

# Big Data in Economics

## Lecture 5: Data cleaning & wrangling: (2) data.table

---

Grant McDermott

University of Oregon | EC 510

# Table of contents

1. Prologue
2. Introduction
3. `data.table` basics
4. Working with rows: `DT[i, ]`
5. Manipulating columns: `DT[, j]`
6. Grouping: `DT[, , by]`
7. Keys
8. Merging datasets
9. Reshaping data
10. `data.table` + tidyverse workflows
11. Summary

# Prologue

---

# Checklist

We'll be using the following packages in today's lecture:

- Already installed: **dplyr**, **ggplot2**, **nycflights13**
- New: **data.table**, **tidyfast**, **dtplyr**, **microbenchmark**

The following code chunk will install (if necessary) and load everything for you.

```
if (!require(pacman)) install.packages('pacman', repos = 'https://cran.rstudio.com')  
pacman::p_load(dplyr, data.table, dtplyr, tidyfast, microbenchmark, ggplot2, nycfl
```

# Introduction

---

# Why learn data.table?

The **tidyverse** is great. As I keep hinting, it will also provide a bridge to many of the big data tools that we'll encounter later in the course (SQL databases, etc.)

So why bother learning another data wrangling package/syntax?

When it comes to **data.table**, I can think of at least five reasons:

1. Concise
2. Insanely fast
3. Memory efficient
4. Feature rich (and stable)
5. Dependency free

Before we get into specifics, here are a few examples to whet your appetite...

# Why learn data.table? (cont.)

## 1) Concise

These two code chunks do the same thing:

```
# library(dplyr) ## Already loaded  
# data(starwars, package = "dplyr") ## Optional to bring the DF into the global en  
starwars %>%  
  filter(species="Human") %>%  
  group_by(homeworld) %>%  
  summarise(mean_height=mean(height))
```

VS

```
# library(data.table) ## Already loaded  
starwars_dt = as.data.table(starwars)  
starwars_dt[species="Human", mean(height), by=homeworld]
```

# Why learn data.table? (cont.)

## 2) Insanely fast

```
collapse_dplyr = function() {
  storms %>%
    group_by(name, year, month, day) %>%
    summarize(wind = mean(wind), pressure = mean(pressure), category = dplyr::first(category))
}

storms_dt = as.data.table(storms)
collapse_dt = function() {
  storms_dt[, .(wind = mean(wind), pressure = mean(pressure), category = first(category)),
    by = .(name, year, month, day)]
}

microbenchmark(collapse_dplyr(), collapse_dt(), times = 10)

## Unit: milliseconds
##           expr           min           lq           mean          median           uq
## collapse_dplyr() 121.53350 125.013755 129.728054 128.35446 133.674008
##   collapse_dt()   1.52316   1.635442   3.380381   1.68409   1.744216
##           max neval cld
## 141.07935    10    b
##  15.02886    10    a
```

**Result:** data.table is 75x faster! (Thanks to [Keith Head](#) for this example.)



# Why learn data.table? (cont.)

## 3) Memory efficient

Measuring and comparing memory use [gets complicated](#). But see [here](#) (esp. from slide 12) for a thorough walkthrough of data.table's memory use and efficiency.

## 4) Features and 5) No dependencies

I'll lump these together, since they really have to do with the stability of your code over time. Just to emphasise the point about [dependencies](#), though:

```
tools::package_dependencies("data.table", recursive = TRUE)[[1]]
```

```
## [1] "methods"
```

```
tools::package_dependencies("dplyr", recursive = TRUE)[[1]]
```

```
## [1] "ellipsis"      "assertthat"  "glue"        "magrittr"    "methods"     "pkgconfig"
## [7] "R6"            "Rcpp"        "rlang"        "tibble"      "tidyselect"  "utils"
## [13] "BH"            "plogr"       "tools"        "cli"         "crayon"      "fansi"
## [19] "lifecycle"    "pillar"      "vctrs"        "purrr"       "grDevices"   "utf8"
## [25] "digest"
```

# Before we continue...

The purpose of this lecture is *not* to convince you that `data.table` is superior to the tidyverse. (Or vice versa.)

For sure, people have strong opinions on the matter and you may find yourself pulling strongly in one direction or the other. And that's okay, but...

My goal is simply to show you another powerful tool that you can use to tackle big data problems efficiently in R.

FWIW, I'm a huge fan of both the tidyverse and `data.table`, and use them about equally in my own work.

- Knowing how to use both of them and how they complement each other has (I believe) made me a much more effective R user/empirical economist/data scientist/etc.

We'll get back to the point about complementarity at the end of the lecture.

# data.table basics

---

# The data.table object

We've already seen that the tidyverse provides its own enhanced version of a data.frame in the form of tibbles.

The same is true for data.table. In fact, data.table functions only work on objects that have been converted to data.tables first.

- Beyond simple visual enhancements (similar to tibbles), the specialised internal structure of data.table objects is a key reason why the package is so fast. (More [here](#) and [here](#).)

To create a data.table, we have a couple of options<sup>1</sup>:

- `data.table(x = 1:10)` creates a new data.table from scratch
- `as.data.table(df)` coerces an existing data frame (here: `df`) to a data.table.
- `setDT(df)` coerces an existing data frame to a data.table *by reference*; i.e. we don't have to (re)assign it.

