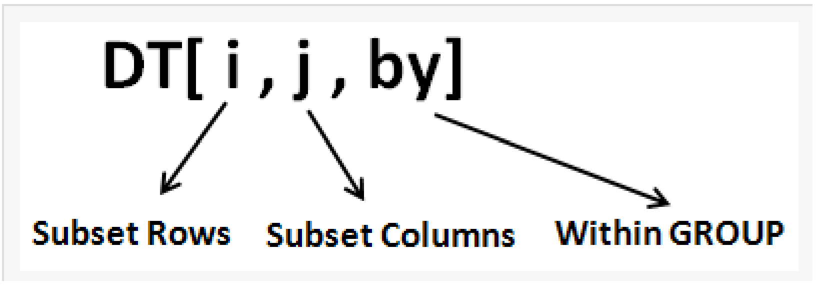
data_dot_table

The data.table R package is considered as the fastest package for data manipulation. This tutorial includes various examples and practice questions to make you familiar with the package. Analysts generally call R programming not compatible with big datasets (> 10 GB) as it is not memory efficient and loads everything into RAM. To change their perception, 'data.table' package comes into play. This package was designed to be concise and painless. There are many benchmarks done in the past to compare dplyr vs data.table. In every benchmark, data.table wins. The efficiency of this package was also compared with python' package (panda). And data.table wins. In CRAN, there are more than 200 packages that are dependent on data.table which makes it listed in the top 5 R's package.

The syntax of data.table is shown in the image below :

% Chosen fairly arbitrarily 

DT[i , j , by]

The first parameter of data.table i refers to rows. It implies subsetting rows. It is equivalent to WHERE clause in SQL The second parameter of data.table j refers to columns. It implies subsetting columns (dropping / keeping). It is equivalent to SELECT clause in SQL. The third parameter of data.table by refers to adding a group so that all calculations would be done within a group. Equivalent to SQL's GROUP BY clause.

The data.table syntax is NOT RESTRICTED to only 3 parameters. There are other arguments that can be added to data.table syntax. The list is as follows

1) with, which

2) allow.cartesian

3) roll, rollends

4) .SD, .SDcols

5) on, mult, nomatch

The above arguments would be explained in the latter part of the post.

```
library(data.table)
library(curl)
```

STEP 1 : Read Data

In data.table package, **fread()** function is available to read or get data from your computer or from a web page. It is equivalent to read.csv() function of base R.

```
mydata = fread("https://github.com/arunsrinivasan/satrdays-workshop/raw/master/flights_2014.csv")
```

STEP 2 : Describe Data

This dataset contains 253K observations and 17 columns. It constitutes information about flights' arrival or departure time, delays, flight cancellation and destination in year 2014.

```
nrow(mydata)
```

```
## [1] 253316
```

```
ncol(mydata)
```

```
## [1] 17
```

```
names(mydata)
```

```
##  [1] "year"      "month"     "day"       "dep_time"  "dep_delay"
##  [6] "arr_time"  "arr_delay" "cancelled" "carrier"   "tailnum"
## [11] "flight"    "origin"    "dest"      "air_time"  "distance"
## [16] "hour"      "min"
```

```
head(mydata)
```

```
##    year month day dep_time dep_delay arr_time arr_delay cancelled carrier
## 1: 2014     1   1      914        14     1238        13         0      AA
## 2: 2014     1   1     1157        -3     1523        13         0      AA
## 3: 2014     1   1     1902         2     2224         9         0      AA
## 4: 2014     1   1      722        -8     1014       -26         0      AA
## 5: 2014     1   1     1347         2     1706         1         0      AA
## 6: 2014     1   1     1824         4     2145         0         0      AA
##    tailnum flight origin dest air_time distance hour min
## 1:  N338AA      1    JFK  LAX      359     2475    9  14
## 2:  N335AA      3    JFK  LAX      363     2475   11  57
## 3:  N327AA     21    JFK  LAX      351     2475   19   2
## 4:  N3EHAA     29    LGA  PBI      157     1035    7  22
## 5:  N319AA    117    JFK  LAX      350     2475   13  47
## 6:  N3DEAA    119    EWR  LAX      339     2454   18  24
```

STEP 3 : Selecting or Keeping Columns

Suppose you need to select only 'origin' column. You can use the code below -

```
dat1 = mydata[ , origin] # returns a vector
```

The above line of code returns a vector not data.table. *To get result in data.table format, run the code below :*

```
dat1 = mydata[ , .(origin)] # returns a data.table
```

