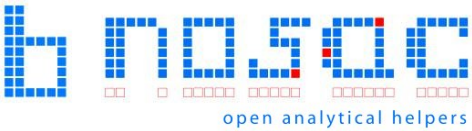
ffbase:
statistical functions for large datasets



Centraal Bureau voor de Statistiek

Jan Wijffels & Edwin de Jonge:
jwijffels@bnosac.be & edwindjonge@gmail.com

BNOSAC - Belgium Network of Open Source Analytical Consultants: www.bnosac.be
& Centraal Bureau voor Statistiek: www.cbs.nl

July 10 2013, useR! 2013

Overview

About Jan)

Founder of `www.bnosac.be` and very recent father of Midas (2013-06-19) and therefore not present at useR! 2013.

- Providing consultancy services in open source analytical engineering
- Poor man's BI: Python/PostgreSQL/Pentaho/R/Hadoop/Sencha/ExtJS...
- Expertise in predictive data mining, biostatistics, geostats, python programming, GUI building, artificial intelligence, process automation, analytical web development
- R implementations & R application maintenance

About Edwin

Working at Statistics Netherlands, that produces all dutch official statistics. Co-author of several R packages:

- `editrules`
- `tabplot`
- `whisker`
- `ffbase`

Large data

- Statistical data is becoming larger.
- If `nrow(data)` $< 10^6$ everything works fine in `R`
- For larger `data.frame`'s you run quickly into memory problems

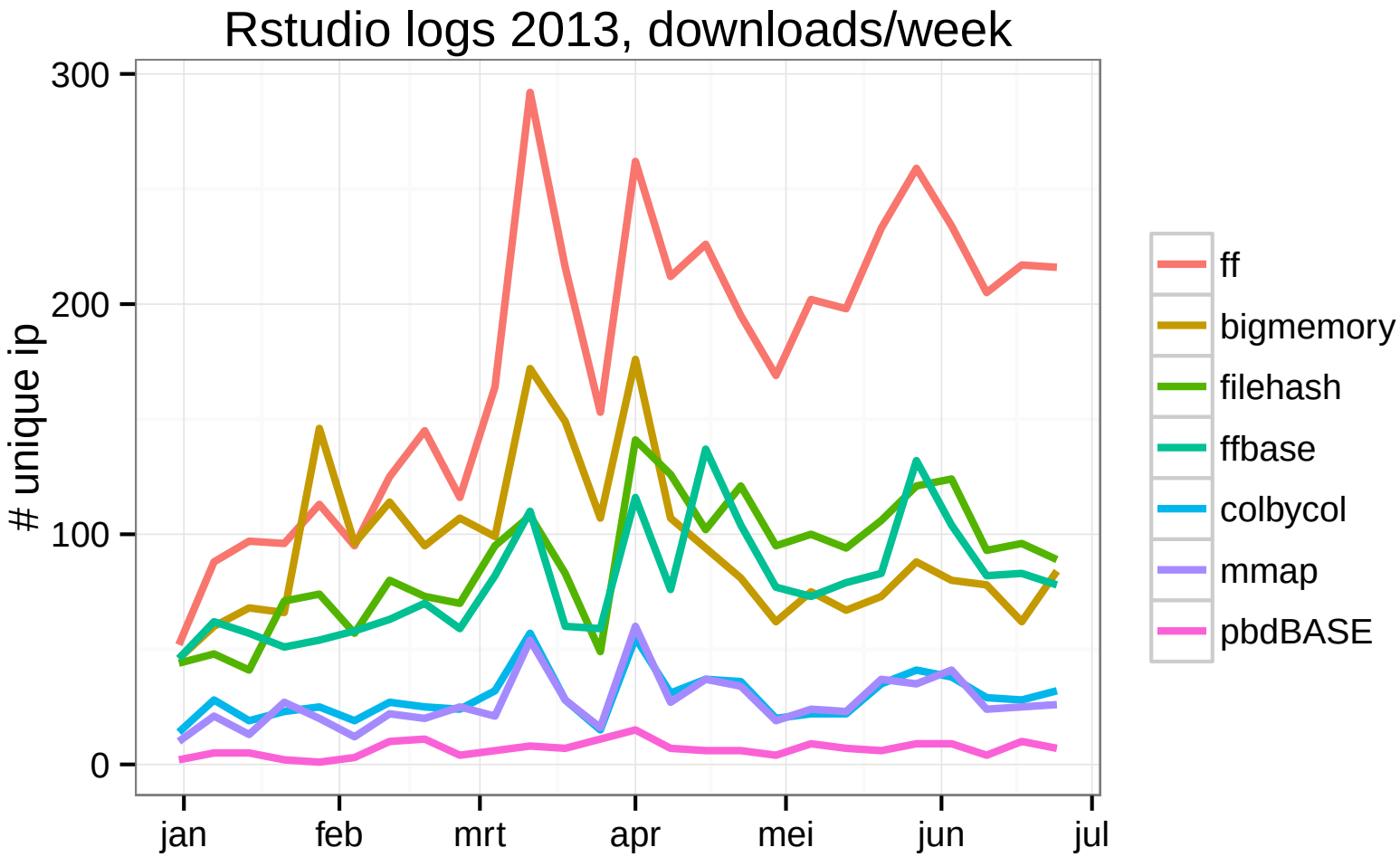
```
# create an integer of length = 1 billion
integer(1e+09)

## Error:  cannot allocate vector of size 3.7 Gb
```

