Rolling Your Rs

Learning R by Imitation
stay updated via rss

Intro to The data.table Package

Posted: June 14, 2016 in **data.table**, **Performance**, **R programming apply lapply tapply**
12

Data Frames

R provides a helpful data structure called the “data frame” that gives the user an intuitive way to organize, view, and access data. Many of the functions that you would use to read in external files (e.g. **read.csv)** or connect to databases (**RMySQL**), will return a data frame structure by default. While there are other important data structures, such as the **vector**, **list** and **matrix,** the data frame winds up being at the heart of many operations not the least of which is aggregation. Before we get into that let me offer a very brief review of data frame concepts:

- A data frame is a set of rows and columns.
- Each column is of the same length and data type
- Every row is of the same length but can be of differing data types
- A data frame has characteristics of both a matrix and a list
- Bracket notation is the customary method of indexing into a data frame

Subsetting Data The Old School Way

Here are some examples of getting specific subsets of information from the built in data frame mtcars. Note that the bracket notation has two dimensions here – one for row and one for column. The comma within any given bracket notation expression separates the two dimensions.

```
# select rows 1 and 2

mtcars[1:2,]
              mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4      21   6  160 110  3.9 2.620 16.46  0  1    4    4
Mazda RX4 Wag  21   6  160 110  3.9 2.875 17.02  0  1    4    4

# select rows 1 and 2 and columns 3 and 5

mtcars[1:2,c(3,5)]
              disp drat
Mazda RX4      160  3.9
Mazda RX4 Wag  160  3.9

# Find the rows where the MPG column is greater than 30

mtcars[mtcars$mpg > 30,]
                mpg cyl disp  hp drat    wt  qsec vs am gear carb
Fiat 128       32.4   4 78.7  66 4.08 2.200 19.47  1  1    4    1
Honda Civic    30.4   4 75.7  52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla 33.9   4 71.1  65 4.22 1.835 19.90  1  1    4    1
Lotus Europa   30.4   4 95.1 113 3.77 1.513 16.90  1  1    5    2

# select columns 10 and 11 for all rows where MPG > 30 and cylinder is
# equal to 4 and then extract columns 10 and 11

mtcars[mtcars$mpg > 30 & mtcars$cyl == 4, 10:11]
              gear carb
Fiat 128         4    1
Honda Civic      4    2
Toyota Corolla   4    1
Lotus Europa     5    2
```

So the bracket notation winds up being a good way by which to index and search within a data frame although to do aggregation requires us to use other functions such as **tapply, aggregate,** and **table.** This isn't necessarily a bad thing just that you have to learn which function is the most appropriate for the task at hand.

```
# Get the mean MPG by Transmission

tapply(mtcars$mpg, mtcars$am, mean)
0    1
17.1 24.4

# Get the mean MPG for Transmission grouped by Cylinder

aggregate(mpg~am+cyl,data=mtcars,mean)
   am cyl  mpg
1  0   4 22.9
2  1   4 28.1
3  0   6 19.1
4  1   6 20.6
5  0   8 15.1
6  1   8 15.4

# Cross tabulation based on Transmission and Cylinder

table(transmission=mtcars$am, cylinder=mtcars$cyl)
             cylinder
transmission  4  6  8
0  3  4 12
1  8  3  2
```

Enter the data.table package

Okay this is nice though wouldn't it be good to have a way to do aggregation within the bracket notation ? In fact there is. There are a couple of packages that could help us to simplify aggregation though we will start with the **data.table** package for now. In addition to being able to do aggregation within the brackets there are some other reasons why it is useful:

- It works well with very large data files
- Can behave just like a data frame
- Offers fast subset, grouping, update, and joins
- Makes it easy to turn an existing data frame into a data table

Since it is an external package you will first need to install it. After that just load it up using the **library** statement

```
library(data.table)

dt <- data.table(mtcars)

class(dt)
[1] "data.table" "data.frame"

dt[,mean(mpg)]   # You can't do this with a normal data frame
[1] 20.09062

mtcars[,mean(mpg)]  # Such a thing will not work with regular data frames
Error in mean(mpg) : object 'mpg' not found
```

