Introduction to data.table

2015-06-24

This vignette introduces the *data.table* syntax, its general form, how to *subset* rows, *select and compute* on columns and perform aggregations *by group*. Familiarity with *data.frame* data structure from base R is useful, but not essential to follow this vignette.

Data analysis using data.table

Data manipulation operations such as *subset*, *group*, *update*, *join* etc., are all inherently related. Keeping these *related operations together* allows for:

- *concise* and *consistent* syntax irrespective of the set of operations you would like to perform to achieve your end goal.
- performing analysis *fluidly* without the cognitive burden of having to map each operation to a particular function from a set of functions available before to perform the analysis.
- *automatically* optimising operations internally, and very effectively, by knowing precisely the data required for each operation and therefore very fast and memory efficient.

Briefly, if you are interested in reducing *programming* and *compute* time tremendously, then this package is for you. The philosophy that *data.table* adheres to makes this possible. Our goal is to illustrate it through this series of vignettes.

Data

In this vignette, we will use NYC-flights14 data. It contains On-Time flights data from the Bureau of Transporation Statistics for all the flights that departed from New York City airports in 2014 (inspired by nycflights13). The data is available only for Jan-Oct'14.

We can use *data.table's* fast file reader `fread` to load *flights* directly as follows:

```
flights <- fread("https://raw.githubusercontent.com/wiki/arunsrinivasan/flights/NYCflights14/flights14.csv")
flights
#         year month day dep_time dep_delay arr_time arr_delay cancelled carrier tailnum flight
#      1: 2014     1   1      914        14     1238        13         0      AA  N338AA      1
#      2: 2014     1   1     1157        -3     1523        13         0      AA  N335AA      3
#      3: 2014     1   1     1902         2     2224         9         0      AA  N327AA     21
#      4: 2014     1   1      722        -8     1014       -26         0      AA  N3EHAA     29
#      5: 2014     1   1     1347         2     1706         1         0      AA  N319AA    117
#     ---
# 253312: 2014    10  31     1459         1     1747       -30         0      UA  N23708   1744
# 253313: 2014    10  31      854        -5     1147       -14         0      UA  N33132   1758
# 253314: 2014    10  31     1102        -8     1311        16         0      MQ  N827MQ   3591
# 253315: 2014    10  31     1106        -4     1325        15         0      MQ  N511MQ   3592
# 253316: 2014    10  31      824        -5     1045         1         0      MQ  N813MQ   3599
#         origin dest air_time distance hour min
#      1:    JFK  LAX      359     2475    9  14
#      2:    JFK  LAX      363     2475   11  57
#      3:    JFK  LAX      351     2475   19   2
#      4:    LGA  PBI      157     1035    7  22
#      5:    JFK  LAX      350     2475   13  47
#     ---
# 253312:    LGA  IAH      201     1416   14  59
# 253313:    EWR  IAH      189     1400    8  54
# 253314:    LGA  RDU       83      431   11   2
# 253315:    LGA  DTW       75      502   11   6
# 253316:    LGA  SDF      110      659    8  24
dim(flights)
# [1] 253316     17
```

Since we will be using this dataset in all the other vignettes as well, it might be better to download the file once and then load it from disk.

Introduction

In this vignette, we will

1. start with basics - what is a *data.table*, its general form, how to *subset* rows, *select and compute* on columns
2. and then we will look at performing data aggregations by group,

1. Basics

a) What is data.table?

data.table is an R package that provides **an enhanced version** of *data.frames*. In the Data section, we already created a *data.table* using `fread()`. We can also create one using the `data.table()` function. Here is an example:

```
DT = data.table(ID = c("b","b","b","a","a","c"), a = 1:6, b = 7:12, c=13:18)
DT
#    ID a  b  c
# 1:  b 1  7 13
# 2:  b 2  8 14
# 3:  b 3  9 15
# 4:  a 4 10 16
# 5:  a 5 11 17
# 6:  c 6 12 18
class(DT$ID)
# [1] "character"
```

You can also convert existing objects to a *data.table* using `as.data.table()`.

> ### Note that:
> - Unlike *data.frames*, columns of `character` type are *never* converted to `factors` by default.
> - Row numbers are printed with a `:` in order to visually separate the row number from the first column.
> - When the number of rows to print exceeds the global option `datatable.print.nrows` (default = 100), it automatically prints only the top 5 and bottom 5 rows (as can be seen in the Data section).
>
>   ```
>   getOption("datatable.print.nrows")
>   # [1] 100
>   ```
>
> - *data.table* doesn't set or use *row names*, ever. We will see as to why in *"Keys and fast binary search based subset"* vignette.

b) General form - in what way is a data.table *enhanced*?

