DISCLOSED

Managing data.frames with package 'ff' and fast filtering with package 'bit'

Oehlschlägel, Adler

Munich, Göttingen

July 2009

SUMMARY

We explain the new capability of package 'ff 2.1.0' to store large dataframes on disk in class 'ffdf'. ffdf objects have a virtual and a physical component. The virtual component defines a behavior like a standard dataframe, while the physical component can be organized to optimize the ffdf object for different purposes: minimal creation time, quickest column access or quickest row access. Furthemore ffdf can be defined without rownames, with in-RAM rownames or with on-disk rownames using a new ff class 'fffc' for fixed width characters.

Package 'bit' provides fast logical filtering: logical vectors in-RAM with only 1-bit memory consumption. It can be used standalone, but also nicely integrates with package 'ff': 'bit' objects can be coerced to boolean 'ff' and vice-versa (as.ff, as.bit), 'bit' objects can also be coerced to 'ff's subscript objects (as.hi). The latter and many other methods support a 'range' argument, which helps batched processing of large objects in small memory chunks.

The following methods are available for objects of class 'bit': logical operators: !, !=, ==, <=, >=, <, >, &, |, xor; aggregation methods: all, any, max, min, range, summary, sum, length; access methods: [[, [[<-, [, [<-; concatenation: c, coercion: as.bit, as.logical, as.integer, which, as.bitwitch. The bit-operations are by factor 32 faster on 32-bit machines. In order to fully exploit this speed, package 'bit' comes with minimal checking.

A second class 'bitwhich' allows storing boolean vectors in a way compatible with R's subscripting, but more efficiently than logical vectors: all==TRUE is represented as TRUE, !any is represented as FALSE, other selections are represented by positive or negative integer subscripts, whatever needs less ram. Logical operators !, &, |, xor use set operations which is efficient for highly skewed (asymmetric) data, where either a small part of the data is selected or excluded and such filters are to be combined.

We show how packages 'ff' and 'snowfall' nicely complement each other: snowfall helps to parallelize chunked processing on 'ff' objects, and 'ff' objects allow exchanging data between snowfall master and slaves without memory duplication. We give an online demo of 'ff', 'bit' and 'snowfall' on a standard notebook with an 80 mio row dataframe – size of a German census :-)

OVERVIEW

Package 'ff' 2.1.0

- provides large, fast disk-based vectors and arrays
- NEW: fast length()<- increase for ff vectors
- NEW: dataframes (ffdf) and fixed width characters (fffc)
- NEW: lean datatypes on CRAN under GPL

Package 'bit' 1.1.0

- Class 'bit': lean in-memory boolean vectors + fast operators
- Class 'bitwhich': alternative for very skewed filters
- Close integration with ff objects and chunked processing