It can also be written like data.frame way

```r
dat1 = mydata[, c("origin"), with=FALSE]
```

Keeping a column based on column position*

```r
dat2 =mydata[, 2, with=FALSE]
```

In this code, we are selecting second column from mydata. ### Keeping Multiple Columns* *The following code tells R to select 'origin', 'year', 'month', 'hour' columns.*

```r
dat3 = mydata[, .(origin, year, month, hour)]
```

Keeping multiple columns based on column position

You can keep second through fourth columns using the code below -

```r
dat4 = mydata[, c(2:4), with=FALSE]
```

Dropping a Column

Suppose you want to include all the variables except one column, say. 'origin'. It can be easily done by adding ! sign (implies negation in R)

```r
dat5 = mydata[, !c("origin"), with=FALSE]
```

Dropping Multiple Columns

```r
dat6 = mydata[, !c("origin", "year", "month"), with=FALSE]
```

Keeping variables that contain 'dep'

You can use %like% operator to find pattern. It is same as base R's grepl() function, SQL's LIKE operator and SAS's CONTAINS function.

```r
dat7 = mydata[,names(mydata) %like% "dep", with=FALSE]
```

STEP 4 : Rename Variables

You can rename variables with setnames() function. In the following code, we are renaming a variable 'dest' to 'destination'.

```r
setnames(mydata,c("dest"),c("destination"),skip_absent=TRUE)
```

To rename multiple variables, you can simply add variables in both the sides.

```r
setnames(mydata, c("dest","origin"), c("Destination", "origin.of.flight"),skip_absent=TRUE)
```

STEP 5 : Subsetting Rows / Filtering

Suppose you are asked to find all the flights whose origin is 'JFK'.

```
names(mydata)
```

```
##  [1] "year"              "month"             "day"
##  [4] "dep_time"          "dep_delay"         "arr_time"
##  [7] "arr_delay"         "cancelled"         "carrier"
## [10] "tailnum"           "flight"            "origin.of.flight"
## [13] "destination"       "air_time"          "distance"
## [16] "hour"              "min"
```

```
setnames(mydata,c("origin.of.flight"),c("origin"),skip_absent=TRUE)
```

```
# Filter based on one variable
dat8 = mydata[origin == "JFK"]
```

Select Multiple Values

Filter all the flights whose origin is either 'JFK' or 'LGA'

```
names(mydata)
```

```
##  [1] "year"        "month"       "day"         "dep_time"    "dep_delay"
##  [6] "arr_time"    "arr_delay"   "cancelled"   "carrier"     "tailnum"
## [11] "flight"      "origin"      "destination" "air_time"    "distance"
## [16] "hour"        "min"
```

```
dat9 = mydata[origin %in% c("JFK", "LGA")]
```

Apply Logical Operator : NOT

The following program selects all the flights whose origin is not equal to 'JFK' and 'LGA'

```
# Exclude Values
dat10 = mydata[!origin %in% c("JFK", "LGA")]
```

Filter based on Multiple variables

If you need to select all the flights whose origin is equal to 'JFK' and carrier = 'AA'

```
dat11 = mydata[origin == "JFK" & carrier == "AA"]
```

STEP 6 : Faster Data Manipulation with Indexing

data.table uses binary search algorithm that makes data manipulation faster.

Binary Search Algorithm
Binary search is an efficient algorithm for finding a value from a sorted list of values. It involves repeatedly splitting in half the portion of the list that contains values, until you found the value that you were searching for.

Suppose you have the following values in a variable :

5, 10, 7, 20, 3, 13, 26 You are searching the value 20 in the above list. See how binary search algorithm works -

1. First, we sort the values
2. We would calculate the middle value i.e. 10.
3. We would check whether 20 = 10? No. 20 < 10.
4. Since 20 is greater than 10, it should be somewhere after 10. So we can ignore all the values that are lower than or equal to 10.
5. We are left with 13, 20, 26. The middle value is 20.
6. We would again check whether 20=20. Yes. the match found.

If we do not use this algorithm, we would have to search 5 in the whole list of seven values.

It is important to set key in your dataset which tells system that data is sorted by the key column. For example, you have employee's name, address, salary, designation, department, employee ID. We can use 'employee ID' as a key to search a particular employee.

SET KEY

In this case, we are setting 'origin' as a key in the dataset mydata.

```
# Indexing (Set Keys)
setkey(mydata, origin)
```

Note : It makes the data table sorted by the column 'origin'

How to filter when key is turned on.

