DISCLOSED

Managing large datasets in R – ff examples and concepts

Jens Oehlschlägel

Vienna

January 2010

Summary

The statistical interpreter R is hungry for RAM and therefore limited to dataset sizes much smaller than available RAM. R packages 'bit' and 'ff' provide the basic infrastructure to handle large data problems in R. In this session we give an introduction into 'bit' and 'ff' – interweaving working examples with short explanation of the most important concepts.

An open-source vision for R

give free access to excellent statistical software to everyone …

… for processing large datasets even with standard hardware

Design goals: base packages for large data objects in R

large data

- many objects (sum(sizes) > RAM and …)
- large objects (size > RAM and virtual address space limitations)

~~"the cloud"~~

standard HW

- limited RAM (or enjoy speedups)
- single disk (or enjoy RAID)
- single processor (or shared processing)

minimal RAM

- required RAM << maximum RAM
- be able to process large data in background

maximum performance

- close to in-RAM performance if size < RAM (system cache)
- still able to process if size > RAM
- avoid costs of redundant access to disk (time, electricity, CO_2)

Source: Oehlschlägel (2010) Managing large datasets in R – ff examples and concepts

Authors

Daniel Adler dadler@uni-goettingen.de

package rgl
package ff since 1.0
dyncall project: dynamic C interface
in work: dynamic compilation and streaming for R

Jens Oehlschlägel Jens_Oehlschlaegel@truecluster.com

package bit since 1.0
package ff since 2.0
prototype R.ff

Development

```
http://r-forge.r-project.org/projects/ff/
```

Production

```
http://cran.r-project.org/web/packages/ff
```

Statistical methods that can be scaled to large data problems using the infrastructure provided by packages bit and ff

Basic infrastructure for large objects	packages bit and ff
Basic infrastructure for chunking	packages bit and ff
Reading and writing csv files	(chunked sequential access with package ff)
Data transformation	(chunked and partially parallelized with package R.ff)
Math and probability distributions	(chunked and parallelized with package R.ff)
Data filtering	(chunked and parallelized, supported by bit and ff)
Descriptive statistics on chunks	(chunked and parallelized, supported by ff)
Biglm and bigglm	(chunked fitting with package biglm)
Bootstrapping	(chunked and parallelized random access)
Bagged predictive modelling	(chunked and parallelized random access)
Bagged clustering	(chunked and parallelized random access with truecluster)
Likelihood maximization	(chunked and parallelized sequential access)
EM-algorithm	(chunked and partially parallelized)
Some linear algebra	(chunked processing with package R.ff)

- matrix transpose
- matrix multiplication
- matrix inversion
- svd for many rows and few columns

new today:
ffsave / ffload

Saving and loading ff archives	(incremental save and selective load with package ff)

Infrastructure in bit and ff addresses a complexity of performance-critical topics

Complexities in scope of ff and bit

• virtual objects by reference	avoid copying and RAM duplication
• disk based objects	manage temporary and permanent objects on disk
• memory efficient data types	minimize storage requirements of data
• memory efficient subscript types	minimize storage requirements of subscripts
• fast chunk access methods	allow fast (random) access to disk – for chunks of data
• fast index (pre)processing	minimize or avoid cost of subscript preparation
• chunk processing infrastructure	foundation for efficient chunked processing
• large csv import/export	interface large datasets
• large data management	conveniently manage all files behind ff

Complexities partially in scope

• parallel processing	parallel access to large datasets (without locking)