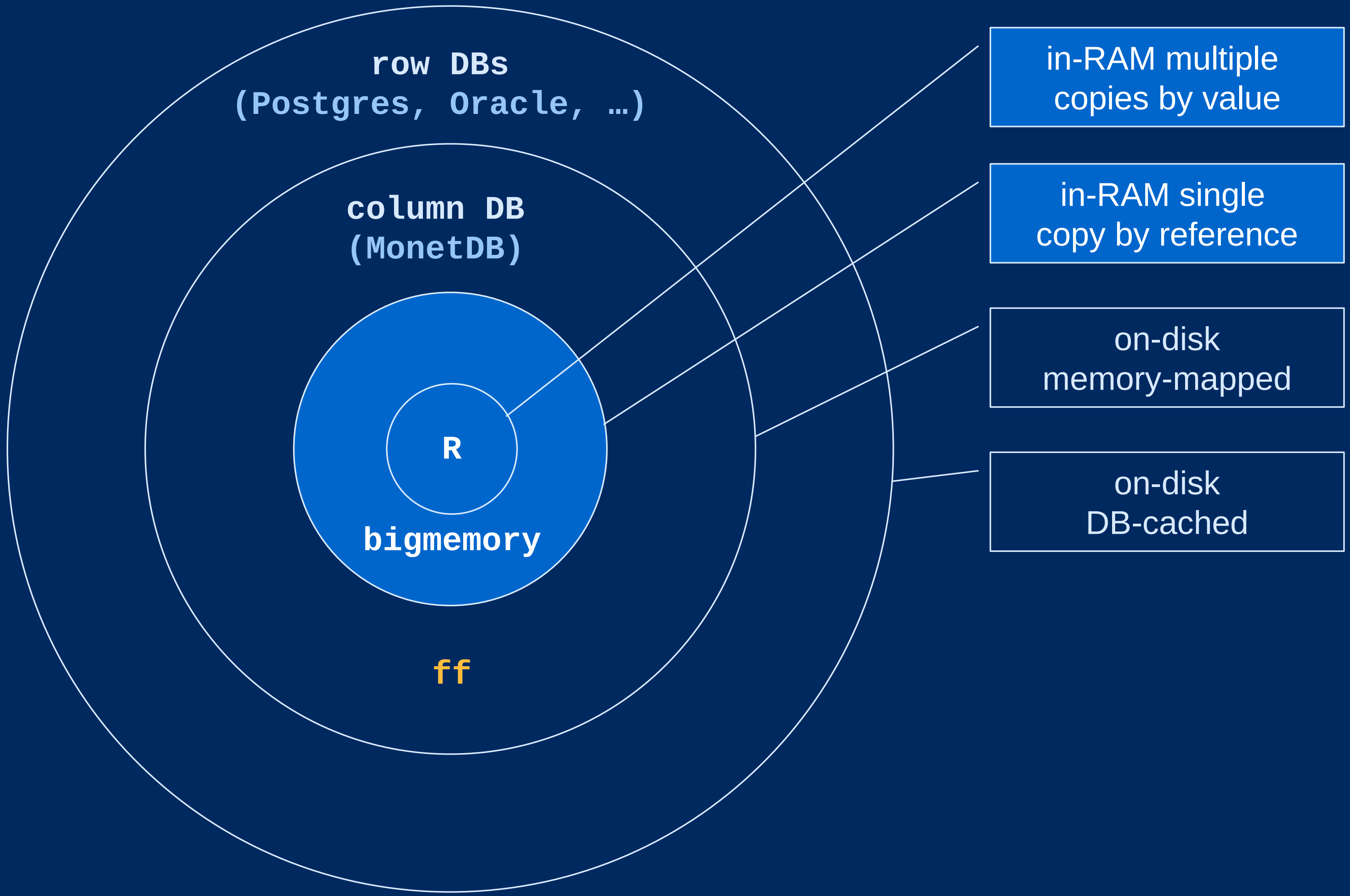
Parallel chunking

- Adding package 'snowfall' to 'ff'
  allows for easy distributed chunked processing

- Adding package 'ff' to 'snowfall'
  allows master sending/receiving datasets to/from slaves
  without memory duplication
  (large bootstrapping, special bagging support, ...)

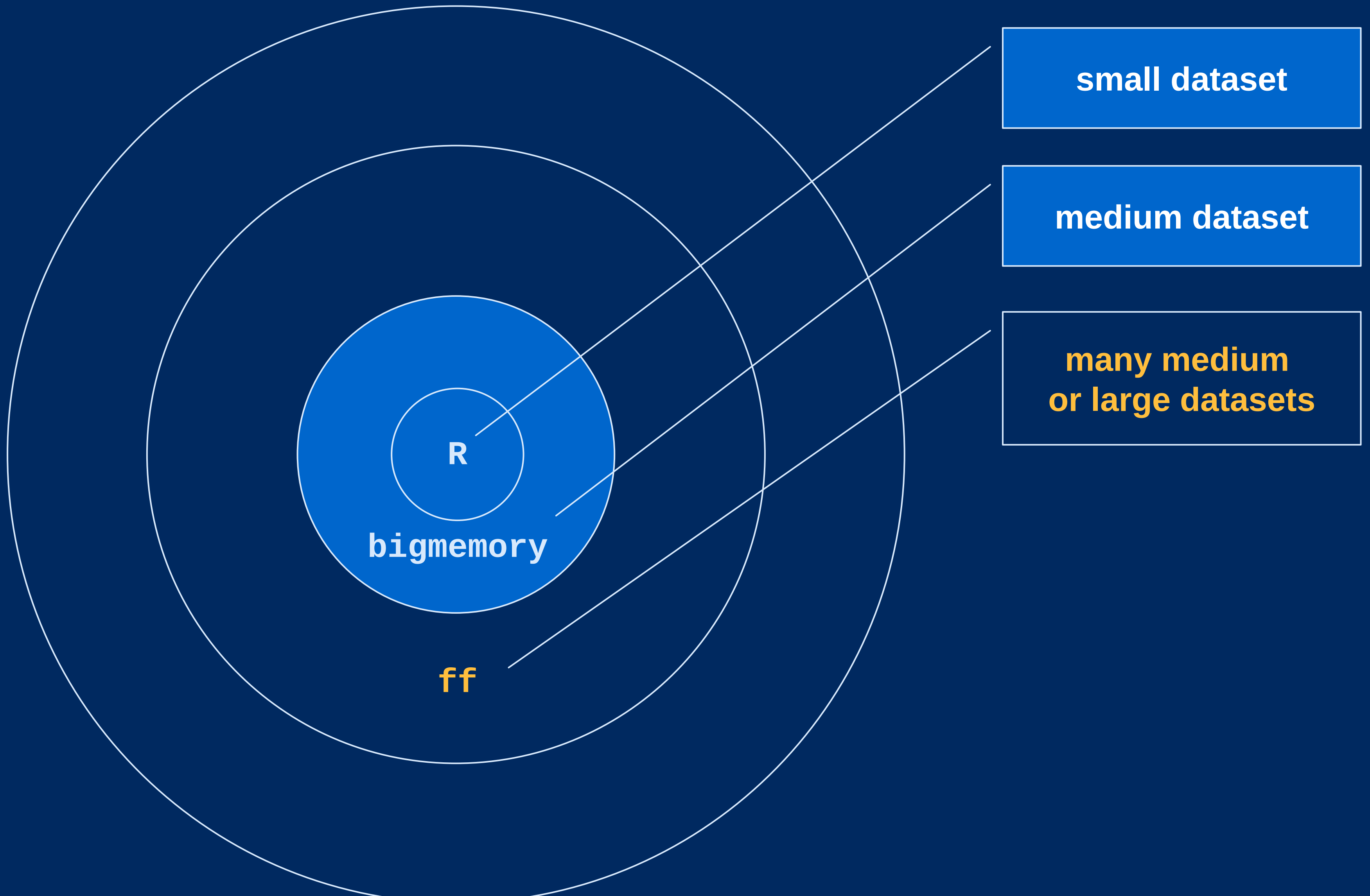
Source: Oehlschlägel, Adler (2009) Managing data.frames with package 'ff' and fast filtering with package 'bit'

Putting 'ff' in perspective with regard to size and some alternatives



Source: Oehlschlägel, Adler (2009) Managing data.frames with package 'ff' and fast filtering with package 'bit'

Comparing 'ff' to RAM-based alternatives: what are they good at?



Source: Oehlschlägel, Adler (2009) Managing data.frames with package 'ff' and fast filtering with package 'bit'

Comparing 'ff' to disk-based alternatives: what are they good at?

