

DISCLOSED

Coordinating package 'ff' for large objects with R base

Oehlschlägel, Adler

Munich, Göttingen

July 2009

This report contains public intellectual property. It may be used, circulated, quoted, or reproduced for distribution as a whole. Partial citations require a reference to the author and to the whole document and must not be put into a context which changes the original meaning. Even if you are not the intended recipient of this report, you are authorized and encouraged to read it and to act on it. Please note that you read this text on your own risk. It is your responsibility to draw appropriate conclusions. The author may neither be held responsible for any mistakes the text might contain nor for any actions that other people carry out after reading this text.

Context of discussion

Package 'ff' 2.1.0

- provides large, fast disk-based vectors and arrays
- NEW: dataframes with up to 2.14 billion rows
- NEW: lean datatypes on CRAN under GPL, e.g. 2bit factors
- NEW: fixed width characters (fffc)
- NEW: fast `length()` increase for ff vectors

Package 'bit' 1.1.0

- Class 'bit': lean in-memory boolean vectors + fast operators
- NEW: class 'ri' (range-index) for chunked-processing
- NEW: class 'bitwhich': alternative for very skewed filters
- NEW: close integration with ff objects and chunked processing

Package 'R.ff'

- GOAL: convert standard R into a system, that provides the most commonly used methods for ff as well as for standard RAM objects (math, operators, data manipulation etc.) and do this PURELY in R.
- QUESTION: is this possible with a separate package, or do we need to more closely integrate R base with large object capabilities?
- ANSWER: seems somewhat possible but is not clever to do this!

EXAMPLE I – preparation

```
library(ff) # loads library(bit)
N <- 8e7; n <- 1e6
countries <- factor(c('FR', 'ES', 'PT', 'IT', 'DE', 'GB', 'NL', 'SE', 'DK', 'FI'))
years <- 2000:2009; genders <- factor(c("male", "female"))

country <- ff(countries, vmode='ubyte', length=N, update=FALSE, filename="d:/tmp/country.ff", finalizer="close")
for (i in chunk(1,N,n)) country[i] <- sample(countries, sum(i), TRUE) # 9 sec

year <- ff(years, vmode='ushort', length=N, update=FALSE, filename="d:/tmp/year.ff", finalizer="close")
for (i in chunk(1,N,n)) year[i] <- sample(years, sum(i), TRUE) # 9 sec

gender <- ff(genders, vmode='quad', length=N, update=FALSE, filename="d:/tmp/gender.ff", finalizer="close")
for (i in chunk(1,N,n)) gender[i] <- sample(genders, sum(i), TRUE) # 9 sec

age <- ff(0, vmode='ubyte', length=N, update=FALSE, filename="d:/tmp/age.ff", finalizer="close")
for (i in chunk(1,N,n)) age[i] <- ifelse(gender[i]=="male", rnorm(sum(i), 40, 10), rnorm(sum(i), 50, 12)) # 90 sec

income <- ff(0, vmode='single', length=N, update=FALSE, filename="d:/tmp/income.ff", finalizer="close")
for (i in chunk(1,N,n)) income[i] <- ifelse(gender[i]=="male", rnorm(sum(i), 34000, 5000), rnorm(sum(i), 30000, 6000)) # 90 sec

x <- fdf(country=country, year=year, gender=gender, age=age, income=income)

fcountry <- bit(N)
fyear <- bit(N)
for (i in chunk(1,N,n)) fcountry[i] <- x$country[i] == 'FR' # 20 sec
for (i in chunk(1,N,n)) fyear[i] <- x$year[i] %in% c(2008,2009) # 20 sec

close(x)
save.image(file="d:/tmp/ffbit.RData")
```

