R: data.tableStatistics 506

About `data.table`

The `data.table` package in R provides an extension of the `data.frame` class that aims to be both more computationally and memory efficient. It is particularly well suited for large in-memory data sets and utilizes *indexed keys* to allow quick search, subset, and aggregate by group operations. The package also automatically employs multicore computations through its back end, enabling a degree of no-hassle parallelism.

The `data.table` package also provides an expressive, compact syntax for working with data. However, compared to `dplyr`, this syntax is less *literate* and may be more difficult for a non-expert to read and make sense of.

Creating `data.table` objects

The **data.table** package provides a function `fread()` for reading delimited files like `read.table()` and `readr::read_delim()`, but returns a `data.table` object instead. As with the `tbl_df` class from **tibble**, `data.table` inherits from `data.frame`. Here is an example using the 2014 New York City Flights data. It is available from download at https://github.com/arunsrinivasan/flights/wiki/NYCflights14/flights14.csv.

```
library(data.table)
nyc14 <- fread("data/flights14.csv")
class(nyc14)
```

```
[1] "data.table" "data.frame"
```

You can also create a `data.table` like a `data.frame` using:

```
n <- 1e3
data.table(a = 1:n,
           b = rnorm(n),
```

```r
              c = sample(letters, n, replace = TRUE))
```

```
         a          b c
   1:    1 -2.33930005 i
   2:    2  0.77922284 h
   3:    3 -1.66062586 i
   4:    4 -0.19608212 m
   5:    5  0.01868968 r
  ---
 996:  996  0.17602767 l
 997:  997  0.58825823 o
 998:  998  0.55546818 l
 999:  999  1.48385524 d
1000: 1000 -0.50649643 y
```

Note that similar to **tibble**, `data.table` objects by default print only a subset of the data.

"Indexing" with brackets

The syntax for the `data.table` package is inspired by the bracket (`[]`) notation for indexing matrices and data frames. At the same time, it aims to allow many common data operations (i.e. `dplyr` verbs) to be expressed within these brackets.

The basic idea is `DT[i, j, by]` where we:

- subset or filter rows in the `i` statement,
- select, transform, or create columns (variables) in the `j` statement
- and group with the `by` statement.

Additional operations can also be expressed within the brackets. Remember that, even for a *data.frame* or *matrix* the left bracket `[` is actually a **function**:

```r
d <- data.frame(a = 1:3, b  = 4:6)
d[1:2, ]
```

```
  a b
1 1 4
2 2 5
```

```r
`[`(d, 1:2, )
```

```
  a b
1 1 4
2 2 5
```

Subsetting

Basic subsetting of `data.table` objects works similarly to `data.frames`.

```
nyc14[1:2, ]
```

```
   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
   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
```

Find all flights from LGA to DTW:

```
lga_dtw <- nyc14[origin == "LGA" & dest == "DTW", ]
```

Get the first and last rows of `lga_dtw`:

```
lga_dtw[c(1, .N)]
```

```
   year month day dep_time dep_delay arr_time arr_delay cancelled carrier
1: 2014     1   1      901        -4     1102       -11         0      DL
2: 2014    10  31     1106        -4     1325        15         0      MQ
   tailnum flight origin dest air_time distance hour min
1:  N917DL    181    LGA  DTW       99      502    9   1
2:  N511MQ   3592    LGA  DTW       75      502   11   6
```

In the above, we used `.N` to index the last row. This is a special symbol defined by `data.table` to hold the number of rows or observations in the "current" group. "Current" here refers to the scope in which it is used; in this example, that is the entire `data.table`.