In contrast to a *data.frame*, you can do *a lot more* than just subsetting rows and selecting columns within the frame of a *data.table*, i.e., within `[ ... ]`. To understand it we will have to first look at the *general form* of *data.table* syntax, as shown below:

```
DT[i, j, by]

##   R:      i                 j        by
## SQL:  where   select | update  group by
```

Users who have a SQL background might perhaps immediately relate to this syntax.

> ### The way to read it (out loud) is:
> Take `DT`, subset rows using `i`, then calculate `j`, grouped by `by`.

Let's begin by looking at `i` and `j` first - subsetting rows and operating on columns.

c) Subset rows in `i`

– Get all the flights with "JFK" as the origin airport in the month of June.

```
ans <- flights[origin == "JFK" & month == 6L]
head(ans)
#    year month day dep_time dep_delay arr_time arr_delay cancelled carrier tailnum flight origin
# 1: 2014     6   1      851        -9     1205        -5         0      AA  N787AA      1    JFK
# 2: 2014     6   1     1220       -10     1522       -13         0      AA  N795AA      3    JFK
# 3: 2014     6   1      718        18     1014        -1         0      AA  N784AA      9    JFK
# 4: 2014     6   1     1024        -6     1314       -16         0      AA  N791AA     19    JFK
# 5: 2014     6   1     1841        -4     2125       -45         0      AA  N790AA     21    JFK
# 6: 2014     6   1     1454        -6     1757       -23         0      AA  N785AA    117    JFK
#    dest air_time distance hour min
# 1:  LAX      324     2475    8  51
# 2:  LAX      329     2475   12  20
# 3:  LAX      326     2475    7  18
# 4:  LAX      320     2475   10  24
# 5:  LAX      326     2475   18  41
# 6:  LAX      329     2475   14  54
```

- Within the frame of a *data.table*, columns can be referred to *as if they are variables*. Therefore, we simply refer to `dest` and `month` as if they are variables. We do not need to add the prefix `flights$` each time. However using `flights$dest` and `flights$month` would work just fine.
- The *row indices* that satisfies the condition `origin == "JFK" & month == 6L` are computed, and since there is nothing else left to do, a *data.table* all columns from `flights` corresponding to those *row indices* are simply returned.
- A comma after the condition is also not required in `i`. But `flights[dest == "JFK" & month == 6L, ]` would work just fine. In *data.frames* however, the comma is necessary.

– Get the first two rows from `flights`.

```
ans <- flights[1:2]
ans
#    year month day dep_time dep_delay arr_time arr_delay cancelled carrier tailnum flight origin
# 1: 2014     1   1      914        14     1238        13         0      AA  N338AA      1    JFK
# 2: 2014     1   1     1157        -3     1523        13         0      AA  N335AA      3    JFK
#    dest air_time distance hour min
# 1:  LAX      359     2475    9  14
# 2:  LAX      363     2475   11  57
```

- In this case, there is no condition. The row indices are already provided in `i`. We therefore return a *data.table* with all columns from `flight` for those *row indices*.

– Sort `flights` first by column `origin` in *ascending* order, and then by `dest` in *descending* order:

We can use the base R function `order()` to accomplish this.

```
ans <- flights[order(origin, -dest)]
head(ans)
#    year month day dep_time dep_delay arr_time arr_delay cancelled carrier tailnum flight origin
# 1: 2014     1   5      836         6     1151        49         0      EV  N12175   4419    EWR
# 2: 2014     1   6      833         7     1111        13         0      EV  N24128   4419    EWR
# 3: 2014     1   7      811        -6     1035       -13         0      EV  N12142   4419    EWR
# 4: 2014     1   8      810        -7     1036       -12         0      EV  N11193   4419    EWR
# 5: 2014     1   9      833        16     1055         7         0      EV  N14198   4419    EWR
# 6: 2014     1  13      923        66     1154        66         0      EV  N12157   4419    EWR
#    dest air_time distance hour min
# 1:  XNA      195     1131    8  36
# 2:  XNA      190     1131    8  33
# 3:  XNA      179     1131    8  11
# 4:  XNA      184     1131    8  10
# 5:  XNA      181     1131    8  33
# 6:  XNA      188     1131    9  23
```

