DISCLOSED

Large atomic data in R: package 'ff'

Adler, Oehlschlägel, Nenadic, Zucchini

Göttingen, Munich
August 2008

SUMMARY

A proof of concept for the 'ff' package has won the large data competition at useR!2007 with its C++ core implementing fast memory mapped access to flat files. In the meantime we have complemented memory mapping with other techniques that allow fast and convenient access to large atomic data residing on disk. ff stores index information efficiently in a packed format, but only if packing saves RAM. HIP (hybrid index preprocessing) transparently converts random access into sorted access thereby avoiding unnecessary page swapping and HD head movements. The subscript C-code directly works on the hybrid index and takes care of mixed packed/unpacked/negative indices in ff objects; ff also supports character and logical indices. Several techniques allow performance improvements in special situations. ff arrays support optimized physical layout for quicker access along desired dimensions: while matrices in the R standard have faster access to columns than to rows, ff can create matrices with a row-wise layout and arbitrary 'dimorder' in the general array case. Thus one can for example quickly extract bootstrap samples of matrix rows. In addition to the usual '[' subscript and assignment '[<-' operators, ff supports a 'swap' method that assigns new values and returns the corresponding old values in one access operation - saving a separate second one. Beyond assignment of values, the '[<-' and 'swap' methods allow adding values (instead of replacing them). This again saves a second access in applications like bagging which need to accumulate votes. ff objects can be created, stored, used and removed, almost like standard R ram objects, but with hybrid copying semantics, which allows virtual 'views' on a single ff object. This can be exploitet for dramatic performance improvements, for example when a matrix multiplication involves a matrix and it's (virtual) transpose. The exact behavior of ff can be customized through global and local 'options', finalizers and more.

The supported range of storage types was extended since the first release of ff, now including support for atomic types 'raw', 'logical', 'integer' and 'double' and ff data structures 'vector' and 'array'. A C++ template framework has been developed to map a broader range of signed and unsigned types to R storage types and provide handling of overflow checked operations and NAs. Using this we will support the packed types 'boolean' (1 bit), 'quad' (2 bit), 'nibble' (4 bit), 'byte' and 'unsigned byte' (8 bit), 'short', 'unsigned short' (16 bit) and 'single' (32bit float) as well as support for (dense) symmetric matrices with free and fixed diagonals. These extensions should be of some practical use, e.g. for efficient storage of genomic data (AGCT as.quad) or for working with large distance matrices (i.e. symmetric matrices with diagonal fixed at zero).

FF 2.0 DESIGN GOALS: BASE PACKAGE FOR LARGE DATA

large data

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

standard HW

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

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 redundant access

Source: Adler, Oehlschlägel, Nenadic, Zucchini (2008) Large atomic data in R: package 'ff'

A SHORT FF DEMO

```
library(ff)

ffVector <- ff(0:1, length=36e6)  # 0,1,0,…       4 byte integers
ffVector

ffMatrix <- ff(vmode="logical", dim=c(6e3,6e3)) # 2 bit logical
ffMatrix

ffPOSIXct <- ff(Sys.time(), length=36e6)        # 8 byte double
ffPOSIXct

bases <- c("A","T","G","C")
ffFactor <- ff("A", levels=bases, length=400e6  # 2 bit quad
, vmode="quad", filename="QuadFactorDemo.ff", overwrite=TRUE)
  # 95 MB with quad instead of 1.5 GB with integer
ffFactor

# accessing parts based on memory mapping and OS file caching
ffFactor[3:400e6] <- c("A","T") # quick recycling at no RAM
ffFactor[1:12]
```

SUPPORTED DATA TYPES

on CRAN
prototype available
not yet implemented

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

SUPPORTED DATA STRUCTURES

on CRAN	
prototype available	
not yet implemented	

	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")`
symmetric matrix with free diag	`ff(1:6, dim=c(3,3), symm=TRUE, fixdiag=NULL)`	`c("ff_symm","ff")`
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")`

