Post01: Dplyr vs. Data.Table in Data Manipulation

Yuechan Huang

October 25, 2017

Introduction

Despite the fact that this class is my first time learning to use command-based software and analyze data, I am inspired by the powerful package **dplyr**, and thus, I decide to perform some tasks in manipulating data using **dplyr** along with another powerful package **data.table** in this post. For more comparision on **dplyr** and **data.table**, you can read it from Stack Overflow and Quora.

dplyr

According to http://dplyr.tidyverse.org/, **dplyr** is a grammar of data manipulation, providing a consistent set of verbs to solve the most common data manipulation tasks.

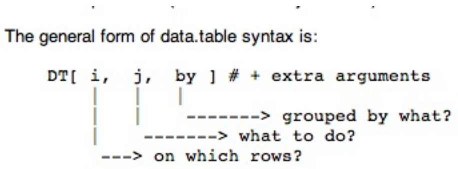
- **select( )**: picks variables based on their names.
- **filter( )**: picks cases based on their values.
- **arrange( )**: changes the ordering of the rows (assending or descending).
- **mutate( )**: adds new variables that aer functions of existing variables.
- **summarise( )**: reduces multiple values down to a single summary.

In addition to these, we learned 2 more verbs in class:

- **slice( )**: selects rows by position.
- **group_by( )**: groups (aggregate) operations.

data.table

According to https://cran.r-project.org/web/packages/data.table/vignettes/datatable-intro.html, the general form of **data.table** is DT[*i*, *j*, *by*] interpreted as: take *DT*, subset rows using *i*, then calculate *j*, grouped by *by* as the picture shown below.



The general form of data.table syntax is:

```
DT[ i,  j,  by ] # + extra arguments
    |   |   |
    |   |    -------> grouped by what?
    |    -------> what to do?
     ---> on which rows?
```

Data Manipulation

```r
# Loading Packages
library(readr)
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```r
library(data.table)
```

```
##
## Attaching package: 'data.table'
```

```
## The following objects are masked from 'package:dplyr':
##
##     between, first, last
```

```r
library(ggplot2)
library(compare)
```

```
##
## Attaching package: 'compare'
```

```
## The following object is masked from 'package:base':
##
##     isTRUE
```

Data Preparation

In this post, I would like to use the data of NBA players from the class.

```
# importing data
dat <- read.csv('nba2017-players.csv', stringsAsFactors = FALSE)
```

Note that **data.table** only work with data.table format, so we need to turn this data into a data.table using data.table() or as.data.table() functions.

```
dt <- data.table(dat)
```

Select Columns/Rows

To select columns, we will use the **select** function in **dplyr**. On the other hand, we can specify the column names using **data.table**.

Select One Variable

Let's select the "height" variable from the data.

```
in_dplyr <- select(dat, height)
in_data.table <- dt[ , height]
```

Let's compare if the results from **dplyr** and **data.table** are the same.

```
compare(in_dplyr, in_data.table, allowAll = TRUE )
```

```
## TRUE
##   coerced from <data.frame> to <data frame>
##   renamed
##   dropped names
```

Based on the answer from the compare() function, we prove that results using **dplyr** and **data.table** are the same.

Select Multiple Variables

```
in_dplyr1 <- select(dat, height, weight, age)
in_data.table1 <- dt[ , .(height, weight, age)]
compare(in_dplyr1, in_data.table1, allowAll = TRUE )
```

```
## TRUE
##   dropped attributes
```

Remove One Variable

Here, I will show you how to remove variables.

```
in_dplyr2 <- select(dat, -height)
in_data.table2 <- dt[ , !'height', with = FALSE]
compare(in_dplyr2, in_data.table2, allowAll = TRUE )
```

```
## TRUE
##   dropped attributes
```

Remove Multiple Variables

```
in_dplyr3 <- select(dat, -c(height, weight, age))
in_data.table3 <- dt[ , !c("height", "weight", "age"), with = FALSE]
compare(in_dplyr3, in_data.table3, allowAll = TRUE )
```

```
## TRUE
##   dropped attributes
```

Again, based on the answer, results using both **dplyr** and **data.table** are the same.