Short demo of 'ff' with 'bit'

```
library(ff) # loads library(bit)
load(file="d:/tmp/ffbit.RData") # load some ff and bit objects
open(x) # open ffd (all embedded vectors)

x[1:10,] # subscripting returns data.frame
sum(.rambytes[vmode(x)]) * 8e7 / 1024^2 # instead of 1.8 GB in RAM
sum(.ffbytes[vmode(x)]) * 8e7 / 1024^2 # only 630 MB disk/fs-cache
object.size(physical(x)) # and 9k in R's RAM

fcountry
filter <- fcountry & fyear
summary(filter) # check filter summary, then use
summary(filter, range=c(1, 1000)) # dito for chunk
summary(x[filter & ri(1, 8e6, N),], maxsum = 12)
# filter combined with range index and used as subscript to ffd

nrow(x)
nrow(x) <- 1e8
x[c(1, 1e8),]
```

NOTE the following peculiarities

```
library(ff) # load ff package
load(file="d:/tmp/ffbit.RData") # load bit objects
open(x) # open bit objects (embedded vectors)

x[1:10,] # subscripting returns data.frame
sum(.rambytes[vmode(x)]) * 8e7 / 1024^2 # instead of 1.8 GB in RAM
sum(.ffbytes[vmode(x)]) * 8e7 / 1024^2 # only 630 MB disk/fs-cache
object.size(physical(x)) # and 9k in R's RAM

fcountry
filter <- fcountry & fyear
summary(filter) # check filter summary, then use
summary(filter, range=c(1, 1000)) # filter summary
summary(x[filter & ri(1, 8e6, N),]) # filter combined with range index
# filter combined with range index

nrow(x)
nrow(x) <- 1e8 # assignment to local variable
x[c(1, 1e8),] # did not create a local copy
# but
# changed the global original
# through a reference
```

subscripting ffdid

return something

very usefull ... but a bit objects

different class (embedded vectors)

'bit' subscripts are
unknown to standard R

to ffdid

assignment to local variable
did not create a local copy
but
changed the global original
through a reference

Is R too functional a language? Sometimes there is too much copying!

```
> n <- 1e8
> y <- double(n)
> gctorture(on = TRUE)
> attr(y, "a") <- 1
> gctorture(on = TRUE)
> attr(y, "b") <- 2
Fehler: kann Vektor der Größe 762.9 MB nicht allozieren
Zusätzlich: Warnmeldungen:
1: In attr(y, "b") <- 2 :
  Reached total allocation of 1535Mb: see help(memory.size)
2: In attr(y, "b") <- 2 :
  Reached total allocation of 1535Mb: see help(memory.size)
3: In attr(y, "b") <- 2 :
  Reached total allocation of 1535Mb: see help(memory.size)
4: In attr(y, "b") <- 2 :
  Reached total allocation of 1535Mb: see help(memory.size)
```

Can a functional language handle large data with all its copying?

Is R a functional programming language at all?

```
> f<- function(x) {  
    x[] <- x[] + 1L  
    return(x)  
}  
  
> y <- matrix(1L, nrow=1, ncol=2)  
> z <- f(y)  
> y[]  
      [,1] [,2]  
[1,]    1    1  
> z[]  
      [,1] [,2]  
[1,]    2    2  
  
> library(bigmemory)  
> y <- big.matrix(1, 2, type = "integer", init=1L)  
> z <- f(y)  
> y[]  
[1] 2 2  
> z[]  
[1] 2 2
```

Don't blame "bigmemory" or "ff"!

```
> f<- function(x) {  
+ x$a <- x$a + 1L  
+ return(x)  
+ }  
>  
> y <- data.frame(a=1)  
> z <- f(x)  
> y$a  
[1] 1  
> z$a  
[1] 2
```

```
> y <- new.env()  
> y$a <- 1  
> z <- f(y)  
> y$a  
[1] 2  
> z$a  
[1] 2
```


ff can do both – but it reads a bit clumsy ...

```
> library(ff)
> fref <- function(x){x[1] <- x[1]+1; x}
> fval <- function(x){x <- clone(x); x[1] <- x[1]+1; x}
> y <- ff(1)
> fref(y)
ff (open) double length=1 (1)
[1]
 2
> y
ff (open) double length=1 (1)
[1]
 2
> fval(y)
ff (open) double length=1 (1)
[1]
 3
> y
ff (open) double length=1 (1)
[1]
 2
```