`order()` is internally optimised

- We can use "-" on a *character* columns within the frame of a *data.table* to sort in decreasing order.
- In addition, `order(...)` within the frame of a *data.table* uses *data.table*'s internal fast radix order `forder()`, which is much faster than `base::order`. Here's a small example to highlight the difference.

```
odt = data.table(col=sample(1e7))
(t1 <- system.time(ans1 <- odt[base::order(col)]))  ## uses order from base R
#    user  system elapsed 
#   8.610   0.056   8.708 
(t2 <- system.time(ans2 <- odt[order(col)]))        ## uses data.table's forder
#    user  system elapsed 
#   0.526   0.024   0.553 
(identical(ans1, ans2))
# [1] TRUE
```

The speedup here is **~16x**. We will discuss *data.table*'s fast order in more detail in the *data.table internals* vignette.

- This is so that you can improve performance tremendously while using already familiar functions.

d) Select column(s) in `j`

– Select `arr_delay` column, but return it as a *vector*.

```
ans <- flights[, arr_delay]
head(ans)
# [1]  13  13   9 -26   1   0
```

- Since columns can be referred to as if they are variables within the frame of data.tables, we directly refer to the *variable* we want to subset. Since we want *all the rows*, we simply skip `i`.
- It returns *all* the rows for the column `arr_delay`.

– Select `arr_delay` column, but return as a *data.table* instead.

```
ans <- flights[, list(arr_delay)]
head(ans)
#    arr_delay
# 1:        13
# 2:        13
# 3:         9
# 4:       -26
# 5:         1
# 6:         0
```

- We wrap the *variables* (column names) within `list()`, which ensures that a *data.table* is returned. In case of a single column name, not wrapping with `list()` returns a vector instead, as seen in the previous example.
- *data.table* also allows using `.()` to wrap columns with. It is an *alias* to `list()`; they both mean the same. Feel free to use whichever you prefer.

  We will continue to use `.()` from here on.

data.tables (and *data.frames*) are internally *lists* as well, but with all its columns of equal length and with a *class* attribute. Allowing `j` to return a *list* enables converting and returning a *data.table* very efficiently.

Tip:

As long as `j-expression` returns a *list*, each element of the list will be converted to a column in the resulting *data.table*. This makes `j` quite powerful, as we will see shortly.

– Select both `arr_delay` and `dep_delay` columns.

```
ans <- flights[, .(arr_delay, dep_delay)]
head(ans)
#    arr_delay dep_delay
# 1:        13        14
# 2:        13        -3
# 3:         9         2
# 4:       -26        -8
# 5:         1         2
# 6:         0         4

## alternatively
# ans <- flights[, list(arr_delay, dep_delay)]
```

> - Wrap both columns within `.()`, or `list()`. That's it.

– Select both `arr_delay` and `dep_delay` columns *and* rename them to `delay_arr` and `delay_dep`.

Since `.()` is just an alias for `list()`, we can name columns as we would while creating a *list*.

```
ans <- flights[, .(delay_arr = arr_delay, delay_dep = dep_delay)]
head(ans)
#    delay_arr delay_dep
# 1:        13        14
# 2:        13        -3
# 3:         9         2
# 4:       -26        -8
# 5:         1         2
# 6:         0         4
```

That's it.

e) Compute or *do* in `j`

– How many trips have had total delay < 0?

```
ans <- flights[, sum((arr_delay + dep_delay) < 0)]
ans
# [1] 141814
```

> What's happening here?
>
> - *data.table*'s `j` can handle more than just *selecting columns* - it can handle *expressions*, i.e., *compute on columns*. This shouldn't be surprising, as columns can be referred to as if they are variables. Then we should be able to *compute* by calling functions on those variables. And that's what precisely happens here.

f) Subset in `i` *and* do in `j`

– Calculate the average arrival and departure delay for all flights with "JFK" as the origin airport in the month of June.

```
ans <- flights[origin == "JFK" & month == 6L,
               .(m_arr=mean(arr_delay), m_dep=mean(dep_delay))]
ans
#       m_arr    m_dep
# 1: 5.839349 9.807884
```

> - We first subset in `i` to find matching *row indices* where `origin` airport equals *"JFK"*, and `month` equals *6*. At this point, we *do not* subset the entire *data.table* corresponding to those rows.
> - Now, we look at `j` and find that it uses only *two columns*. And what we have to do is to compute their `mean()`. Therefore we subset just those columns corresponding to the matching rows, and compute their `mean()`.
>
> Because the three main components of the query (`i`, `j` and `by`) are *together* inside `[...]`, *data.table* can see all three and

> optimise the query altogether *before evaluation*, not each separately. We are able to therefore avoid the entire subset, for both speed and memory efficiency.

