Do you program in R and normally use DPLYR for data wrangling, manipulation or whatever term you call it? Have you heard all the hype about data.table and how this package can significantly improve the performance run time of your R scripts? Have you been meaning to get round to learning data.table and have never managed it? The answer to these questions is that you don't have to. This is exactly the purpose of the DTPLYR package.

Loading in DTPLYR

To load the package we are going to bring in some of the libraries we will need to work with:

```
#install.packages("dtplyr")
library(data.table)
library(dtplyr)
library(dplyr, warn.conflicts = FALSE)
library(NHSRdatasets)
```

Here I bring in the dtplyr package alongside data.table, dplyr and NHSRdatasets. The major benefit of data.table over dplyr is that it loads larger datasets much quicker.

Generating a large dataset to work with

The next step is to use the stranded_data data from the NHSRdatasets package. This is a set of patients who have been in hospital, as an inpatient, longer than 7 days.

The following lines load in the data and then create duplicates of the data frame. I have added a system.time() wrapper here to see how long the code takes to run:

```
#Save the data to a data frame called df
df <- NHSRdatasets::stranded_data

#Create another 100,500 copies of the original data
system.time(
  df <- df[rep(seq_len(nrow(df)), each = 100500), ] ##TAKES A WHILE TO
LOAD
)
```

Create a lazy data table

Now, we are going to use dtplyr to create a lazy data table. It is lazy, because you don't need to know anything about the data.table package to convert it to this type, which under the hood is essentially a data.frame class.

```
# Create a lazy data table
strand_dt_lazy <- lazy_dt(df)
```

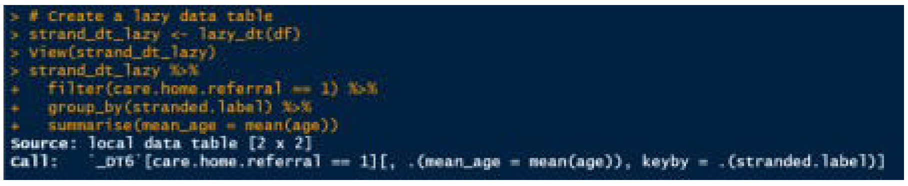
DPLYR filtering on data.table object

The next step is to do some custom filtering on our new lazy data table object:

```
strand_dt_lazy %>%
  filter(care.home.referral == 1) %>%
  group_by(stranded.label) %>%
  summarise(mean_age = mean(age))
```

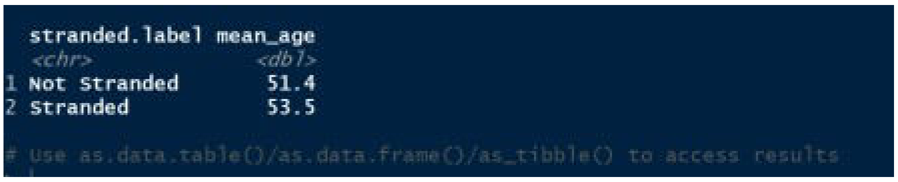
Here two things happen. You get:

1. The converted data.table code in the first print to the console:



```
> # Create a lazy data table
> strand_dt_lazy <- lazy_dt(df)
> View(strand_dt_lazy)
> strand_dt_lazy %>%
+   filter(care.home.referral == 1) %>%
+   group_by(stranded.label) %>%
+   summarise(mean_age = mean(age))
Source: local data table [2 x 2]
Call:   `_DT6`[care.home.referral == 1][, .(mean_age = mean(age)), keyby = .(stranded.label)]
```

2. The results of the filtering operation:



```
  stranded.label mean_age
  <chr>             <dbl>
1 Not Stranded       51.4
2 Stranded           53.5

# Use as.data.table()/as.data.frame()/as_tibble() to access results
```

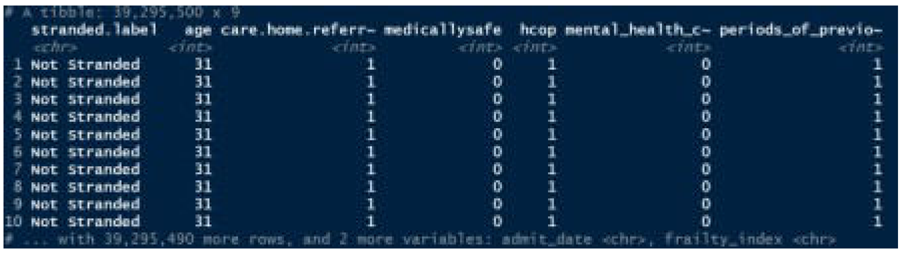
At the bottom it says use the as.data.table, as.data.frame or as_tibble commands to store these results. This is what I will do in the next step, but on a subset of the whole data.

Access our dtplyr results in a tibble

As indicated in the last print, we now need to access the results as a tibble. The code below is how to use this, essentially at the end of your filter, just add another pipe and make sure you cast it to the relevant data frame object i.e. tibble, data.frame or data.table:

```
# To convert to tibble, use as_tibble at the end of the code
strand_dt_lazy %>%
  filter(care.home.referral == 1) %>%
  group_by(stranded.label) %>%
  as_tibble()
```

The results are outputted in super lightning speed:



```
# A tibble: 39,295,500 x 9
   stranded.label   age care.home.referr~ medicallysafe  hcop mental_health_c~ periods_of_previo~
   <chr>          <int>             <int>         <int> <int>            <int>              <int>
 1 Not Stranded      31                 1             0     1                0                  1
 2 Not Stranded      31                 1             0     1                0                  1
 3 Not Stranded      31                 1             0     1                0                  1
 4 Not Stranded      31                 1             0     1                0                  1
 5 Not Stranded      31                 1             0     1                0                  1
 6 Not Stranded      31                 1             0     1                0                  1
 7 Not Stranded      31                 1             0     1                0                  1
 8 Not Stranded      31                 1             0     1                0                  1
 9 Not Stranded      31                 1             0     1                0                  1
10 Not Stranded      31                 1             0     1                0                  1
# ... with 39,295,490 more rows, and 2 more variables: admit_date <chr>, frailty_index <chr>
```

Wrapping up

This was just a short blog to highlight the usefulness of dtplyr and show how it can be used to convert dplyr code to data.table syntax, and also how it builds data.table syntax behind the scenes.