... and ask for explicit cloning of RAM objects would destroy R as a functional programming language

What about an easy to read and explicit syntax?

```
ff[i]
# dispatches to "[.ff"

ff[i] <- value
# dispatches to "[<-.ff" and clones ff before assigning value

value -> ff[i]
# dispatches to "->[.ff" and assigns value by reference
```

How to pass a return object into a function or expression for reuse?

```
# currently R.ff does
newff <- aff + bff
oldff <- "+"(aff, bff, FF_RETURN=oldff)

# then R.ff could read
newff <- aff + bff
aff + bff -> oldff
```

What should `ff[i]` return?

Class `ff`

- + returns consistent class
- kills performance because we need to copy from larger to smaller `ff` on disk and then coerce to standard RAM object:
`as.ram(ff[i])`

Standard vector

- Current implementation
- + returns much faster
 - returns inconsistent class

Class `ff` with virtual window

- + returns much faster (but still needs coercion)
- returns relatively consistent class (still `ff`, just with different `vw`)
- `vw` complicates virtual-to-physical subscript translation
- `vw` does not allow for arbitrary subscripts

Class `ff` with any virtual selection

- + fully consistent
- more indirect subscript processing costing performance, e.g. by attaching a 'bit' vector storing the virtual selection, translating each new subscript to 'bit' and return '&' of both

[.AsIs

```
# Currently R has:

> get("[.AsIs")
function (x, i, ...)
  structure(NextMethod("["), class = class(x))

# assuming that [.class always returns the same class

# ff fixes this by

assignInNamespace(
  "[.AsIs"
  , function (x, i, ...){
    ret <- NextMethod("[")
    oldClass(ret) <- c("AsIs", oldClass(ret))
    ret
  }
  , "base"
  )

# Should go into Base R
```

Double inheritance and triple method dispatch

```
> class(x$gender)
[1] "ff_vector" "ff"
> class(x$gender[1:10])
[1] "factor"
# x$gender inherits from both - 'ff' and 'factor'
# i.e. we have a two-dimensional class inheritance
# currently:
# when subscripting, we restore class attributes of the vector
# however, second dimension is invisible in first class call
# and adding an S3 class fails

# in
x$gender[filter]
# "[" needs to be dispatched
# on classes of the object 'ff' and 'factor'
# and on class 'bit' of the subscript
# i.e. we have dispatch on three classes
# currently:
# we coerce the subscript to class 'hi', ff's hybrid index class
```

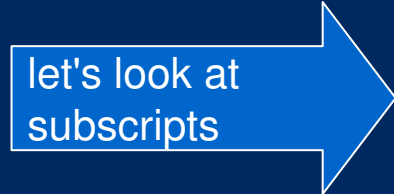
Are we done by simply using S4 classes? 6-dimensional dispatch?

Matrix has
3-dimensional dispatch

- storage.type
- dense vs. sparse
- type of matrix

R.ff could add yet another
3-dimensions

- RAM vs. disk based
- whole vs. chunked processing
- type of subscripts



let's look at
subscripts

What about the speed of the S4 class system?

Does S4 really solves all issues that arise in context of large objects?

- E.g. chunked iterating over all data
- Upper limit of chunk size to reduce RAM-need
- Lower limit on chunk size to avoid costly disk access
- Distributed calculation on multiple cores / cluster nodes
- Partitioned physical storage

We need more data-types and subscript-types

old data types

- logical
- integer (positive, negative)
- character (names)

data type
!=
subscript type

- `as.integer(logical)`
- `which(logical)` => make generic either `which` or `as.which`

new subscript
types

- 64bit integer for *virtual* addressing
(not necessarily 64bit pointers for *physical* addressing)
- `which` (need generic 'which' or 'as.which' for strictly positive)
- bit
- `whichbit` (FALSE or negint or posint or TRUE)
- `ri` (range index identifying a virtual chunk)
- `hi` (hybrid index storing the sorted, compressed physical access positions)
- ... ?