– How many trips have been made in 2014 from “JFK” airport in the month of June?

```
ans <- flights[origin == "JFK" & month == 6L, length(dest)]
ans
# [1] 8422
```

The function `length()` requires an input argument. We just needed to compute the number of rows in the subset. We could have used any other column as input argument to `length()` really.

This type of operation occurs quite frequently, especially while grouping as we will see in the next section, that *data.table* provides a *special symbol* `.N` for it.

> ### Special symbol `.N`:
> `.N` is a special in-built variable that holds the number of observations in the current group. It is particularly useful when combined with `by` as we’ll see in the next section. In the absence of group by operations, it simply returns the number of rows in the subset.

So we can now accomplish the same task by using `.N` as follows:

```
ans <- flights[origin == "JFK" & month == 6L, .N]
ans
# [1] 8422
```

> - Once again, we subset in `i` to get the *row indices* where `origin` airport equals *“JFK”*, and `month` equals *6*.
> - We see that `j` uses only `.N` and no other columns. Therefore the entire subset is not materialised. We simply return the number of rows in the subset (which is just the length of row indices).
> - Note that we did not wrap `.N` with `list()` or `.()`. Therefore a vector is returned.
>
> We could have accomplished the same operation by doing `nrow(flights[origin == "JFK" & month == 6L])`. However, it would have to subset the entire *data.table* first corresponding to the *row indices* in `i` *and then* return the rows using `nrow()`, which is unnecessary and inefficient. We will cover this and other optimisation aspects in detail under the *data.table design* vignette.

g) Great! But how can I refer to columns by names in `j` (like in a *data.frame*)?

You can refer to column names the *data.frame* way using `with = FALSE`.

– Select both `arr_delay` and `dep_delay` columns the *data.frame* way.

```
ans <- flights[, c("arr_delay", "dep_delay"), with=FALSE]
head(ans)
#    arr_delay dep_delay
# 1:        13        14
# 2:        13        -3
# 3:         9         2
# 4:       -26        -8
# 5:         1         2
# 6:         0         4
```

The argument is named `with` after the R function `with()` because of similar functionality. Suppose you’ve a *data.frame* `DF` and you’d like to subset all rows where `x > 1`.

```
DF = data.frame(x = c(1,1,1,2,2,3,3,3), y = 1:8)

## (1) normal way
DF[DF$x > 1, ] # data.frame needs that ',' as well
#   x y
# 4 2 4
# 5 2 5
# 6 3 6
# 7 3 7
# 8 3 8

## (2) using with
DF[with(DF, x > 1), ]
#   x y
# 4 2 4
# 5 2 5
# 6 3 6
# 7 3 7
# 8 3 8
```

> - Using `with()` in (2) allows using `DF`'s column `x` as if it were a variable.
>
>   Hence the argument name `with` in *data.table*. Setting `with=FALSE` disables the ability to refer to columns as if they are variables, thereby restoring the "*data.frame* mode".
> - We can also *deselect* columns using `-` or `!`. For example:
>
>   ```
>   ## not run
>
>   # returns all columns except arr_delay and dep_delay
>   ans <- flights[, !c("arr_delay", "dep_delay"), with=FALSE]
>   # or
>   ans <- flights[, -c("arr_delay", "dep_delay"), with=FALSE]
>   ```
>
> - From `v1.9.5+`, we can also select by specifying start and end column names, for e.g, `year:day` to select the first three columns.
>
>   ```
>   ## not run
>
>   # returns year,month and day
>   ans <- flights[, year:day, with=FALSE]
>   # returns day, month and year
>   ans <- flights[, day:year, with=FALSE]
>   # returns all columns except year, month and day
>   ans <- flights[, -(year:day), with=FALSE]
>   ans <- flights[, !(year:day), with=FALSE]
>   ```
>
>   This is particularly handy while working interactively.

`with = TRUE` is default in *data.table* because we can do much more by allowing `j` to handle expressions - especially when combined with `by` as we'll see in a moment.