SUPPORTED INDEX EXPRESSIONS

implemeneted	
not implemented	

```
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)) ]`
hybrid index	`x[ hi ,1]`
zeros	`x[ 0 ]`
NAs	`x[ NA ]`

FF DOES SEVERAL ACCESS 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 generics**
  - **`clone(), update(), swap(), add()`**

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

C++ backend

Memory Mapped 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`** of large files

DOUBLE VECTOR CHUNKED SEQUENTIAL ACCESS TIMINGS [sec]

		plain R	bigmemory	ff2.0	ff1.0	R.huge
76 MB	read by 1e6 of 1e7	0,3	4,5	0,40	0,25	165,0
76 MB	write	0,3	1,1	0,20	0,70	110,0
0,75 GB	read by 1e6 of 1e8	2,5	42,5	4,00	1,97	1600,0
0,75 GB	write	2,5	12,3	2,00	7,57	1150,0
3,50 GB	read by 1e6 of 4*1e8	failed	crashed	99,78	90,00	skipped
3,50 GB	write	:	:	188,16	420,00	:
7,50 GB	read by 1e6 of 1e9	:	:	229,00	skipped	:
7,50 GB	write	:	:	916,00	:	:

as fast as in-memory methods

faster than older disk methods

* HP nc6400 Notebook 2GB RAM, Windows XP, x86 dual core ~2327 Mhz (of which 50% is used)

Source: Adler, Oehlschlägel, Nenadic, Zucchini (2008) Large atomic data in R: package 'ff'

DOUBLE VECTOR CHUNKED RANDOM ACCESS TIMINGS [sec]

			plain R	bigmemory	ff2.0	ff1.0	R.huge
76 MB	read	10x1e6 of 1e7	2,5	7,1	8,11	62,3	180,5
76 MB	write		2,5	3,1	7,40	63,2	123,7
76 MB	read	1000x1e4 of 1e7	2,7	7,3	24,30	62,1	172,2
76 MB	write		2,6	3,6	23,10	62,0	2800,0
0,75 GB	read	10x1e6 of 1e8	5,8	11,0	18,90	77,8	184,6
0,75 GB	write		5,8	7,0	18,50	77,1	277,9
0,75 GB	read	1000x1e4 of 1e8	2,8	7,6	48,30	72,8	220,0
0,75 GB	write		2,6	3,6	47,20	73,0	20000,0
3,50 GB	read	1x1e7 of 4e8	failed	crashed	103,00	skipped	skipped
3,50 GB	write		:	:	261,00	:	:
3,50 GB	read	10x1e6 of 4e8	:	:	935,00	:	:
3,50 GB	write		:	:	5340,00		:
3,50 GB	read	1000x1e4 of 4e8		:	32000,00		
3,50 GB	write				70000,00		
7,50 GB	read	10x1e6 of 1e9		:	2200,00		
7,50 GB	write			:	9471,00		:
7,50 GB	read	1000x1e4 of 1e9	:	:	67000,00	:	:
7,50 GB	write		:	:	135000,00	:	:

acceptable if chunks are large enough

faster than older disk methods

* HP nc6400 Notebook 2GB RAM, Windows XP, x86 dual core ~2327 Mhz (of which 50% is used)

Source: Adler, Oehlschlägel, Nenadic, Zucchini (2008) Large atomic data in R: package 'ff'

DOUBLE MATRIX ROW ACCESS TIMINGS, ROW 1..1000 [sec]

	7,6 MB read from 1000^2	7,6 MB write to	0,75 GB read from 10000^2	0,75 GB write to	3 GB read from 20000^2	3 GB write to	6,7 GB read from 30000^2	6,7 GB write to
`plain R, single rows`	0,03	0,03	failed	...	...	...	...	...
`bigmemory, single rows`	0,80	0,60	4,90	1,40	crashed	...	...	...
`bigmemory, by 100 rows`	0,33	0,08	3,10	0,69	crashed	...	...	...
`dimorder=2:1, ff2.0, by 100 rows`	**0,08**	**0,04**	**0,82**	**0,42**	**1,55**	**0,86**	**2,20**	**1,20**
`dimorder=2:1, ff2.0, single rows`	**0,95**	**0,85**	**1,40**	**1,20**	**1,86**	**1,59**	**2,33**	**1,97**
`ff2.0, by 100 rows`	**0,09**	**0,07**	**1,35**	**0,95**	**4,64**	**4,04**	**11,50**	**11,00**
`ff2.0, single rows`	**2,50**	**2,50**	**53,00**	**53,00**	**330,00**	**313,00**	**skipped**	**skipped**
`ff1.0, single rows`	85,00	230,00	skipped	...	...	...	...	...
`R.huge, single rows`	96,00	80,00	skipped	...	...	...		
`R.huge, by 100 rows`	4,70	4,50	50,00	261,80	skipped			
`byrow R.huge, single rows`	5,60	5,40	37,70	37,40	skipped			
`byrow R.huge, by 100 rows`	4,33	4,33	37,10	38,90	skipped			

faster than older disk methods

as fast as in-memory if chunksize and dimorder fine

* HP nc6400 Notebook 2GB RAM, Windows XP, x86 dual core ~2327 Mhz (of which 50% is used)

Source: Adler, Oehlschlägel, Nenadic, Zucchini (2008) Large atomic data in R: package 'ff'

FF BEATS LOCAL DATABASES FOR TYPICAL REPORTING TASKS

Timings in seconds

	2 Mio Access	2 Mio no index SQLite	2 Mio R ff	5 Mio Access	5 Mio no index SQLite	5 Mio R ff	10 Mio R ff
MB on Disk without indices	320	673	289		1685	724	1448
MB on Disk including indices	430			1073			