- If it fits in memory plain `R` is just fine!
- If it doesn't fit on your hard disk, you'll need some big data stuff (Hadoop, `rmr`, `RHadoop`)
- For data sizes range of 10^6 - 10^9 several options provided by external packages.
- Popular one is `ff`[1]

[1] Daniel Adler, Christian Gläser, Oleg Nenadic, Jens Oehlschlägel, Walter Zucchini

Popularity based on RStudio download log files



`ff` package provides:

- numerical, integer, boolean and factor variables of type `ff` stored on disk (via memory mapping)
- a sort of data.frame of type `ffdf`
- efficient indexing, retrieval and sorting of `ff` vectors and `ffdf`.
- efficient chunk-wise retrieval of data.
- `ff`-based matrix storage.
- vectors up-to length of $2 \cdot 10^9$.

What's the catch?

`ff` is nice, but:

- Handling `ff` vectors often results in non-standard `R` code
- It offers no statistical functions on `ff` and `ffdf` objects
  - No `mean`, `max`, `min`, `sd`, etc. on `ff` vectors
- Requires that you process the data chunkwise.
- Has no support for `character` vectors.

Typical chunkwise code for ff

```
library(ff)
x <- ff(0, length = 1e+08)

# calculating the max value of x
m <- -Inf
for (i in chunk(x)) {
    m <- max(x[i], m, na.rm = T)
}
```

Value vs reference

Note that out-of-memory objects have issues with value vs reference semantics.

```
x <- ff(0, length = 1e+07)
y <- x
y[1] <- 100
print(x[1])

## [1] 100
```

This is not what a normal R vector would do! Trade-of between copying large object and side-effects.

ffbase[2] attempts to

- add basic statistical functions to ff
- make code as standard R as possible
- make working with `ff` more pleasant.
- connect `ff` with `big*` methods.

[2] de Jonge/Wijffels/van der Laan

Basic operations

Most methods works via S3 dispatch (but not all...)

- `mean`,`min`,`max`,`range`, `sum`, `all`, `cumsum`, `cumprod`, `quantile`, `tabulate.ff`, `table.ff`,
- `cut`, `c`, `unique`, `duplicated`, `Math.Ops`

```
x <- 1:10
x_ff <- ff(x)
mean(x)

## [1] 5.5

mean(x_ff)

## [1] 5.5
```

Idiosyncratic R

- Use S3 dispatch (works only for generic methods...)
- `ffbase` adds `subset`[3], `with`, `within` and `transform` to `ff`
- Makes code interchangeable with normal R code:

```
iris_ff <- as.ffdf(iris)
iris_ff <- transform( iris_ff
                    , Sepal.Ratio = Sepal.Width/Sepal.Length
                    )
str(iris_ff[,])

## 'data.frame': 150 obs. of  6 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2
##  $ Species     : Factor w/ 3 levels "setosa","versicolor"
##  $ Sepal.Ratio : num  0.686 0.612 0.681 0.674 0.72 ...
```

