Wrangling data at scale with data.table

Gresa Smolica and Amin Oueslati

Hertie School of Governance

Introduction to Data Science 2022

Problems with large data in R

R commonly fails to process large data sets – troubling beginners and experts alike

```
> ## II. Movie and user effect model ##
>
> # Plot penaly term user effect #
>
> avg_users <- edx %>%
+   left_join(avg_movie_rating, by='movieId') .... [TRUNCATED]
Error: cannot allocate vector of size 68.7 Mb
```

```
> wids_tidy1 <- wids_tidy %>%
+   gather(key = "Diagnosis", value = "Diagnosed", aids:solid_tumor_with_metastasis)
Error: cannot allocate vector of size 442.2 Mb
```

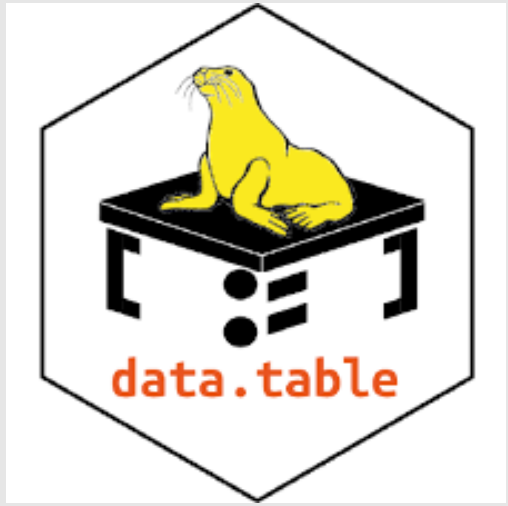
```
Error cannot allocate vector of size 113.9 Mb
```

"Do I need a new laptop?"

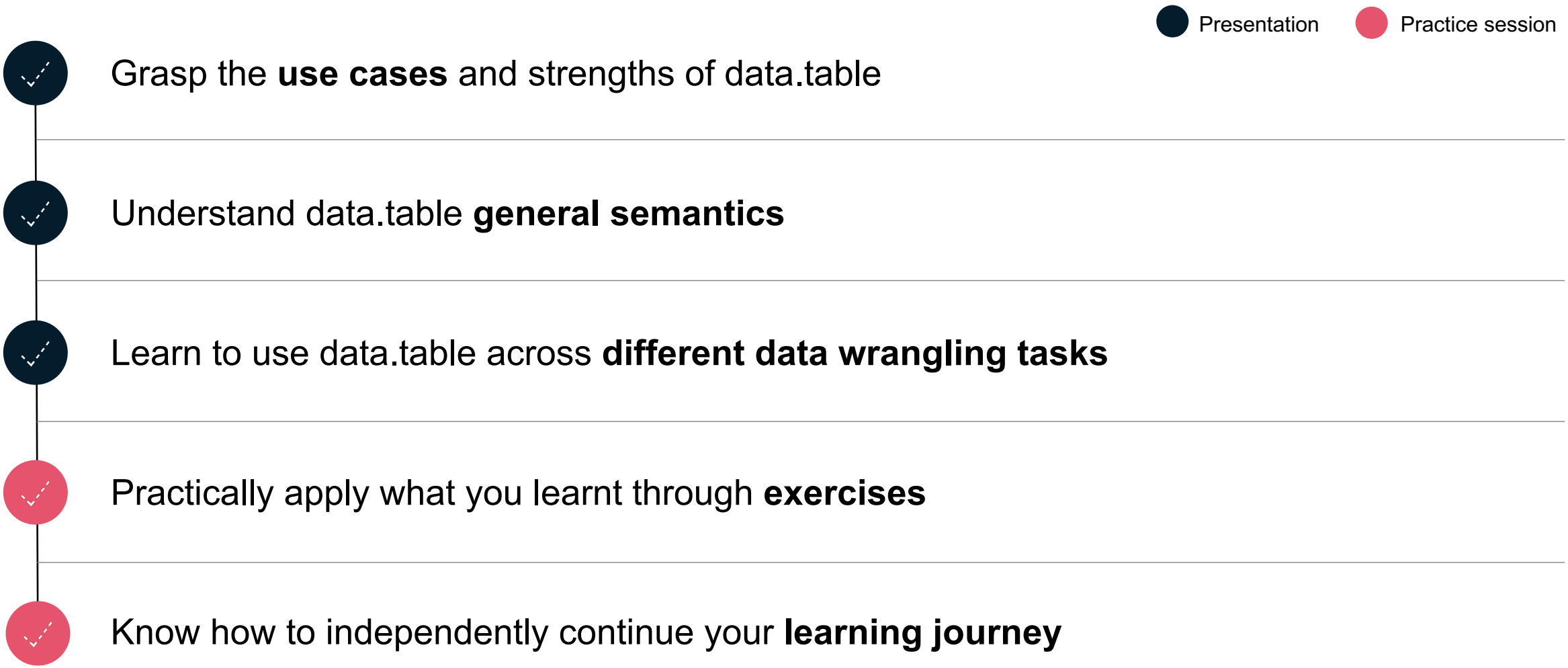
"What's all about these Apple M1 processors?"