<sup>1</sup> CSV files imported into R via the amazingly fast `fread()` function are automatically converted to data.table class too. We'll cover `fread()` in the next lecture in data I/O, though.

# What does "modify by reference" mean?

That last bullet leads us to an important concept that underlies much of `data.table`'s awesomeness: It tries, as much as possible, to *modify by reference*.

What does this mean? We don't have time to go into details here, but the very (very) short version is that there are basically two ways of changing or assigning objects in R.

1. **Copy-on-modify:** Creates a copy of your data. Implies extra computational overhead.\*
2. **Modify-in-place:** Avoids creating copies and simply changes the data where it sits in memory.

When we say that `data.table` "modifies by reference", that essentially means it modifies objects in place. This translates to lower memory overhead and faster computation time!

P.S. Further reading if this stuff interests you: (a) [Reference semantics](#) `data.table` vignette, (b) [Names and Values](#) section of *Advanced R* (Hadley Wickham), (c) Nice [blog post](#) by Tyson Barrett that's accessible to beginners.

\* In truth, we need to distinguish between *shallow* and *deep copies*. But that's more than I want you to worry about here.

# data.table syntax

All data.tables accept the same basic syntax:

DT[i, j, by]

On which rows?

What to do?

Grouped by what?

dplyr "equivalents":

- filter(); slice(); arrange()
- select(); mutate()
- group\_by()

While the tidyverse tends to break up operations step-by-step, data.table aims to do everything in one concise expression.

- We can execute complex data wrangling commands as a single, fluid thought.
- Although, as we'll see in a bit, you can certainly chain (pipe) multiple operations together too.

# A Quick Example

We'll dive into the details (and quirks) of `data.table` shortly.

But first, a quick side-by-side comparison example with `dplyr`, since that will help to orientate us for the rest of the lecture. Using our `starwars` dataset, say we want to know:

What is the average height of the human characters by gender?

## dplyr

```
data(starwars, package = "dplyr")
starwars %>%
  filter(species="Human") %>%
  group_by(gender) %>%
  summarise(V1 = mean(height, na.rm=T))
```

## data.table

```
starwars_dt = as.data.table(starwars)
starwars_dt[
  species="Human",
  mean(height, na.rm=T),
  by = gender]
```

# A Quick Example

We'll dive into the details (and quirks) of `data.table` shortly.

But first, a quick side-by-side comparison with `dplyr`, since that will help to orientate us for the rest of the lecture. Using our `starwars` dataset, say we want to know:

What is the average height of the human characters by gender?

## dplyr

```
data(starwars, package = "dplyr")
starwars %>%
  filter(species="Human") %>%
  group_by(gender) %>%
  summarise(V1 = mean(height, na.rm=T))
```

## data.table

```
starwars_dt = as.data.table(starwars)
starwars_dt[
  species="Human", ## i
  mean(height, na.rm=T),
  by = gender]
```



# A Quick Example

We'll dive into the details (and quirks) of `data.table` shortly.

But first, a quick side-by-side comparison with `dplyr`, since that will help to orientate us for the rest of the lecture. Using our `starwars` dataset, say we want to know:

What is the average height of the human characters by gender?

## dplyr

```
data(starwars, package = "dplyr")
starwars %>%
  filter(species="Human") %>%
  group_by(gender) %>%
  summarise(V1 = mean(height, na.rm=T))
```

## data.table

```
starwars_dt = as.data.table(starwars)
starwars_dt[
  species="Human",
  mean(height, na.rm=T), ## j
  by = gender]
```

# A Quick Example

We'll dive into the details (and quirks) of `data.table` shortly.

But first, a quick side-by-side comparison with `dplyr`, since that will help to orientate us for the rest of the lecture. Using our `starwars` dataset, say we want to know:

What is the average height of the human characters by gender?

## dplyr

```
data(starwars, package = "dplyr")
starwars %>%
  filter(species="Human") %>%
  group_by(gender) %>%
  summarise(V1 = mean(height, na.rm=T))
```

## data.table

```
starwars_dt = as.data.table(starwars)
starwars_dt[
  species="Human",
  mean(height, na.rm=T),
  by = gender] ## by
```

# A Quick Example

We'll dive into the details (and quirks) of `data.table` shortly.

But first, a quick side-by-side comparison with `dplyr`, since that will help to orientate us for the rest of the lecture. Using our `starwars` dataset, say we want to know:

What is the average height of the human characters by gender?

## dplyr

```
data(starwars, package = "dplyr")
starwars %>%
  filter(species="Human") %>%
  group_by(gender) %>%
  summarise(V1 = mean(height, na.rm=T))
```

```
## # A tibble: 2 x 2
##   gender      V1
## * <chr>    <dbl>
## 1 feminine  160.
## 2 masculine 182.
```

## data.table

```
starwars_dt = as.data.table(starwars)
starwars_dt[
  species="Human",
  mean(height, na.rm=T),
  by = gender]
```

```
##           gender      V1
## 1: masculine 182.3478
## 2:  feminine 160.2500
```

# Working with rows: DT[i, ]

---

# Subset by rows (filter)

Subsetting by rows is very straightforward in `data.table`. Everything works pretty much the same as you'd expect if you're coming from `dplyr`.