15 ColSums FullTableScan 100%	14,40	4,20	2,18	36,10	>120	4,11	39,23
3 ColSums FullTableScan 100%	3,08	3,90	0,44	7,73	>120	1,06	2,21
15 ColSums 2 SelectDims 90%	13,70	7,17	2,32	34,47	>120	7,58	18,61
3 ColSums 2 SelectDims 90%	3,45	4,38	1,41	9,06	>120	3,46	7,01
15 ColSums 2 SelectDims 10%	2,02	4,03	0,94	5,36	>120	2,34	4,67
3 ColSums 2 SelectDims 10%	0,84	3,61	0,85	2,33	>120	2,03	4,09
15 ColSums 4 SelectDims 10%	2,14	4,19	1,33	5,58	>120	3,31	19,42
3 ColSums 4 SelectDims 10%	1,01	3,69	1,19	2,86	>120	3,01	5,96
0.01% Records 2 Select Dimensions	0,03	3,90	0,39	0,05	>120	0,77	8,07
0.01% Records 4 Select Dimensions	0,08	4,00	0,36	0,17	>120	0,89	9,01
Worst Case	14,40	7,17	2,32	36,10	>120	7,58	39,23

45 columns: 1x Integer, 14 x Smallint x 100 values, 30 x Float
ff timings from within R, MS Access and SQLite timings without interfacing from R

faster if more than tiny part accessed

* HP nc6400 Notebook 2GB RAM, Windows XP, x86 dual core ~2327 Mhz (of which 50% is used)

Source: Adler, Oehlschlägel, Nenadic, Zucchini (2008) Large atomic data in R: package 'ff'

FF BEATS LOCAL DATABASES FOR TYPICAL REPORTING TASKS

Timings in seconds

	1 Mio disabl. index Access	1 Mio Access	1 Mio SQLite	1 Mio disabl. index SQLite	1 Mio R ff
MB on Disk without indices	160	160	337	337	144
MB on Disk including indices	215	215	514	514	
15 ColSums FullTableScan 100%	7,25	7,25	3,50	3,50	0,95
3 ColSums FullTableScan 100%	1,50	1,50	2,02	2,02	0,22
15 ColSums 2 SelectDims 90%	8,10	6,90	8,50	3,75	1,14
3 ColSums 2 SelectDims 90%	2,98	1,73	7,30	2,30	0,62
15 ColSums 2 SelectDims 10%	2,60	1,03	2,50	2,00	0,44
3 ColSums 2 SelectDims 10%	2,00	0,41	2,30	1,77	0,42
15 ColSums 4 SelectDims 10%	3,40	1,08	4,30	2,30	0,66
3 ColSums 4 SelectDims 10%	2,90	0,58	4,00	2,00	0,61
0.01% Records 2 Select Dimensions	1,77	0,03	0,25	2,00	0,26
0.01% Records 4 Select Dimensions	2,22	0,05	1,02	2,00	0,25
Worst Case	8,10	7,25	8,50	3,75	1,14

45 columns: 1x Integer, 14 x Smallint x 100 values, 30 x Float