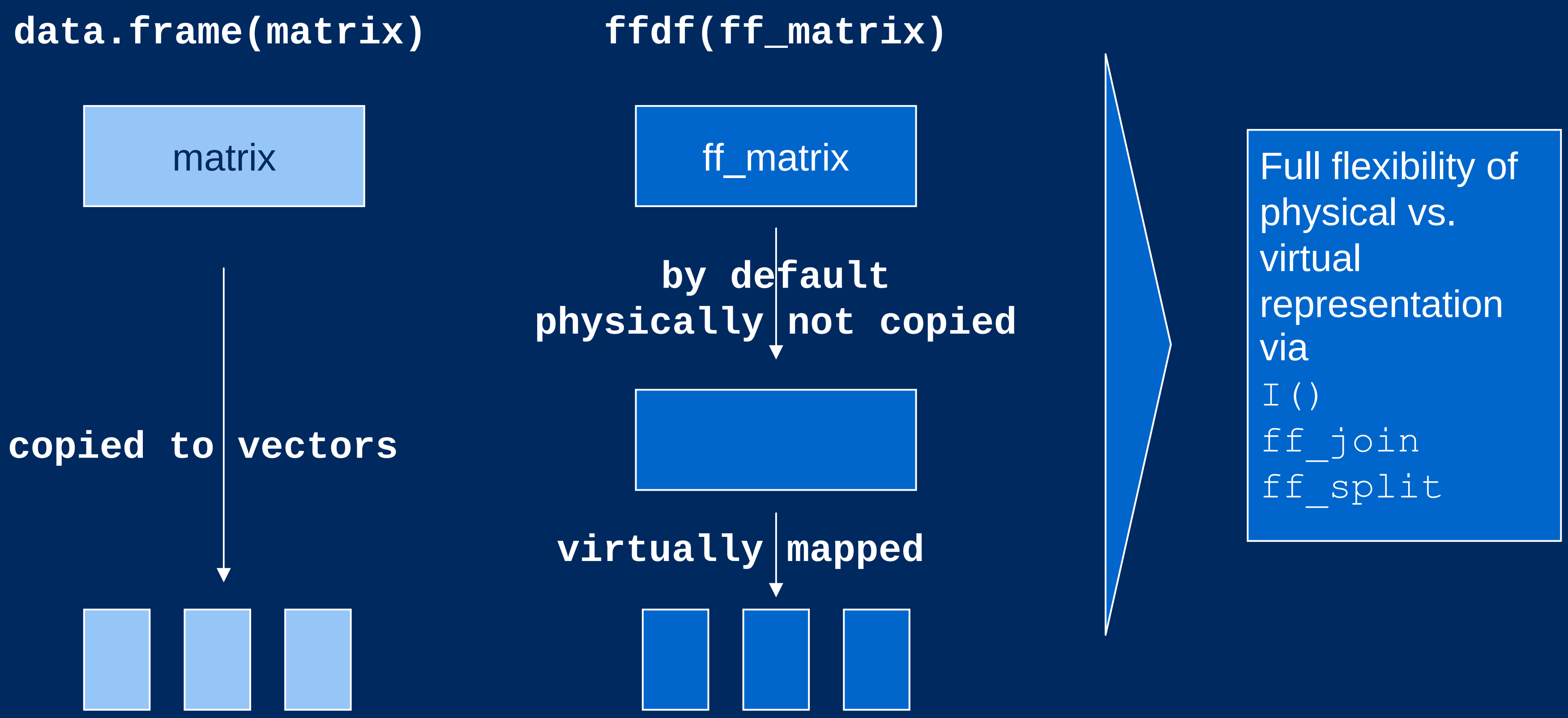
many small OLTP queries
(e.g. find and update single row)

large simple OLAP queries
(e.g. column-sums across majority of rows)

large complex read and write operations
(e.g. kernel-smoothing)

Source: Oehlschlägel, Adler (2009) Managing data.frames with package 'ff' and fast filtering with package 'bit'

ffdf dataframes separate virtual layout from physical storage



Source: Oehlschlägel, Adler (2009) Managing data.frames with package 'ff' and fast filtering with package 'bit'

EXAMPLE I – create ff vectors with 80 Mio elements as input to ffdf

```
library(ff)                    # loads library(bit)
options(fffinalizer='close') # let snowfall not delete on remove
N <- 8e7                       # sample size
n <- 1e6                       # chunk size

genders <- factor(c("male","female"))

gender <- ff(genders, vmode='quad', length=N, update=FALSE)
for (i in chunk(1,N,n)){
  print(i)
  gender[i] <- sample(genders, sum(i), TRUE)
}
gender

# load the other prepared ff vectors
load(file="d:/tmp/ff.RData")
open(year); open(country); open(age); open(income)
ls()
```

EXAMPLE I – create and access ffdf data.frame with 80 Mio rows

```
# create a data.frame
x <- ffdf(country=country, year=year, gender=gender, age=age
, income=income)
x
vmode(x)
# only 630 MB on disk instead of 1.8 GB in RAM
# => factor 3 RAM savings in file-system cache
sum(.ffbytes[vmode(x)]) * 8e7 / 1024^2
sum(.rambytes[vmode(x)]) * 8e7 / 1024^2
object.size(physical(x))

x$country                      # return  1  ff column
x[["country"]]                 # dito

x[c("country", "year")]        # return ffdf with selected columns

x[1:10, c("country", "year")] # return  2 RAM data.frame columns
x[1:10,]                       # return 10 data.frame rows
x[1,,drop=TRUE]                # return  1 row as list

# all these have <- assignment functions
```

EXAMPLE I – ff objects can be grown at no penalty

```
nrow(x)
system.time( nrow(x) <- 1e8 )
# after 0 seconds we have a dataframe with 100 Mio rows
x

nrow(x) <- 8e7
# back to original size for the following example
```

Useful for e.g. chunked reading of a csv

Difficult to do with in-memory objects

Source: Oehlschlägel, Adler (2009) Managing data.frames with package 'ff' and fast filtering with package 'bit'