So notice that we can actually find the mean MPG directly within the bracket notation. We don't have to go outside of the brackets to do this. So what about reproducing the previous **tapply** example:

```
tapply(mtcars$mpg,mtcars$am,mean)
 0    1
17.1 24.4

# Here is how we would do this with the data table "dt"

dt[,mean(mpg),by=am]
   am   V1
1:  1 24.4
2:  0 17.1

# We could even extend this to group by am and cyl

dt[,mean(mpg),by=.(am,cyl)]
    am cyl  V1
1:  1   6 20.6
2:  1   4 28.1
3:  0   6 19.1
4:  0   8 15.0
5:  0   4 22.9
6:  1   8 15.4

# If we want to more clearly label the computed average

dt[,.(avg=mean(mpg)),by=.(am,cyl)]
    am cyl  avg
1:  1   6 20.6
2:  1   4 28.1
3:  0   6 19.1
4:  0   8 15.0
5:  0   4 22.9
6:  1   8 15.4
```

Similarities to SQL

It doesn't require many examples to prove that we don't have to use the **aggregate** or **tapply** functions to do any of the work once we have created a data table. Unlike default data frames the bracket notation for a data table object has three dimensions which correspond to what one might see in an SQL statement. Don't worry – you do not have to be an SQL expert to use data.table. In reality you don't have to know it at all although if you do then using data.table becomes much easier.

So in terms of SQL we would say something like select “j” (columns or an operation on some columns) where those columns in a row(s) “i” satisfy some specified condition on the rows. And if the “by” index is supplied it indicates how to group the result. Well this might sound a little involved so something more practical might be in order to illustrate what I’m talking about here.

R :	i	j	by
SQL :	WHERE	SELECT	GROUP BY

So let’s revisit the previous examples and see how it relates to the SQL model – This is helpful in understanding the paradigm associated with data table objects:

```
dt[,mean(mpg),by=am]
   am   V1
1:  1 24.4
2:  0 17.1

# The above is analogous to an SQL statement like

select am,avg(mpg) from mtcars group by am

# The following example

dt[,.(avg=mean(mpg)),by=.(am,cyl)]
    am cyl  avg
1:  1   6 20.6
2:  1   4 28.1
3:  0   6 19.1
4:  0   8 15.0
5:  0   4 22.9
6:  1   8 15.4

# is analogous to an SQL statement like:

select am,avg(mpg) as avg from mtcars group by am,cyl

# The following example

dt[mpg > 20,.(avg=mean(mpg)),by=.(am,cyl)]
    am cyl  avg
1:  1   6 21.0
2:  1   4 28.1
3:  0   6 21.4
4:  0   4 22.9

# would be analogous to the following SQL statement

select am,avg(mpg) as avg from mtcars where mpg > 20 group by am,cyl
```

As previously mentioned one does not need to know SQL to use data.table. However, if you do it can help you understand some of the motivations behind the package.

Tabulation

Here are some more examples that illustrate how we can count and tabulate things. Within a data table the special variable **.N** represents the count of rows. If there is a group by index then it presents the number of rows within that grouping variable.

```
dt[, .N] # How many rows
[1] 32

dt[, .N, by=cyl]  # How many cars in each cylinder group
   cyl  N
1:   6  7
2:   4 11
3:   8 14

# For rows where the wt is > 1.5 tons count the number of cars by
# transmission type.

dt[wt > 1.5, .(count=.N), by=am]
   am count
1:  1    13
2:  0    19
```