- `DT[x = "string", ]`: Subset to rows where variable `x` equals "string"
- `DT[y > 5, ]`: Subset to rows where variable `y` is greater than 5
- `DT[1:10, ]`: Subset to the first 10 rows

Multiple filters/conditions are fine too:

- `DT[x="string" & y>5, ]`: Subset to rows where `x` is "string" AND `y` is greater than 5

Note that we don't actually need commas when we're only subsetting on `i` (i.e. no `j` or `by` components).

- `DT[x="string"]` is equivalent to `DT[x="string", ]`
- `DT[1:10]` is equivalent to `DT[1:10, ]`
- etc.

# Order by rows (arrange)

```
starwars_dt[order(birth_year)] ## (temporarily) sort by youngest to oldest  
starwars_dt[-order(birth_year)] ## (temporarily) sort by oldest to youngest
```

While ordering as per the above is very straightforward, `data.table` also provides an optimised `setorder()` function for reordering *by reference*.

```
setorder(starwars_dt, birth_year, na.last = TRUE)  
head(starwars_dt[, name:birth_year]) ## Only print subset to stay on the slide
```

```
##           name height mass hair_color skin_color eye_color birth_year  
## 1: Wicket Systri Warrick      88    20      brown      brown      brown        8  
## 2:           IG-88      200   140       none      metal        red       15  
## 3:      Luke Skywalker      172    77      blond      fair       blue       19  
## 4:      Leia Organa      150    49      brown      light      brown       19  
## 5:      Wedge Antilles      170    77      brown      fair      hazel       21  
## 6:           Plo Koon      188    80       none      orange     black       22
```

# Manipulating columns: $DT[, j]$

---

# One slot (j) to rule them all

Recall some of the dplyr verbs that we used to manipulate our variables in different ways:

- `select()`
- `mutate()`
- `summarise()`
- `count()`

`data.table` recognizes that all of these verbs are just different versions of telling R...

*"Do something to this variable in my dataset"*

... and it let's you do all of those operations in one place: the `j` slot.

However, this concision requires a few syntax tweaks w.r.t. how we change and assign variables in our dataset.

- Some people find this off-putting (or, at least, weird) when they first come to `data.table`.
- I hope to convince you that these syntax tweaks aren't actually that difficult to grok and give us a *lot* of power in return.



# Modifying columns :=

To add, delete, or change columns in `data.table`, we use the `:=` operator.

- Known as the *walrus* operator (geddit??)

For example,

- `DT[, xsq := x^2]`: Create a new column (`xsq`) from an existing one (`x`)
- `DT[, x := as.character(x)]`: Change an existing column

**Important:** `:=` is *modifying by reference*, i.e. in place. So we don't have to (re)assign the object to save these changes.

However, we also won't see these changes printed to screen unless we ask R explicitly.

```
DT = data.table(x = 1:2)
# DT[, xsq := x^2] ## Modifies in place but doesn't print the result
DT[, x_ssq := x^2][] ## Adding [] prints the result.
```

```
##      x x_ssq
## 1:  1      1
## 2:  2      4
```

# Modifying columns := (cont.)

## Sub-assign by reference

One really cool implication of `:=` is data.table's **sub-assign by reference** functionality. As a simple example, consider another fake dataset.

```
DT2 = data.table(a = -2:2, b = LETTERS[1:5])
```

Now, imagine we want to locate all rows where "a" is negative and replace the corresponding "b" cell with NA.

- In dplyr you'd have to do something like `...mutate(b = ifelse(a < 0, NA, b))`.
- In data.table, simply specify which rows to target (`i`) and then sub-assign (`j`) directly.

```
DT2[a < 0, b := NA][] ## Again, just adding the second [] to print to screen
```

```
##      a      b
## 1: -2 <NA>
## 2: -1 <NA>
## 3:  0      C
## 4:  1      D
## 5:  2      E
```

# Modifying columns := (cont.)

To modify multiple columns simultaneously, we have two options.

1. LHS := RHS form: `DT[, c("var1", "var2") := .(val1, val2)]`

2. Functional form: `DT[, ':= ' (var1=val1, var2=val2)]`

Personally, I *much* prefer the functional form and so that's what I'll use going forward. E.g.

```
DT[, ':= ' (y = 3:4, y_name = c("three", "four"))]  
DT ## Another way to print the results instead of appending []
```

```
##      x x_sq y y_name  
## 1: 1      1 3  three  
## 2: 2      4 4   four
```

Note, however, that dynamically assigning dependent columns in a single step (like we did with `dplyr::mutate`) doesn't work.

```
DT[, ':= ' (z = 5:6, z_sq = z^2)][[]]
```

```
## Error in eval(jsub, SEnv, parent.frame()): object 'z' not found
```

# Aside: Chaining data.table operations

That last example provides as good a time as any to mention that you can chain multiple data.table operations together.