Packages 'ff' + 'bit' support a variety of important access scenarios

	random access	sequential access	unpredictable search condition	BI drill-down
R	fast if fits in-memory	fast if fits in-memory	fast if small data	combine logicals
bigmemory	as fast as possible if fits in-memory	as fast as possible if fits in-memory	-	-[2]
ff	**as fast as possible if large chunks**	**as fast as possible if chunked**	**-**	**combine bit filters**
MonetDB	-	as fast as possible if many rows	as fast as possible if many rows[1]	-
row DBs	-	-	b*-tree, bitmap	combine bitmaps

```
WHERE country = 'France'
                ↓
WHERE country = 'France'
  AND   year IN (2008, 2009)
```

1 so far not delivered compiled with experimental 'cracking' option

2 might also benefit from bit filters in future releases

Source: Oehlschlägel, Adler (2009) Managing data.frames with package 'ff' and fast filtering with package 'bit'

Package 'bit' supports lean in-RAM storage of booleans and fast combination of booleans

in R's memory

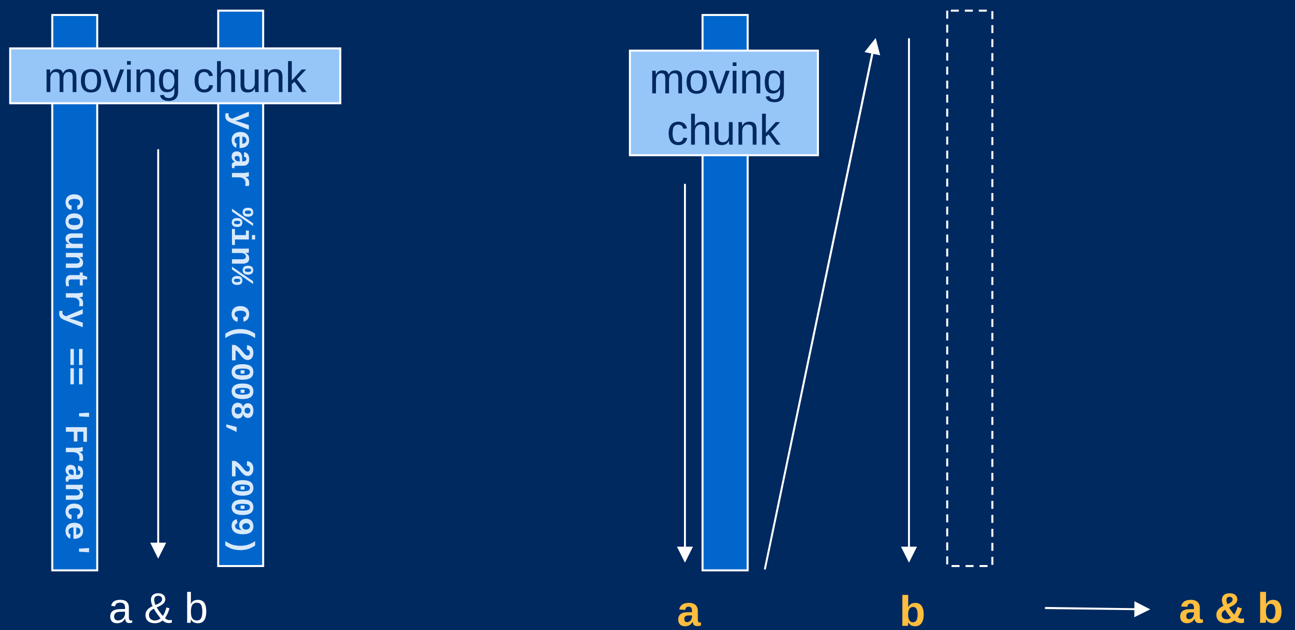
in fs-cache

Disadvantage of processing two conditions at once

- double load on memory-mapped file-system-cache
- double wait time after user action

Advantage of processing two conditions one by one

- half load on memory-mapped file-system-cache
- half wait time between user actions



Source: Oehlschlägel, Adler (2009) Managing data.frames with package 'ff' and fast filtering with package 'bit'