1st year MDs students,
prefer to remain anonymous

NO!
Just use **data.table**



Learning objectives for the workshop



3 reasons for using data.table

Versatile for data wrangling

Swiss Army Knife for data wrangling tasks

Functionally, data.table is comparable to **tidyverse**

Performance-oriented

Very **fast and memory efficient**

Outperforms dplyr, data.frame or pandas on **large data sets**

Consistent

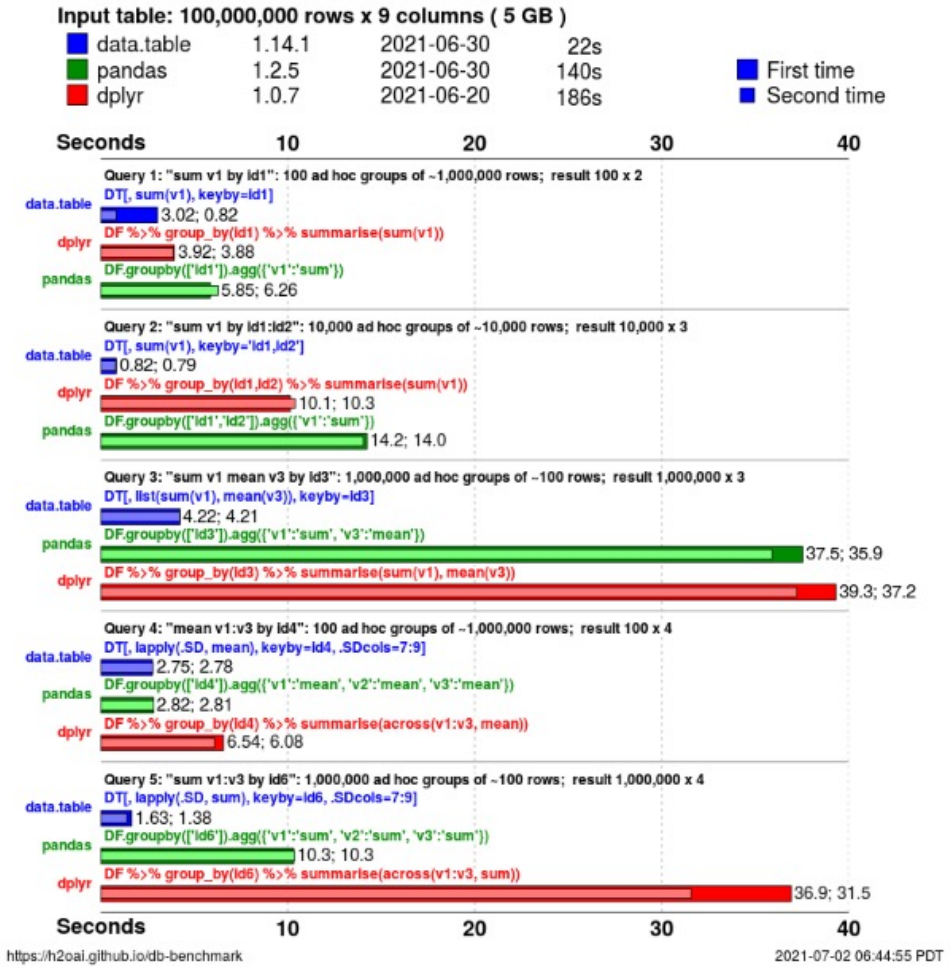
No dependencies other than base R

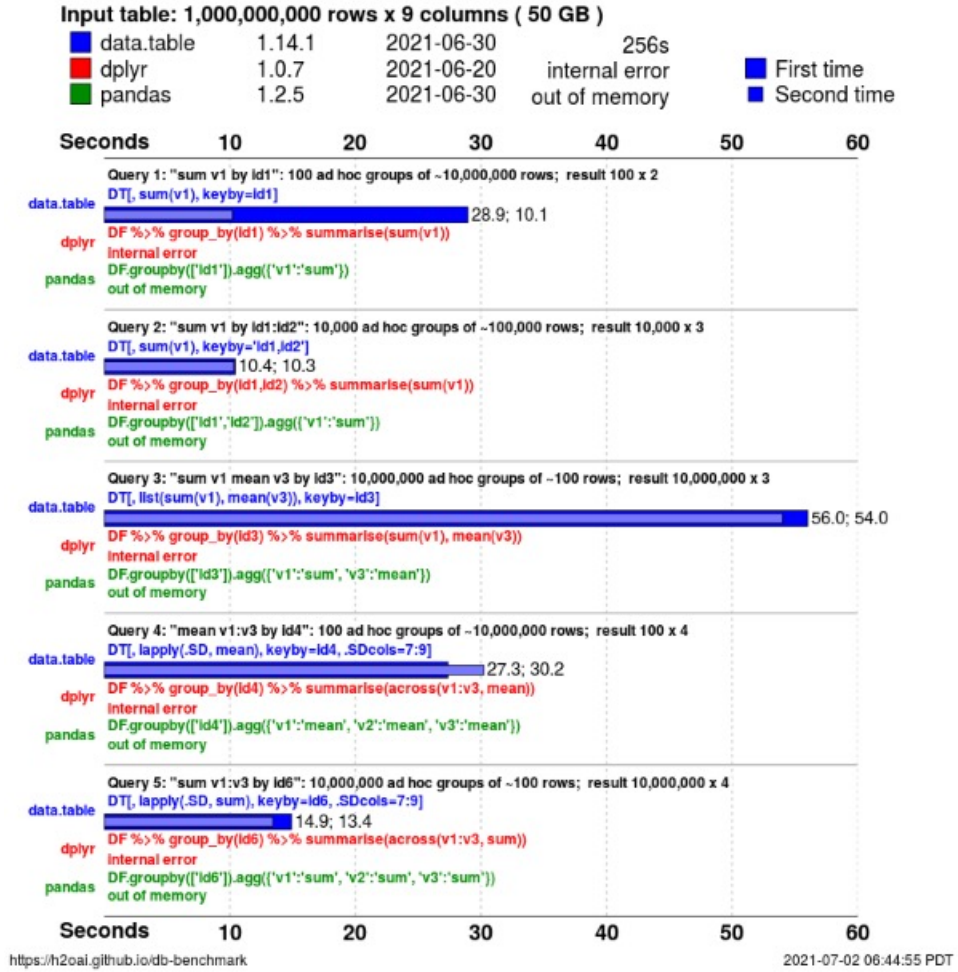
Compatible with other packages, anything that works with a df

Consistent syntax, slightly less intuitive than tidyverse

Ok, but how fast is it really?