The native data.table way is simply to append consecutive `[]` terms.

```
DT[, z := 5:6][, z_sq := z^2][]
```

```
##      x x_sq y y_name z z_sq
## 1:  1     1 3  three 5    25
## 2:  2     4 4   four 6    36
```

But if you prefer the **magrittr** pipe, then that's also possible. Just prefix each step with `.:`

```
# library(magrittr) ## Not needed since we've already loaded %>% via dplyr
DT %>%
  .[, xyz := x+y+z] %>%
  .[, xyz_sq := xyz^2] %>%
  .[]
```

```
##      x x_sq y y_name z z_sq xyz xyz_sq
## 1:  1     1 3  three 5    25   9     81
## 2:  2     4 4   four 6    36  12    144
```

# Modifying columns := (cont.)

To remove a column from your dataset, set it to NULL.

```
DT[, y_name := NULL]
```

```
DT
```

```
##      x x_sq y z z_sq xyz xyz_sq
## 1:  1    1 3 5   25    9     81
## 2:  2    4 4 6   36   12    144
```

# Subsetting on columns (select)

We can also use the `j` slot to subset our data on columns. I'll return to the starwars dataset for these examples...

Subset by column position:

```
starwars_dt[, c(1:3, 10)] %>% head(2)
```

```
##           name height mass homeworld
## 1: Luke Skywalker   172   77  Tatooine
## 2:           C-3PO   167   75  Tatooine
```

Or by name:

```
# starwars_dt[, c("name", "height", "mass", "homeworld")] ## Also works
# starwars_dt[, list(name, height, mass, homeworld)] ## So does this
starwars_dt[, .(name, height, mass, homeworld)] %>% head(2)
```

```
##           name height mass homeworld
## 1: Luke Skywalker   172   77  Tatooine
## 2:           C-3PO   167   75  Tatooine
```

# Aside: What's with the `.()`?

We've now seen `.()` in a couple places, e.g the previous slide and [this slide](#) from earlier if you were paying close attention.

- `.()` is just a `data.table` shortcut for `list()`.

We'll be using `.()` quite liberally once we start working subsetting and/or grouping by multiple variables at a time.

You can think of it as one of `data.table`'s syntactical quirks. But, really, it's just there to give you more options. You can often — if not always — use these three forms interchangeably in `data.table`:

- `.(var1, var2, ... )`
- `list(var1, var2, ... )`
- `c("var1", "var2", ... )`

I like the `.()` syntax best — less typing! — but each to their own.

Okay, back to subsetting on columns...

# Subsetting on columns (cont.)

You can also exclude columns through negation. Try this next code chunk yourself:

```
starwars_dt[, !c("name", "height")]
```

And, you can rename a column as follows (again, run this yourself):

```
setnames[starwars_dt, old = "name", new = "alias"][]  
## Safer to change it back, just in case we use "name" on a later slide again  
setnames[starwars_dt, old = "alias", new = "name"]
```

## General comment

While it offers some important performance gains, I have to admit that I find data.table's column subsetting approach less user-friendly than dplyr's `select()` and `rename()`... On the plus, side, you can still use dplyr verbs on data.tables.

```
starwars_dt %>% select(homeworld, everything()) ## Try this yourself
```

I don't want to preempt myself, though. I'll get back to dplyr+data.table functionality at the end of the lecture....



# Aggregating

Finally, we can do aggregating manipulations in `j`.

```
starwars_dt[, mean(height, na.rm=T)]
```

```
## [1] 174.358
```

Note that we don't keep anything unless we assign the result to a new object. If you want to add the new aggregated column to your original dataset, use `:=`.

```
starwars_dt[, mean_height := mean(height, na.rm=T)] %>% ## Add mean height as a co  
  .[, .(name, height, mean_height)] %>% ## Just display a few columns on the slide  
  head(5)
```

```
##           name height mean_height  
## 1: Luke Skywalker   172      174.358  
## 2:           C-3PO   167      174.358  
## 3:           R2-D2    96      174.358  
## 4:   Darth Vader   202      174.358  
## 5:   Leia Organa   150      174.358
```

# Aggregating (cont.)

`data.table` also provides specialised convenience **symbols** for common aggregation tasks in `j`.

For example, we can quickly count the number of observations using `.N`.

```
starwars_dt[, .N]
```

```
## [1] 87
```

Of course, this is a pretty silly example since it's just going to give us the total number of rows in the dataset. Like most forms of aggregation, `.N` is much more interesting when it is applied by group.

- This provides a nice segue to our next section...

Group by: DT[, , by]

---

# by

data.table's `by` argument functions very similarly to the `dplyr::group_by` equivalent. Try these next few examples in your own R console:

- `starwars_dt[, mean(height, na.rm=T), by = species]`: Collapse by single variable
- `starwars_dt[, .(species_height = mean(height, na.rm=T)), by = species]`: As above, but explicitly name the summary variable
- `starwars_dt[, mean(mass, na.rm=T), by = height>190]`: Conditionals work too.
- `starwars_dt[, species_n := .N, by = species][[]]`: Add an aggregated column to the data (here: number of observations by species group)

To perform aggregations by multiple variables, we'll use the `.()` syntax again.

```
starwars_dt[, .(mean_height = mean(height, na.rm=T)), by = .(species, homeworld)]  
head(4)
```

```
##      species homeworld mean_height  
## 1:   Human   Tatooine    179.2500  
## 2:   Droid   Tatooine    132.0000  
## 3:   Droid     Naboo     96.0000  
## 4:   Human  Alderaan    176.3333
```

# Efficient subsetting with .SD

We've seen how to group by multiple variables. But what if we want to *summarise* multiple variables, regardless of how we are grouping?