Also, notice the difference from standard `data.frame` or matrix sub-setting, in that we did not leave a blank for columns: `lga_dta[c(1, .N)]` versus `lga_dta[c(1, .N), ]`. With `data.frames`, a single argument to `` `[` `` treats the object as a list and returns as expected.

```r
nyc14[1, ]
```

```
   year month day dep_time dep_delay arr_time arr_delay cancelled carrier
1: 2014     1   1      914        14     1238        13         0      AA
   tailnum flight origin dest air_time distance hour min
1:  N338AA      1    JFK  LAX      359     2475    9  14
```

```r
nyc14[1]
```

```
   year month day dep_time dep_delay arr_time arr_delay cancelled carrier
1: 2014     1   1      914        14     1238        13         0      AA
   tailnum flight origin dest air_time distance hour min
1:  N338AA      1    JFK  LAX      359     2475    9  14
```

```r
nyc14df <- as.data.frame(nyc14)
nyc14df[1, ]
```

```
  year month day dep_time dep_delay arr_time arr_delay cancelled carrier
1 2014     1   1      914        14     1238        13         0      AA
  tailnum flight origin dest air_time distance hour min
1  N338AA      1    JFK  LAX      359     2475    9  14
```

```r
head(nyc14df[1]) # don"t print too much!
```

```
  year
1 2014
2 2014
3 2014
4 2014
5 2014
6 2014
```

You can also use the `i` clause to order a *data.table*:

```r
lga_dtw[order(-month, -day, dep_time)]
```

```
      year month day dep_time dep_delay arr_time arr_delay cancelled carrier
   1: 2014    10  31      718        -2      904        -9         0      DL
   2: 2014    10  31      900        -5     1051       -22         0      DL
   3: 2014    10  31      939        -6     1138        -2         0      MQ
   4: 2014    10  31     1106        -4     1325        15         0      MQ
   5: 2014    10  31     1113         8     1317         8         0      DL
  ---
3659: 2014     1   1     1302        17     1503        21         0      DL
3660: 2014     1   1     1350        -5     1600         5         0      MQ
3661: 2014     1   1     1628        -1     1829        -8         0      DL
3662: 2014     1   1     1920       130     2137       147         0      MQ
3663: 2014     1   1     2037        52     2242        52         0      MQ
      tailnum flight origin dest air_time distance hour min
   1:  N320US    831    LGA  DTW       74      502    7  18
   2:  N818DA    189    LGA  DTW       79      502    9   0
   3:  N528MQ   3478    LGA  DTW       80      502    9  39
   4:  N511MQ   3592    LGA  DTW       75      502   11   6
   5:  N3758Y   2449    LGA  DTW       74      502   11  13
  ---
3659:  N331NW   1131    LGA  DTW       98      502   13   2
3660:  N839MQ   3340    LGA  DTW      101      502   13  50
3661:  N310NW   2231    LGA  DTW       98      502   16  28
3662:  N833MQ   3530    LGA  DTW      103      502   19  20
3663:  N856MQ   3603    LGA  DTW      105      502   20  37
```

Column Selection

Get the departure and arrival times, flight number, and carrier for all flights from LGA to DTW:

```
nyc14[origin == "LGA" & dest == "DTW",
      list(dep_time, arr_time, carrier, flight)]
```

```
      dep_time arr_time carrier flight
   1:      901     1102      DL    181
   2:      555      745      DL    731
   3:     1302     1503      DL   1131
   4:     1628     1829      DL   2231
   5:      849     1058      MQ   3478
  ---
3659:     1613     1757      DL   2231
```

```
3660:      939      1138       MQ   3478
3661:     1912      2104       MQ   3603
3662:     1346      1535       MQ   3631
3663:     1106      1325       MQ   3592
```

Notice the use of `list()` to select columns. A synonym for `list()` within `data.table` is `.()` to save typing and enhance readability:

```
nyc14[origin == "LGA" & dest == "DTW",
      .(dep_time, arr_time, carrier, flight)]
```

```
      dep_time arr_time carrier flight
   1:      901     1102      DL    181
   2:      555      745      DL    731
   3:     1302     1503      DL   1131
   4:     1628     1829      DL   2231
   5:      849     1058      MQ   3478
  ---
3659:     1613     1757      DL   2231
3660:      939     1138      MQ   3478
3661:     1912     2104      MQ   3603
3662:     1346     1535      MQ   3631
3663:     1106     1325      MQ   3592
```

