Speeding up (big) data manipulation with data.table package

Vasily Tolkachev

Zurich University of Applied Sciences (ZHAW)

Institute for Data Analysis and Process Design (IDP)

vasily.tolkachev@gmail.com

www.idp.zhaw.ch

21.01.2016

About me

- Sep. 2014 – Present: Research Assistant in Statistics at

Zürcher Hochschule
für Angewandte Wissenschaften

zh aw School of Engineering

IDP Institut für Datenanalyse
und Prozessdesign

- Sep. 2011 – Aug. 2014: MSc Statistics & Research Assistant at

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

- Sep. 2008 – Aug. 2011: BSc Mathematical Economics at

UNIVERSITY OF LONDON

- https://ch.linkedin.com/in/vasily-tolkachev-20130b35

Motivating example from stackoverflow

lapply and do.call running very slowly?

4

4

I have a data frame that is some 35,000 rows, by 7 columns. it looks like this:

```
head(nuc)
```

```
  chr feature    start      end   gene_id    pctAT    pctGC length
1   1     CDS 67000042 67000051 NM_032291 0.600000 0.400000     10
2   1     CDS 67091530 67091593 NM_032291 0.609375 0.390625     64
3   1     CDS 67098753 67098777 NM_032291 0.600000 0.400000     25
4   1     CDS 67101627 67101698 NM_032291 0.472222 0.527778     72
5   1     CDS 67105460 67105516 NM_032291 0.631579 0.368421     57
6   1     CDS 67108493 67108547 NM_032291 0.436364 0.563636     55
```

gene_id is a factor, that has about 3,500 unique levels. I want to, for each level of gene_id get the `min(start)`, `max(end)`, `mean(pctAT)`, `mean(pctGC)`, and `sum(length)`.

I tried using lapply and do.call for this, but it's taking forever +30 minutes to run. the code I'm using is:

```
nuc_prof = lapply(levels(nuc$gene_id), function(gene){
  t = nuc[nuc$gene_id==gene, ]
  return(list(gene_id=gene, start=min(t$start), end=max(t$end), pctGC =
              mean(t$pctGC), pct = mean(t$pctAT), cdslength = sum(t$length)))
})
nuc_prof = do.call(rbind, nuc_prof)
```

I'm certain I'm doing something wrong to slow this down. I haven't waited for it to finish as I'm sure it can be faster. Any ideas?

data.table solution

```
dt = data.table(nuc, key="gene_id")

dt[,list(A = min(start),
         B = max(end),
         C = mean(pctAT),
         D = mean(pctGC),
         E = sum(length)),
       by = key(dt)]
#      gene_id        A        B         C         D   E
# 1: NM_032291 67000042 67108547 0.5582567 0.4417433 283
# 2:       ZZZ 67000042 67108547 0.5582567 0.4417433 283
```

- takes ~ 3 seconds to run !
- easy to program
- easy to understand

Huge advantages of data.table

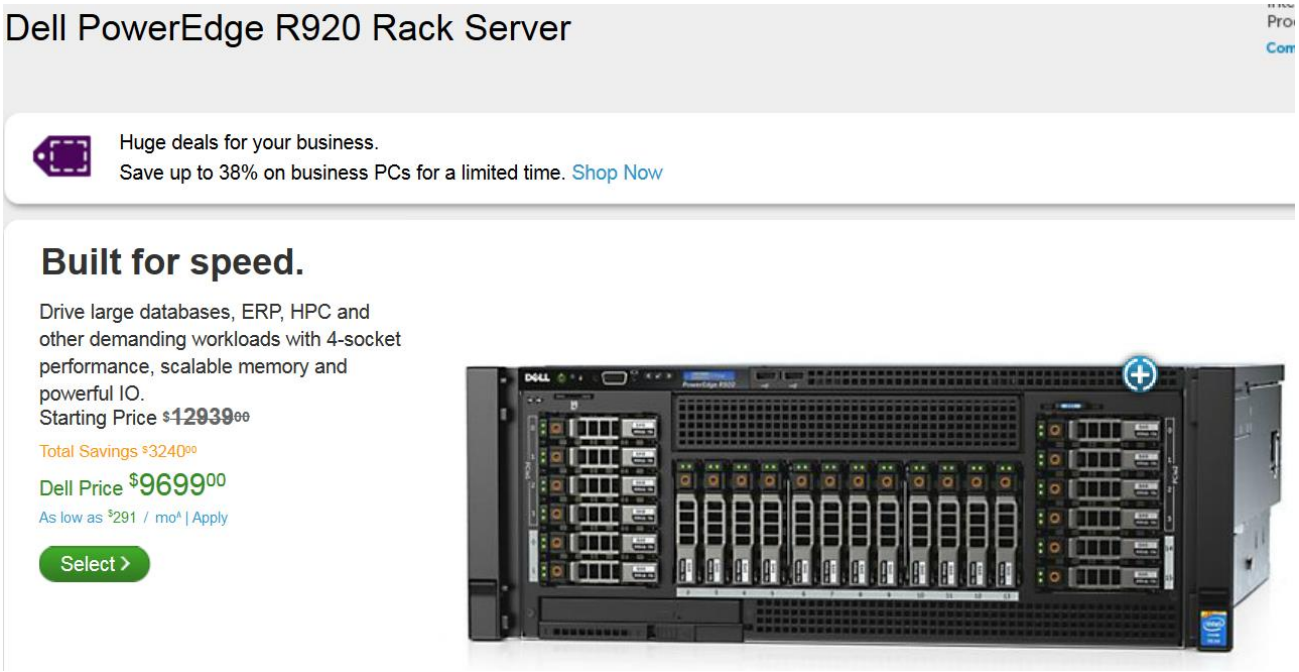
- easier & faster to write the code (no need to write data frame name multiple times)
- easier to read & understand the code
- shorter code
- fast split-apply-combine operations on large data, e.g. 100GB in RAM (up to $2^{31} \approx 2$ billion rows in current R version, provided that you have the RAM)
- fast add/modify/delete columns by reference by group without copies
- fast and smart file reading function (**`fread`**)
- flexible syntax
- easier & faster than other advanced data manipulation packages like dplyr, plyr, readr.
- backward-compatible with code using data.frame
- named one of the success factors by Kaggle competition winners

How to get a lot of RAM

- 240 GB of RAM: https://aws.amazon.com/ec2/details/

amazon web services™ | EC2

- 6 TB of RAM: http://www.dell.com/us/business/p/poweredge-r920/pd



Some limitations of data.table

- although **`merge.data.table`** is faster than **`merge.data.frame`**, it requires the key variables to have the same names