One solution is to again use `.()` and write everything out, e.g.

```
## Run yourself if you'd like to check the output
starwars_dt[,
  .(mean(height, na.rm=T), mean(mass, na.rm=T), mean(birth_year, na.rm=T)
  by = species]
```

But this soon become tedious. Imagine we have even more variables. Do we really have to write out `mean( ... , na.rm=T)` for each one?

Fortunately, the answer is "no". `data.table` provides a special `.SD` symbol for **s**ubsetting **d**ata. In truth, `.SD` can do a **lot more** than what I'm about to show you, but here's how it would work in the present case...

*See next slide.*

# Efficient subsetting with .SD (cont.)

```
starwars_dt[,  
  lapply(.SD, mean, na.rm=T),  
  by = species,  
  .SDcols = c("height", "mass", "birth_year")] %>%  
  head(2) ## Just keep everything on the slide
```

```
##   species  height    mass birth_year  
## 1:   Human 176.6452 82.78182   53.41200  
## 2:   Droid 131.2000 69.75000   53.33333
```

First, we specify what we want to *do* on our data subset (i.e. `.SD`). In this case, we want the mean for each element, which we obtain by iterating over with the base `lapply()` function.<sup>1</sup>

Then, we specify *which columns* to subset with the `.SDcols` argument.

P.S. One annoyance I have is that the `.()` syntax doesn't work for `.SDcols`. However, we can at least feed it consecutive columns without quotes, e.g. `.SDcols = height:mass`. See

[here](#) We'll learn more about iteration once we get to the programming section of the course.

# Efficient subsetting with .SD (cont.)

Just to add: We need only specify `.SDcols` if we want to subset specific parts of the data.

If we instead want to apply the same function on *all* the variables in our dataset, then `.SD` by itself will suffice.

As a quick example, recall our earlier DT object that contains only numeric variables.

```
DT
```

```
##      x x_sq y z z_sq xyz xyz_sq
## 1:  1    1 3 5   25   9     81
## 2:  2    4 4 6   36  12    144
```

We can obtain the mean for each variable as follows.

```
DT[, lapply(.SD, mean)]
```

```
##      x x_sq    y    z z_sq  xyz xyz_sq
## 1: 1.5  2.5 3.5 5.5 30.5 10.5  112.5
```

# keyby

The last thing I want to mention w.r.t. `by` is its close relative: `keyby`.

The `keyby` argument works exactly like `by` — you can use it as a drop-in replacement — except that it orders the results and creates a **key**.

- Setting a key for a `data.table` will allow for various (and often astonishing) performance gains<sup>1</sup>

Keys are important enough that I want to save them for their own section, though...

<sup>1</sup> Note that you won't see an immediate performance gain with `keyby`, but subsequent operations will certainly benefit. (Of course, you can get an immediate boost by setting the key ahead of time, but I'll explain all that on the next slide...)



# Keys

---

# What are keys?

Keys are a way of ordering the data that allows for *extremely* fast subsetting.

The data.table [vignette](#) describes them as "supercharged rownames". I know that might sound a bit abstract, but here's the idea in a nutshell...

Imagine that we want to filter a dataset based on a particular value (e.g. find all the human characters in our starwars dataset).

- Normally, we'd have to search through the whole dataset to identify matching cases.
- But, if we've set an appropriate key, then the data are already ordered in such a way that we (i.e. our computer) only has to search through a much smaller subset.

**Analogy:** Think of the way a filing cabinet might divide items by alphabetical order: Files starting "ABC" in the top drawer, "DEF" in the second drawer, etc. To find *Alice's* file, you'd only have to search the top draw. For *Fred*, the second draw, and so on.

Not only is this much quicker, but the same idea also carries over to *all other* forms of data manipulation that rely on subsetting (aggregation by group, joins, etc.)

P.S. We'll get there later in the course, but keys are also the secret sauce in databases.

# How do I set a key?

You can set a key when you first create a data.table. E.g.

- `DT = data.table(x = 1:10, y = LETTERS[1:10], key = "x")`
- `DT = as.data.table(DF, key = "x")`
- `setDT(DF, key = "x")`

Or, you can set keys on an existing data.table with the `setkey()` function.

- `setkey(DT, x)`: Note that the key doesn't have to be quoted this time

**Important:** Since keys just describe a particular ordering of the data, you can set a key on *multiple* columns. (More [here](#).) E.g.

- `DT = as.data.table(DF, key = c("x", "y"))`
- `setkey(DT, x, y)`: Again, no quotes needed

P.S. Use the `key()` function to see what keys are currently set for your data.table. You can only ever have one key per table at a time, but it's very easy to change them using one of the above commands.

# Example

Recall the **speed benchmark** that we saw at the very beginning of the lecture: `data.table` ended up being 75x faster than `dplyr` for a fairly standard summarising task.