We can also do sorting quite easily and very fast.

```
# Present the 5 cars with the best MPG

head(dt[order(-mpg)],5)
 mpg cyl disp  hp drat   wt qsec vs am gear carb
1: 33.9   4 71.1  65 4.22 1.83 19.9  1  1    4    1
2: 32.4   4 78.7  66 4.08 2.20 19.5  1  1    4    1
3: 30.4   4 75.7  52 4.93 1.61 18.5  1  1    4    2
4: 30.4   4 95.1 113 3.77 1.51 16.9  1  1    5    2
5: 27.3   4 79.0  66 4.08 1.94 18.9  1  1    4    1

# Since data table inherits from a data frame we could have also done

dt[order(-mpg)][1:5]
mpg cyl disp  hp drat   wt qsec vs am gear carb
1: 33.9   4 71.1  65 4.22 1.83 19.9  1  1    4    1
2: 32.4   4 78.7  66 4.08 2.20 19.5  1  1    4    1
3: 30.4   4 75.7  52 4.93 1.61 18.5  1  1    4    2
4: 30.4   4 95.1 113 3.77 1.51 16.9  1  1    5    2
5: 27.3   4 79.0  66 4.08 1.94 18.9  1  1    4    1

# We could sort on multiple keys. Here we find the cars with the best
# gas mileage and then sort those on increasing weight

dt[order(-mpg,wt)][1:5]
   mpg cyl disp  hp drat   wt qsec vs am gear carb
1: 33.9   4 71.1  65 4.22 1.83 19.9  1  1    4    1
2: 32.4   4 78.7  66 4.08 2.20 19.5  1  1    4    1
3: 30.4   4 95.1 113 3.77 1.51 16.9  1  1    5    2
4: 30.4   4 75.7  52 4.93 1.61 18.5  1  1    4    2
5: 27.3   4 79.0  66 4.08 1.94 18.9  1  1    4    1
```

Chicago Crime Statistics

Let's look at a more realistic example. I have a file that relates to Chicago crime data that you can download if you wish (that is if you want to work this example). It is about 81 megabytes so it isn't terribly large.

```
url <- "https://raw.githubusercontent.com/steviep42/youtube/master/YOUTUBE.DI
download.file(url,"chi_crimes.zip")
system("unzip chi_crimes.zip")
system.time(df.crimes <- read.csv("chi_crimes.csv", header=TRUE,sep=","))
```

So you might try reading it in with the typical import functions in R such as **read.csv** which many people use as a default when reading in CSV files. I'm reading this file in on a five year old Mac Book with about 8 GB of RAM. Your speed may vary. We'll first read in the file using **read.csv** and then use the **fread** function supplied by data.table to see if there is any appreciable difference

```
system.time(df.crimes <- read.csv("chi_crimes.csv", header=TRUE,sep=","))

user system elapsed
30.251 0.283 30.569

nrow(df.crimes)
[1] 334141

# Now let's try fread

system.time(dt.crimes <- fread("chi_crimes.csv", header=TRUE,sep=","))

user system elapsed
1.045 0.037 1.362

attributes(dt.crimes)$class # dt.crimes is also a data.frame
[1] "data.table" "data.frame"

nrow(df.crimes)
[1] 334141

dt.crimes[,.N]
[1] 334141
```

That was a fairly significant difference. If the file were much larger we would see an even larger time difference which for me is a good thing since I routinely read in large files. Consequently **fread** has become a default for me even if I don't wind up using the aggregation capability of the data.table package. Also note that the **fread** function returns a data.table object by default. Now let's investigate some crime using some of the things we learned earlier. This data table has information on every call to the Chicago police in the year 2012. So we'll want to see what factors there are in the data frame/table so we can do some summaries across groups.

```
names(dt.crimes)
 [1] "Case Number"          "ID"                    "Date"
[4] "Block"                "IUCR"                  "Primary Type"
[7] "Description"          "Location Description" "Arrest"
[10] "Domestic"             "Beat"                  "District"
[13] "Ward"                 "FBI Code"              "X Coordinate"
[16] "Community Area"       "Y Coordinate"          "Year"
[19] "Latitude"             "Updated On"            "Longitude"
[22] "Location"

# Let's see how many unique values there are for each column. Looks
# like 30 FBI codes so maybe we could see the number of calls per FBI
# code. What about District ? There are 25 of those.

sapply(dt.crimes,function(x) length(unique(x)))
```

```
        Case Number                  ID                Date
             334114              334139              121484
              Block                IUCR        Primary Type
              28383                 358                  30
        Description Location Description              Arrest
                296                 120                   2
           Domestic                Beat            District
                  2                 302                  25
               Ward            FBI Code        X Coordinate
                 51                  30               60704
     Community Area        Y Coordinate                Year
                 79               89895                   1
           Latitude          Updated On           Longitude
             180396                1311              180393
           Location
             178534
```