Benchmarking of data.table, tidyverse and pandas on grouping task





Source: https://h2oai.github.io/db-benchmark/

Basic data.table syntax

DT[i, j, by]

DT denotes the **data table object**

i: indexes **rows**

j: indexes **columns**, reference for any computation by column

by: ‘group by’, refers to any **variable** by which to **group the operation**

Further syntax and tips

The **commas matter**, as they indicate whether we are referring to i, j or by

In data.table we refer to column names as if they were variables, **no need for quotation marks or $-signs**

To return a data.table object, we **must wrap the columns** under j with **list**() or as shortcut .()

Data wrangling tasks (1/6): Sub-setting rows

Tidyverse equivalent: **filter**

```
## Sub-set all male students from Albania
male_albanian_studens <- student_data[country == "ALB" & gender == "male"]

## as you can see, a comma after the condition in "i"is not required
## but student_data[country == "ALB" & gender == "male",] would work just fine
```

country <fctr>	student_id <fctr>	gender <fctr>	math <dbl>	read <dbl>	science <dbl>
ALB	800251	male	490.187	375.984	445.039
ALB	800402	male	462.464	434.352	421.731
ALB	803546	male	482.501	425.131	515.942
ALB	804776	male	459.804	306.028	328.261
ALB	805287	male	441.037	271.213	391.562

Subset all rows which meet the **condition(s)** in i

A **comma** after the condition in i is not required

```
## Order students in increasing order by their math score
incr_order <- male_albanian_students[order(math)]
```

```
## Order students in decreasing order by their math score
decr_ord <- male_albanian_studens[order(-math)]
```

country <fctr>	student_id <fctr>	gender <fctr>	math <dbl>	read <dbl>	science <dbl>
ALB	803745	male	788.671	530.154	558.363
ALB	803432	male	730.705	592.545	603.320
ALB	801075	male	725.921	542.705	540.001
ALB	802315	male	693.042	644.056	638.658
ALB	801983	male	688.525	599.703	597.521

We order rows with **order()-command**

By default, the **order is increasing**, which can be **reversed** with a **minus sign** before the order command

Data wrangling tasks (1/6): Sub-setting rows con' %between% and %inrange%

%between% and %inrange% are useful to **subset rows** conditional on a value falling within a **certain range**

When used with a **scalar**, %between% and %inrange% **perform identically**

However, both operators also **accept vectors**, in which case the operate differently:

- **%between% evaluates each row** and returns T/F if the value of interest falls between the two vectors - the **range** defined by the two vectors **varies for each row**
- More on %inrange% in the practice material

```
between_vector <- male_albanian_studens[math %between%
list(science, read)]
between_vector[1:5]
```

country <fctr>	student_id <fctr>	gender <fctr>	math <dbl>	read <dbl>	science <dbl>
ALB	801814	male	366.898	409.533	366.636
ALB	801470	male	510.842	514.861	460.068
ALB	800519	male	508.325	516.116	489.465
ALB	801021	male	438.600	443.990	431.792
ALB	803960	male	451.147	478.775	425.916

Data wrangling tasks (2/6): Extracting and modifying columns

Tidyverse equivalent: **select**

```
# returning a vector
student_id_vector <- student_data[, student_id]
```

```
# returning a data.table, wrapping with .()
select_columns <- student_data[,
  .(student_id, country, gender, read, science)]
select_columns[1:5]
```

student_id <fctr>	country <fctr>	gender <fctr>	read <dbl>	science <dbl>
800251	ALB	male	375.984	445.039
800402	ALB	male	434.352	421.731
801902	ALB	female	359.191	392.223
803546	ALB	male	425.131	515.942
804776	ALB	male	306.028	328.261

Select all **columns** which are specified under **j** – the prior comma matters (!)

Wrapping the columns with .() makes sure a data table is returned, otherwise this would be a vector; we can also use list()

```
# drop selected columns
select_columns_short <- select_columns[, !c("read", "science")]
select_columns_short
```

student_id <fctr>	country <fctr>	gender <fctr>
800251	ALB	male
800402	ALB	male
801902	ALB	female
803546	ALB	male
804776	ALB	male

By adding a **! sign** before the list of columns, we **drop** those columns specified