EXAMPLE II – create, combine and coerce filters with 80 Mio bits

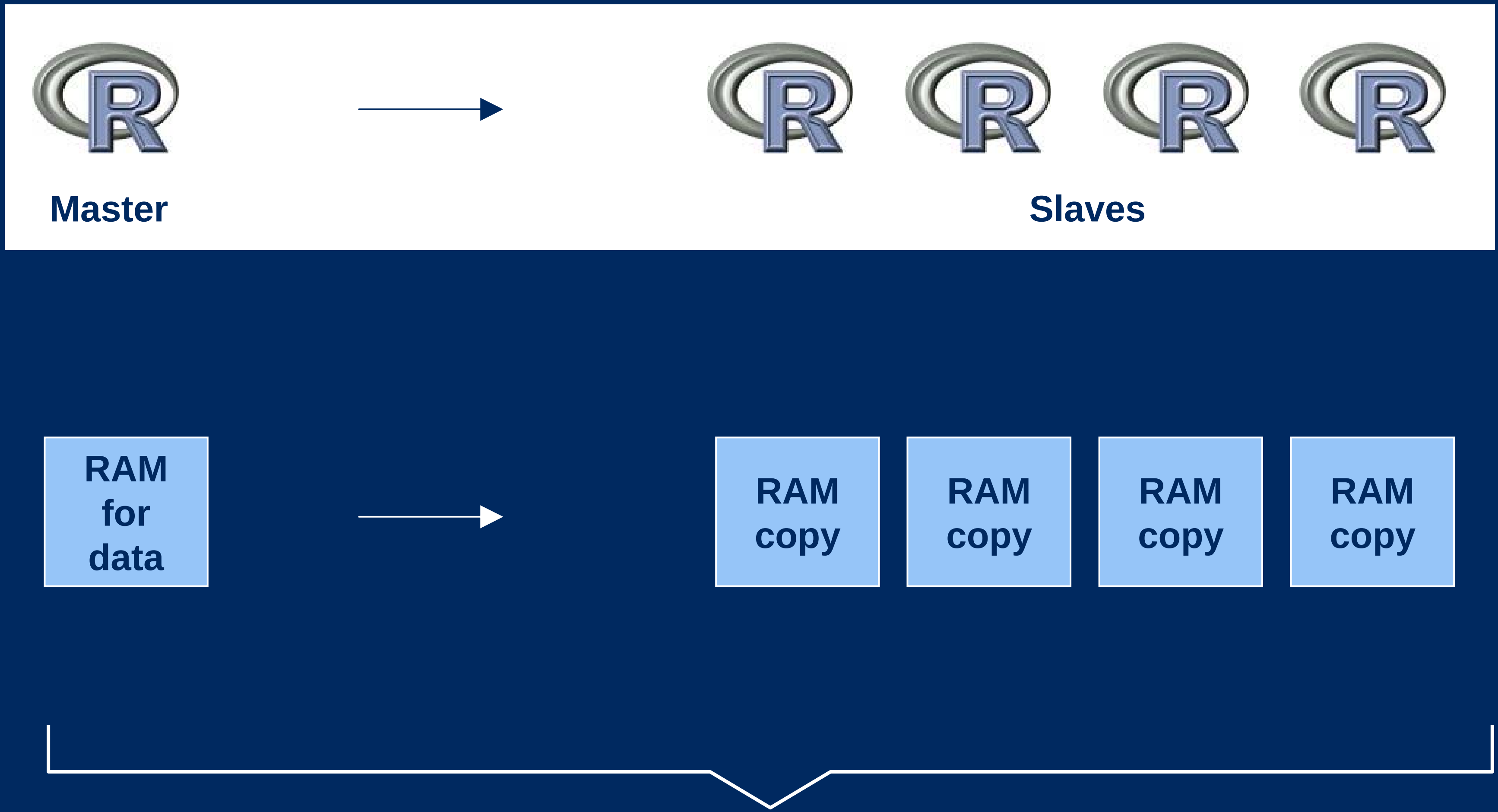
```
# create bit object
fcountry <- bit(N)
fyear <- bit(N)
# process chunks and write to bit object
system.time( for (i in chunk(1,N,n)){
    fcountry[i] <- x$country[i] == 'FR'
  } )
system.time(for (i in chunk(1,N,n)){
    fyear[i] <- x$year[i] %in% c(2008,2009)
  })
# combine with boolean operator
system.time( filter <- fcountry & fyear )

summary(filter)                   # check filter summary, then use
summary(filter, range=c(1, 8e6))  # dito for chunk
# filter combined with range index and automatically coerced as.hi
summary(x[filter & ri(1,8e6, N),], maxsum = 12)

# coercing
h <- as.hi(filter)  # coerce chunk: as.hi(filter, range=c(1,8e6))
as.bit(h)
f <- as.ff(filter)
as.bit(f)
```

Source: Oehlschlägel, Adler (2009) Managing data.frames with package 'ff' and fast filtering with package 'bit'

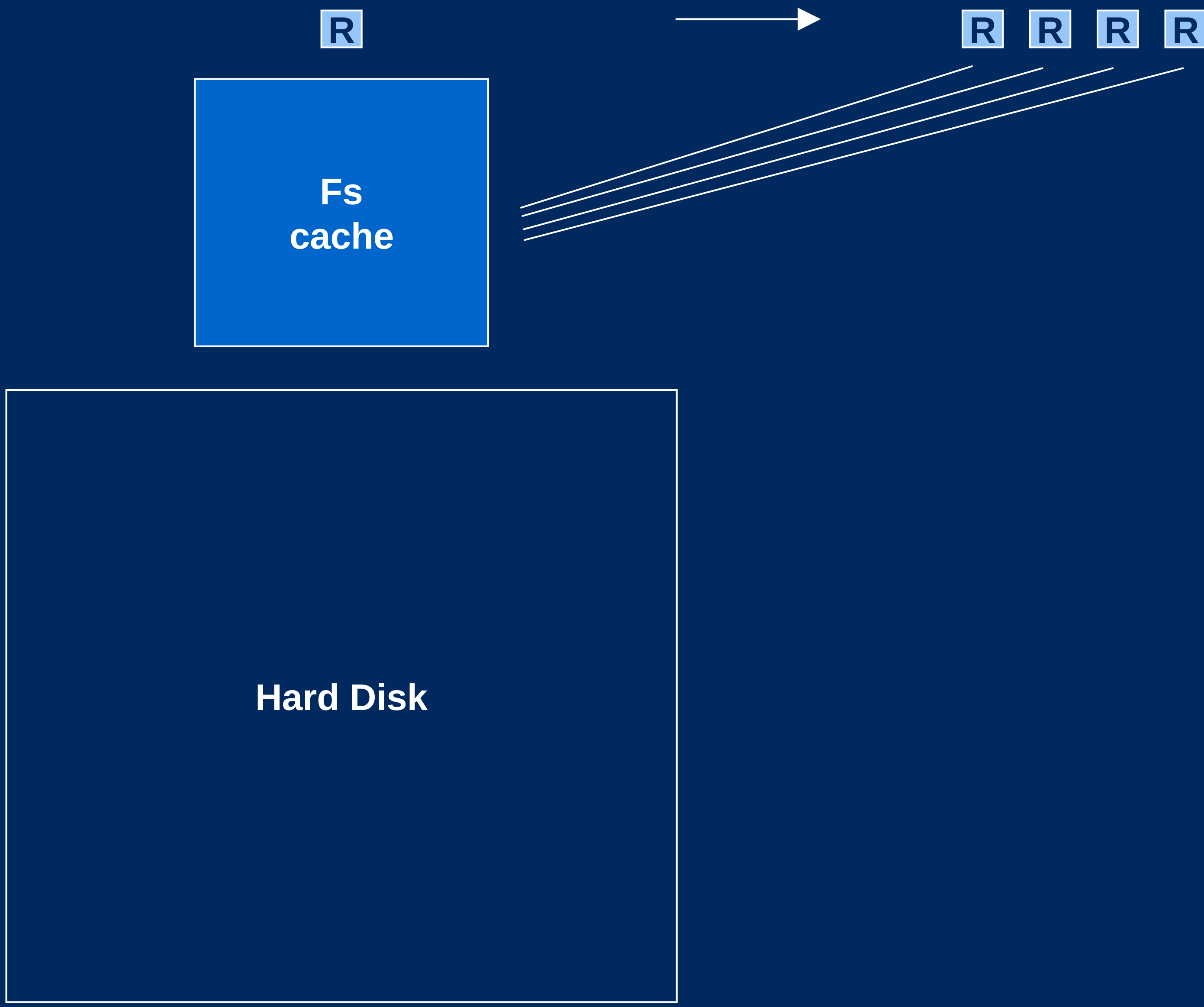
PARALLEL BOOTSTRAP with snowfall (R Journal 1/1)