Let's redo the benchmark, but this time include a version where we pre-assign a key. For optimal performance, the key should match the same variables that we're grouping/subsetting on.

- Again, a key can be set on multiple variables, although the lead grouping variable (in the below case: "name") is the most important.

```
## First create a keyed version of the storms data.table.
## Note that key variables match the 'by' grouping variables below.
storms_dt_key = as.data.table(storms, key = c("name", "year", "month", "day"))
## Collapse function for this keyed data.table. Everything else stays the same.
collapse_dt_key = function() {
  storms_dt_key[, .(wind = mean(wind), pressure = mean(pressure), category = first(category
    by = .(name, year, month, day))]
}
## Run the benchmark on all three functions.
microbenchmark(collapse_dplyr(), collapse_dt(), collapse_dt_key(), times = 10)
```

See next slide for results

# Example (cont.)

```
## Unit: microseconds
##           expr           min           lq           mean           median           uq
## collapse_dplyr() 114507.903 118471.268 126757.730 121386.061 129442.557
##   collapse_dt()   1510.639   1605.432   2050.183   1719.483   1906.047
## collapse_dt_key()   846.293    881.082   1252.658   1047.224   1079.106
##           max neval cld
## 168398.229     10    b
##   5050.027     10    a
##   3324.596     10    a
```

The keyed data.table version is now **144x** (!!!) faster than dplyr.

- That thing you feel... is your face melting.

It's not just this toy example. In working with real-life data, my experience is that setting keys almost always leads to huge speed-ups... and those gains tend to scale as the datasets increase in size.

**Bottom line:** data.table is already plenty fast. But use keys if you're really serious about performance.

# Merging datasets

---

# Merge (aka join) options

`data.table` provides two ways to merge datasets.

- `DT1[DT2, on = "id"]`
- `merge(DT1, DT2, by = "id")`

I prefer the latter because it offers extra functionality (see `?merge.data.table`), but each to their own.<sup>1</sup>

I'm going to keep things brief by simply showing you how to repeat the same left join that we **practiced** with `dplyr` in the last lecture, using data from the **nycflights13** package.

```
# library(nycflights13) ## Already loaded
flights_dt = as.data.table(flights)
planes_dt = as.data.table(planes)
```

<sup>1</sup> For a really good summary of the different join options (left, right, full, anti, etc.) using these two methods, as well as their `dplyr` equivalents, see [here](#).

# Left join example (cont.)

Here's a comparison with the dplyr equivalents from last week. I'll let you run and compare these yourself. (Note that the row orders will be different.)

## dplyr

```
left_join(  
  flights,  
  planes,  
  by = "tailnum"  
)
```

## data.table

```
merge(  
  flights_dt,  
  planes_dt,  
  all.x = TRUE, ## omit for inner join  
  by = "tailnum")
```

If you run these, you'll see that both methods handle the ambiguous "year" columns by creating "year.x" and "year.y" variants. We avoided this in dplyr by using `rename()`. How might you avoid the same thing in data.table? **Possible answer:** Use `setnames()`.

```
merge(  
  setnames(flights_dt, old = "year", new = "year_built"),  
  planes_dt,  
  all.x = TRUE,  
  by = "tailnum")
```



# Use keys for lightning fast joins

The only other thing I'll point out is that setting **keys** can lead to dramatic speed-ups for merging data.tables. I'll demonstrate using an inner join this time.

```
merge_dt = function() merge(flights_dt, planes_dt, by = "tailnum")

flights_dt_key = as.data.table(flights, key = "tailnum")
planes_dt_key = as.data.table(planes, key = "tailnum")
merge_dt_key = function() merge(flights_dt_key, planes_dt_key, by = "tailnum")

microbenchmark(merge_dt(), merge_dt_key(), times = 10)
```

```
## Unit: milliseconds
```

```
##           expr      min       lq      mean   median       uq      max  neval
##   merge_dt() 41.21015 45.57901 99.65385 48.08860 151.66506 363.86659    10
## merge_dt_key() 27.80020 28.33882 31.09487 29.46449  35.01835  37.23726    10
##   cld
##     a
##     a
```

Okay, not *such* a dramatic speed-up in this simple case, but trust me: For really big datasets with complicated joins (e.g. multiple columns), setting keys will make a huge difference. (FWIW, the same is true for dplyr. See [here](#).)

# Reshaping data

---

# Reshaping options with data.table

In the tidyverse lecture, we saw how to reshape data using the `tidyr::pivot*` functions.

`data.table` offers its own functions for flexibly reshaping data:

- `dcast()`: convert wide data to long data
- `melt()`: convert long data to wide data

However, I also want to flag the **tidyfast** package by Tyson Barrett, which implements `data.table` versions of the `tidyr::pivot*` functions (among other things).