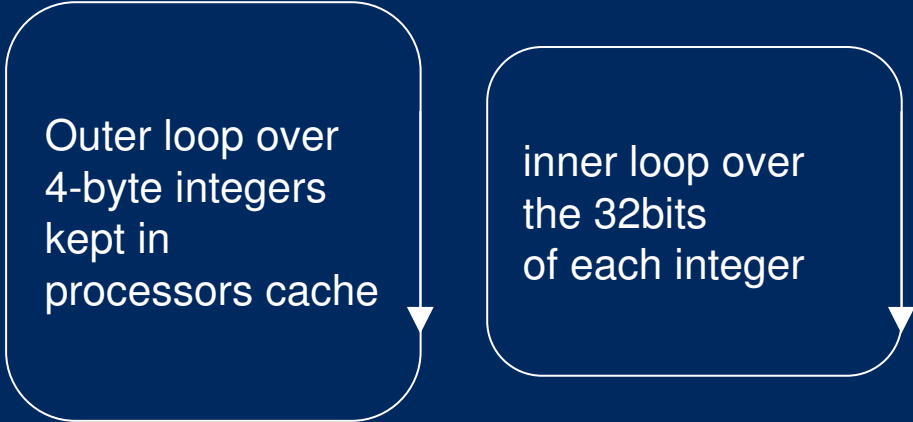
virtual subscript
!=
physical position

- Is a consequence of
- virtual windows (vw)
 - Different mappings from virtual structure to physical storage (e.g. non-standard dimorder, block cyclic layout for PBLAS)

Remember that subscripting is implicit looping

```
# take  
ff2[bit] <- ff1[bit]
```

Outer loop over
4-byte integers
kept in
processors cache



inner loop over
the 32bits
of each integer

```
# what we want is a  
# possibly compiled, chunked and parallelized iteration aka  
foreach (i in bit) %dopar% {ff2[i] <- ff1[i]}
```

```
# and what we want to avoid is evaluation of i1:i2 in  
foreach (i in i1:i2) %dopar% {ff2[i] <- ff1[i]}
```


Thesis: speed optimization with C-code is evil: let's kill .Call()

- An old rule in S says: do use the appropriate access function, do not rely on the internal structure of an object
- .Call is a severe violation of this – sensible – rule
- Calling C-code to obtain a faster looping of an operation over the elements of a vector assumes the vector is in-RAM and kills compatibility because it bypasses necessary method dispatches (e.g. if a 'vector' is 'ff' instead of standard RAM)
- Doing method dispatches during the loop would clearly kill performance
- Can't we have the dispatches in a looped expression BEFORE the loop and then do a compiled iteration with callbacks to the resolved functions?
- In order to do this, we would need to know that the dispatch would never change during the loop, i.e. that the classes of the involved objects do not change during the loop.
- Such a context information could be provided to the interpreter using *expression attributes*
- This would allow to write algorithms for untyped – but stable – objects

- Daniel Adler and Philipp Tassilo have proven with their revolutionary Rdyncall that a static foreign language interface like .Call is not needed to call binary code

Challenge: efficient processing might need physical iteration sequence different from virtual iteration sequence

```
# take
cumsum(vector[sample(length(vector))])

# or
cumsum(vt(matrix)) # virtually transposed matrix

# naively iterated would crazily jump between file positions

# the latter can be indirectly chunked
# by using block cyclic design of the matrix

# the former requires chunking of the loop itself
```

Example for chunked looping

```
y <- ff(vmode="integer", length=100)
chunk(1, 100, 30)
for (i in chunk(1, 100, 30)) y[i] <- i[1]:i[2]
b <- bit(100)
for (i in chunk(1, 100, 30)) b[i] <- y[i] > 50
summary(b)
```

What is the best chunk size – given available RAM, physical partition and number of cores?