Complexities in scope of R.ff

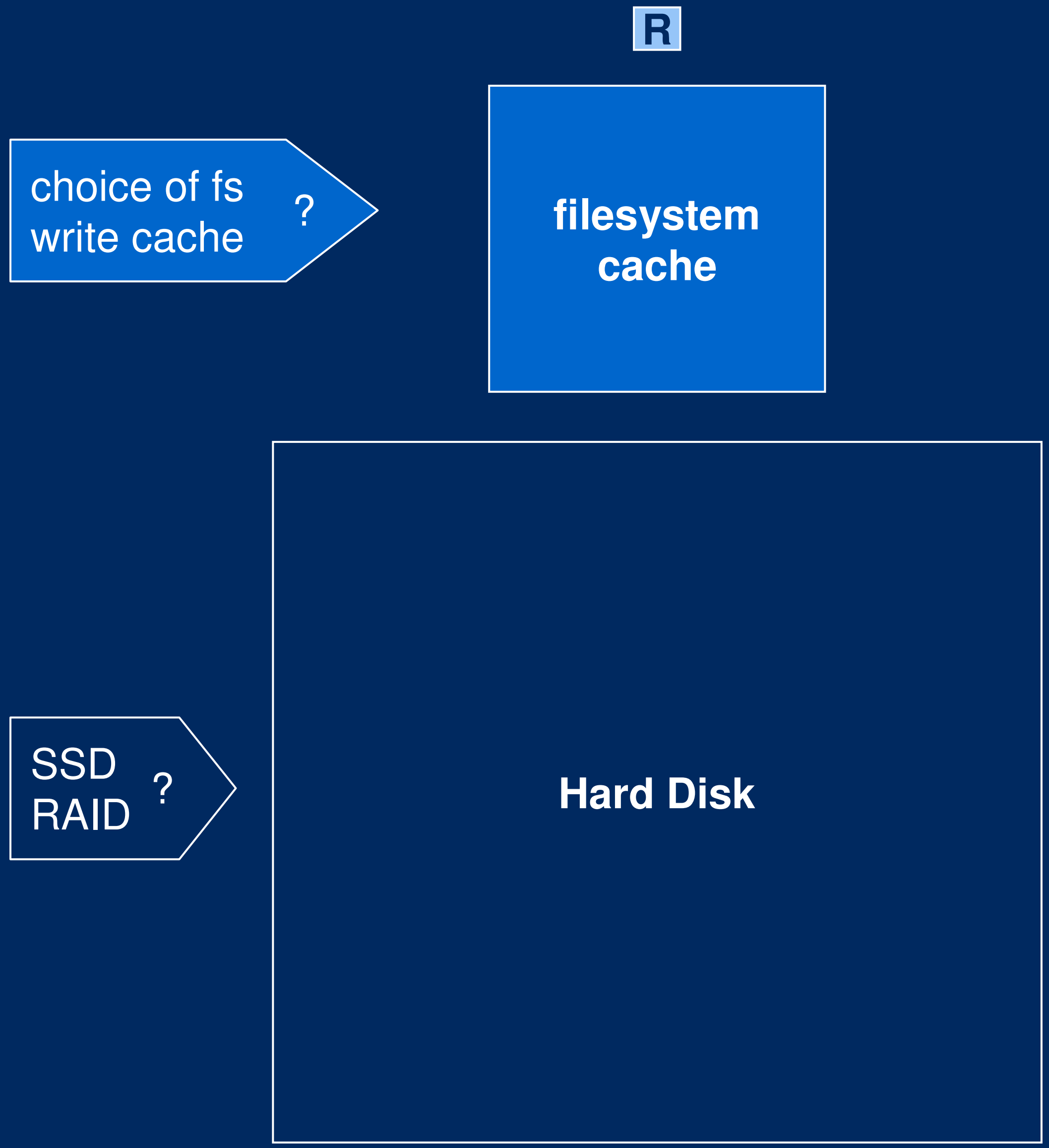
• basic processing of large objects	elementwise operations and more
• some linear algebra for large objects	t, vt, matmul, matinv, few column svd

Currently not in scope

• full linear algebra for large objects	observe package bigmemory

Source: Oehlschlägel (2010) Managing large datasets in R – ff examples and concepts

Basic memory organisation



Architecture behind packages ff and bit implements several performance optimizations

R frontend — Hybrid Index Preprocessing ...

- HIP
  - *parsing* of index expressions instead of memory consuming evaluation
  - *ordering* of access positions and re-ordering of returned values
  - rapid rle *packing* of indices if and only if rle representation uses less memory compared to raw storage
- Hybrid copying semantics
  - virtual `dim/dimorder()`
  - virtual windows `vw()`
  - virtual transpose `vt()`
- New performance generics
  - `clone()`, `update()`, `swap()`, `add()`, `chunk()`, `bigsample()`
- Efficient coercions

C interface — ... fast access methods ...

- C-code accelerating is.unsorted() and rle() for integers: `intisasc()`, `intisdesc()`, `intrle()`
- C-code for looping over hybrid index can handle mixed raw and rle packed indices in arrays and avoids multiplication
- C-code for looping over bit: outer loop fixes word in processor cache, inner loop handles bits

C++ backend — ... memory mapped compressed pages

- Tunable `pagesize` and system `caching=c("mmnoflush", "mmeachflush")`
- Custom datatype `bit-level en/decoding`, ‚`add`' arithmetics and `NA` handling
- `Ported` to Windows, Mac OS, Linux and BSDs
- `Large File Support` (>2GB) on Linux
- `Paged shared memory` allows parallel processing
- `Fast creation and modification` of large files on sparse filesystems

Source: Oehlschlägel (2010) Managing large datasets in R – ff examples and concepts

Basic example: creating atomic vectors

```
# R example
ri <- integer(10)
```

```
# ff examples
library(ff)
fi <- ff(vmode="integer", length=10)
fb <- ff(vmode="byte", length=10)

rb <- byte(10) # in R this is integer
fb <- ff(rb)

vmode(ri)
vmode(fi)
vmode(rb)
vmode(fb)

cbind(.rambytes, .ffbytes)[c("integer","byte"),]
?vmode
```

Atomic data types supported by ff

implemented
not implemented

vmode(x)		
boolean	1 bit logical	no NA
logical	2 bit logical	with NA
quad	2 bit unsigned integer	no NA
nibble	4 bit unsigned integer	no NA
byte	8 bit signed integer	with NA
ubyte	8 bit unsigned integer	no NA
short	16 bit signed integer	with NA
ushort	16 bit unsigned integer	no 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	fixed widths, tbd.	