5 times RAM on Quadcore == max dataset size is 1/5th

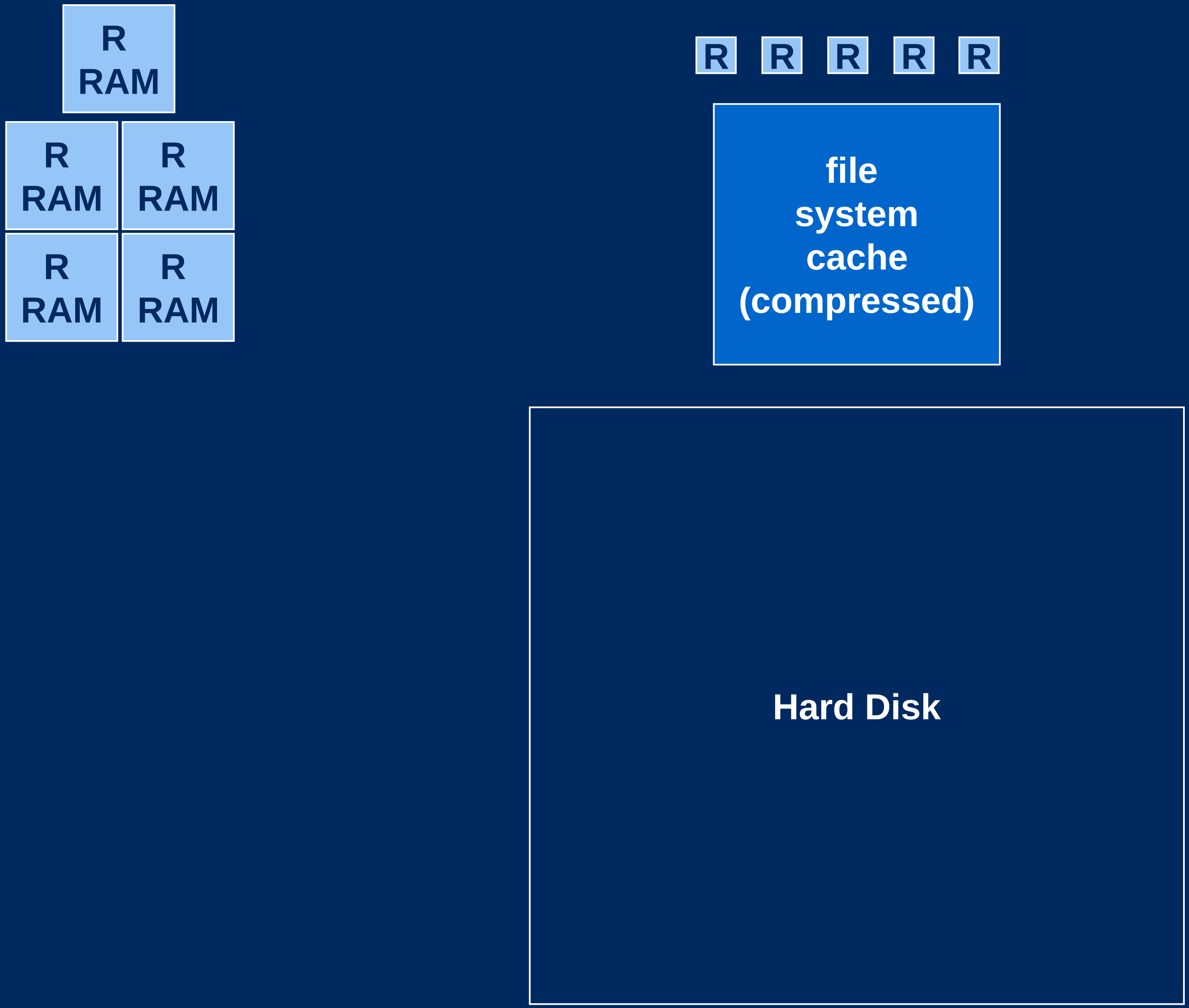
Source: Oehlschlägel, Adler (2009) Managing data.frames with package 'ff' and fast filtering with package 'bit'

Negligible RAM duplication for parallel execution on ff with snowfall



Source: Oehlschlägel, Adler (2009) Managing data.frames with package 'ff' and fast filtering with package 'bit'