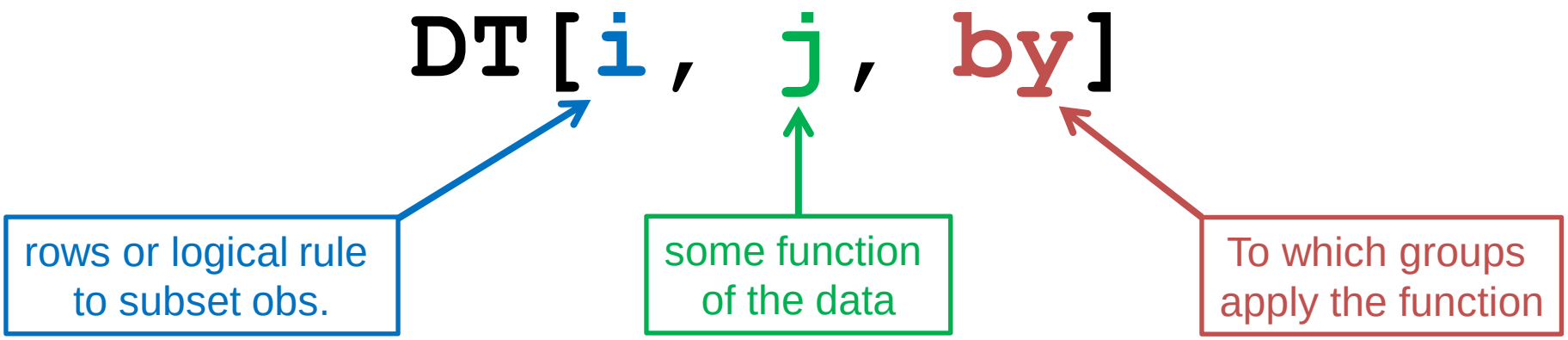
- (In my experience) it may not be compatible with sp and other spatial data packages, as converting sp object to data.table looses the polygons sublist

- Used to be excellent when combined with dplyr's pipeline (%>%) operator for nested commands, but now a bit slower.

- file reading function (**`fread`**) currently does not support some compressed data format (e.g. `.gz, .bz2`)

- It's still limited by your computer's and R limits, but exploits them maximally

General Syntax



Take **DT**,

subset rows using **i**,

then calculate **j**

grouped by **by**

Examples 1

- Let's take a small dataset *Boston* from package MASS as a starting point.

```
> data = data.table(Boston)
> data
        crim zn indus chas   nox    rm  age    dis rad tax ptratio  black lstat medv
  1: 0.00632 18  2.31    0 0.538 6.575 65.2 4.0900   1 296    15.3 396.90  4.98 24.0
  2: 0.02731  0  7.07    0 0.469 6.421 78.9 4.9671   2 242    17.8 396.90  9.14 21.6
  3: 0.02729  0  7.07    0 0.469 7.185 61.1 4.9671   2 242    17.8 392.83  4.03 34.7
  4: 0.03237  0  2.18    0 0.458 6.998 45.8 6.0622   3 222    18.7 394.63  2.94 33.4
  5: 0.06905  0  2.18    0 0.458 7.147 54.2 6.0622   3 222    18.7 396.90  5.33 36.2
 ---
502: 0.06263  0 11.93    0 0.573 6.593 69.1 2.4786   1 273    21.0 391.99  9.67 22.4
503: 0.04527  0 11.93    0 0.573 6.120 76.7 2.2875   1 273    21.0 396.90  9.08 20.6
504: 0.06076  0 11.93    0 0.573 6.976 91.0 2.1675   1 273    21.0 396.90  5.64 23.9
505: 0.10959  0 11.93    0 0.573 6.794 89.3 2.3889   1 273    21.0 393.45  6.48 22.0
506: 0.04741  0 11.93    0 0.573 6.030 80.8 2.5050   1 273    21.0 396.90  7.88 11.9
```

- Accidentally typing the name of a large data table doesn't crush R.

```
> class(data)
[1] "data.table" "data.frame"
```

- It's still a data frame, but if you prefer to use it in the code with data.frames, convertion to data.frame is necessary:

```
> as.data.frame(data)
```

- Converting a data frame to data.table:

```
> data.table(Boston)
  as.data.table(Boston)
```

Examples 2

- Subset rows from 11 to 20:

comma not needed when subsetting rows

```
> data[11:20]
       crim   zn indus chas   nox    rm  age    dis rad tax ptratio  black lstat medv
 1: 0.22489 12.5  7.87    0 0.524 6.377 94.3 6.3467   5 311    15.2 392.52 20.45 15.0
 2: 0.11747 12.5  7.87    0 0.524 6.009 82.9 6.2267   5 311    15.2 396.90 13.27 18.9
 3: 0.09378 12.5  7.87    0 0.524 5.889 39.0 5.4509   5 311    15.2 390.50 15.71 21.7
 4: 0.62976  0.0  8.14    0 0.538 5.949 61.8 4.7075   4 307    21.0 396.90  8.26 20.4
 5: 0.63796  0.0  8.14    0 0.538 6.096 84.5 4.4619   4 307    21.0 380.02 10.26 18.2
 6: 0.62739  0.0  8.14    0 0.538 5.834 56.5 4.4986   4 307    21.0 395.62  8.47 19.9
 7: 1.05393  0.0  8.14    0 0.538 5.935 29.3 4.4986   4 307    21.0 386.85  6.58 23.1
 8: 0.78420  0.0  8.14    0 0.538 5.990 81.7 4.2579   4 307    21.0 386.75 14.67 17.5
 9: 0.80271  0.0  8.14    0 0.538 5.456 36.6 3.7965   4 307    21.0 288.99 11.69 20.2
10: 0.72580  0.0  8.14    0 0.538 5.727 69.5 3.7965   4 307    21.0 390.95 11.28 18.2
```

- In this case the result is a vector:

quotes not needed for variable names

```
> data[11:20, age]
 [1] 94.3 82.9 39.0 61.8 84.5 56.5 29.3 81.7 36.6 69.5
```

- To get a data.table, use **`list()`**

```
> data[11:20, list(age)]
     age
 1: 94.3
 2: 82.9
 3: 39.0
 4: 61.8
 5: 84.5
 6: 56.5
 7: 29.3
 8: 81.7
 9: 36.6
10: 69.5
```

- The usual data.frame style is done with **`with = FALSE`**

```
> data[11:20, 1, with = FALSE]
       crim
 1: 0.22489
 2: 0.11747
 3: 0.09378
 4: 0.62976
 5: 0.63796
 6: 0.62739
 7: 1.05393
 8: 0.78420
 9: 0.80271
10: 0.72580
```