Compounds

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

Source: Oehlschlägel (2010) Managing large datasets in R – ff examples and concepts

Advanced example: creating atomic vectors

```
# R example
rf <- factor(levels= c("A","T","G","C"))
length(rf) <- 10
rf
```

```
# ff examples
frf <- ff(rf)
length(frf) <- 1e8
frf
frf[11:1e8] <- NA
ff(vmode="quad", length=1e8, levels=c("A","T","G","C"))
ff(vmode="quad", length=10
 , levels=c("A","B","C","D"), ordered=TRUE)

ff(Sys.Date()+0:9, length=10)
ff(Sys.time()+0:9, length=10)
ff(0:9, ramclass="Date")
ff(0:9, ramclass=c("POSIXt", "POSIXct"))
str(ff(as.POSIXct(as.POSIXlt(Sys.time(), "GMT")), length=12))
```

Source: Oehlschlägel (2010) Managing large datasets in R – ff examples and concepts

Limiting R's RAM consumption through chunked processing

```
# ff example
> str(chunk(fd))
List of 50
 $ :Class 'ri'  int [1:3] 1 2000000 100000000
 $ :Class 'ri'  int [1:3] 2000001 4000000 100000000
 $ :Class 'ri'  int [1:3] 4000001 6000000 100000000
[snipped]

# simple as seq but balancing chunk size
> args(chunk.default)
function (from = NULL, to = NULL, by = NULL, length.out = NULL
, along.with = NULL, overlap = 0L, method = c("bbatch", "seq"), ...)

# automatic calculation of chunk size
> args(chunk.ff_vector)
function (x
, RECORDBYTES = .rambytes[vmode(x)]
, BATCHBYTES = getOption("ffbatchbytes")
, ...)

# by default limited at 1% of available RAM
> getOption("ffbatchbytes") / 1024^2 / memory.limit()
[1] 0.01
```

Source: Oehlschlägel (2010) Managing large datasets in R – ff examples and concepts

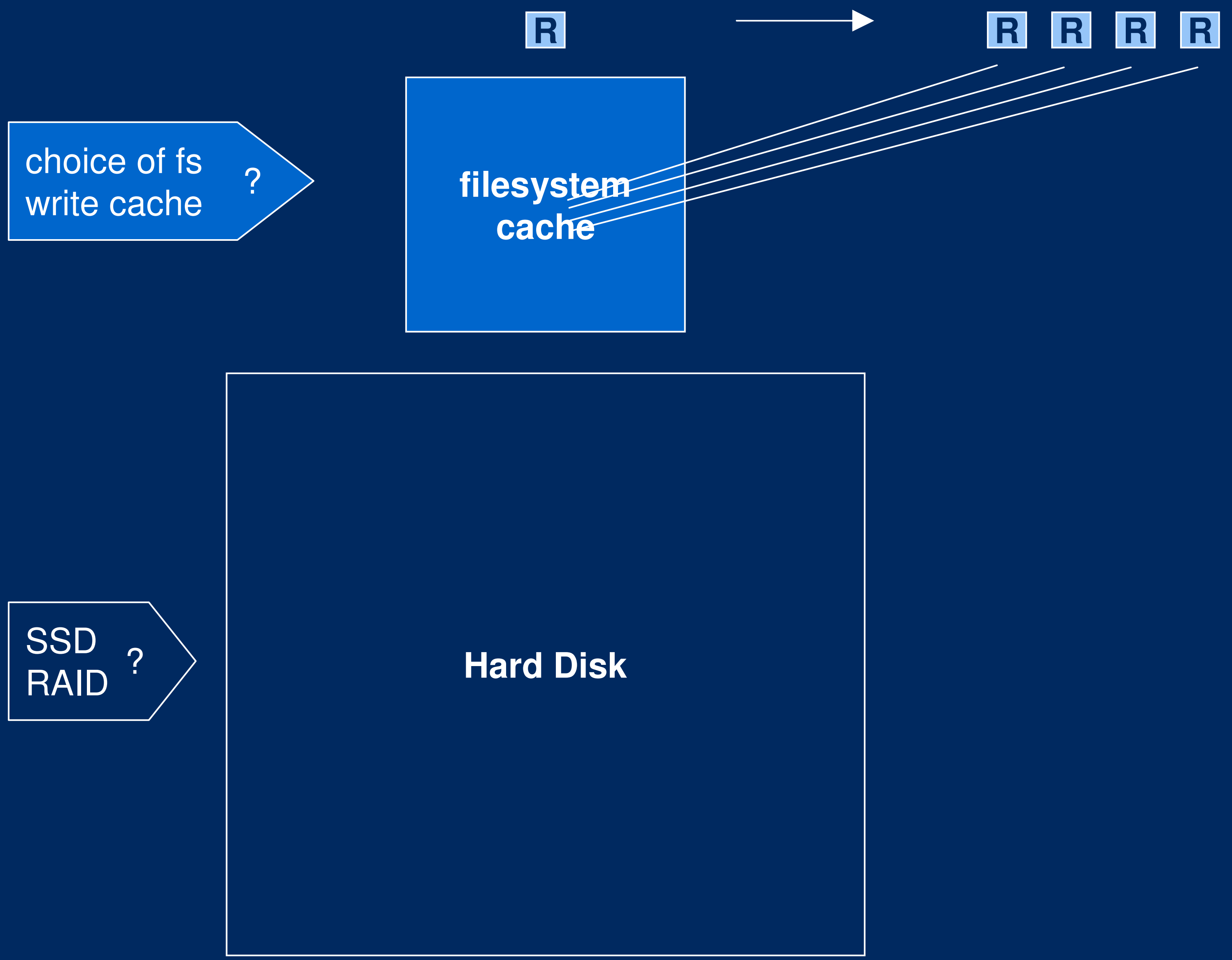
Basic example: working with atomic vectors

```
# R example
rd <- double(100)
rd[] <- runif(100)  # write
rd[] # this is the proper non-lazy way to read
```

```
# ff example
fd <- ff(vmode="double", length=1e8)

system.time(
    for (i in chunk(fd)) fd[i] <- runif(sum(i))
)

system.time(
s <- lapply( chunk(fd)
, function(i)quantile(fd[i], c(0.05, 0.95)) )
)

crbind(s)
```