Select Rows

Besides selecting columns, we can also select rows.

```
in_dplyr4 <- slice(dat, 1:5)
in_data.table4 <- dt[1:5, ]
compare(in_dplyr4, in_data.table4, allowAll = TRUE )
```

```
## TRUE
##   dropped attributes
```

Remove Rows

```
in_dplyr5 <- slice(dat, -(1:5))
in_data.table5 <- dt[-(1:5), ]
compare(in_dplyr5, in_data.table5, allowAll = TRUE )
```

```
## TRUE
##   dropped attributes
```

Fliter Data

Let's select players with height greater than 70 inches and age less than 28.

```
in_dplyr6 <- filter(dat, height > 70 & age < 28)
in_data.table6 <- dt[height > 70 & age < 28]
compare(in_dplyr6, in_data.table6, allowAll = TRUE )
```

```
## TRUE
##   dropped attributes
```

Order Data

Besides selecting data, we can put data in either decresing or increasing order. Here, let's put age in increasing order.

```
# Ascending
in_dplyr7 <- arrange(dat, age)
in_data.table7 <- setorder(dt, age)
compare(in_dplyr7, in_data.table7, allowAll = TRUE )
```

```
## TRUE
##   dropped attributes
```

This time, let's put height in decreasing order.

```
# Descending
in_dplyr8 <- arrange(dat, desc(height))
in_data.table8 <- setorder(dt, -height)
compare(in_dplyr8, in_data.table8, allowAll = TRUE )
```

```
## TRUE
##   sorted
##   renamed rows
##   dropped row names
##   dropped attributes
```

Adding / Updating Columns

We can also create new variables as well as updating columns.

```
# create new variables
in_dplyr9 <- mutate(dat, height_weight = height / weight, salary2 = salary / 2)
in_data.table9 <- dt[ , c("height_weight", "salary2") := list(height / weight, salary / 2)]
compare(in_dplyr9, in_data.table9, allowAll = TRUE )
```

```
## TRUE
##   sorted
##   renamed rows
##   dropped row names
##   dropped attributes
```

```
# update column
in_dplyr10 <- mutate(dat, salary = salary / 2)
in_data.table10 <- dt[ , salary := salary / 2]
compare(in_dplyr10, in_data.table10, allowAll = TRUE )
```

```
## TRUE
##   shortened comparison
##   sorted
##   renamed rows
##   dropped row names
##   dropped attributes
```

Summarise and Group_by

Note that Group_by() is often used with Summarise(). While summarise() applies a function on multiple columns in order to summarize values like standard deviation, mean, median, etc, group_by() allows us to perform data aggregations.

```r
# summarise() with group_by()
in_dplyr12 <- summarise(group_by(dat, team),
                      avg = mean(height),
                      min = min(age),
                      max = max(weight))
in_data.table12 <- dt[ , .(avg = mean(height),
                      min = min(age),
                      max = max(weight)),
                      by = team]
in_dplyr12
```

```
## # A tibble: 30 x 4
##     team      avg   min   max
##    <chr>    <dbl> <dbl> <dbl>
##  1   ATL 79.14286    22   265
##  2   BOS 78.20000    20   253
##  3   BRK 78.66667    21   275
##  4   CHI 78.53333    21   275
##  5   CHO 78.80000    21   257
##  6   CLE 78.86667    21   260
##  7   DAL 79.13333    22   260
##  8   DEN 79.40000    19   270
##  9   DET 79.53333    20   290
## 10   GSW 79.86667    20   270
## # ... with 20 more rows
```

```r
in_data.table12
```

```
##     team      avg min max
##  1:  NYK 80.00000  21 250
##  2:  CLE 78.86667  21 260
##  3:  DET 79.53333  20 290
##  4:  NOP 79.50000  20 270
##  5:  DEN 79.40000  19 270
##  6:  MIL 80.35714  19 265
##  7:  SAC 78.46667  19 265
##  8:  LAL 80.00000  19 275
##  9:  PHO 78.53333  19 260
## 10:  UTA 79.46667  21 265
## 11:  POR 79.42857  21 280
## 12:  DAL 79.13333  22 260
## 13:  MEM 79.26667  20 260
## 14:  ORL 78.93333  20 260
## 15:  TOR 79.06667  21 265
## 16:  GSW 79.86667  20 270
## 17:  MIN 79.71429  20 250
## 18:  PHI 79.33333  21 275
## 19:  CHO 78.80000  21 257
## 20:  OKC 79.26667  20 255
## 21:  BOS 78.20000  20 253
## 22:  BRK 78.66667  21 275
## 23:  MIA 79.00000  20 265
## 24:  SAS 79.13333  20 260
## 25:  CHI 78.53333  21 275
## 26:  WAS 79.50000  21 250
## 27:  LAC 78.80000  19 265
## 28:  IND 78.50000  20 289
## 29:  ATL 79.14286  22 265
## 30:  HOU 78.28571  20 245
##     team      avg min max
```

```r
compare(in_dplyr12, in_data.table12, allowAll = TRUE )
```