Examples 3

- Find all rows where **`tax`** variable is equal to 216:

```
> data[tax == 216]
      crim   zn indus chas    nox    rm  age    dis rad tax ptratio  black lstat medv
1: 0.01951 17.5  1.38    0 0.4161 7.104 59.5 9.2229   3 216    18.6 393.24  8.05 33.0
2: 0.21038 20.0  3.33    0 0.4429 6.812 32.2 4.1007   5 216    14.9 396.90  4.85 35.1
3: 0.03578 20.0  3.33    0 0.4429 7.820 64.5 4.6947   5 216    14.9 387.31  3.76 45.4
4: 0.03705 20.0  3.33    0 0.4429 6.968 37.2 5.2447   5 216    14.9 392.23  4.59 35.4
5: 0.06129 20.0  3.33    1 0.4429 7.645 49.7 5.2119   5 216    14.9 377.07  3.01 46.0
```

- Find the range of **`crim`** (criminality) variable:

```
> data[, range(crim)]
[1]  0.00632 88.97620
```

- Display values of **`rad`** (radius) variable:

```
> data[, table(rad)]
rad
  1   2   3   4   5   6   7   8  24
 20  24  38 110 115  26  17  24 132
```

- Add a new variable with **`:=`**

```
> data[, rad.f := as.factor(rad) ]
> data[, levels(rad.f)]
[1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "24"
```

Tip: with **`with = FALSE`**, you could also select all columns between some two:

```
> data[, indus:age, with = FALSE]
      indus chas   nox    rm  age
   1:  2.31    0 0.538 6.575 65.2
   2:  7.07    0 0.469 6.421 78.9
   3:  7.07    0 0.469 7.185 61.1
   4:  2.18    0 0.458 6.998 45.8
   5:  2.18    0 0.458 7.147 54.2
  ---
 502: 11.93    0 0.573 6.593 69.1
 503: 11.93    0 0.573 6.120 76.7
 504: 11.93    0 0.573 6.976 91.0
 505: 11.93    0 0.573 6.794 89.3
 506: 11.93    0 0.573 6.030 80.8
```

- i.e. we defined a new factor variable(**`rad.f`**) in the data table from the integer variable radius (**`rad`**), which describes accessibility to radial highways.

Examples 4

- Compute mean of house prices for every level of **`rad.f`**:

```
> data[, .( mean(nox), sd(age), mad(black) ), by = rad.f]
   rad.f        V1       V2        V3
1:     1 0.4628900 25.70204  3.639783
2:     2 0.4849167 23.61596  6.501201
3:     3 0.4524237 25.29279  3.491523
4:     5 0.5708835 26.99779  8.554602
5:     4 0.5043109 30.83723  4.981536
6:     8 0.4925000 21.09286 12.424188
7:     6 0.5148462 24.09800  3.461871
8:     7 0.4410000 26.71751  8.673210
9:    24 0.6724167 12.62581 34.337016
```

- Recall that **`j`** argument is a *function*, so in this case it's a function calling a variable **`medv`**:

```
> data[, medv ]
  [1] 24.0 21.6 34.7 33.4 36.2 28.7 22.9 27.1 16.5 18.9 15.0 18.9 21.7 20.4 18.2 19.9
 [17] 23.1 17.5 20.2 18.2 13.6 19.6 15.2 14.5 15.6 13.9 16.6 14.8 18.4 21.0 12.7 14.5
 [33] 13.2 13.1 13.5 18.9 20.0 21.0 24.7 30.8 34.9 26.6 25.3 24.7 21.2 19.3 20.0 16.6
 [49] 14.4 19.4 19.7 20.5 25.0 23.4 18.9 35.4 24.7 31.6 23.3 19.6 18.7 16.0 22.2 25.0
 [65] 33.0 23.5 19.4 22.0 17.4 20.9 24.2 21.7 22.8 23.4 24.1 21.4 20.0 20.8 21.2 20.3
 [81] 28.0 23.9 24.8 22.9 23.9 26.6 22.5 22.2 23.6 28.7 22.6 22.0 22.9 25.0 20.6 28.4
```

- Below it's a function which is equal to 5:

```
> data[, 5 ]
[1] 5
```

Here's the standard way to select 5th variable

```
> data[, 5, with = FALSE ]
       nox
  1: 0.538
  2: 0.469
  3: 0.469
  4: 0.458
  5: 0.458
 ---
502: 0.573
503: 0.573
504: 0.573
505: 0.573
506: 0.573
```

Examples 5

- Select several variables (result is a data.table)

```
> data[, list(nox, age, black)]
        nox  age  black
  1: 0.538 65.2 396.90
  2: 0.469 78.9 396.90
  3: 0.469 61.1 392.83
  4: 0.458 45.8 394.63
  5: 0.458 54.2 396.90
 ---
502: 0.573 69.1 391.99
503: 0.573 76.7 396.90
504: 0.573 91.0 396.90
505: 0.573 89.3 393.45
506: 0.573 80.8 396.90
```

Or equivalently:

```
> data[, .(nox, age, black)]
        nox  age  black
  1: 0.538 65.2 396.90
  2: 0.469 78.9 396.90
  3: 0.469 61.1 392.83
  4: 0.458 45.8 394.63
  5: 0.458 54.2 396.90
 ---
502: 0.573 69.1 391.99
503: 0.573 76.7 396.90
504: 0.573 91.0 396.90
505: 0.573 89.3 393.45
506: 0.573 80.8 396.90
```

- Compute several functions:

```
> data[, .( mean(nox), sd(age), mad(black) )]
          V1       V2       V3
1: 0.5546951 28.14886 8.094996
```

- Compute these functions for groups (levels) of **`rad.f`**:

```
> data[, .( mean(nox), sd(age), mad(black) ), by = rad.f]
   rad.f        V1       V2        V3
1:     1 0.4628900 25.70204  3.639783
2:     2 0.4849167 23.61596  6.501201
3:     3 0.4524237 25.29279  3.491523
4:     5 0.5708835 26.99779  8.554602
5:     4 0.5043109 30.83723  4.981536
6:     8 0.4925000 21.09286 12.424188
7:     6 0.5148462 24.09800  3.461871
8:     7 0.4410000 26.71751  8.673210
9:    24 0.6724167 12.62581 34.337016
```

Examples 6

- Compute functions for every level of **`rad.f`** and return a data.table with column names:

```
> data[, .( Var1 = mean(nox), Var2 = sd(age), Var3 = mad(black) ), by = rad.f]
   rad.f      Var1     Var2      Var3
1:     1 0.4628900 25.70204  3.639783
2:     2 0.4849167 23.61596  6.501201
3:     3 0.4524237 25.29279  3.491523
4:     5 0.5708835 26.99779  8.554602
5:     4 0.5043109 30.83723  4.981536
6:     8 0.4925000 21.09286 12.424188
7:     6 0.5148462 24.09800  3.461871
8:     7 0.4410000 26.71751  8.673210
9:    24 0.6724167 12.62581 34.337016
```