Source: Oehlschlägel (2010) Managing large datasets in R – ff examples and concepts

Negligible RAM duplication for parallel execution on ff objects



Advanced example: working parallel with atomic vectors

```
library(snowfall)
finalizer(fd)
# let slaves not delete fd on shutdown
finalizer(fd) <- "close"

sfInit(parallel=TRUE, cpus=2, type="SOCK")
sfLibrary(ff)
sfExport("fd")  # do not export the same ff multiple times
sfClusterEval(open(fd)) # explicitely opening avoids a gc problem
system.time(
    sfLapply( chunk(fd), function(i){
        fd[i] <- runif(sum(i))
        invisible()
    })
)
system.time(
    s <- sfLapply( chunk(fd)
    , function(i) quantile(fd[i], c(0.05, 0.95)) )
)
sfClusterEval(close(fd)) # for completeness
csummary(s)
sfStop()
```

Supported index expressions

implemeneted	(white box)
not implemented	(grey box)

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

expression	Example
(which) positive integers	`x[ 1, 1]`
negative integers	`x[ -(2:12) ]`
logical	`x[ c(TRUE, FALSE, FALSE), 1]`
character	`x[ "a", 1]`
integer matrices	`x[ rbind(c(1,1)) ]`
bit	`x[ i & i, 1]`
bitwhich	`x[ as.bitwhich(i), 1]`
range index	`x[ ri(1,1), 1]`
hybrid index ← ff's internal index	`x[ as.hi(1) ,1]`
zeros	`x[ 0 ]`
NAs	`x[ NA ]`

Basic example: working with bit filters

```
# R example
l <- rd > 0.99
rd[l]
1e8 * .rambytes["logical"] / (1024^2) # 381 MB for logical
```

```
# ff example
b1 <- b2 <- bit(length(fd))
system.time( b1[] <- c(FALSE, TRUE) )
system.time( for (i in chunk(fd)) b2[i] <- fd[i] > 0.99 )
system.time( b <- b1 & b2 )
object.size(b) / (1024^2)
system.time( x <- fd[b] )
x[1:10]

sum(b) / length(b)  # less dense than 1/32

w <- as.bitwhich(b)
sum(w) / length(w)
object.size(w) / (1024^2)
system.time( x <- fd[w] )
x[1:10]
```

Advanced example: working with hybrid indexing

```
# ff example
hp <- as.hi(b)          # ignores pack=FALSE
object.size(hp) / (1024^2)
system.time( x <- fd[hp] )
x[1:10]

hu <- as.hi(w, pack=FALSE)
object.size(hu) / (1024^2)
system.time( x <- fd[hu] )
x[1:10]

# Don't do
as.hi(1:1e8)

# Do
as.hi(quote(1:1e8))
hi(1, 1e8)
ri(1, 1e8)
chunk(1, 1e8, by=1e8)
```

Supported data structures

soon on CRAN	(white box)
prototype available	(blue box)
not yet implemented	(grey box)

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

Basic example: working with arrays (and matrices)

```
# R example: physically stored by column
array(1:12, dim=c(3,4))       # read by column
matrix(1:12, 3,4, byrow=TRUE) # read by row
```

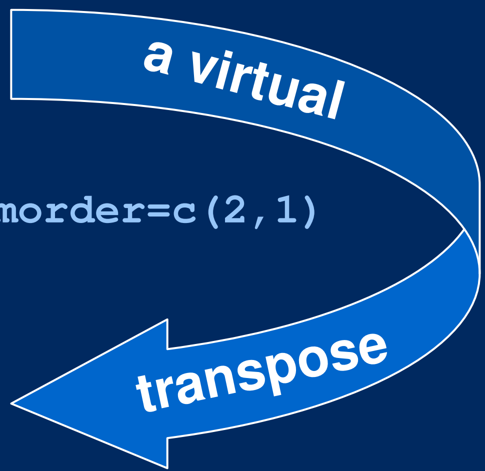
```
# ff example: physically stored by column - like columnar OLAP
ff(1:12, dim=c(3,4))                       # read by column
ff(1:12, dim=c(3,4), bydim=c(2,1))         # read by row

# ff example: physically stored by row - like OLTP database
ff(1:12, dim=c(3,4), dimorder=c(2,1))      # read by column
ff(1:12, dim=c(3,4), dimorder=c(2,1), bydim=c(2,1)) # read by row

fm <- ff(1:12, dim=c(3,4), dimorder=c(2,1))
get.ff(fm, 1:12) # note the physical order
fm[1:12]         # [. exhibits standard R behaviour
ncol(fm) <- 1e8  # not possible with this dimorder
nrow(fm) <- 1e8  # possible with this dimorder

fm <- ff(vmode="double", dim=c(1e4, 1e4))
system.time( fm[1,] <- 1 )  # column store: slow
system.time( fm[,1] <- 1 )  # column store: fast
# even more pronounced difference for caching="mmeachflush"
```