ff timings from within R, MS Access and SQLite timings without interfacing from R

* HP nc6400 Notebook 2GB RAM, Windows XP, x86 dual core ~2327 Mhz (of which 50% is used)

Source: Adler, Oehlschlägel, Nenadic, Zucchini (2008) Large atomic data in R: package 'ff'

FF FUTURE …

large processing → R.ff (next presentation)

obvious extensions →
- Fixed-width character with `internal_dim=c(width, dim)`
- Indexing (b*tree and bitmap with e.g. Fastbit)
- Dataframes, 2nd of
  - Modification of R's dataframe code to wrap arbitrary atomics ?
  - Specific data.frame emulation class on top of ff !
  - Generalize `ff_array` to `ff_mixed` ?

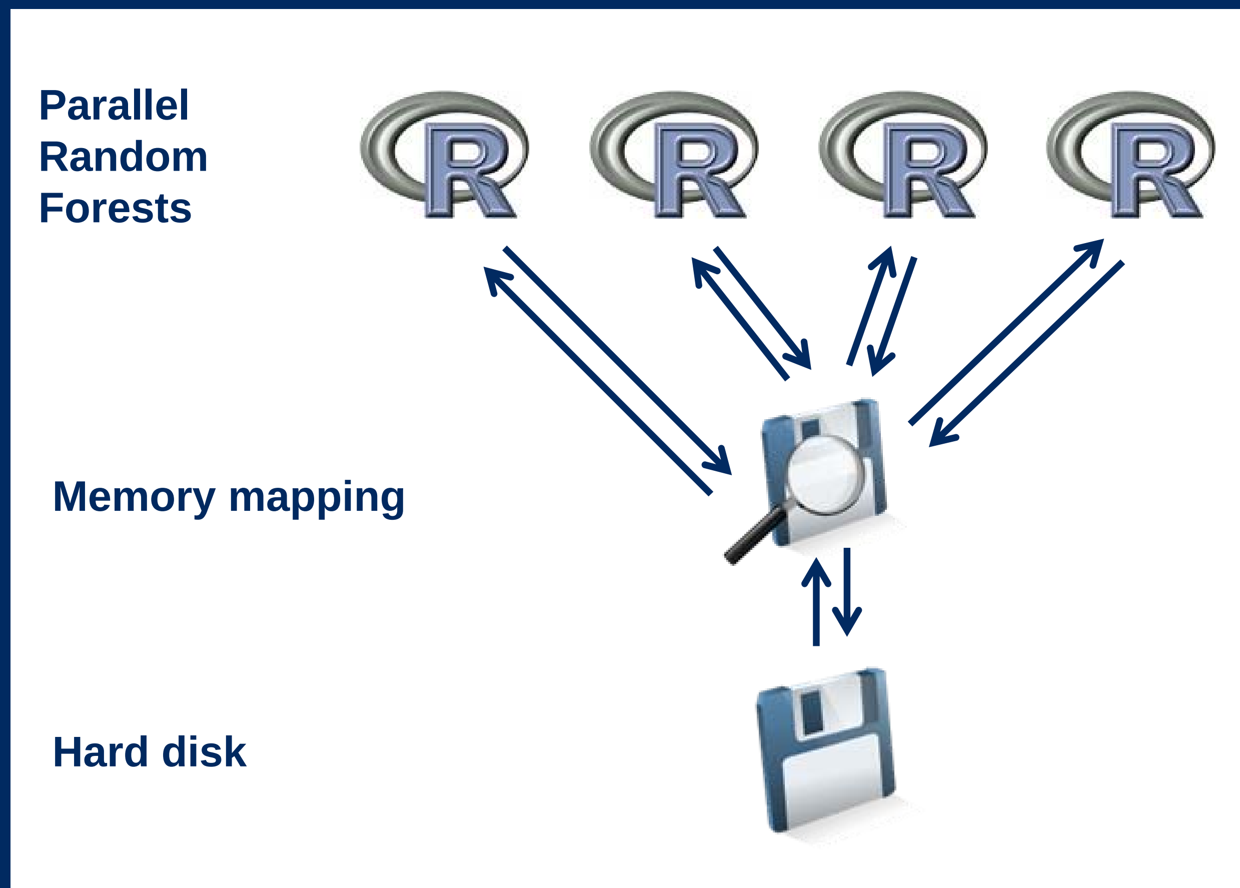
svd and friends? → Volunteers?

development resources ? → Dual licensing for high-performance data types and data structures to ff supporters

… AND BEYOND

biglm available → fit 2 GB data with ~50 MB RAM (see working example in `?ff`)

bagging [add=T] →



TEAM / CREDITS

Version 1.0

Daniel Adler	dadler@uni-goettingen.de
Oleg Nenadic	onenadi@uni-goettingen.de
Walter Zucchini	wzucchi@uni-goettingen.de
Christian Gläser	christian_glaeser@gmx.de

Version 2.0

Jens Oehlschlägel	Jens_Oehlschlaegel@truecluster.com
	R package redesign; Hybrid Index Preprocessing; transparent object creation and finalization; vmode design; virtualization and hybrid copying; arrays with dimorder and bydim; symmetric matrices; factors and POSIXct; virtual windows and transpose; new generics update, clone, swap, add, as.ff and as.ram; ffapply and collapsing functions. R-coding, C-coding and Rd-documentation.
Daniel Adler	dadler@uni-goettingen.de
	C++ generic file vectors, vmode implementation and low-level bit-packing / unpacking, arithmetic operations and NA handling, Memory-Mapping and backend caching modes. C++ coding and platform ports. R-code extensions for opening existing flat files readonly and shared.

SOME DETAILS NOT PRESENTED IN THE SESSION

FF CLASS STRUCTURE WITH HYBRID COPYING SEMANTICS