2. Aggregations

We've already seen `i` and `j` from *data.table*'s general form in the previous section. In this section, we'll see how they can be combined together with `by` to perform operations *by group*. Let's look at some examples.

a) Grouping using `by`

– How can we get the number of trips corresponding to each origin airport?

```
ans <- flights[, .(.N), by=.(origin)]
ans
#    origin     N
# 1:    JFK 81483
# 2:    LGA 84433
# 3:    EWR 87400

## or equivalently using a character vector in 'by'
# ans <- flights[, .(.N), by="origin"]
```

- We know `.N` is a special variable that holds the number of rows in the current group. Grouping by `origin` obtains the number of rows, `.N`, for each group.
- By doing `head(flights)` you can see that the origin airports occur in the order *"JFK"*, *"LGA"* and *"EWR"*. The original order of grouping variables is preserved in the result.
- Since we did not provide a name for the column returned in `j`, it was named `N` automatically by recognising the special symbol `.N`.
- `by` also accepts character vector of column names. It is particularly useful to program with, for e.g., designing a function with the columns to be group by as a function argument.
- When there's only one column or expression to refer to in `j` and `by`, we can drop the `.()` notation. This is purely for convenience. We could instead do:

  ```
  ans <- flights[, .N, by=origin]
  ans
  #    origin     N
  # 1:    JFK 81483
  # 2:    LGA 84433
  # 3:    EWR 87400
  ```

  We'll use this convenient form wherever applicable hereafter.

– How can we calculate the number of trips for each origin airport for carrier code *"AA"*?

The unique carrier code *"AA"* corresponds to *American Airlines Inc.*

```
ans <- flights[carrier == "AA", .N, by=origin]
ans
#    origin     N
# 1:    JFK 11923
# 2:    LGA 11730
# 3:    EWR  2649
```

- We first obtain the row indices for the expression `carrier == "AA"` from `i`.
- Using those *row indices*, we obtain the number of rows while grouped by `origin`. Once again no columns are actually materialised here, because the `j-expression` does not require any columns to be actually subsetted and is therefore fast and memory efficient.

– How can we get the total number of trips for each `origin, dest` pair for carrier code *"AA"*?

```
ans <- flights[carrier == "AA", .N, by=.(origin,dest)]
head(ans)
#    origin dest    N
# 1:    JFK  LAX 3387
# 2:    LGA  PBI  245
# 3:    EWR  LAX   62
# 4:    JFK  MIA 1876
# 5:    JFK  SEA  298
# 6:    EWR  MIA  848

## or equivalently using a character vector in 'by'
# ans <- flights[carrier == "AA", .N, by=c("origin", "dest")]
```

> - `by` accepts multiple columns. We just provide all the columns by which to group by.

– How can we get the average arrival and departure delay for each `orig,dest` pair for each month for carrier code *"AA"*?

```
ans <- flights[carrier == "AA",
        .(mean(arr_delay), mean(dep_delay)),
        by=.(origin, dest, month)]
ans
#      origin dest month         V1         V2
#   1:    JFK  LAX     1   6.590361 14.2289157
#   2:    LGA  PBI     1  -7.758621  0.3103448
#   3:    EWR  LAX     1   1.366667  7.5000000
#   4:    JFK  MIA     1  15.720670 18.7430168
#   5:    JFK  SEA     1  14.357143 30.7500000
#  ---
# 196:    LGA  MIA    10  -6.251799 -1.4208633
# 197:    JFK  MIA    10  -1.880184  6.6774194
# 198:    EWR  PHX    10  -3.032258 -4.2903226
# 199:    JFK  MCO    10 -10.048387 -1.6129032
# 200:    JFK  DCA    10  16.483871 15.5161290
```

> - We did not provide column names for expressions in `j`, they were automatically generated (`V1`, `V2`).
> - Once again, note that the input order of grouping columns is preserved in the result.

Now what if we would like to order the result by those grouping columns `origin`, `dest` and `month`?

b) keyby

data.table retaining the original order of groups is intentional and by design. There are cases when preserving the original order is essential. But at times we would like to automatically sort by the variables we grouped by.

– So how can we directly order by all the grouping variables?

```
ans <- flights[carrier == "AA",
        .(mean(arr_delay), mean(dep_delay)),
        keyby=.(origin, dest, month)]
ans
#      origin dest month         V1         V2
#   1:    EWR  DFW     1   6.427673 10.0125786
#   2:    EWR  DFW     2  10.536765 11.3455882
#   3:    EWR  DFW     3  12.865031  8.0797546
#   4:    EWR  DFW     4  17.792683 12.9207317
#   5:    EWR  DFW     5  18.487805 18.6829268
#  ---
# 196:    LGA  PBI     1  -7.758621  0.3103448
# 197:    LGA  PBI     2  -7.865385  2.4038462
# 198:    LGA  PBI     3  -5.754098  3.0327869
# 199:    LGA  PBI     4 -13.966667 -4.7333333
# 200:    LGA  PBI     5 -10.357143 -6.8571429
```