Now – I just used the sapply function to tell me how many unique values each column assumes. This is so we can identify potential summary factors. This is a common activity and I used a common R-like approach although for the newcomer it looks a little verbose (Welcome to R my friend) and it turns out that we can accomplish the same thing using data.table itself. But first let’s do some exploration and then we’ll come back to this. So how many calls per District were there ? Let’s then order this result such that we see the Districts with the highest number of calls first and in descending order.

```
dt.crimes[,.N,by=District][order(-N)]
1:        8 22386
2:       11 21798
3:        7 20150
4:        4 19789
5:       25 19658
6:        6 19232
7:        3 17649
8:        9 16656
9:       19 15608
10:       5 15258
11:      10 15016
12:      15 14385
13:      18 14178
14:       2 13448
15:      14 12537
16:       1 12107
17:      16 10753
18:      22 10745
19:      17  9673
20:      24  9498
21:      12  8774
22:      13  7084
23:      20  5674
24:      NA  2079
25:      31     6
District     N
```

Next let's randomly sample 500 rows and then find the mean number of calls to the cops as grouped by FBI.Code (whatever that corresponds to) check https://www2.fbi.gov/ucr/nibrs/manuals/v1all.pdf (https://www2.fbi.gov/ucr/nibrs/manuals/v1all.pdf) to see them all.

```
dt.crimes[sample(1:.N,500), .(mean=mean(.N)), by="FBI Code"]
  FBI Code mean
1:       14   60
2:       19    3
3:       24    6
4:       26   47
5:       06  109
6:      08B   83
7:       07   27
8:      08A   22
9:       05   34
10:       18   44
11:      04B   10
12:       03   19
13:       11   15
14:      04A    7
15:       09    1
16:       15    6
17:       16    3
18:       10    1
19:       17    1
20:       02    2

# Here we count the number of calls for each day of the year and order from tl
# to the lowest. We see then that the day with the most calls to the police wa
# New Years Day. Whereas Christmas Day had the fewest calls

dt.crimes[,.N,by=substr(Date,1,10)][order(-N)]
         substr    N
  1: 01/01/2012 1297
  2: 06/01/2012 1163
  3: 07/01/2012 1158
  4: 09/01/2012 1124
  5: 07/15/2012 1122
 ---
362: 12/27/2012  679
363: 01/20/2012  677
364: 11/22/2012  653
365: 12/26/2012  619
366: 12/25/2012  520

# Here is another example that would emulate the HAVING clause in SQL

dt.crimes[,.N,by=substr(Date,1,10)][N > 1122][order(-N)]
       substr    N
1: 01/01/2012 1297
2: 06/01/2012 1163
3: 07/01/2012 1158
4: 09/01/2012 1124
```

Other Things

Keep in mind that data.table isn't just for aggregation. You can do anything with it that you can do with a normal data frame. This includes creating new columns, modify existing ones, and create your own functions to do aggregation, and many other activities.

```
# Get the next to the last row from the data table

dt[.N-1]
mpg cyl disp  hp drat   wt qsec vs am gear carb
1:  15   8  301 335 3.54 3.57 14.6  0  1    5    8

dt[cyl %in% c(4,6)]
mpg cyl  disp  hp drat   wt qsec vs am gear carb
1: 21.0   6 160.0 110 3.90 2.62 16.5  0  1    4    4
2: 21.0   6 160.0 110 3.90 2.88 17.0  0  1    4    4
3: 22.8   4 108.0  93 3.85 2.32 18.6  1  1    4    1
4: 21.4   6 258.0 110 3.08 3.21 19.4  1  0    3    1
5: 18.1   6 225.0 105 2.76 3.46 20.2  1  0    3    1
6: 24.4   4 146.7  62 3.69 3.19 20.0  1  0    4    2
7: 22.8   4 140.8  95 3.92 3.15 22.9  1  0    4    2
8: 19.2   6 167.6 123 3.92 3.44 18.3  1  0    4    4
9: 17.8   6 167.6 123 3.92 3.44 18.9  1  0    4    4
10: 32.4   4  78.7  66 4.08 2.20 19.5  1  1    4    1
11: 30.4   4  75.7  52 4.93 1.61 18.5  1  1    4    2
12: 33.9   4  71.1  65 4.22 1.83 19.9  1  1    4    1
13: 21.5   4 120.1  97 3.70 2.46 20.0  1  0    3    1
14: 27.3   4  79.0  66 4.08 1.94 18.9  1  1    4    1
15: 26.0   4 120.3  91 4.43 2.14 16.7  0  1    5    2
16: 30.4   4  95.1 113 3.77 1.51 16.9  1  1    5    2
17: 19.7   6 145.0 175 3.62 2.77 15.5  0  1    5    6
18: 21.4   4 121.0 109 4.11 2.78 18.6  1  1    4    2

# Summarize different variables at once

dt[,.(avg_mpg=mean(mpg),avg_wt=mean(wt))]
avg_mpg avg_wt
1:    20.1   3.22

dt[,cyl:=NULL]  # Removes the cyl column

head(dt)
mpg disp  hp drat   wt qsec vs am gear carb
1: 21.0  160 110 3.90 2.62 16.5  0  1    4    4
2: 21.0  160 110 3.90 2.88 17.0  0  1    4    4
3: 22.8  108  93 3.85 2.32 18.6  1  1    4    1
4: 21.4  258 110 3.08 3.21 19.4  1  0    3    1
5: 18.7  360 175 3.15 3.44 17.0  0  0    3    2
6: 18.1  225 105 2.76 3.46 20.2  1  0    3    1
```