You don't need to refer the key column when you apply filter.

```
data12 = mydata[c("JFK", "LGA")]
```

STEP 7 : Performance Comparison

You can compare performance of the filtering process (With or Without KEY).

```
system.time(mydata[origin %in% c("JFK", "LGA")])
```

```
##    user  system elapsed
##   0.044   0.004   0.009
```

```
system.time(mydata[c("JFK", "LGA")])
```

```
##    user  system elapsed
##   0.068   0.008   0.055
```

STEP 8 : Indexing Multiple Columns

We can also set keys to multiple columns like we did below to columns 'origin' and 'dest'. See the example below.

```
names(mydata)
```

```
##  [1] "year"        "month"       "day"         "dep_time"    "dep_delay"
##  [6] "arr_time"    "arr_delay"   "cancelled"   "carrier"     "tailnum"
## [11] "flight"      "origin"      "destination" "air_time"    "distance"
## [16] "hour"        "min"
```

```
setnames(mydata,c("Destination"),c("dest"),skip_absent=TRUE)
```

```
#setkey(mydata, origin, dest)
```

Filtering while setting keys on Multiple Columns

```
# First key column 'origin' matches "JFK" and second key column 'dest' matches "MIA"
#a <- mydata[.("JFK", "MIA")]
```

It is equivalent to the following code :

```
#from base R
#b <- mydata[origin == "JFK" & dest == "MIA"]
```

To identify the column(s) indexed by

```
key(mydata)
```

```
## [1] "origin"
```

Result : It returns origin and dest as these are columns that are set keys.

STEP 9 : Sorting Data

We can sort data using **setorder()** function, By default, it sorts data on ascending order.

```
mydata01 = setorder(mydata, origin)
```

Sorting Data on descending order

In this case, we are sorting data by ‘origin’ variable on descending order.

```
mydata02 = setorder(mydata, -origin)
```

Sorting Data based on multiple variables

In this example, we tells R to reorder data first by origin on ascending order and then variable ’carrier’on descending order.

```
mydata03 = setorder(mydata, origin, -carrier)
```

STEP 10 : Adding Columns (Calculation on rows)

You can do any operation on rows by adding := operator. In this example, we are subtracting 'dep_delay' variable from 'dep_time' variable to compute scheduled departure time.

```
mydata[, dep_sch:=dep_time - dep_delay]
```

Adding Multiple Columns

```
mydata002 = mydata[, c("dep_sch","arr_sch"):=list(dep_time - dep_delay, arr_time - arr_delay)]
```

STEP 11 : IF THEN ELSE

The 'IF THEN ELSE' conditions are very popular for recoding values. In data.table package, it can be done with the following methods : **Method I :**

```
mydata[, flag:= 1*(min < 50)]
```

Method II :

```
mydata[, flag:= ifelse(min < 50, 1,0)]
```

It means to set flag= 1 if min is less than 50. Otherwise, set flag =0.

How to write Sub Queries (like SQL)

We can use this format - DT[ ] [ ] [ ] to build a chain in data.table. It is like sub-queries like SQL.

```
ab <- mydata[, dep_sch:=dep_time - dep_delay][,.(dep_time,dep_delay,dep_sch)]
```

First, we are computing scheduled departure time and then selecting only relevant columns.

Summarize or Aggregate Columns

Like SAS PROC MEANS procedure, we can generate summary statistics of specific variables. In this case, we are calculating mean, median, minimum and maximum value of variable arr_delay.

```
mydata[, .(mean = mean(arr_delay, na.rm = TRUE),
median = median(arr_delay, na.rm = TRUE),
min = min(arr_delay, na.rm = TRUE),
max = max(arr_delay, na.rm = TRUE))]
```

```
##        mean median  min  max
## 1: 8.146702     -4 -112 1494
```

Summarize Multiple Columns

```
mydata[, .(mean(arr_delay), mean(dep_delay))]
```

```
##          V1       V2
## 1: 8.146702 12.46526
```

If you need to calculate summary statistics for a larger list of variables, you can use .SD and .SDcols operators. The .SD operator implies 'Subset of Data'.

```
mydata[, lapply(.SD, mean), .SDcols = c("arr_delay", "dep_delay")]
```

```
##    arr_delay dep_delay
## 1:  8.146702  12.46526
```

In this case, we are calculating mean of two variables - arr_delay and dep_delay.