Thus same RAM will allow much larger datasets if using ff



Source: Oehlschlägel, Adler (2009) Managing data.frames with package 'ff' and fast filtering with package 'bit'

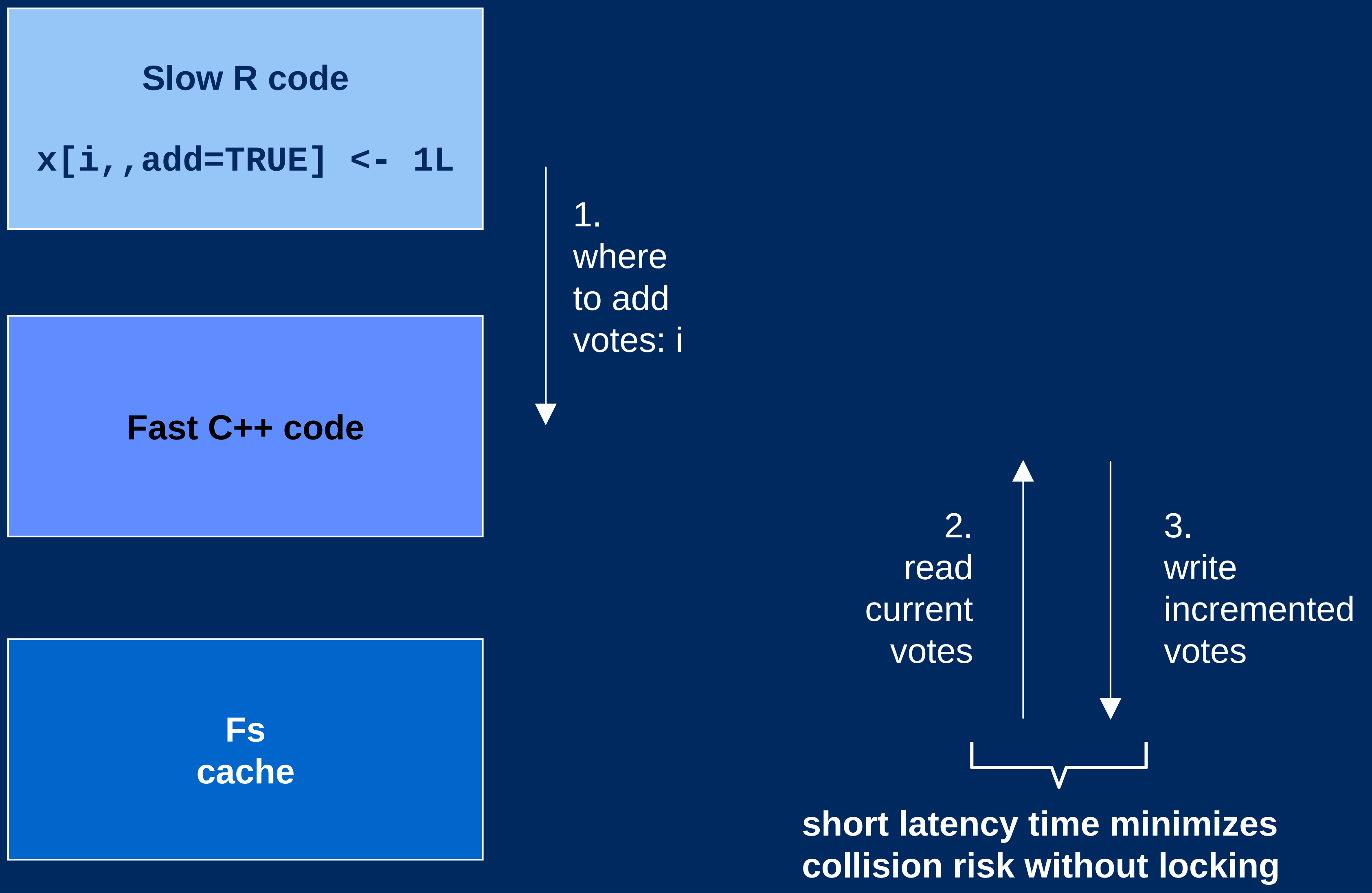
EXAMPLE III – parallel subsampling with 'ff' and 'snowfall'

```
library(snowfall)
wrapper <- function(n){
  colMeans(x[sample(nrow(x), n, TRUE), c("age","income")])
}

sfInit(parallel=TRUE, cpus=2, type="SOCK")
sfLibrary(ff)
sfExport("x")
sfClusterSetupRNG()
system.time(y <- sfLapply(rep(10000, 200), wrapper))
sfStop()

z <- do.call("rbind", y)
summary(z)
```

Low latency-times for adding votes in bagging



Source: Oehlschlägel, Adler (2009) Managing data.frames with package 'ff' and fast filtering with package 'bit'

EXAMPLE IV – rare collisions in parallel bagging with ff and snowfall

```
library(ff)
library(snowfall)
N <- 10000000   # sample size
n <- 100000     # sub-sample size
r <- 10         # number of subsamples
x <- ff(0L, length=N)  # worst case: all votings are collected in
the same column (like in perfect prediction)
wrapper <- function(i){
  x[sample(N, n), add=TRUE] <- 1L
  NULL
}
sfInit(parallel=TRUE, cpus=2, type="SOCK")
sfLibrary(ff)
sfExport("x")
sfExport("N")
sfExport("n")
sfClusterSetupRNG()
system.time(sfLapply(1:r, wrapper))
sfStop()
e <- r*n; m <- e - sum(x[]); cat("expected votes", e, "absolute
votes lost", m, "  votes lost% =", 100 * m/e, "= 1 /", e/m, "\n")
```