[3] This creates a copy of all selected data

Under the hood

ffbase rewrites the expression into chunked expression

```
transform( iris_ff
         , Sepal.Ratio = Sepal.Width/Sepal.Length
         )
```

into[4]

```
iris_ff2 <- iris_ff
for (.i in chunk(iris_ff2)){
  iris_ff2[.i,] <- transform( iris_ff2[.i,]
                            , Sepal.Ratio=Sepal.Width/Sepal.Length
                            )
}
return(iris_ff2)
```

[4] greatly simplified

filtering: ffwhich

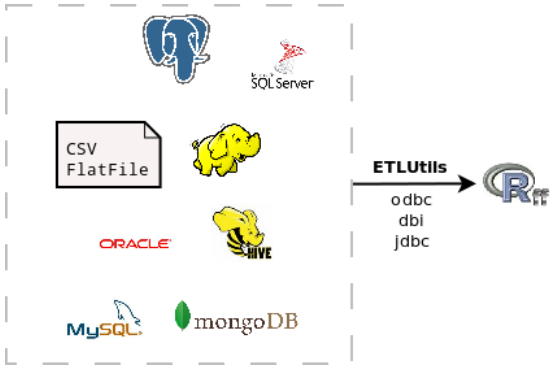
Often we need an index for a subselection, but even this may be to big for memory.

- `ffwhich(dat, expression)` returns a `ff` index vector
- result can be used to index `ffdf` data.frame.

```
idx <- ffwhich(iris_ff, Sepal.Width > 2)
iris_ff[idx, ]
```

Importing data

- Package `ff`: `read.table.ffdf`, `read.csv.ffdf` etcetera
- Package `ETLutils`: `read.dbi.ffdf`, `read.odbc.ffdf`, `read.jdbc.ffdf` SQL Databases (SQLite / PostgreSQL / Oracle / MySQL / SQL Server (read.odbc.ffdf) / Hive (read.jdbc.ffdf) / ...)



ffbase

ffbase adds

- laf_to_ffdf using LaF for importing large csv and fwf files
- ffappend for appending vectors to an existing ff object

```
x <- ffappend(x, 1:10)
```

- ffdfappend for appending data.frame's to an existing ffdf

```
dat <- ffdfappend(dat, iris)
# Note the pattern of assigning the result of the function a
# ff object to itself
```

ff storage

- When data is in `ff` format processing is fast!
- Time bottle neck can be loading the data into `ff`
- Keeping data in `ff` format can save time

`ff` stores all `ff` vectors on disk, however filenames are not user-friendly.

```
basename(filename(iris_ff$Sepal.Length))

## [1] "ffdf1ad05b442183.ff"
```

Furthermore each `ff` vector stores the absolute path.

- Makes moving data around more difficult
- `ff` provides: `ffsave` and `ffload`, which archives and unarchives ff vectors and ffdf data.frames into a zip file.
  Note that this still can be time-consuming.

ffbase has:

- save.ffdf and load.ffdf that store and load ffdf data.frames into a directory with sensible R names.

```
save.ffdf(iris_ff)
basename(filename(iris_ff$Sepal.Length))

## [1] "iris_ff$Sepal.Length.ff"
```

- pack.ffdf and unpack.ffdf that do the same but zip/unzip the result.

Several methods for `ff`

`ffbase` allows statistical models directly on ff data.[5]

- Classification + Regression with `bigglm.ffdf` (*biglm* + *ffbase* packages)
- Least angle regression with `biglars.fit` (*biglars* + *ffbase* packages)
- Randomforest classification with `bigrfc` (*bigrf* + *ffbase* packages)
- Clustering with clustering features of package *stream*

[5] These work on large data but you can also sample from your ffdf to build your stat model and predict chunkwise.

Example of bigglm

Using biglm (with bigglm.ffdf from ffbase)

```
class(iris_ff)

## [1] "ffdf"

nrow(iris_ff)  # that is 1e7!

## [1] 1000000

mymodel <- bigglm(Sepal.Length ~ Petal.Length, data = iris_ff)
coef(mymodel)

##  (Intercept) Petal.Length
##       4.3051       0.4093
```

Thank you!
Questions?