Attention: here we have to refer to the column names as strings

Data wrangling tasks (3/6): Creating new columns

Tidyverse equivalent: **mutate**

```
# creating a new column
total_score <- male_albanian_students[,
  total_score := read + science]
```

gender <fctr>	math <dbl>	read <dbl>	science <dbl>	total_score <dbl>
male	490.187	375.984	445.039	821.023
male	462.464	434.352	421.731	856.083
male	482.501	425.131	515.942	941.073
male	459.804	306.028	328.261	634.289
male	441.037	271.213	391.562	662.775

The **:= assignment symbol** allows us to **create new columns** through the modification/combination of one ore more columns

As part of the assignment we can optionally specify the **name of the new column**

```
# creating multiple new columns
multiple_columns <- school_data[ , `:=` (
  total_fun = fund_gov + fund_fees + fund_donation,  total_students =
enrol_boys + enrol_girls)]
```

sch_wgt <dbl>	school_size <dbl>	total_fun <dbl>	total_students <dbl>
1	2367	100	2367
1	813	100	813
1	1003	98	1003
1	315	100	315
1	498	80	498

To create **several new columns** at once, we use this convention `:=`

Data wrangling tasks (4/6): Descriptive analysis

Tidyverse equivalent: **summarise**

```
# school with most enrolled boys
max_boys <- clean_school_data[, max(enrol_boys)]

# school with fewest enrolled girls
min_girls <- clean_school_data[, min(enrol_girls)]

# adding logical operators: number of schools with more girls than boys
n_schools_more_girls <- clean_school_data[, sum(enrol_boys < enrol_girls)]
```

```
[1] "School with most enrolled boys: 8500"
[1] "School with fewest enrolled girls: 0"
[1] "Number of schools with more enrolled girls than boys: 33517"
```

Perform a **descriptive analysis** of j (sum, maximum, minimum, etc.)

Optionally We can assign a **name to the outcome variable** from the descriptive analysis

```
# special operator N
french_schools_2018 = clean_school_data[country== "FRA" & year == 2018, .N]
```

```
[1] 174
```

Special symbol: **.N** returns the **number of observations** in the current group

Data wrangling tasks (5/6): Descriptive analysis by group

Tidyverse equivalent: **group_by**

```
# using all the operations learned so far:
KAZ <- clean_school_data[country == "KAZ",
        .(mean(enrol_boys), mean(enrol_girls)),
        by = .(year, public_private)]
```

year <fctr>	public_private <fctr>	V1 <dbl>	V2 <dbl>
2009	public	356.1211	357.3263
2009	private	236.3750	355.5000
2012	public	367.0148	352.9951
2012	private	430.4286	493.4286
2018	public	356.6547	332.8354
2018	private	374.8974	375.2051

Perform a **descriptive analysis** by some **grouping variable** specified under *by*

We are not limited to a single group, but can **specify several groups**

```
# conditional grouping
by_condition <- clean_school_data[country == "ALB",
        .(mean(staff_shortage)),
        by = .(fund_donation > 50, total_students < 500)]
```

fund_donation <lgl>	total_students <lgl>	V1 <dbl>
FALSE	FALSE	-0.09020168
FALSE	TRUE	-0.20778860
TRUE	TRUE	-0.39405000
TRUE	FALSE	-0.15485000

by also accepts **conditional expressions**, i.e., create groups around conditions

The outcome is a **matrix** which shows all possible **true/false combinations**

Data wrangling tasks (6/6): Advanced applications

```
# repeated operations with .SD, .SDcols and lapply
mean_columns <- c("enrol_boys", "enrol_girls", "fund_fees")
# specifying the columns to iterate over
selected_means <- clean_school_data[year == 2018,
        lapply(.SD, mean),
        by = .(country),
        .SDcols = mean_columns]
```

country <fctr>	enrol_boys <dbl>	enrol_girls <dbl>	fund_fees <dbl>
ALB	231.8041	200.8763	16.6597938
QAZ	719.4483	610.8448	1.6034483
ARG	264.7633	256.6078	27.6749117
AUS	515.7223	477.7769	21.7454545
BIH	197.5504	177.3876	3.9922481
BRA	384.4560	403.5173	10.7866667
BRN	434.3962	411.6415	31.6792453
BGR	325.1810	283.2845	1.3793103
BLR	289.6752	260.4145	3.6324786
CHL	464.2586	404.6810	32.7672414