Identifying Possible Factors in a Data Frame

Okay – I demonstrated above a way to identify the number of unique values contained in each column as a means to determine what the factors/categories might be. I used the following

```
sapply(dt.crimes,function(x) length(unique(x)))
```

However, the data.table package has a special variable called **.SD** and a function called **uniqueN** that can help us. Let's explore what these do. First let me create a smaller data frame which makes it easier to see what is going on

```
set.seed(123)
mydt <- data.table(group=sample(LETTERS[1:3],6,T),gender=sample(c("M","F"),6,
                   measure=round(rnorm(6,10),2))

mydt
   group gender measure
1:     A      F   10.46
2:     C      F    8.73
3:     B      F    9.31
4:     C      M    9.55
5:     C      F   11.22
6:     A      M   10.36

# The following groups the data.table by the "gender" column

mydt[,.SD,by=gender]
   gender group measure
1:      F     A   10.46
2:      F     C    8.73
3:      F     B    9.31
4:      F     C   11.22
5:      M     C    9.55
6:      M     A   10.36
```

In this case the **.SD**variable is a way to group the data by a **key** as if we were creating an index. (Note that the data.table package has a **setkey** function for this to formalize the creation of an index).
What about this example ?

```
mydt[,print(.SD),by=gender]
   group measure
1:     A   10.46
2:     C    8.73
3:     B    9.31
4:     C   11.22
   group measure
1:     C    9.55
2:     A   10.36
Empty data.table (0 rows) of 1 col: gender
```

This is somewhat similar to the native **split** function in R that let's one split a data frame on a given factor and store the results in a list. In this case, however, the "splits" aren't really stored any where because we are simply just printing them. A more useful example might be:

```
mydt[,lapply(.SD,mean),by=gender,.SDcols="measure"]
   gender measure
1:      F    9.93
2:      M    9.96
```

Oh wow – so the **.SD** pulls out all the columns in the data frame except gender and then applies the mean function to the columns specified by the **.SDcols** variable. So this is another way of doing some aggregation (although perhaps a bit intimidating for those not familiar with the lapply function. (If that is the case I have a cure for you – go read this post (https://rollingyours.wordpress.com/2014/10/20/the-lapply-command-101/)). Well we might be getting a bit off track here because I wanted to show you how to replace the functionality of the sapply example from above. Here it is. The **uniqueN** function determines how many unique values there are in a column. The **lapply** function applies **uniqueN** to each column.

```
mydt[,lapply(.SD,uniqueN)]
   group gender measure
1:     3      2       6
```

C'est la fin ?