Columns can also be selected using a character vector of column names.

```
nyc14[origin == "LGA" & dest == "DTW",
      c("dep_time", "arr_time", "carrier", "flight")]
```

```
      dep_time arr_time carrier flight
   1:      901     1102      DL    181
   2:      555      745      DL    731
   3:     1302     1503      DL   1131
   4:     1628     1829      DL   2231
   5:      849     1058      MQ   3478
  ---
3659:     1613     1757      DL   2231
3660:      939     1138      MQ   3478
3661:     1912     2104      MQ   3603
3662:     1346     1535      MQ   3631
3663:     1106     1325      MQ   3592
```

Note that a vector of variable names (which are **not** characters) will return a vector. E.g. `DT[, var]` (or `DT$var`) returns a vector similar to how `data.frame$var` does. However, `DT[, c(var1, var2)]` returns the equivalent of `c(data.frame$var1, data.frame$var2)` which probably isn't what you want.

We can deselect columns using negation (`-` or `!`).

```r
ncol(nyc14)
```

```
[1] 17
```

```r
nyc14b <- nyc14[, -c("tailnum")]
ncol(nyc14b)
```

```
[1] 16
```

```r
nyc14b <- nyc14b[, !c("cancelled", "year", "day", "hour", "min")]
ncol(nyc14b)
```

```
[1] 11
```

Note that this only works with the character vector name.

```r
nyc14[, -list(tailnum)]
```

```
Error in -list(tailnum): invalid argument to unary operator
```

```r
nyc14[, -.(tailnum)]
```

```
Error in -list(tailnum): invalid argument to unary operator
```

Computing with Columns

The `j`-clause can be used to compute with column variables like `dplyr::summarize()`. Below, we find the mean and IQR of departure delays for flights between LGA and DTW during this period:

```r
lga_dtw[ , .(median = median(dep_delay),
             p25 = quantile(dep_delay, .25),
             p75 = quantile(dep_delay, .75))]
```

```
   median p25 p75
1:     -3  -6   4
```

Note that both `median` and `quantile` are base R functions. Any function which takes in a vector and returns a scalar can be used in this fashion. Note also the use of `.()` instead of `list()`.

The `j`-clause can also be used to compute with column variables much like `dplyr::transmute()`. Here, we create new columns indicating whether the arrival or departure delays were greater than 15 minutes:

```r
nyc14[, .(delay15 = dep_delay > 15 | arr_delay > 15)]
```

```
        delay15
     1:   FALSE
     2:   FALSE
     3:   FALSE
     4:   FALSE
     5:   FALSE
    ---
253312:   FALSE
253313:   FALSE
253314:    TRUE
253315:   FALSE
253316:   FALSE
```

Reference semantics

To get behavior like `dplyr::mutate()` we need reference semantics. This allows adding/updating/removing columns in-place.

The short version is: modifying columns of a `data.frame` creates copies. This can be slow and/or memory exhaustive. By using the `:=` operator, **data.table** avoids making any copies.

```r
ncol(nyc14)
```

```
[1] 17
```

```r
nyc14[, delay30 := dep_delay > 30 | arr_delay > 30]
ncol(nyc14)
```

```
[1] 18
```

```r
nyc14[1:2, ]
```

```
   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
   tailnum flight origin dest air_time distance hour min delay30
1:  N338AA      1    JFK  LAX      359     2475    9  14   FALSE
2:  N335AA      3    JFK  LAX      363     2475   11  57   FALSE
```

We can see how much faster the `data.table` approach is

```r
nyc2 <- fread("data/flights14.csv")
nyc3 <- fread("data/flights14.csv")
library(microbenchmark)
microbenchmark(
  baseR = {
    nyc2$delay30 <- nyc2$dep_delay > 30 | nyc2$arr_delay > 30
  },
  data.table = {
    nyc3[, delay30 := dep_delay > 30 | arr_delay > 30]
  })
```

```
Warning in microbenchmark(baseR = {: less accurate nanosecond times to avoid
potential integer overflows
```

```
Unit: milliseconds
       expr      min       lq     mean   median       uq       max neval
      baseR 1.710233 2.614426 6.006976 3.405972 5.157042 30.117288   100
 data.table 1.117004 1.209623 1.344081 1.289409 1.350294  3.758306   100
```

```r
identical(nyc2, nyc3)
```

```
[1] TRUE
```

Note that the creation of the new column was done "in place" - nothing was returned.