- `tidyfast::dt_pivot_longer()`: wide to long
- `tidyfast::dt_pivot_wider()`: long to wide

I'll demonstrate reshaping with both options on the same fake "stocks" data that we created last time:

```
stocks = data.table(time = as.Date('2009-01-01') + 0:1,  
                    X = rnorm(2, 0, 1),  
                    Y = rnorm(2, 0, 2),  
                    Z = rnorm(2, 0, 4))
```

# Reshape from wide to long

Our data are currently in wide format.

```
stocks
```

```
##           time           X           Y           Z
## 1: 2009-01-01  0.3522752  4.068978  7.3494008
## 2: 2009-01-02 -0.3102111 -1.763021  0.7670371
```

To convert this into long format, we could do either of the following:

```
# See ?melt.data.table for options
melt(stocks, id.vars = "time")
```

```
##           time variable      value
## 1: 2009-01-01      X  0.3522752
## 2: 2009-01-02      X -0.3102111
## 3: 2009-01-01      Y  4.0689778
## 4: 2009-01-02      Y -1.7630212
## 5: 2009-01-01      Z  7.3494008
## 6: 2009-01-02      Z  0.7670371
```

```
stocks %>%
  dt_pivot_longer(X:Z, names_to="stock", values_to="price")
```

```
##           time stock      price
## 1: 2009-01-01      X  0.3522752
## 2: 2009-01-02      X -0.3102111
## 3: 2009-01-01      Y  4.0689778
## 4: 2009-01-02      Y -1.7630212
## 5: 2009-01-01      Z  7.3494008
## 6: 2009-01-02      Z  0.7670371
```

# Reshape from long to wide

Let's quickly save the long-format stocks data.table. I'll use the `melt()` approach and also throw in some extra column-naming options, just so you can see those in action.

```
stocks_long = melt(stocks, id.vars = "time",  
                   variable.name = "stock", value.name = "price")
```

```
stocks_long
```

```
##           time stock      price  
## 1: 2009-01-01      X 0.3522752  
## 2: 2009-01-02      X -0.3102111  
## 3: 2009-01-01      Y 4.0689778  
## 4: 2009-01-02      Y -1.7630212  
## 5: 2009-01-01      Z 7.3494008  
## 6: 2009-01-02      Z 0.7670371
```

```
dcast(stocks_long,  
      time ~ stock,  
      value.var = "price")
```

```
stocks_long %>%  
  dt_pivot_wider(names_from=stock,  
                values_from=price)
```

```
##           time      X      Y      Z ##           time      X      Y      Z  
## 1: 2009-01-01 0.3522752 4.068978 7.3494008 ## 1: 2009-01-01 0.3522752 4.068978 7.3494008  
## 2: 2009-01-02 -0.3102111 -1.763021 0.7670371 ## 2: 2009-01-02 -0.3102111 -1.763021 0.7670371
```

# data.table + tidyverse workflows

---

# Choosing a workflow that works for you

When it comes to data work in R, we truly are spoilt for choice.

We have two incredible data wrangling ecosystems to choose from.

- **tidyverse** (esp. **dplyr** and **tidyr**)
- **data.table**

Over the last two lectures, we've explored some of the key features of each. It's only natural that people might find themselves gravitating to one or the other.

- Some people love the expressiveness and modularity of the tidyverse.
- Others love the concision and power of data.table.

And that's cool. But I'll repeat a point I made earlier: I use both ecosystems about equally in my own work and honestly believe that this has been to my great benefit.

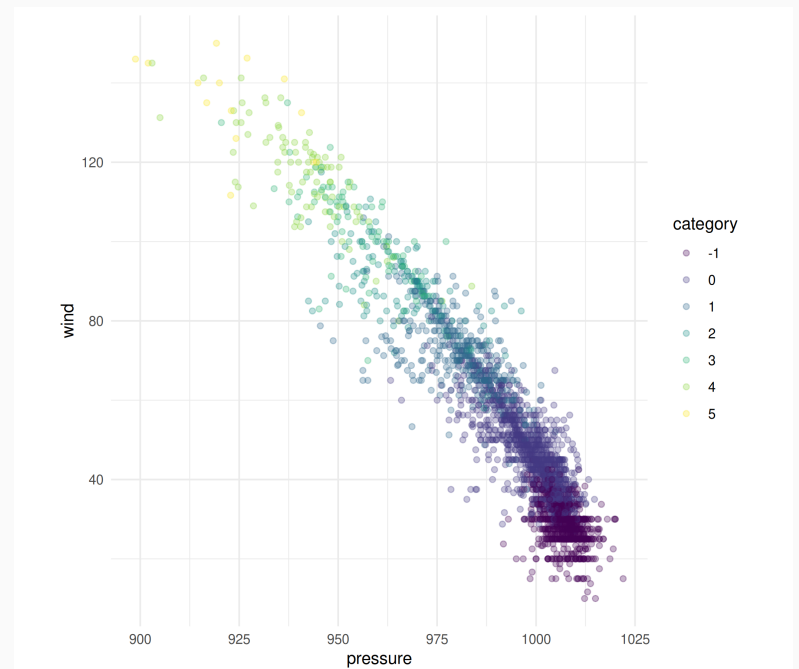
These next few slides offer a few additional thoughts on how data.table and the tidyverse can be combined profitably in your own workflow.

# Pick and choose

The first point is perhaps the most obvious one: The tidyverse consists of *multiple* packages. Just because you prefer to use `data.table` instead of `dplyr+tidyr` for your data wrangling needs, doesn't mean that other tidyverse packages are off limits too.