Candidates for expression attributes

```
# require strict sequential processing  
# does not exist in SQL - R is more flexible  
{expr1, expr2, .sequential=TRUE}
```

```
# allow parallelization  
{expr1, expr2, .sequential=FALSE}
```

```
# how to parallelize  
{expr1, expr2, .parallel="multicore"}  
{expr1, expr2, .parallel="MPI"}
```

```
# when to dispatch  
{expr1, expr2, .dispatch="static"}  
{expr1, expr2, .dispatch="dynamic"}
```

```
# when to parse  
{expr1, expr2, .parse="compile"}  
{expr1, expr2, .parse="interpret"}
```

Candidates for function attributes

```
runif(n=4, min=c(1, 10, 100), max=c(2, 20, 200))
```

not
recycled

recyled

recyled

return
length
taken
from
this
scalar
or length
of this
object

return
type
taken
from
??

Other topics

- efficient and identical ordering: quicksort vs. merge sort
- indexing (b-tree, quad-tree, r-trees, ...)
- summary is too dynamic in its return structure
- factor treatment is inconsistent, compare `c()` and `rbind.data.frame()`
- `ffapply`: expressions or function calls?
- transparent partitioning of `ff` into smaller files could help performance, could this be synchronized with `pblas`?

**SOME DETAILS
NOT PRESENTED
IN THE SESSION**

Atomic data types supported by ff

implemented
 not implemented

<code>vmode(x)</code>		
<code>boolean</code>	1 bit logical	no NA
<code>logical</code>	2 bit logical	with NA
<code>quad</code>	2 bit unsigned integer	no NA
<code>nibble</code>	4 bit unsigned integer	no NA
<code>byte</code>	8 bit signed integer	with NA
<code>ubyte</code>	8 bit unsigned integer	no NA
<code>short</code>	16 bit signed integer	with NA
<code>ushort</code>	16 bit unsigned integer	no NA
<code>integer</code>	32 bit signed integer	with NA
<code>single</code>	32 bit float	
<code>double</code>	64 bit float	
<code>complex</code>	2x64 bit float	
<code>raw</code>	8 bit unsigned char	
<code>character</code>	fixed widths, tbd.	

Compounds

- `factor`
- `ordered`
- `custom compounds`
 - `Date`
 - `POSIXct`
 - `POSIXlt` (not atomic)

Supported data structures

soon on CRAN



prototype available



not yet implemented



	example	class(x)
vector	<code>ff(1:12)</code>	<code>c("ff_vector", "ff")</code>
array	<code>ff(1:12, dim=c(2,2,3))</code>	<code>c("ff_array", "ff")</code>
matrix	<code>ff(1:12, dim=c(3,4))</code>	<code>c("ff_matrix", "ff_array", "ff")</code>
data.frame	<code>ffdf(sex=a, age=b)</code>	<code>c("ffdf", "ff")</code>
symmetric matrix with free diag	<code>ff(1:6, dim=c(3,3), symm=TRUE, fixdiag=NULL)</code>	
symmetric matrix with fixed diag	<code>ff(1:3, dim=c(3,3), symm=TRUE, fixdiag=0)</code>	
distance matrix		<code>c("ff_dist", "ff_symm", "ff")</code>
mixed type arrays instead of data.frames		<code>c("ff_mixed", "ff")</code>

Supported index expressions

implemeneted
not implemented

```
x <- ff(1:12, dim=c(3,4), dimnames=list(letters[1:3], NULL))  
i <- as.bit(c(TRUE, FALSE, FALSE))
```

<u>expression</u>	<u>Example</u>
(which) positive integers	<code>x[1, 1]</code>
negative integers	<code>x[-(2:12)]</code>
logical	<code>x[c(TRUE, FALSE, FALSE), 1]</code>
character	<code>x["a", 1]</code>
integer matrices	<code>x[rbind(c(1,1))]</code>
bit	<code>x[i & i, 1]</code>
bitwhich	<code>x[as.bitwhich(i), 1]</code>
range index	<code>x[ri(1,1), 1]</code>
hybrid index	<code>x[as.hi(1) ,1]</code>
zeros	<code>x[0]</code>
NAs	<code>x[NA]</code>

ff's internal index