram2ffcode), 114
ramattribs, 25, 36, 37, 43, 45, 46, 111, 115
ramattribs_excludes (ramattribs), 115
ramclass, 25, 36, 37, 43, 45, 46, 97
ramclass (ramattribs), 115
ramclass_excludes (ramattribs), 115
ramdorder (ffdfsorder), 59
ramdforder (ffdfsorder), 59
ramorder, 60, 67, 120
ramorder.default, 116
ramsort, 60, 72, 118, 127
ramsort.default, 119
raw, 153
rbind, 20
rbind.data.frame, 135
read.csv, 121, 122
read.csv.ffdf (read.table.ffdf), 121
read.csv2.ffdf (read.table.ffdf), 121
read.delim.ffdf (read.table.ffdf), 121
read.delim2.ffdf (read.table.ffdf), 121
read.ff, 46
read.ff (readwrite.ff), 126
read.table, 121–123
read.table.ffdf, 121, 133, 135, 158
readwrite.ff, 36–38, 46, 84, 99, 105, 126, 139, 140
Recall, 87
recodeLevels, 46
recodeLevels (sortlevels), 134
reg.finalizer, 25, 30, 78, 80, 81
regtest.fforder, 127
regtest.vmode (vmode), 151
remove, 25, 43, 44
rep, 133
repfromto, 52, 133
reppnam, 133
ri, 22, 23
rlepack, 13, 86
rm, 80
row, 100
row.names, 27, 28, 56, 89, 109, 158
row.names.ffdf (dimnames.ffdf), 32
INDEX

row.names<- .ff df (dimnames .ff df), 32
row.names, 33

sample, 18, 19
save, 44, 70
set .ff, 36, 46, 99, 105
set .ff (getset .ff), 84
shellsort .default (ramorder .default), 116
shellSort .default (ramsort .default), 119
short, 153
short (vector .vmode), 148
sort, 60, 72, 91, 119, 120
sort .list, 117, 120
sortLevels, 46, 57, 123, 134
splitPathFile, 136
standardPathFile (splitPathFile), 136
stop, 53, 54
storage .mode, 17, 98, 148, 153
str, 45, 56, 114
str .ff (print .ff), 113
str .ff df (print .ff), 113
str .hi (hi), 85
subscript2integer (unsort), 144
sum, 20–22
summary, 20
swap, 5, 36, 38, 46, 99, 105, 139
swap .default, 98
symmetric, 13, 16, 46, 141, 141
symmetric .ff, 45
symmIndex2vectorIndex, 142, 151
t, 46, 155
t .ff (vt), 154
tempdir, 47
tempfile, 137, 138
tempPathFile (splitPathFile), 136

ubyte, 152
ubyte (vector .vmode), 148
unclass, 143, 144
unclass<- (unclass _), 143
unclass _, 143
undim, 143, 143
unlink, 44
unname, 144
unsort, 144
unsplitPathFile (splitPathFile), 136

update, 26, 45, 56
update .ff, 26, 34, 41, 43, 55, 77, 145
update .ff df (update .ff), 145
ushort, 153
ushort (vector .vmode), 148
vecprint, 101, 147
vector, 44, 48, 133, 149
vector .vmode, 18, 25, 42, 148, 152
vector2array, 6, 7, 149, 151
vectorCompatible, 146
vectorCompatible (dimorderCompatible), 34
vectorIndex2arrayIndex, 8, 150, 150
vectorStandard (dimorderCompatible), 34
virtual, 29, 31, 34, 40, 41, 44–46, 56, 91, 93,
97, 98, 107, 112, 116, 123, 135,
154–156
virtual .ff (physical .ff), 110
virtual .ff df (physical .ff df), 111
virtual<- .ff (physical .ff), 110
vmode, 9, 11, 17, 18, 21, 22, 25, 41–45, 51, 56,
62, 63, 67, 74, 93, 98, 102, 112, 115,
121, 122, 151, 154, 157
vmode .ff df, 153
vmode<- (vmode), 151
vt, 41, 46, 154
v w, 13, 16, 31, 33, 34, 36, 41, 46, 85, 86, 93,
99, 107, 146, 155, 155
v w .ff, 45
v w<- (v w), 155

warning, 54, 73
write .csv, 157, 158
write .csv (write .table .ff df), 157
write .csv 2, 158
write .csv 2 (write .table .ff df), 157
write .ff, 46, 105
write .ff (readwrite .ff), 126
write .table, 157, 158
write .table .ff df, 123, 157

ymismatch, 54
ymismatch (mismatch), 104