FF FUTURE

what we work at

- transparent partitioning of ff objects
- simplified processing of ff objects (R.ff)

what we not plan in the near future

- Native fixed-width characters or variable-width characters
- Complex type
- Generalize `ff_array` to `ff_mixed` structure
- Indexing (b*tree and bitmap with e.g. Fastbit)
- svd and friends[1]

what others easily could do

- ffcsv package providing efficient import/export of csv files
- ffsql package providing exchange with SQL databases
- statistical and graphical methods that work with ff objects

1 As an exception to this rule, R.ff will contain a svd routine – suitable in specific contexts – donated by John Nash

CONCLUSION

large data.frames

- R now has a data.frame class ffdf allowing for 2.14 bil. rows
- Memory need for file-system cache can be reduced by using lean data types (boolean, byte, small, single etc.)

fast selections

- Package 'bit' provides three classes for managing selections on large objects quickly, in a way appropriate to R rather than re-inventing what is available elsewhere.

chunking + parallel execution

- Package 'bit' helps with easy chunking and package 'ff' and 'snowfall' complement each other for speeding-up calculations on large datasets.

Source: Oehlschlägel, Adler (2009) Managing data.frames with package 'ff' and fast filtering with package 'bit'

AUTHORS

Jens Oehlschlägel Jens_Oehlschlaegel@truecluster.com

ff 2.0
bit 1.0[1]
bit 1.1
ff 2.1

Daniel Adler dadler@uni-goettingen.de

ff 1.0
ff 2.0
ff 2.1

1 Thanks to Stavros Macrakis for some helpful comments on bit 1.0

SOME DETAILS
NOT PRESENTED
IN THE SESSION

SUPPORTED DATA TYPES

native	(white outline)
indirect via raw matrix	(dark shading)
not implemented	(light shading)

vmode(x)			
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		fixed widths, tbd.	

```
# example
x <- ff(0:3
, vmode="quad")
```

Compounds

factor
ordered
POSIXct
POSIXlt

Source: Oehlschlägel, Adler (2009) Managing data.frames with package 'ff' and fast filtering with package 'bit'

SUPPORTED DATA STRUCTURES

soon on CRAN	(white outline box)
prototype available	(medium shade)
not yet implemented	(light shade)

	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)`	`c("ff_dist","ff_symm","ff")`
distance matrix		`c("ff_mixed", "ff")`
mixed type arrays instead of data.frames		

SUPPORTED INDEX EXPRESSIONS

implemeneted	(white box)
not implemented	(grey box)

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

expression	Example
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[ bit1 & bit2 ,]`
bitwhich	`x[ as.bitwhich(...) ,]`
range index	`x[ ri(chunk_start,chunk_end) ,]`
hybrid index	`x[ as.hi(...) ,1]`
zeros	`x[ 0 ]`
NAs	`x[ NA ]`

Source: Oehlschlägel, Adler (2009) Managing data.frames with package 'ff' and fast filtering with package 'bit'

INDICATION AND CONTRA-INDICATION for 'ff'

Reasons for using ff

- Fast access to large data volumes directly in R
  - Data too large for RAM
  - Too many datasets
  - Too many copies of the same data
- Sharing data between parallel R slaves running on a multi-core machine (snowfall)

Reasons for not using ff

- Speed matters with small datasets and everything fits into RAM (multiple times possibly)
- Dataset size requires more than 2.14 billion elements per atomic or more than 2.14 / fixed-width billion elements per atomic character
- Data needed at the same time in the fs-cache exhausts available memory (900MB under Win32) and swapping exhausts acceptable execution time.
- B*-tree like searching is required (use row database)
- Simple large queries only (use column-DB like MonetDB or row-DB with bitmap indexing.
- Transparent locking required (use bigmemory or row-DB)

INDICATION AND CONTRA-INDICATION for 'bit'

Reasons for using bit

- Saving RAM for booleans
- Faster boolean operations

Reasons for not using bit

- NAs needed (tri-boolean)
- Simple condition only needed once for subscripting

Source: Oehlschlägel, Adler (2009) Managing data.frames with package 'ff' and fast filtering with package 'bit'

Performance tests 0.19 GB doubles

Windows XP 32 bit 3GB RAM RGui 2.8.1

5000 x 5000	R	**ff**	bigmemory	filebacked
Create	0.40	**0.00**	0.00	78.90
Colwrite	0.75	**2.55**	2.02	2.20
Colread	0.55	**2.17**	3.42	3.45
Rowwrite	0,60	**3.95**	2.13	2.40
Rowread	0.70	**3.70**	3.50	4.10

250000 x 100	R	**ffdf**	**ff**	bigmemory	filebacked
Create	79.66	**0.50**	**0.02**	0.03	1.92
Colwrite	46.00	**2.87**	**2.22**	2.20	2.35
Colread	0	**3.02**	**2.16**	3.85	3.92
Rowwrite	48.50	**11.23**	**2.44**	1.45	1.50
Rowread	0.95	**15.83**	**2.21**	3.90	4.05

Performance tests 3.05 GB doubles (x 16)

Windows XP 32 bit 3GB RAM RGui 2.8.1

20000 x 20000	factor =>	**ff**		
Create		**0.00**		
Colwrite	x 32	**77**		
Colread	x 37	**78**		
Rowwrite	x 5200	**20800**		
Rowread	x 81	**403**		

4000000 x 100	factor =>	**ffdf**	**ff**	<= factor	
Create	x 4	**2**	**0.02**		
Colwrite	x 32	**91**	**85**	x 38	
Colread	x 33	**100**	**77**	x 36	
Rowwrite	x 69	**775**	**1748**	x 722	
Rowread	x 52	**820**	**704**	x 320	

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"))

# 9 sec
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)
for (i in chunk(1,N,n))
  gender[i] <- sample(genders, sum(i), TRUE)


# 90 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))

close(age); close(income); close(country); close(year)
save(age, income, country, year, countries, years, genders, N, n, file="d:/tmp/ff.RData")
```