> - All we did was to change `by` to `keyby`. This automatically orders the result by the grouping variables in increasing order. Note that `keyby()` is applied after performing the operation, i.e., on the computed result.
>
> **Keys:** Actually `keyby` does a little more than *just ordering*. It also *sets a key* after ordering by setting an *attribute* called `sorted`. But we'll learn more about `keys` in the next vignette.
>
> For now, all you've to know is you can use `keyby` to automatically order by the columns specified in `by`.

c) Chaining

Let's reconsider the task of getting the total number of trips for each `origin, dest` pair for carrier *"AA"*.

```r
ans <- flights[carrier == "AA", .N, by = .(origin, dest)]
```

– How can we order `ans` using the columns `origin` in ascending order, and `dest` in descending order?

We can store the intermediate result in a variable, and then use `order(origin, -dest)` on that variable. It seems fairly straightforward.

```r
ans <- ans[order(origin, -dest)]
head(ans)
#    origin dest    N
# 1:    EWR  PHX  121
# 2:    EWR  MIA  848
# 3:    EWR  LAX   62
# 4:    EWR  DFW 1618
# 5:    JFK  STT  229
# 6:    JFK  SJU  690
```

> - Recall that we can use "-" on a *character* column in `order()` within the frame of a *data.table*. This is possible to due *data.table*'s internal query optimisation.
> - Also recall that `order(...)` with the frame of a *data.table* is *automatically optimised* to use *data.table*'s internal fast radix order `forder()` for speed. So you can keep using the already *familiar* base R functions without compromising in speed or memory efficiency that *data.table* offers. We will cover this in more detail in the *data.table internals* vignette.

But this requires having to assign the intermediate result and then overwriting that result. We can do one better and avoid this intermediate assignment on to a variable altogther by `chaining` expressions.

```r
ans <- flights[carrier == "AA", .N, by=.(origin, dest)][order(origin, -dest)]
head(ans, 10)
#     origin dest    N
#  1:    EWR  PHX  121
#  2:    EWR  MIA  848
#  3:    EWR  LAX   62
#  4:    EWR  DFW 1618
#  5:    JFK  STT  229
#  6:    JFK  SJU  690
#  7:    JFK  SFO 1312
#  8:    JFK  SEA  298
#  9:    JFK  SAN  299
# 10:    JFK  ORD  432
```

> - We can tack expressions one after another, *forming a chain* of operations, i.e., `DT[ ... ][ ... ][ ... ]`.
> - Or you can also chain them vertically:
>
>   ```r
>   DT[ ...
>    ][ ...
>    ][ ...
>    ]
>   ```

d) Expressions in `by`

– Can `by` accept *expressions* as well or just take columns?

Yes it does. As an example, if we would like to find out how many flights started late but arrived early (or on time), started and arrived late etc...

```
ans <- flights[, .N, .(dep_delay>0, arr_delay>0)]
ans
#    dep_delay arr_delay      N
# 1:      TRUE      TRUE  72836
# 2:     FALSE      TRUE  34583
# 3:     FALSE     FALSE 119304
# 4:      TRUE     FALSE  26593
```

- The last row corresponds to `dep_delay > 0 = TRUE` and `arr_delay > 0 = FALSE`. We can see that 26593 flights started late but arrived early (or on time).
- Note that we did not provide any names to `by-expression`. And names have been automatically assigned in the result.
- You can provide other columns along with expressions, for example: `DT[, .N, by=.(a, b>0)]`.

e) Multiple columns in `j` - `.SD`

– Do we have to compute `mean()` for each column individually?

It is of course not practical to have to type `mean(myCol)` for every column one by one. What if you had a 100 columns to compute `mean()` of?

How can we do this efficiently? To get there, refresh on this tip - *“As long as j-expression returns a list, each element of the list will be converted to a column in the resulting data.table”*. Suppose we can refer to the *data subset for each group* as a variable *while grouping*, then we can loop through all the columns of that variable using the already familiar base function `lapply()`. We don’t have to learn any new function.

Special symbol `.SD`:

data.table provides a *special* symbol, called `.SD`. It stands for **S**ubset of **D**ata. It by itself is a *data.table* that holds the data for *the current group* defined using `by`.

Recall that a *data.table* is internally a list as well with all its columns of equal length.