Before we finish this posting it is worth mentioning that the data table package also provides support for setting a key much in the same way one would create an index in a relational database to speed up queries. This is for situations wherein you might have a really large data table and expect to routinely interrogate it using the same column(s) as keys. In the next posting I will look at the **dplyr** package to show another way to handle large files and accomplish intuitive aggregation. Some R experts represents **data.table** as being as competitor of **dplyr** although one could mix the two. What I like about **data.table** is that it allows you to build sophisticated queries, summaries, and aggregations within the bracket notations. It has the added flexibility of allowing you to employ existing R functions or any that you decide to write.

Comments

Intro to The data.table Package – Mubashir Qasim says:
June 15, 2016 at 7:19 am
[…] article was first published on Rolling Your Rs, and kindly contributed to […]

Reply

Lasse says:
June 15, 2016 at 8:21 am

Nice post – I've not really been a big fan of data table, but this explains it in a good way. However, I still feel it is hard package to get to learn as a Rookie, not least because it also saves its objects in a very non-intuitive way. And it is hard to use other functions on data.table objects?

Reply

Stevie P says:
June 15, 2016 at 3:17 pm

Thanks for reading. I understand your point of view – I do and it's probably why I find myself using package connections to databases to retrieve data and/or dplyr more so than data.table. However, with data.table I do like the fact that I can do pretty much everything I need to do within the bracket notation. So on that account it does present a unified approach. Plus I like the fread function and have found it to be quite resilient. Like anything else the more experience you get with it the easier i becomes to grok.

Reply

P Lijnzaad says:
June 15, 2016 at 8:23 am

> Each row is of the same length and data type
> Every column is of the same length but can be of differing data types

This should be exactly the other way around!

Reply

Stevie P says:
June 15, 2016 at 3:13 pm

Indeed ! Thanks for reading and thanks for the catch !

Reply

myschizobuddy says:
June 15, 2016 at 6:28 pm

Will you discuss in your future post how would you use data.table from inside a function. When you do that with dplyr it turns into an ugly toad. You have to use lazyeval:interp to construct your parameters before you pass them to dplyr. I find that extremely complicated. Does data.table suffer from the same mess

Reply

eo says:
June 30, 2016 at 11:57 pm

Does data.table offer any advantage over a data frame when using grep to obtain rows with a particular string or regular expression? Can you give an example on how to use grep on a data.table?

Reply

Stevie P says:
July 1, 2016 at 7:24 am

Thanks for reading. Keep in mind that any R function from any R package can be used in data.table queries (as long as it makes sense to do so of course). So something like the example below is possible. The grep can be as involved as you need it to be. data.table doesn't care. Also consider that using the grep inside a native R data frame of substantial size will begin to slow down. data.table lookups will be fast.

```
library(data.table)
DT = data.table(x=rep(c("ab","ba","cb"),each=3), y=c(1,3,6), v=1:9)
DT[,grep("a$",x,value=T)]
```

data.table has some built in functions such as:

DT[like(x,”a”)]
x y v
1: ab 1 1
2: ab 3 2
3: ab 6 3

DT[y > 2 & like(x,”a”)]
x y v
1: ab 3 2
2: ab 6 3
3: ba 3 5
4: ba 6 6
4: ba 1 4
5: ba 3 5
6: ba 6 6

Reply

jalapic says:
July 4, 2016 at 3:38 pm
Very Nice Tutorial Just a small point about the code. For this line—

dt.crimes[,.N,by=substr(Date,1,10)][order(-N)]

— probably better to go with:

gsub(” .*$”, “”, Date) rather than using ‘substr’

The way it’s written originally it will pick up the first number of the time if a date has single digits for day and month. That way you will not be grouping uniquely by day.

Great post.

Reply

kateryna alyoshina says:
March 16, 2017 at 11:34 pm
hello, does this still work?
url <- "http://steviep42.bitbucket.org/YOUTUBE.DIR/chi_crimes.csv"
download.file(url,"chi_crimes.csv")
I get a 0 data.frame out of it.

thanks a lot for the very useful post

Reply

Stevie P says:
March 16, 2017 at 11:52 pm
I will fix this – Bitbucket changed some access things which has broken my links to data. Sorry for the inconvenience

Reply

Stevie P says:
March 17, 2017 at 2:31 am

Okay I fixed the link to point to the correct location. I changed the logic to reflect that I'm now using a .zip file for the chi_crime data

Reply