```
> x <- ff(1:12, dim=c(3,4))
> str(x)
 list()
 - attr(*, "physical")=Class 'ff_pointer' <externalptr>
  ..- attr(*, "vmode")= chr "integer"
  ..- attr(*, "maxlength")= int 12
  ..- attr(*, "pattern")= chr "ff"
  ..- attr(*, "filename")= chr "PathToFFFolder\\FFFilename"
  ..- attr(*, "pagesize")= int 65536
  ..- attr(*, "finalizer")= chr "deleteopen"
  ..- attr(*, "finonexit")= logi TRUE
  ..- attr(*, "readonly")= logi FALSE
 - attr(*, "virtual")= list()
  ..- attr(*, "Length")= int 12
  ..- attr(*, "Dim")= int [1:2] 3 4
  ..- attr(*, "Dimorder")= int [1:2] 1 2
  ..- attr(*, "Symmetric")= logi FALSE
  ..- attr(*, "VW")= NULL
 - attr(*, "class")= chr [1:3] "ff_matrix" "ff_array" "ff"

> y <- x
```

SPECIAL COPY SEMANTICS: PARTIAL SHARING

physical attributes shared:
`filename, vmode, maxlength, is.sorted, na.count`

on copy

virtual attributes independent:
`length*, dim, …`

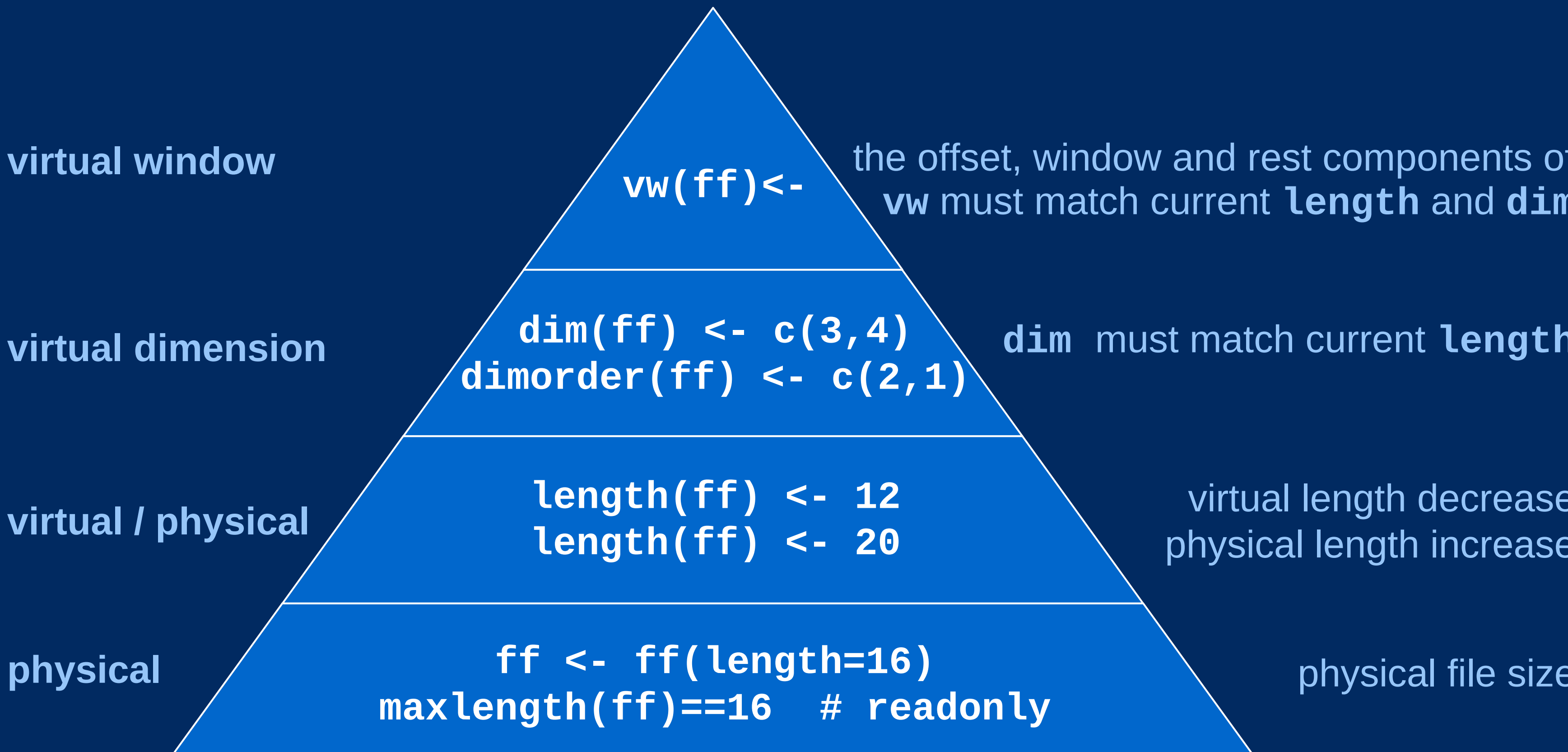
virtual attributes independent:
`… dimorder, vw`