To **repeat an operation** across many columns, we use **.SD.,** which returns **a list** for each column, differentiated by any grouping variable

If we want to limit the column-wise computation to selected columns only (e.g., only numerical ones), we specify these through .SDcols

To iterate over each of those lists, we use ***lapply***

Chaining

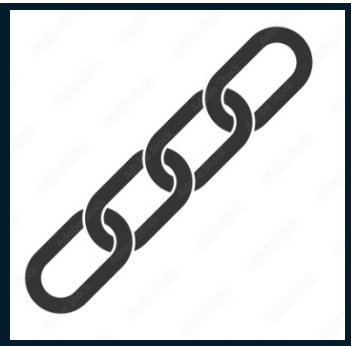
```
DT[...
   ][...
     ][...
       ]

# chaining: ordering by decreasing mean staff shortage by country
chaining <- clean_school_data[,
                              .(staff_shortage = mean(staff_shortage)),
                              by = .(country)
                  ][order(-staff_shortage)
                    ]
```

country <fctr>	staff_shortage <dbl>
KGZ	1.243214719
TUR	1.066562098
MAR	1.026273684
JOR	0.995370562
QCH	0.772223176
QCN	0.684336486
QVE	0.639240476
QRS	0.628721739
LUX	0.620728780
THA	0.543139001

To avoid saving intermediate results, data.table uses **chaining** (pipe-operator in tidyverse)

The syntax can be read as ‘result from **first []-operation** is **used in subsequent []-operation**’

Chaining vertically enhances readability

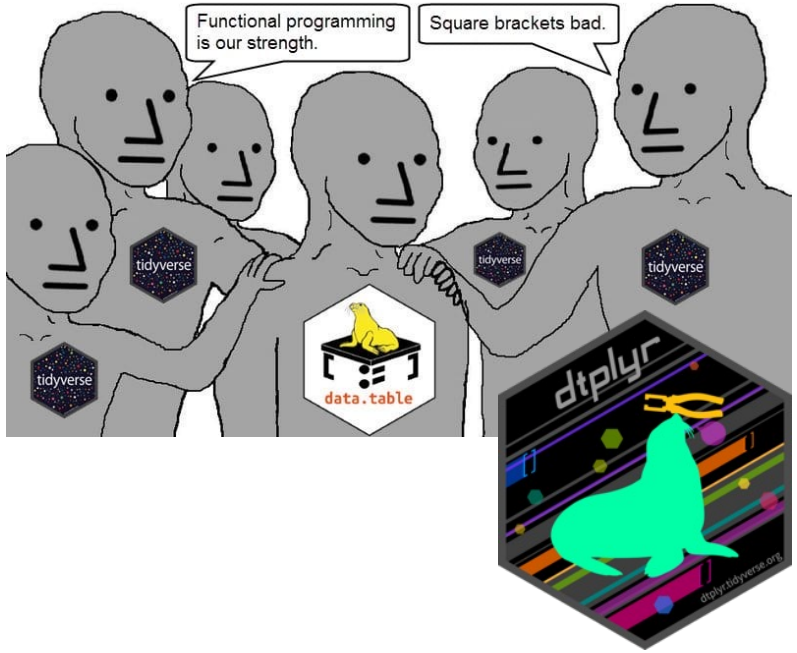


Summary and outlook

Becoming fluent in a new package always **takes time**. Should you invest the time? The **answer is yes**, if you plan to work with **large data sets in R**

dtplyr strives to resolve the performance-usability conflict by using a **data.table backend** but the **dplyr command logic**

Then no data.table after all? Probably, it is **still worth learning data.table** as (i) parts of dtplyr are still being developed, (ii) dtplyr will always be somewhat slower than data.table



Next you get a chance to wrangle some data in our **practical session**, where we will also share **resources** for you to continue your learning journey

Source: https://github.com/tidyverse/dtplyr