Summarize all numeric Columns

By default, .SD takes all continuous variables (excluding grouping variables)

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

```
## Warning in mean.default(X[[i]], ...): argument is not numeric or logical:
## returning NA

## Warning in mean.default(X[[i]], ...): argument is not numeric or logical:
## returning NA

## Warning in mean.default(X[[i]], ...): argument is not numeric or logical:
## returning NA

## Warning in mean.default(X[[i]], ...): argument is not numeric or logical:
## returning NA
```

```
##    year    month     day dep_time dep_delay arr_time arr_delay cancelled
## 1: 2014 5.638578 15.8937 1338.238  12.46526  1494.33  8.146702         0
##    carrier tailnum  flight origin destination air_time distance     hour
## 1:      NA      NA 1840.82     NA          NA 156.7228 1099.445 13.06343
##         min  dep_sch  arr_sch      flag
## 1: 31.89482 1325.773 1486.183 0.7474143
```

Summarize with multiple statistics

```
#mydata[, sapply(.SD, function(x) c(mean=mean(x), median=median(x)))]
```

STEP 12 : GROUP BY (Within Group Calculation)

Summarize by group 'origin

```
mydata[, .(mean_arr_delay = mean(arr_delay, na.rm = TRUE)), by = origin]
```

```
##    origin mean_arr_delay
## 1:    EWR      10.026121
## 2:    JFK       7.731465
## 3:    LGA       6.601968
```

Use key column in a by operation

Instead of 'by', you can use keyby= operator.

```
mydata[, .(mean_arr_delay = mean(arr_delay, na.rm = TRUE)), keyby = origin]
```

```
##    origin mean_arr_delay
## 1:    EWR      10.026121
## 2:    JFK       7.731465
## 3:    LGA       6.601968
```

Summarize multiple variables by group 'origin'

```
mydata[, .(mean(arr_delay, na.rm = TRUE), mean(dep_delay, na.rm = TRUE)), by = origin]
```

```
##    origin        V1       V2
## 1:    EWR 10.026121 15.21248
## 2:    JFK  7.731465 11.44617
## 3:    LGA  6.601968 10.60500
```

Or it can be written like below -

```
mydata[, lapply(.SD, mean, na.rm = TRUE), .SDcols = c("arr_delay", "dep_delay"), by = origin]
```

```
##    origin arr_delay dep_delay
## 1:    EWR 10.026121  15.21248
## 2:    JFK  7.731465  11.44617
## 3:    LGA  6.601968  10.60500
```

Remove Duplicates

You can remove non-unique / duplicate cases with unique() function. Suppose you want to eliminate duplicates based on a variable, say. carrier.

```
setkey(mydata, "carrier")
dup <- unique(mydata)
```

Suppose you want to remove duplicated based on all the variables. You can use the command below -

```
setkey(mydata, NULL)
#dup1 <- unique(mydata)
```

Note : Setting key to NULL is not required if no key is already set.

STEP 13 : Extract values within a group

The following command selects first and second values from a categorical variable carrier.

```
dc <- mydata[, .SD[1:2], by=carrier]
```

Select LAST value from a group

```
e <- mydata[, .SD[.N], by=carrier]
```

STEP 14 : SQL's RANK OVER PARTITION

In SQL, Window functions are very useful for solving complex data problems. RANK OVER PARTITION is the most popular window function. It can be easily translated in data.table with the help of frank() function. frank() is similar to base R's rank() function but much faster. See the code below.

```
dt = mydata[, rank:=frank(-distance,ties.method = "min"), by=carrier]
```

In this case, we are calculating rank of variable 'distance' by 'carrier'. We are assigning rank 1 to the highest value of 'distance' within unique values of 'carrier'.

STEP 15 : Cumulative SUM by GROUP

We can calculate cumulative sum by using cumsum() function.

```
dat = mydata[, cum:=cumsum(distance), by=carrier]
```

STEP 16 : Lag and Lead

The lag and lead of a variable can be calculated with shift() function. The syntax of shift() function is as follows - shift(variable_name, number_of_lags, type=c("lag", "lead"))

```
DT <- data.table(A=1:5)
DT[ , X := shift(A, 1, type="lag")]
DT[ , Y := shift(A, 1, type="lead")]
```

STEP 17 : Between and LIKE Operator

We can use %between% operator to define a range. It is inclusive of the values of both the ends.

```
DT = data.table(x=6:10)
DT[x %between% c(7,9)]
```

```
##    x
## 1: 7
## 2: 8
## 3: 9
```

The %like% is mainly used to find all the values that matches a pattern.

```
DT = data.table(Name=c("dep_time","dep_delay","arrival"), ID=c(2,3,4))
DT[Name %like% "dep"]
```

```
##         Name ID
## 1:  dep_time  2
## 2: dep_delay  3
```

STEP 17 : Merging / Joins

The merging in data.table is very similar to base R merge() function. The only difference is data.table by default takes common key variable as a primary key to merge two datasets. Whereas, data.frame takes common variable name as a primary key to merge the datasets.