- Add many new variables with `` `:=`() ``.
  If a variable attains only a single value, copy it for each observation:

```
> data[, `:=`( Var1 = mean(nox), Var2 = sd(age), Var3 = mad(black) )]
> data
        crim zn indus chas   nox    rm  age    dis rad tax ptratio  black lstat medv
  1: 0.00632 18  2.31    0 0.538 6.575 65.2 4.0900   1 296    15.3 396.90  4.98 24.0
  2: 0.02731  0  7.07    0 0.469 6.421 78.9 4.9671   2 242    17.8 396.90  9.14 21.6
  3: 0.02729  0  7.07    0 0.469 7.185 61.1 4.9671   2 242    17.8 392.83  4.03 34.7
  4: 0.03237  0  2.18    0 0.458 6.998 45.8 6.0622   3 222    18.7 394.63  2.94 33.4
  5: 0.06905  0  2.18    0 0.458 7.147 54.2 6.0622   3 222    18.7 396.90  5.33 36.2
     rad.f      Var1     Var2     Var3
  1:     1 0.5546951 28.14886 8.094996
  2:     2 0.5546951 28.14886 8.094996
  3:     2 0.5546951 28.14886 8.094996
  4:     3 0.5546951 28.14886 8.094996
  5:     3 0.5546951 28.14886 8.094996
```

- Updating or deletion of old variables/columns is done the same way

Examples 7

- Compute a more complicated function for groups. It's a weighted mean of house prices, with **`dis`** (distances to Boston employment centers) as weights:

```
> data[, sum(medv * dis)/sum(dis), by = rad.f ]
   rad.f       V1
1:     1 24.78636
2:     2 28.62271
3:     3 27.18226
4:     5 25.88535
5:     4 22.28278
6:     8 28.19286
7:     6 21.66519
8:     7 26.58133
9:    24 16.67276
```

- Dynamic variable creation. Now let's create a variable of weighted means (**`mean_w`**), and then use it to create a variable for weighted standard deviation (**`std_w`**).

```
> data[,  `:=`(mean_w = mean_w <- sum(medv * dis)/sum(dis),
+              std_w = sqrt( sum( dis * (medv - mean_w)^2 )/sum(dis) ) ),
+         by = rad.f ][]
        crim zn indus chas   nox    rm  age    dis rad tax ptratio  black lstat medv
  1: 0.00632 18  2.31    0 0.538 6.575 65.2 4.0900   1 296    15.3 396.90  4.98 24.0
  2: 0.02731  0  7.07    0 0.469 6.421 78.9 4.9671   2 242    17.8 396.90  9.14 21.6
  3: 0.02729  0  7.07    0 0.469 7.185 61.1 4.9671   2 242    17.8 392.83  4.03 34.7
  4: 0.03237  0  2.18    0 0.458 6.998 45.8 6.0622   3 222    18.7 394.63  2.94 33.4
  5: 0.06905  0  2.18    0 0.458 7.147 54.2 6.0622   3 222    18.7 396.90  5.33 36.2
     rad.f      Var1     Var2     Var3   mean_w    std_w
  1:     1 0.5546951 28.14886 8.094996 24.78636 7.613547
  2:     2 0.5546951 28.14886 8.094996 28.62271 7.335980
  3:     2 0.5546951 28.14886 8.094996 28.62271 7.335980
  4:     3 0.5546951 28.14886 8.094996 27.18226 8.062266
  5:     3 0.5546951 28.14886 8.094996 27.18226 8.062266
```

Examples 8

- What if variable names are too long and you have a non-standard function where they are used multiple times?
- Of course, it's possible to change variable names, do the analysis and then return to the original names, but if this isn't an option, one needs to use a list for variable names **`.SD`**, and the variables are specified in **`.SDcols`**:

use these instead of variable names

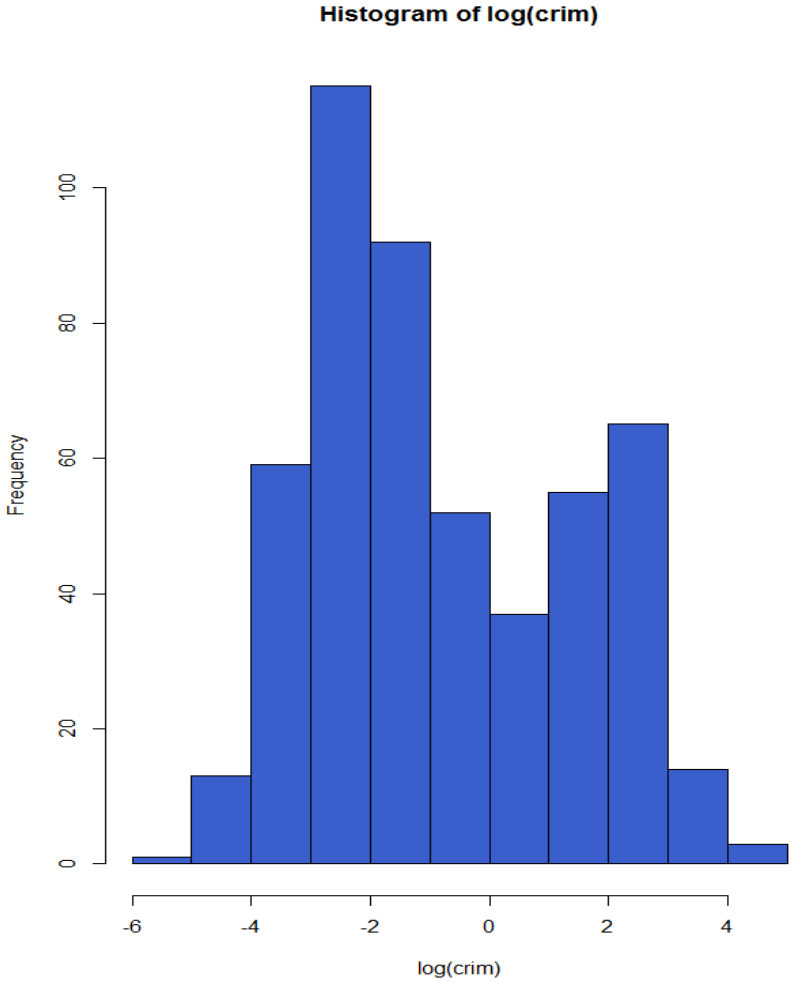
give variable names here

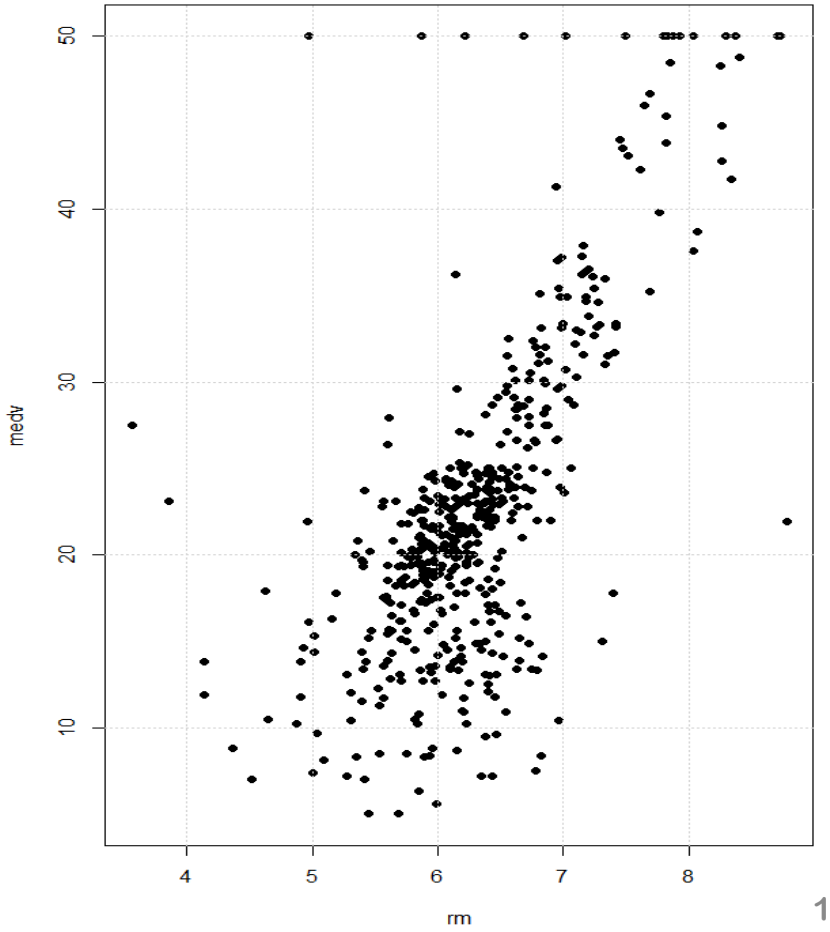
```
> data[, `:=`( x = sum(.SD[[1]]^2) / sum(.SD[[1]]),
+              y = sum(.SD[[2]]^2) / sum(.SD[[2]]) ),
+         by = rad.f,
+         .SDcols = c("medv", "age") ][]
        crim zn indus chas   nox    rm  age    dis rad tax ptratio  black lstat medv
  1: 0.00632 18  2.31    0 0.538 6.575 65.2 4.0900   1 296    15.3 396.90  4.98 24.0
  2: 0.02731  0  7.07    0 0.469 6.421 78.9 4.9671   2 242    17.8 396.90  9.14 21.6
  3: 0.02729  0  7.07    0 0.469 7.185 61.1 4.9671   2 242    17.8 392.83  4.03 34.7
  4: 0.03237  0  2.18    0 0.458 6.998 45.8 6.0622   3 222    18.7 394.63  2.94 33.4
     rad.f      Var1     Var2     Var3   mean_w    std_w        x        y
  1:     1 0.5546951 28.14886 8.094996 24.78636 7.613547 26.87566 58.96314
  2:     2 0.5546951 28.14886 8.094996 28.62271 7.335980 29.04783 73.02262
  3:     2 0.5546951 28.14886 8.094996 28.62271 7.335980 29.04783 73.02262
  4:     3 0.5546951 28.14886 8.094996 27.18226 8.062266 30.34496 61.94252
```

Examples 9

- Multiple expressions in `j` could be handled with `{ }`:

```
> par(mfrow = c(1,2))
> data[, { hist(log(crim), col = "royalblue3")
+          plot(rm, medv, pch = 16)
+          grid()
+        } ]
```





Examples 10

- Hence, a separate data.table with dynamically created variables can be done by

```
> data[, { list(mean_w = mean_w <- sum(medv * dis)/sum(dis),
+               std_w = sqrt( sum( dis * (medv - mean_w)^2 )/sum(dis) )
+               )},
+       by = rad.f ]
   rad.f   mean_w    std_w
1:     1 24.78636 7.613547
2:     2 28.62271 7.335980
3:     3 27.18226 8.062266
4:     5 25.88535 8.230743
5:     4 22.28278 6.906212
6:     8 28.19286 9.206210
7:     6 21.66519 2.150142
8:     7 26.58133 6.658226
9:    24 16.67276 7.317581
```

- Changing a subset of observations. Let's create another factor variable **`crim.f`** with 3 levels standing for low, medium and severe crime rates per capita:

```
> data[           , crim.f := "low"]
> data[ crim >= 1 , crim.f := "medium"]
> data[ crim >= 10, crim.f := "severe"]
> data[           , crim.f := as.factor(crim.f)][]
        crim zn indus chas   nox    rm  age    dis rad tax ptratio  black lstat medv
  1: 0.00632 18  2.31    0 0.538 6.575 65.2 4.0900   1 296    15.3 396.90  4.98 24.0
  2: 0.02731  0  7.07    0 0.469 6.421 78.9 4.9671   2 242    17.8 396.90  9.14 21.6
  3: 0.02729  0  7.07    0 0.469 7.185 61.1 4.9671   2 242    17.8 392.83  4.03 34.7
  4: 0.03237  0  2.18    0 0.458 6.998 45.8 6.0622   3 222    18.7 394.63  2.94 33.4
     rad.f      Var1     Var2     Var3   mean_w    std_w        x        y crim.f
  1:     1 0.5546951 28.14886 8.094996 24.78636 7.613547 26.87566 58.96314    low
  2:     2 0.5546951 28.14886 8.094996 28.62271 7.335980 29.04783 73.02262    low
  3:     2 0.5546951 28.14886 8.094996 28.62271 7.335980 29.04783 73.02262    low
  4:     3 0.5546951 28.14886 8.094996 27.18226 8.062266 30.34496 61.94252    low
```

Examples 11. Chaining

```
DT[i, j, by][i, j, by]
```

- It's a very powerful way of doing multiple operations in one command
- The command for **`crim.f`** on the previous slide can thus be done by

```
> data[ , crim.f := "low"] [ crim >= 1, crim.f := "medium"]
> data[ crim >= 10, crim.f := "severe"][, crim.f := as.factor(crim.f)]
> levels(data$crim.f)
[1] "low"    "medium" "severe"
```