```
> a <- ff(1:12)
> b <- a
> dim(b) <- c(3,4)
> a[] <- a[] + 1
> a
ff (open) integer length=12(12)
 [1]  [2]  [3]  [4]  [5]  [6]  [7]  [8]  [9] [10] [11] [12]
   2    3    4    5    6    7    8    9   10   11   12   13
> b
ff (open) integer length=12(12) dim=c(3,4) dimorder=c(1,2)
     [,1] [,2] [,3] [,4]
[1,]    2    5    8   11
[2,]    3    6    9   12
[3,]    4    7   10   13
```

* one exception: if length(ff) is increased, a new ff object is created and the physical sharing is lost

Source: Adler, Oehlschlägel, Nenadic, Zucchini (2008) Large atomic data in R: package 'ff'

A PHYSICAL TO VIRTUAL HIERARCHY



Source: Adler, Oehlschlägel, Nenadic, Zucchini (2008) Large atomic data in R: package 'ff'

WHILE R‘s RAM STORAGE IS ALWAYS IN COLUMN-MAJOR ORDER, FF ARRAYS CAN BE STORED IN ARBITRARY DIMORDER …

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

> x[]
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12


> x[1:12]
 [1]  1  2  3  4  5  6  7
8  9 10 11 12
```

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

> x[]
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

# NOTE: 1:12 is unpacked
> x[1:12]
 [1]  1  2  3  4  5  6  7
8  9 10 11 12
# BEWARE the difference
> read.ff(x, 1, 12)
 [1]  1  4  7 10  2  5  8
11  3  6  9 12
```

… WHICH SOMETIMES CAN HELP SPEEDING UP

```
> 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,])))
   user  system elapsed
   1.39    1.26    2.66
> system.time(lapply(1:n, function(i)sum(b[i,])))
   user  system elapsed
   0.54    0.07    0.60

> system.time(lapply(1:n, function(i){i<-(i-1)*(m/n)+1;
sum(a[,i:(i+m/n-1)])}))
   user  system elapsed
   0.48    0.03    0.52
> system.time(lapply(1:n, function(i){i<-(i-1)*(m/n)+1;
sum(b[,i:(i+m/n-1)])}))
   user  system elapsed
   0.56    0.01    0.61
```

Source: Adler, Oehlschlägel, Nenadic, Zucchini (2008) Large atomic data in R: package 'ff'

BYDIM GENERALIZES BYROW …

```
> matrix(1:12, nrow=3, ncol=4
, byrow=FALSE)
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

```
> matrix(1:12, nrow=3, ncol=4
, byrow=TRUE)
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
```

```
> ff(1:12, dim=c(3,4)
, bydim=c(1,2))
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

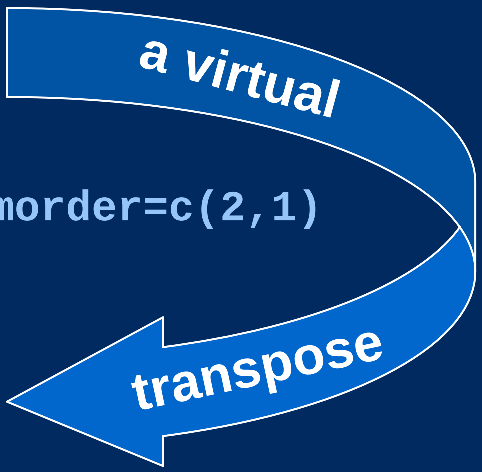
```
> ff(1:12, dim=c(3,4)
, bydim=c(2,1))
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
```

… EVEN FOR ACCESSING THE DATA

```
> x <- ff(1:12, dim=c(3,4), bydim=c(2,1))
> x[]  # == x[,,bydim=c(1,2)]
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12

# consistent interpretation in subscripting
> x[,, bydim=c(2,1)]
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
> as.vector(x[,,bydim=c(2,1)])
 [1]  1  2  3  4  5  6  7  8  9 10 11 12

# consistent interpretation in assigments
x[,, bydim=c(1,2)] <- x[,, bydim=c(1,2)]
x[,, bydim=c(2,1)] <- x[,, bydim=c(2,1)]
```

THE POWER OF PARTIAL SHARING: DIFFERENT ‘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

> b <- vt(a)  # shortcut
> filename(a) == filename(b)
[1] TRUE
```



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")`**

OPTIONS

```
getOption("fftempdir")    == "D:/.../Temp/RtmpidNQq9"
getOption("fffinonexit")  == TRUE
getOption("ffpagesize")   == 65536         # getdefaultpagesize()
getOption("ffcaching")    == "mmnoflush"   # or "mmeachflush"
getOption("ffdrop")       == TRUE          # or always drop=FALSE
getOption("ffbatchbytes") == 16104816      # 1% of RAM
```

UPDATE, CLONING AND COERCION

```
# fast plug in of temporary calculation into original ff
update(origff, from=tmpff, delete=TRUE)

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

# 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)
```

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 vw support for ram compatibility	**`[.(,add=FALSE)`**	**`[<-.(,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)

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

# 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       # Finally needed
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*

minus using structure() up 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

* as of version 2.6.2

Source: Adler, Oehlschlägel, Nenadic, Zucchini (2008) Large atomic data in R: package 'ff'