Let’s use the *data.table* `DT` from before to get a glimpse of what `.SD` looks like.

```
DT
#    ID a  b  c
# 1:  b 1  7 13
# 2:  b 2  8 14
# 3:  b 3  9 15
# 4:  a 4 10 16
# 5:  a 5 11 17
# 6:  c 6 12 18

DT[, print(.SD), by=ID]
#    a b  c
# 1: 1 7 13
# 2: 2 8 14
# 3: 3 9 15
#    a  b  c
# 1: 4 10 16
# 2: 5 11 17
#    a  b  c
# 1: 6 12 18
# Empty data.table (0 rows) of 1 col: ID
```

- `.SD` contains all the columns *except the grouping columns* by default.
- It is also generated by preserving the original order - data corresponding to `ID = "b"`, then `ID = "a"`, and then `ID = "c"`.

To compute on (multiple) columns, we can then simply use the base R function `lapply()`.

```
DT[, lapply(.SD, mean), by=ID]
#    ID   a    b    c
# 1:  b 2.0  8.0 14.0
# 2:  a 4.5 10.5 16.5
# 3:  c 6.0 12.0 18.0
```

> - `.SD` holds the rows corresponding to columns *a*, *b* and *c* for that group. We compute the `mean()` on each of these columns using the already familiar base function `lapply()`.
> - Each group returns a list of three elements containing the mean value which will become the columns of the resulting `data.table`.
> - Since `lapply()` returns a *list*, there is no need to wrap it with an additional `.()` (if necessary, refer to this tip).

We are almost there. There is one little thing left to address. In our `flights` *data.table*, we only wanted to calculate the `mean()` of two columns `arr_delay` and `dep_delay`. But `.SD` would contain all the columns other than the grouping variables by default.

– How can we specify just the columns we would like to compute the `mean()` on?

> .SDcols
>
> Using the argument `.SDcols`. It accepts either column names or column indices. For example, `.SDcols = c("arr_delay", "dep_delay")` ensures that `.SD` contains only these two columns for each group.
>
> Similar to the with = FALSE section, you can also provide the columns to remove instead of columns to keep using `-` or `!` sign as well as select consecutive columns as `colA:colB` and deselect consecutive columns as `!(colA:colB) or`-(colA:colB)`.

Now let us try to use `.SD` along with `.SDcols` to get the `mean()` of `arr_delay` and `dep_delay` columns grouped by `origin`, `dest` and `month`.

```
flights[carrier == "AA",                       ## Only on trips with carrier "AA"
        lapply(.SD, mean),                     ## compute the mean
        by=.(origin, dest, month),             ## for every 'origin,dest,month'
        .SDcols=c("arr_delay", "dep_delay")] ## for just those specified in .SDcols
#      origin dest month  arr_delay  dep_delay
#   1:    JFK  LAX     1   6.590361 14.2289157
#   2:    LGA  PBI     1  -7.758621  0.3103448
#   3:    EWR  LAX     1   1.366667  7.5000000
#   4:    JFK  MIA     1  15.720670 18.7430168
#   5:    JFK  SEA     1  14.357143 30.7500000
#  ---
# 196:    LGA  MIA    10  -6.251799 -1.4208633
# 197:    JFK  MIA    10  -1.880184  6.6774194
# 198:    EWR  PHX    10  -3.032258 -4.2903226
# 199:    JFK  MCO    10 -10.048387 -1.6129032
# 200:    JFK  DCA    10  16.483871 15.5161290
```

f) Subset `.SD` for each group:

– How can we return the first two rows for each `month`?

```
ans <- flights[, head(.SD, 2), by=month]
head(ans)
#    month year day dep_time dep_delay arr_time arr_delay cancelled carrier tailnum flight origin
# 1:     1 2014   1      914        14     1238        13         0      AA  N338AA      1    JFK
# 2:     1 2014   1     1157        -3     1523        13         0      AA  N335AA      3    JFK
# 3:     2 2014   1      859        -1     1226         1         0      AA  N783AA      1    JFK
# 4:     2 2014   1     1155        -5     1528         3         0      AA  N784AA      3    JFK
# 5:     3 2014   1      849       -11     1306        36         0      AA  N784AA      1    JFK
# 6:     3 2014   1     1157        -3     1529        14         0      AA  N787AA      3    JFK
#    dest air_time distance hour min
# 1:  LAX      359     2475    9  14
# 2:  LAX      363     2475   11  57
# 3:  LAX      358     2475    8  59
# 4:  LAX      358     2475   11  55
# 5:  LAX      375     2475    8  49
# 6:  LAX      368     2475   11  57
```

> - `.SD` is a *data.table* that holds all the rows for *that group*. We simply subset the first two rows as we have seen here already.
> - For each group, `head(.SD, 2)` returns the first two rows as a *data.table* which is also a list. So we do not have to wrap it with `.()`.

g) Why keep `j` so flexible?