Hybrid copying semantics: physical and virtual attributes

```
x <- ff(1:12, dim=c(3,4))
x

> str(physical(x))
List of 9
 $ vmode    : chr "integer"                              ff(); vmode()
 $ maxlength: int 12                                 ff(); maxlength()
 $ pattern  : chr "ff"                                     pattern()<-
 $ filename : chr                                         filename()<-
 "D:/DOCUME~1/JENSOE~1/LOCALS~1/Temp/RtmpKDilKg/ff462a1d5a.ff"
 $ pagesize : int 65536                              open(, pagesize=)
 $ finalizer: chr "delete"                               finalizer()<-
 $ finonexit: logi TRUE                                           ff()
 $ readonly : logi FALSE                             open(, readonly=)
 $ caching  : chr "mmnoflush"                         open(, caching=)

> str(virtual(x))
List of 4
 $ Length   : int 12                                        length()<-
 $ Dim      : int [1:2] 3 4                                    dim()<-
 $ Dimorder : int [1:2] 1 2                               dimorder()<-
 $ Symmetric: logi FALSE                             ff(); symmetric()
```

Hybrid coyping semantics in action: different virtual ‘views’ into same ff

```
a <- ff(1:12, dim=c(3,4))
b <- a
dim(b) <- c(4,3)
dimorder(b) <- c(2:1)

> a
ff (open) integer length=12(12) dim=c(3,4) dimorder=c(1,2)
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> b
ff (open) integer length=12(12) dim=c(4,3) dimorder=c(2,1)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
[4,]   10   11   12
```




```
vt(a)  # shortcut to virtually transpose
t(a)   # == clone(vt(a))
```

Basic example: working with data.frames

```
# R example
id <- 1:12
gender <- sample(factor(c("male","female","unknown")), 12, TRUE)
rating <- matrix(sample(1:6, 12*10, TRUE), 12, 10)
colnames(rating) <- paste("r", 1:10, sep="")
df <- data.frame(id, gender, rating)
df[1:3,]
```

```
# ff example
fid <- as.ff(id); fgender <- as.ff(gender); frating <- as.ff(rating)
fdf <- ffdf(id=fid, gender=fgender, frating)
identical(df, fdf[,])
fdf[1:3,]    # data.frame
fdf[,1:4]    # data.frame
fdf[1:4]     # ffdf
fdf[]        # ffdf
fdf[[2]]     # ff
fdf$gender   # ff
```

Source: Oehlschlägel (2010) Managing large datasets in R – ff examples and concepts

Advanced example: physical structure of data.frames

```
# R example
# remember that 'rating' was a matrix
# but data.frame has copied to columns (unless we use I(rating))
str(df)
```

```
# ff example
physical(fdf)  # ffdf has *not* copied anything
# lets' physically copy
fdf2 <- ffdf(id=fid, gender=fgender, frating, ff_split=3)
physical(fdf2)
filename(fid)
filename(fdf$id)
filename(fdf2$id)
nrow(fdf2) <- 1e6
fdf2[1e6,] <- fdf[1,]
fdf2
nrow(fdf2) <- 12
# understand this error: pros and cons of embedded ff_matrix
nrow(fdf) <- 16
# understand what this does to the original fid and fgender
fdf3 <- fdf[1:2]
nrow(fdf3) <- 16
fgender
nrow(fdf3) <- 12
```

Basic example: reading and writing csv

```
# R example
write.csv(df, file="df.csv")
cat(readLines("df.csv"), sep="\n")
df2 <- read.csv(file="df.csv")
df2
```

```
# ff example
write.csv.ffdf          # we leverage R's original functions
args(write.table.ffdf)  # arguments of the workhorse function

write.csv.ffdf(fdf, file="fdf.csv")
cat(readLines("fdf.csv"), sep="\n")

fdf2 <- read.csv.ffdf(file="fdf.csv")
fdf2
identical(fdf[,], fdf2[,])
```

Advanced example: physical specification when reading a csv

```
# ff example
vmode(fdf2)
args(read.table.ffdf)
fdf2 <- read.csv.ffdf(file="fdf.csv"
, asffdf_args = list(vmode=list(quad="gender", nibble=3:12))
, first.rows = 1
, next.rows = 4
, VERBOSE=TRUE)
fdf2
fdf3 <- read.csv.ffdf(file="fdf.csv"
, asffdf_args = list( vmode=list(nibble=3:12)
                    , ff_join=list(3:12)
                )
)
fdf3
# understand "unknown factor values mapped to first level"
read.csv.ffdf(file="fdf.csv"
, asffdf_args = list(vmode=list(quad="gender", nibble=3:12))
, appendLevels = FALSE
, first.rows = 2
, VERBOSE=TRUE)
```