- Or in one go:

```
data[…, …][…, …][…, …][…, …]
```

```
data[ , crim.f := "low"] [
  crim >= 1, crim.f := "medium"] [
    crim >= 10, crim.f := "severe"][,
      , crim.f := as.factor(crim.f)]
```

Examples 12

- Now that we have 2 factor variables, **`crim.f`** and **`rad.f`**, we can also apply functions in **`j`** on two groups:

```
> data[, .(mean(medv), sd(medv)), by = .(rad.f, crim.f) ]
    rad.f crim.f       V1        V2
 1:     1    low 24.36500  8.024454
 2:     2    low 26.83333  7.874376
 3:     3    low 27.92895  8.324692
 4:     5    low 26.43294  8.163557
 5:     4    low 22.06735  6.982154
 6:     4 medium 15.83333  3.472577
 7:     8    low 30.35833  9.727724
 8:     6    low 20.97692  2.312801
 9:     5 medium 23.65000 11.963126
10:     7    low 27.10588  6.493215
11:    24 medium 19.21667  9.259525
12:    24 severe 12.34074  5.217840
```

- It appears that there is one remote district with severe crime rates.
- **`.N`** function counts the number observations in a group:

```
> data[, .N, by = .(rad.f, crim.f) ]
    rad.f crim.f  N
 1:     1    low 20
 2:     2    low 24
 3:     3    low 38
 4:     5    low 85
 5:     4    low 98
 6:     4 medium 12
 7:     8    low 24
 8:     6    low 26
 9:     5 medium 30
10:     7    low 17
11:    24 medium 78
12:    24 severe 54
```

Examples 13

- Another useful function is **`.SD`** which contains values of all variables except the one used for grouping:

```
> data[, .SD, by =  crim.f ]
      crim.f     crim zn indus chas   nox    rm  age    dis rad tax ptratio  black lstat medv rad.f
   1:    low  0.00632 18  2.31    0 0.538 6.575 65.2 4.0900   1 296    15.3 396.90  4.98 24.0     1
   2:    low  0.02731  0  7.07    0 0.469 6.421 78.9 4.9671   2 242    17.8 396.90  9.14 21.6     2
   3:    low  0.02729  0  7.07    0 0.469 7.185 61.1 4.9671   2 242    17.8 392.83  4.03 34.7     2
   4:    low  0.03237  0  2.18    0 0.458 6.998 45.8 6.0622   3 222    18.7 394.63  2.94 33.4     3
   5:    low  0.06905  0  2.18    0 0.458 7.147 54.2 6.0622   3 222    18.7 396.90  5.33 36.2     3
  ---
 502: severe 15.57570  0 18.10    0 0.580 5.926 71.0 2.9084  24 666    20.2 368.74 18.13 19.1    24
 503: severe 13.07510  0 18.10    0 0.580 5.713 56.7 2.8237  24 666    20.2 396.90 14.76 20.1    24
 504: severe 15.02340  0 18.10    0 0.614 5.304 97.3 2.1007  24 666    20.2 349.48 24.91 12.0    24
 505: severe 10.23300  0 18.10    0 0.614 6.185 96.7 2.1705  24 666    20.2 379.70 18.03 14.6    24
 506: severe 14.33370  0 18.10    0 0.614 6.229 88.0 1.9512  24 666    20.2 383.32 13.11 21.4    24
```

- Use **`setnames()`** and **`setcolorder()`** functions to change column names or reorder them:

```
> setnames(data, c("rm", "zn") , c("rooms_average", "proportion_zoned"))[]
        crim proportion_zoned indus chas   nox rooms_average  age    dis rad tax ptratio  black lstat medv rad.f crim.f
   1: 0.00632              18  2.31    0 0.538         6.575 65.2 4.0900   1 296    15.3 396.90  4.98 24.0     1    low
   2: 0.02731               0  7.07    0 0.469         6.421 78.9 4.9671   2 242    17.8 396.90  9.14 21.6     2    low
   3: 0.02729               0  7.07    0 0.469         7.185 61.1 4.9671   2 242    17.8 392.83  4.03 34.7     2    low
   4: 0.03237               0  2.18    0 0.458         6.998 45.8 6.0622   3 222    18.7 394.63  2.94 33.4     3    low
   5: 0.06905               0  2.18    0 0.458         7.147 54.2 6.0622   3 222    18.7 396.90  5.33 36.2     3    low
  ---
 502: 0.06263               0 11.93    0 0.573         6.593 69.1 2.4786   1 273    21.0 391.99  9.67 22.4     1    low
 503: 0.04527               0 11.93    0 0.573         6.120 76.7 2.2875   1 273    21.0 396.90  9.08 20.6     1    low
 504: 0.06076               0 11.93    0 0.573         6.976 91.0 2.1675   1 273    21.0 396.90  5.64 23.9     1    low
 505: 0.10959               0 11.93    0 0.573         6.794 89.3 2.3889   1 273    21.0 393.45  6.48 22.0     1    low
 506: 0.04741               0 11.93    0 0.573         6.030 80.8 2.5050   1 273    21.0 396.90  7.88 11.9     1    low
```

Examples 14. Key on one variable

- The reason why data.table works so fast is the use of keys. All observations are internally indexed by the way they are stored in RAM and sorted using Radix sort.

- *Any* column can be set as a key (list & complex number classes not supported), and duplicate entries are allowed.

- **`setkey(DT, colA)`** introduces an index for column A and sorts the data.table by it increasingly. In contrast to data.frame style, this is done without extra copies and with a very efficient memory use.

- After that it's possible to use

  binary search by providing index values directly **`data["1"],`** which is 100-1000… times faster than

  vector scan **`data[rad.f == "1"]`**

- Setting keys is necessary for joins and significantly speeds up things for big data. However, it's not necessary for **`by =`** aggregation.