Sample Data

```
(dt1 <- data.table(A = letters[rep(1:3, 2)], X = 1:6, key = "A"))
```

```
##    A X
## 1: a 1
## 2: a 4
## 3: b 2
## 4: b 5
## 5: c 3
## 6: c 6
```

```
(dt2 <- data.table(A = letters[rep(2:4, 2)], Y = 6:1, key = "A"))
```

```
##    A Y
## 1: b 6
## 2: b 3
## 3: c 5
## 4: c 2
## 5: d 4
## 6: d 1
```

Inner Join : It returns all the matching observations in both the datasets.

```
merge(dt1, dt2, by="A")
```

```
##    A X Y
## 1: b 2 6
## 2: b 2 3
## 3: b 5 6
## 4: b 5 3
## 5: c 3 5
## 6: c 3 2
## 7: c 6 5
## 8: c 6 2
```

Left Join : It returns all observations from the left dataset and the matched observations from the right dataset.

```
merge(dt1, dt2, by="A", all.x = TRUE)
```

```
##     A X  Y
##  1: a 1 NA
##  2: a 4 NA
##  3: b 2  6
##  4: b 2  3
##  5: b 5  6
##  6: b 5  3
##  7: c 3  5
##  8: c 3  2
##  9: c 6  5
## 10: c 6  2
```

Right Join : It returns all observations from the right dataset and the matched observations from the left dataset.

```
merge(dt1, dt2, by="A", all.y = TRUE)
```

```
##      A  X Y
##  1: b  2 6
##  2: b  2 3
##  3: b  5 6
##  4: b  5 3
##  5: c  3 5
##  6: c  3 2
##  7: c  6 5
##  8: c  6 2
##  9: d NA 4
## 10: d NA 1
```

Full Join : It return all rows when there is a match in one of the datasets.

```
merge(dt1, dt2, all=TRUE)
```

```
##      A  X  Y
##  1: a  1 NA
##  2: a  4 NA
##  3: b  2  6
##  4: b  2  3
##  5: b  5  6
##  6: b  5  3
##  7: c  3  5
##  8: c  3  2
##  9: c  6  5
## 10: c  6  2
## 11: d NA  4
## 12: d NA  1
```

STEP 18 : Convert a data.table to data.frame

You can use setDF() function to accomplish this task.

```
#setDF(mydata)
```

Similarly, you can use setDT() function to convert data frame to data table.

```
#set.seed(123)
#X = data.frame(A=sample(3, 10, TRUE),B=sample(letters[1:3], 10, TRUE))
```

```
#setDT(X, key = "A")
```

STEP 19 : Other Useful Functions

Reshape Data : It includes several useful functions which makes data cleaning easy and smooth. To reshape or transpose data, you can use dcast.data.table() and melt.data.table() functions. These functions are sourced from reshape2 package and make them efficient. It also add some new features in these functions.

Rolling Joins : It supports rolling joins. They are commonly used for analyzing time series data. A very R packages supports these kind of joins.

Examples for Practise

Q1. Calculate total number of rows by month and then sort on descending order

```
#mydata[, .N, by = month] [order(-N)]
```

The .N operator is used to find count.

Q2. Find top 3 months with high mean arrival delay

```
#mydata[, .(mean_arr_delay = mean(arr_delay, na.rm = TRUE)), by = month][order(-mean_arr_delay)][1:3]
```

Q3. Find origin of flights having average total delay is greater than 20 minutes

```
#mydata[, lapply(.SD, mean, na.rm = TRUE), .SDcols = c("arr_delay", "dep_delay"), by = origin][(arr_del
```

Q4. Extract average of arrival and departure delays for carrier == 'DL' by 'origin' and 'dest' variables

```
#mydata[carrier == "DL",lapply(.SD, mean, na.rm = TRUE),by = .(origin, dest),.SDcols = c("arr_delay", "
```

Q5. Pull first value of 'air_time' by 'origin' and then sum the returned values when it is greater than 300

```
#mydata[, .SD[1], .SDcols="air_time", by=origin][air_time > 300, sum(air_time)]
```