Basic example: file locations and file survival

```
# R example: objects are not permanent
# rm(df) simply removes the object
# when closing R with q() everything is gone
```

```
# ff example: object data in files is POTENTIALLY permanent
# rm(fdf) just removes the R object,
# the next gc() triggers a finalizer which acts on the file
# when closing R with q()
# the attribute finonexit decides whether the finalizer is called
# finally fftempdir is unlinked
physical(fd)
dir(getOption("fftempdir"))

# changing file locations and finalizers
sapply(physical(fdf2), finalizer)
filename(fdf2)                      # filename(ff) <- changes one ff
pattern(fdf2) <- "./cwdpat_"
filename(fdf2)                      # ./ renamed to cwdpat_ in getwd()
sapply(physical(fdf2), finalizer) # AND set finalizers to "close"
pattern(fdf2) <- "temppat_"
filename(fdf2)                      # renamed to temppat_ in fftempdir
sapply(physical(fdf2), finalizer) # AND set finalizers to "delete"
```

Closing example: managing ff archives

```
# R example
# save.image() saves all R objects (but not ff files)
# load() restores all R objects (if ff files exist, ff objects work)
```

```
# ff example

# get rid of the large objects (takes too long for demo)
delete(frf); rm(frf)
delete(fd); rm(fd)
delete(fm); rm(fm)
rm(b, b1, b2, w, x, hp, hu)

ffsave.image(file="myff")  # => myff.RData + myff.ffData
# close R
# open R
library(ff)
str(ffinfo(file="myff"))
ffload(file="myff", list="fdf2")
sapply(physical(fdf2), finalizer)

yaff <- ff(1:12)
ffsave(yaff, file="myff", add=TRUE)
ffdrop(file="myff")
```

SOME DETAILS NOT PRESENTED IN THE SESSION

Before you start, make sure you read the important warnings in

```
library(ff)
?LimWarn
```

The default behavior of ff can be configured in `options()`

```
getOption("fftempdir")     == "D:/.../Temp/RtmpidNQq9"
getOption("ffextension")   == "ff"
getOption("fffinonexit")   == TRUE
getOption("ffdrop")        == TRUE         # or always drop=FALSE
getOption("ffpagesize")    == 65536        # getdefaultpagesize()
getOption("ffcaching")     == "mmnoflush"  # or "mmeachflush"
getOption("ffbatchbytes") == 16104816     # default 1% of RAM Win32
                                           # set this under Linux !!
```

Behavior on `rm()` and on `q()`

If we create or open an ff file, C++ resources are allocated, the file is opened and a finalizer is attached to the external pointer, which will be executed at certain events to release these resources.

Available finalizers

`close`	releases C++ resources and closes file (default for named files)
`delete`	releases C++ resources and deletes file (default for temp files)
`deleteIfOpen`	releases C++ resources and deletes file only if file was open

Finalizer is executed

`rm(); gc()`	at next garbage collection after removal of R-side object
`q()`	at the end of an R-session (only if **`finonexit=TRUE`**)

Wrap-up of temporary directory

`.onUnload`	**`getOption("fftempdir")`** is unliked and all ff-files therein deleted

You need to understand these mechanisms, otherwise you might suffer …

…	unexpected loss of ff files
…	GBs of garbage somewhere in temporary directories

Check and set the defaults to your needs …

…	**`getOption("fffinonexit")`**

Update, cloning and coercion

```
# plug in temporary result into original ff object
update(origff, from=tmpff, delete=TRUE)

# deep copy with no shared attributes
y <- clone(x)

# three variants to cache complete ff object
# into R-side RAM and write back to disk later

# variant deleting ff
ramobj <- as.ram(ffobj); delete(ffobj)
# some operations purely in RAM
ffobj  <- as.ff(ramobj)

# variant retaining ff
ramobj <- as.ram(ffobj); close(ffobj)
# some operations purely in RAM
ffobj  <- as.ff(ramobj, overwrite=TRUE)

# variant using update
ramobj <- as.ram(ffobj)
update(ffobj, from=ramobj)
```

Atomic access functions, methods and generics

	reading	writing	combined reading and writing
single element	**get.ff**	**set.ff**	**getset.ff**
contiguous vector	**read.ff**	**write.ff**	**readwrite.ff**
indexed access with HIP and vw for ram compatibility	**[**	**[<-.(,add=FALSE)** **add(x,i,value)**	**swap(,add=FALSE)** **swap.default**

HIP optimized disk access

Hybrid Index Preprocessing (HIP)

ffobj[1:1000000000] will silently submit the index information to as.hi(quote(1:1000000000)) which does the HIP:

- rather parses than expands index expressions like 1:1000000000
- stores index information either plain or as rle-packed index increments (therefore 'hybrid')
- sorts the index and stores information to restore order

Benefits

- minimized RAM requirements for index information
- all elements of ff file accessed in strictly increasing position