Examples 15. Keys on multiple variables

- Any number of columns can be set as key using **`setkey()`**. This way rows can be selected on 2 keys.
- **`setkey(DT, colA, colB)`** introduces indexes for both columns and sorts the data.table by column A, then by column B within each group of column A:

- Then *binary search* on two keys is

```
> setkey(data, rad.f, crim.f)
> data[ .("7", "low") ]
       crim zn indus chas   nox    rm  age    dis rad tax ptratio  black lstat medv rad.f crim.f
 1: 0.20608 22  5.86    0 0.431 5.593 76.5 7.9549   7 330    19.1 372.49 12.50 17.6     7    low
 2: 0.19133 22  5.86    0 0.431 5.605 70.2 7.9549   7 330    19.1 389.13 18.46 18.5     7    low
 3: 0.33983 22  5.86    0 0.431 6.108 34.9 8.0555   7 330    19.1 390.18  9.16 24.3     7    low
 4: 0.19657 22  5.86    0 0.431 6.226 79.2 8.0555   7 330    19.1 376.14 10.15 20.5     7    low
 5: 0.16439 22  5.86    0 0.431 6.433 49.1 7.8265   7 330    19.1 374.71  9.52 24.5     7    low
 6: 0.19073 22  5.86    0 0.431 6.718 17.5 7.8265   7 330    19.1 393.74  6.56 26.2     7    low
```

Vector Scan vs. Binary Search

Vector Scan	Binary search
`data[rad.f ==“7” & crim.f == “low”]`	`setkey(data, rad.f, crim.f)` `data[ .(“7”, “low")]`
$O(n)$	$O(\log(n))$

- The reason *vector scan* is so inefficient is that is searches first for entries “7” in **`rad.f`** variable row-by-row, then does the same for **`crim.f`**, then takes element-wise intersection of logical vectors.
- *Binary search*, on the other hand, searches already on *sorted* variables, and hence cuts the number of observations by half at each step.
- Since rows of each column of data.tables have corresponding locations in RAM memory, the operations are performed in a very cache efficient manner.
- In addition, since the matching row indices are obtained directly without having to create huge logical vectors (equal to the number of rows in a data.table), it is quite memory efficient as well.

What to avoid

- Avoid **`read.csv`** function which takes hours to read in files > 1 Gb. Use **`fread`** instead. It's a lot smarter and more efficient, e.g. it can guess the separator.
- Avoid **`rbind`** which is again notoriously slow. Use **`rbindlist`** instead.
- Avoid using data.frame's vector scan inside data.table:

```
data[ data$rad.f == "7" & data$crim.f == "low", ]
```

(even though data.table's vector scan is faster than data.frame's vector scan, this slows it down.)

- In general, avoid using $ inside the data.table, whether it's for subsetting, or updating some subset of the observations:

```
data[ data$rad.f == "7", ] = data[ data$rad.f == "7", ] + 1
```

- For speed use `:=` by group, don't transform() by group or cbind() afterwards
- data.table used to work with dplyr well, but now it is usually slow:

```
data %>% filter(rad == 1)
```

Speed comparison

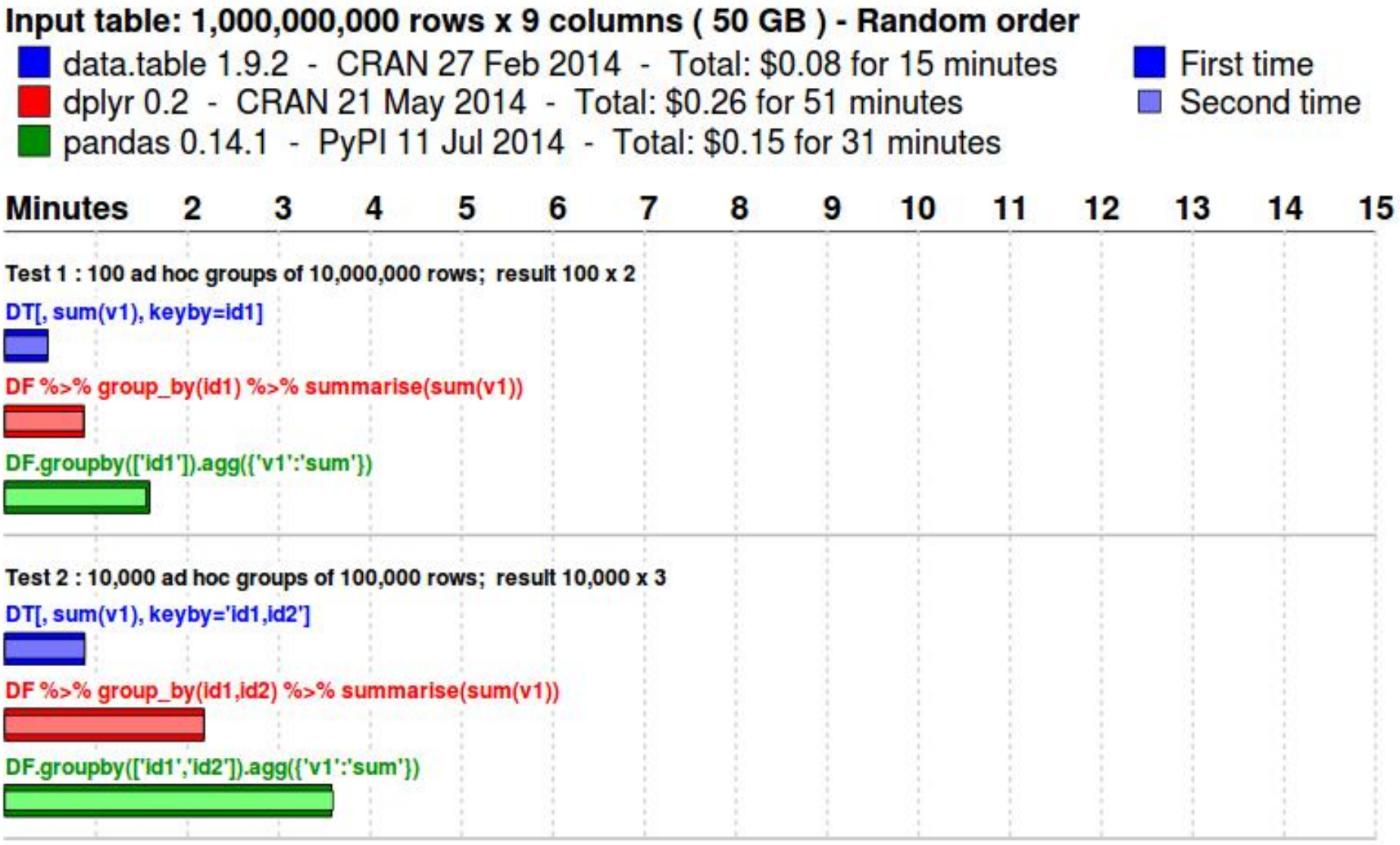
- Create artificial data which is randomly ordered. No pre-sort. No indexes. No key.
- 5 simple queries are run: large groups and small groups on different columns of different types. Similar to what a data analyst might do in practice; i.e., various ad hoc aggregations as the data is explored and investigated.
- Each package is tested separately in its own fresh session.
- Each query is repeated once more, immediately. This is to isolate cache effects and confirm the first timing. The first and second times are plotted. The total runtime of all 5 tests is also displayed.
- The results are compared and checked allowing for numeric tolerance and column name differences.
- It is the toughest test the developers could think of but happens to be realistic and very common.