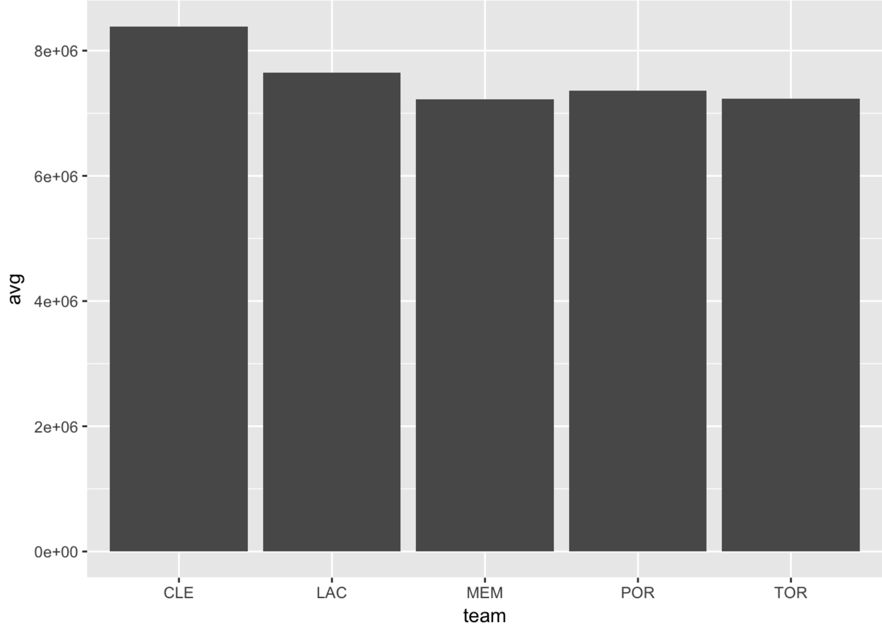
```
## Warning: Setting row names on a tibble is deprecated.

## Warning: Setting row names on a tibble is deprecated.
```

```
## TRUE
##   [min] coerced from <integer> to <numeric>
##   [max] coerced from <integer> to <numeric>
##   sorted
##   renamed rows
##   dropped row names
##   dropped attributes
```

Pipe Operator %>%

In addition, **dplyr** also have another powerful tool: pipe operation %>%, which can avoid many unnecessary work and better understanding of what's going on.

```
dat %>%
  group_by(team) %>%
  summarise(avg = mean(salary)) %>%
  arrange(desc(avg)) %>%
  head(5) %>%
  ggplot(aes(x = team, y = avg)) + geom_bar(stat = 'identity')
```



Summary

In this post, we perform the same tasks using both **data.table** and **dplyr** packages. If you want to know more, you can check out the cheatsheets for dplyr and data.table. While **dplyr** is more elegant and resembles with natural language, **data.table** is succinct. Though **data.table** is similar to base R functions because it builds on base R functions, but **data.table** is much more faster and can save a lot of time.

Reference

- https://www.quora.com/Which-is-better-to-use-for-data-manipulation-dplyr-package-or-data-table-library
- https://stackoverflow.com/questions/21435339/data-table-vs-dplyr-can-one-do-something-well-the-other-cant-or-does-poorly
- https://s3.amazonaws.com/assets.datacamp.com/img/blog/data+table+cheat+sheet.pdf
- https://cran.r-project.org/web/packages/data.table/vignettes/datatable-intro.html
- https://github.com/ucb-stat133/stat133-fall-2017/blob/master/labs/lab05-dplyr-ggplot-basics.Rmd
- https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf
- http://dplyr.tidyverse.org/