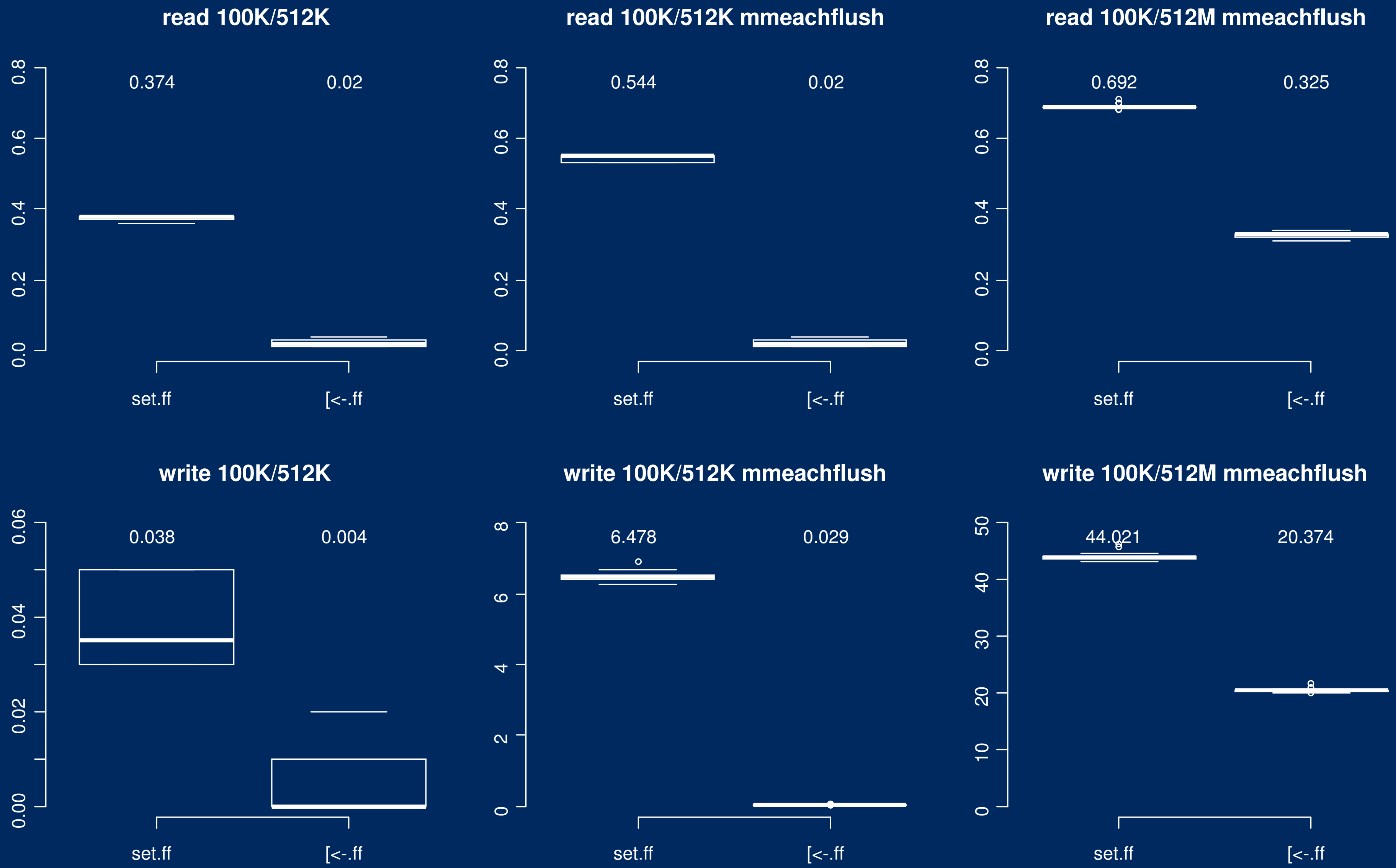
Costs

- RAM needed for HI may double RAM for plain index (due to re-ordering)
- RAM needed during HIP may be higher than final index (due to sorting)

Currently preprocessing is almost purely in R-code
(only critical parts in fast C-code: intisasc, intisdesc, inrle)

Random access to permanent vector of integer: cost of HIP in [.ff and [<-.ff is well-invested compared to naive access in get.ff and set.ff

seconds



Source: Oehlschlägel (2010) Managing large datasets in R – ff examples and concepts

Parsing of index expressions

```
# The parser knows `c()' and `:', nothing else
# [.ff calls as.hi like as.hi(quote(index.expression))

# efficient index expressions
a <- 1
b <- 100
as.hi(quote(c(a:b, 100:1000)))   # parsed (packed)
as.hi(quote(c(1000:100, 100:1))) # parsed and reversed (packed)

# neither ascending nor descending sequences
as.hi(quote(c(2:10,1)))   # parsed, but then expanded and sorted
                          # plus RAM for re-ordering

# parsing aborted when finding expressions with length>16
x <- 1:100; as.hi(quote(x)) # x evaluated, then rle-packed
as.hi(quote((1:100)))       #() stopped here, ok in a[(1:100)]

# parsing skipped
as.hi(1:100)                # index expanded , then rle-packed
# parsing and packing skipped
as.hi(1:100, pack=FALSE)              # index expanded
as.hi(quote(1:100), pack=FALSE)       # index expanded
```