Speed comparison. Data

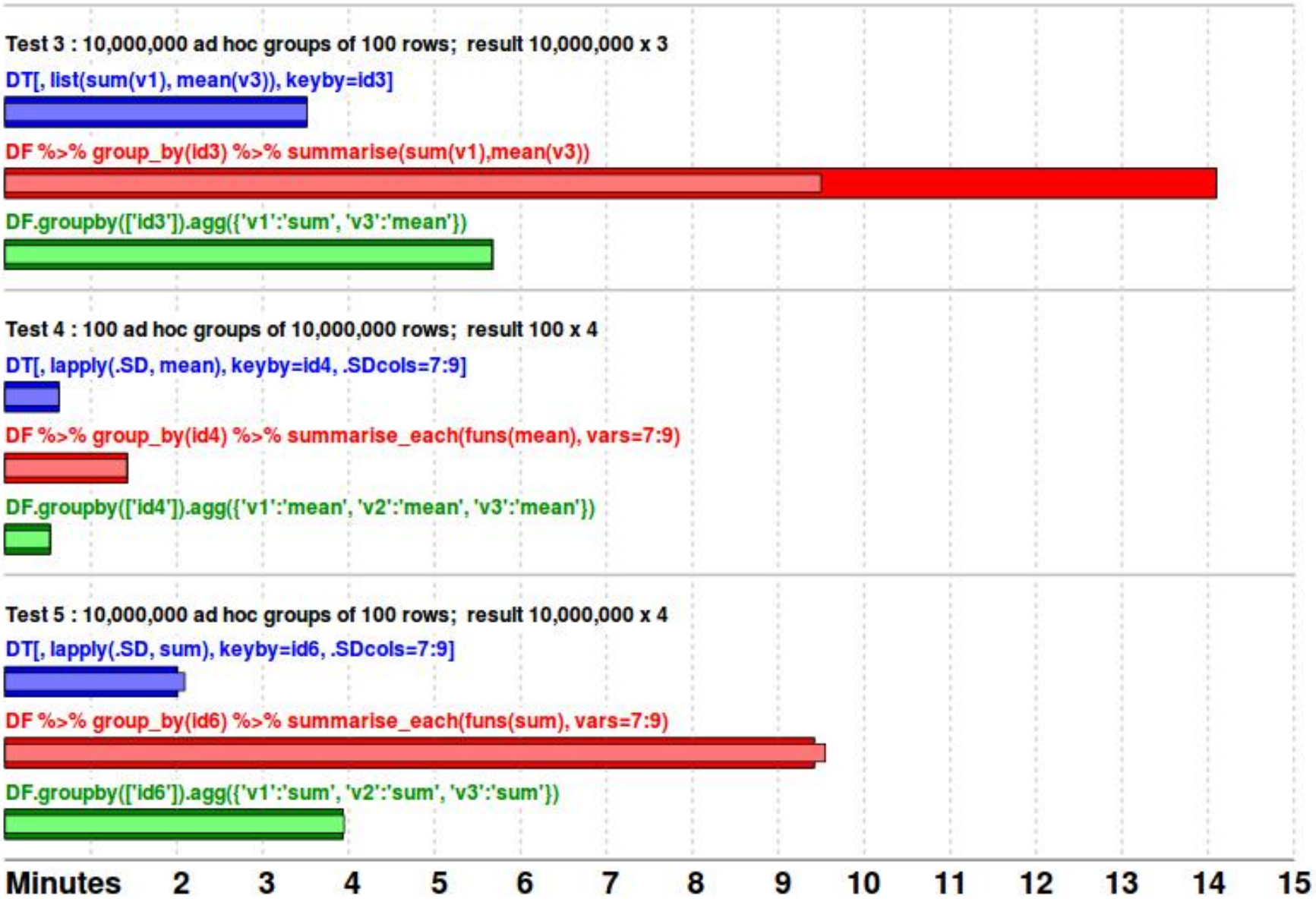
- The artificial dataset looks like:

```
> str(DT)
Classes ‘data.table’ and 'data.frame':  20000000 obs. of  9 variables:
 $ id1: chr  "id027" "id038" "id058" "id091" ...
 $ id2: chr  "id096" "id053" "id009" "id078" ...
 $ id3: chr  "id0000013671" "id0000009982" "id0000156961" "id0000163410" ...
 $ id4: int  10 78 93 11 10 28 67 89 92 8 ...
 $ id5: int  35 14 11 42 98 87 67 72 65 22 ...
 $ id6: int  118504 117110 150665 192840 130654 5484 199643 67509 183684 36116 ...
 $ v1 : int  1 1 2 2 1 2 1 2 4 5 ...
 $ v2 : int  1 1 1 3 4 1 2 2 1 2 ...
 $ v3 : num  78.2 83 92.7 41.2 90 ...
 - attr(*, ".internal.selfref")=<externalptr>
>
> DT
             id1   id2           id3 id4 id5    id6 v1 v2      v3
       1: id027 id096 id0000013671  10  35 118504  1  1 78.2043
       2: id038 id053 id0000009982  78  14 117110  1  1 83.0069
       3: id058 id009 id0000156961  93  11 150665  2  1 92.6780
       4: id091 id078 id0000163410  11  42 192840  2  3 41.1685
       5: id021 id063 id0000046351  10  98 130654  1  4 89.9839
      ---
19999996: id028 id063 id0000028651  62  13 175825  1  1  7.6364
19999997: id096 id029 id0000126326  20  21  33834  1  3 62.1945
19999998: id069 id099 id0000191415  87  69 112773  2  3 49.2211
19999999: id008 id069 id0000010304  71  58  85497  1  5 98.3131
20000000: id086 id064 id0000119991  33  67 197489  2  5 89.9101
```

Speed comparison



Speed comparison



References

- Matt Dowle's data.table git account with newest vignettes.
  https://github.com/Rdatatable/data.table/wiki
- Matt Dowle's presentations & conference videos.
  https://github.com/Rdatatable/data.table/wiki/Presentations
- Official introduction to data.table:
  https://github.com/Rdatatable/data.table/wiki/Getting-started
- Why to set keys in data.table:
  http://stackoverflow.com/questions/20039335/what-is-the-purpose-of-setting-a-key-in-data-table
- Performance comparisons to other packages:
  https://github.com/Rdatatable/data.table/wiki/Benchmarks-%3A-Grouping
- Comprehensive data.table summary sheet:
  https://s3.amazonaws.com/assets.datacamp.com/img/blog/data+table+cheat+sheet.pdf
- An unabridged comparison of dplyr and data.table:
  http://stackoverflow.com/questions/21435339/data-table-vs-dplyr-can-one-do-something-well-the-other-cant-or-does-poorly/27840349#27840349

Thanks a lot for
your attention and interest!