So that we have a consistent syntax and keep using already existing (and familiar) base functions instead of learning new functions. To illustrate, let us use the *data.table* `DT` we created at the very beginning under What is a data.table? section.

– How can we concatenate columns `a` and `b` for each group in `ID`?

```
DT[, .(val = c(a,b)), by=ID]
#     ID val
#  1:  b   1
#  2:  b   2
#  3:  b   3
#  4:  b   7
#  5:  b   8
#  6:  b   9
#  7:  a   4
#  8:  a   5
#  9:  a  10
# 10:  a  11
# 11:  c   6
# 12:  c  12
```

> - That's it. There is no special syntax required. All we need to know is the base function `c()` which concatenates vectors and the tip from before.

– What if we would like to have all the values of column `a` and `b` concatenated, but returned as a list column?

```
DT[, .(val = list(c(a,b))), by=ID]
#    ID         val
# 1:  b 1,2,3,7,8,9
# 2:  a  4, 5,10,11
# 3:  c        6,12
```

> - Here, we first concatenate the values with `c(a,b)` for each group, and wrap that with `list()`. So for each group, we return a list of all concatenated values.
> - Note those commas are for display only. A list column can contain any object in each cell, and in this example, each cell is itself a vector and some cells contain longer vectors than others.

Once you start internalising usage in `j`, you will realise how powerful the syntax can be. A very useful way to understand it is by playing around, with the help of `print()`.

For example:

```
## (1) look at the difference between
DT[, print(c(a,b)), by=ID]
# [1] 1 2 3 7 8 9
# [1]  4  5 10 11
# [1]  6 12
# Empty data.table (0 rows) of 1 col: ID

## (2) and
DT[, print(list(c(a,b))), by=ID]
# [[1]]
# [1] 1 2 3 7 8 9
#
# [[1]]
# [1]  4  5 10 11
#
# [[1]]
# [1]  6 12
# Empty data.table (0 rows) of 1 col: ID
```

In (1), for each group, a vector is returned, with length = 6,4,2 here. However (2) returns a list of length 1 for each group, with its first element holding vectors of length 6,4,2. Therefore (1) results in a length of `6+4+2 = 12`, whereas (2) returns `1+1+1=3`.

Summary

The general form of *data.table* syntax is:

```
DT[i, j, by]
```

We have seen so far that,

Using `i`:

- We can subset rows similar to a *data.frame* - except you don't have to use `DT$` repetitively since columns within the frame of a *data.table* are seen as if they are *variables*.
- We can also sort a *data.table* using `order()`, which internally uses *data.table*'s fast order for performance.

We can do much more in `i` by keying a *data.table*, which allows blazing fast subsets and joins. We will see this in the *"Keys and fast binary search based subsets"* and *"Joins and rolling joins"* vignette.

Using `j`:

1. Select columns the *data.table* way: `DT[, .(colA, colB)]`.
2. Select columns the *data.frame* way: `DT[, c("colA", "colB"), with=FALSE]`.
3. Compute on columns: `DT[, .(sum(colA), mean(colB))]`.
4. Provide names if necessary: `DT[, .(sA =sum(colA), mB = mean(colB))]`.
5. Combine with `i`: `DT[colA > value, sum(colB)]`.

Using `by`:

- Using `by`, we can group by columns by specifying a *list of columns* or a *character vector of column names* or even *expressions*. The flexibility of `j`, combined with `by` and `i` makes for a very powerful syntax.
- `by` can handle multiple columns and also *expressions*.
- We can `keyby` grouping columns to automatically sort the grouped result.
- We can use `.SD` and `.SDcols` in `j` to operate on multiple columns using already familiar base functions. Here are some examples:

  1. `DT[, lapply(.SD, fun), by=., .SDcols=...]` - applies `fun` to all columns specified in `.SDcols` while grouping by the columns specified in `by`.
  2. `DT[, head(.SD, 2), by=.]` - return the first two rows for each group.
  3. `DT[col > val, head(.SD, 1), by=.]` - combine `i` along with `j` and `by`.

And remember the tip:

As long as `j` returns a *list*, each element of the list will become a column in the resulting *data.table*.

We will see how to *add/update/delete* columns *by reference* and how to combine them with `i` and `by` in the next vignette.