Case in point: Almost every hardcore `data.table` user I know is a hardcore **ggplot2** user too.

```
## library(ggplot2) # already loaded
storms_dt[, .(wind = mean(wind),
              pressure = mean(pressure),
              category = first(category)),
           by = .(name, year, month, day)] %>%
  ggplot(aes(x = pressure, y = wind, col=category)) +
  geom_point(alpha = 0.3) +
  theme_minimal()
```





# Don't be a fanatic

Closely related to the second first point: Don't try to shoehorn every problem into a tidyverse or data.table framework.<sup>1</sup>

Having worked extensively with both packages, I think it fair to say that there are things the tidyverse (dplyr+tidyr) does better, and there are things that data.table does better.

- If you find a great solution on Stackoverflow that uses the "other" package... use it!
- Don't be a fanatic. Insisting on ecosystem purity is rarely worth it.

Plus, as I hinted earlier, you *can* use tidyverse verbs on data.tables. Try yourself:

```
starwars_dt %>% group_by(homeworld) %>% summarise(height = mean(height, na.rm=T))
```

This does incur a performance penalty. But luckily there's an ever better solution...

<sup>1</sup> Remember my admonition from last time: "A combination of tidyverse and base R is often the best solution to a problem." Add data.table to that list.

# dtplyr

Do you love dplyr's syntax, but want data.table's performance? Well, you're in luck!

Hadley Wickham's **dtplyr** package provides a data.table "back-end" for dplyr.

- Basically, write your code as if you were using dplyr and then it gets automatically translated to (and evaluated as) data.table code.

If this sounds appealing to you (and it should) I strongly encourage you to check out the [package website](#) for details. But here's quick example, using our benchmark from earlier.

```
# library(dtplyr) ## Already loaded
storms_dtplyr = lazy_dt(storms) ## dtplyr requires objects to be set as "lazy" data.tables
collapse_dtplyr = function() {
  storms_dtplyr %>%
    group_by(name, year, month, day) %>%
    summarize(wind = mean(wind), pressure = mean(pressure), category = first(category)) %>%
    as_tibble()
}
## Just compare dtplyr with normal dplyr and data.table versions (i.e. no keys)
microbenchmark::microbenchmark(collapse_dplyr(), collapse_dt(), collapse_dtplyr(), times = 10)
```

*See next slide for results*

# dtplyr (cont.)

```
## Unit: milliseconds
##           expr           min           lq           mean           median           uq
## collapse_dplyr() 112.908046 115.827278 120.153585 117.788253 123.898402
## collapse_dt()    1.566148    1.601430    2.166914    1.758606    2.068824
## collapse_dtplyr() 2.725430    2.885401    3.900646    3.225255    4.656426
##           max neval cld
## 136.205525    10    b
##   5.569121    10    a
##   8.081461    10    a
```

Not quite as fast as the native `data.table` method, but a 36x speed-up for free? I'd take it!

*Aside:* `dtplyr` automatically prints its `data.table` translation to screen. This can be super helpful when you first come over to `data.table` from the tidyverse.

```
lazy_dt(starwars) %>% filter(species="humans") %>% group_by(gender) %>% summarise

## Source: local data table [?? x 2]
## Call:   _DT1[species = "humans"][, .(height = mean(height, na.rm = ..T)),
##        keyby = .(gender)]
##
## # ... with 2 variables: gender <chr>, height <dbl>
##
```

# Summary

---

# Summary

`data.table` is a powerful data wrangling package that combines concise syntax with incredible performance. It is also very lightweight, despite being feature rich.

The basic syntax is `DT[i, j, by]`

- `i` On which rows?
- `j` What to do?
- `by` Grouped by what?

`data.table` also (re)introduces some new ideas like modify by reference (e.g. `:=`), as well as syntax (e.g. `.`, `.SDcols`).

- All of these ideas support `data.table`'s core goals: Maximise performance and flexibility, whilst maintaining a concise and consistent syntax.

Pro tip: Use keys to order your data and yield dramatic speed-ups.

The tidyverse and `data.table` are often viewed as substitutes, but you can profitably combine both into your workflow... even if you favour one for (most of) your data wrangling needs.

# Further resources

As hard as it may be to believe, there's a ton of `data.table` features that we didn't cover today. Some, we'll get to in later lectures (e.g. the insanely fast `fread()` and `fwrite()` CSV I/O functions). Others, we won't, but hopefully I've given you enough of a grounding to continue exploring on your own.

Here are some recommended further resources:

- <http://r-datatable.com> (Official website. See the vignettes, especially.)
- <https://github.com/Rdatatable/data.table#cheatsheets> (Cheatsheet.)
- <https://atrebas.github.io/post/2019-03-03-datatable-dplyr> (Really nice, side-by-side comparisons of common `data.table` and `dplyr` operations.)

And related packages:

- <https://tysonbarrett.com/tidyfast>
- <https://dtplyr.tidyverse.org>

P.S. Your next assignment is up: Impress me with your data wrangling skills using either the tidyverse or `data.table` (or both!)

# Next lecture: Big data I/O

---