RAM considerations

```
# ff is currently limited to length(ff)==.Machine$max.integer

# storing 370 MB integer data
> a <- ff(0L, dim=c(1000000,100))

# obviously 370 MB for return value
b <- a[]

# zero RAM for index or recycling
a[] <- 1    # thanks to recycling in C
a[] <- 0:1
a[1:100000000] <- 0:1  # thanks to HIP
a[100000000:1] <- 1:0

# Attention: 370 MB for recycled value
a[, bydim=c(2,1)] <- 0:1

# don't do this
a[offset+(1:100000000)] <- 1 # better: a[(o+1):(o+n)] <- 1

# 5x 370MB  during HIP        # Without chunking final costs are
a[sample(100000000)] <- 1     # 370 MB index + 370 MB re-order
a[sample(100000000)] <- 0:1   # dito + 370 MB recycling
```

Lessons from RAM investigation

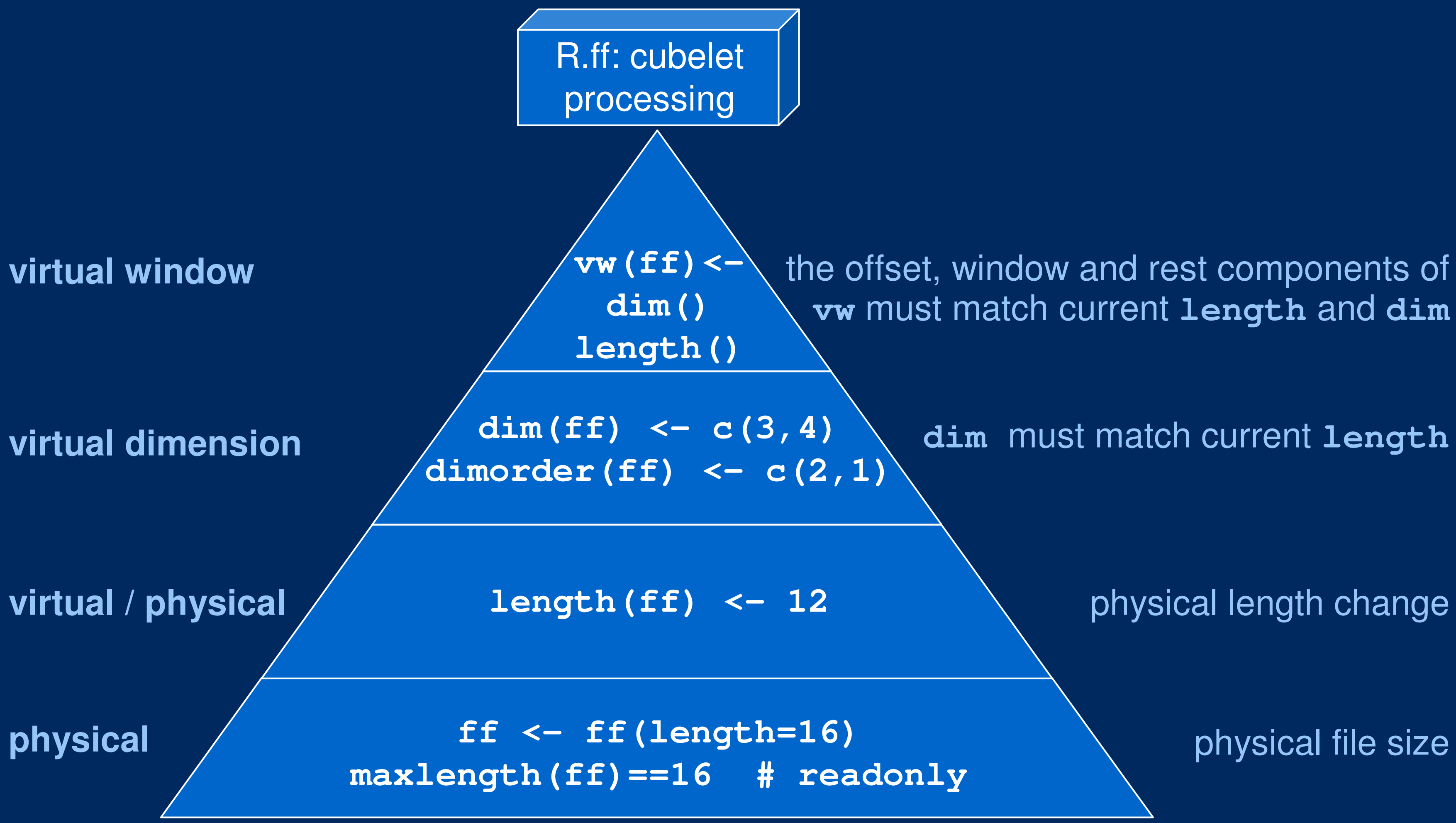
rle() requires up to **9x** its input RAM*

without using structure() reduces to **7x** RAM

intrle() uses an optimized C version, needs
up to **2x** RAM and is by factor 50 faster.
Trick: intrle returns NULL if compression
achieved is worse than 33.3%. Thus the RAM
needed is maximal
- 1/1 for the input vector
- 1/3 for collecting values
- 1/3 for collecting lengths
- 1/3 buffer for copying to return value

* Measured in version 2.6.2

A physical to virtual hierarchy



Source: Oehlschlägel (2010) Managing large datasets in R – ff examples and concepts