The `:=` is a function, so you can use it in its functional form to make multiple modifications. E.g.

```
# Not evaluated
DT[, `:=`(new1 = xxx,
          new2 = yyy,
          new3 = zzz)
```

Technically you can also do `DT[, c("new1", "new2", "new3") := list(xxx, yyy, zzz)]` but I find this notation difficult to read, especially as `xxx`, `yyy` and `zz` get long.

by

To perform operations group-wise use a `by` argument after the `j` statement. Let's find the percent of flights with delays of 30 minutes or more by carrier.

```
nyc14[, .(del30_pct = 100 * mean(delay30)), by = dest]
```

```
     dest del30_pct
  1:  LAX 15.525842
  2:  PBI 19.123218
  3:  MIA 13.839645
  4:  SEA 16.671909
  5:  SFO 18.795666
 ---
105:  ANC 23.076923
106:  TVC 21.428571
107:  HYA  9.333333
108:  SBN  0.000000
109:  DAL  0.000000
```

We can use a list to specify multiple grouping variables.

```
nyc14[, .(del30_pct = 100 * mean(delay30)), by = .(origin, dest)]
```

```
     origin dest del30_pct
  1:    JFK  LAX  13.89107
  2:    LGA  PBI  17.20850
  3:    EWR  LAX  19.47468
  4:    JFK  MIA  14.07273
  5:    JFK  SEA  14.49036
 ---
217:    LGA  AVL   0.00000
218:    LGA  GSP  33.33333
219:    LGA  SBN   0.00000
220:    EWR  SBN   0.00000
221:    LGA  DAL   0.00000
```

Pay attention to how the result is ordered - or rather isn't ordered. Specifically, that the result is *not* alphabetically ordered. Rather, it retains the original ordering as much as possible to minimize memory usage.

```r
head(nyc14[, dest], 20)
```

```
 [1] "LAX" "LAX" "LAX" "PBI" "LAX" "LAX" "LAX" "LAX" "MIA" "SEA" "MIA" "SFO"
[13] "BOS" "LAX" "BOS" "ORD" "IAH" "AUS" "DFW" "ORD"
```

Using `keyby`

To order according to the values in the `by` argument, use `keyby` which sets a key with the data.table ordered by this key. More on keys can be found below.

```r
delay_pct1 <- nyc14[, .(del30_pct = 100 * mean(delay30)), by = dest]
key(delay_pct1)
```

```
NULL
```

```r
delay_pct2 <- nyc14[, .(del30_pct = 100 * mean(delay30)), keyby = dest]
key(delay_pct2)
```

```
[1] "dest"
```

```r
cbind(delay_pct1, delay_pct2)
```

```
     dest del30_pct dest del30_pct
  1:  LAX 15.525842  ABQ  25.53957
  2:  PBI 19.123218  ACK  10.10830
  3:  MIA 13.839645  AGS   0.00000
  4:  SEA 16.671909  ALB  27.81065
  5:  SFO 18.795666  ANC  23.07692
 ---
105:  ANC 23.076923  TPA  19.42496
106:  TVC 21.428571  TUL  32.83582
107:  HYA  9.333333  TVC  21.42857
108:  SBN  0.000000  TYS  33.58025
109:  DAL  0.000000  XNA  14.77987
```

Chaining

As with standard `data.frame` indexing, we can compose `data.table` bracketed expressions using chaining.

```
## Find max departure delay by flight among all flights from LGA to DTW
## Then, select flights within the shortest 10% of max_delay
nyc14[origin == "LGA" & dest == "DTW",
      .(max_delay = max(dep_delay)),
      by = .(carrier, flight)
     ][, .(carrier, flight, max_delay,
           max_delay_q10 = quantile(max_delay, .1))
     ][max_delay < max_delay_q10, -"max_delay_q10"]
```

```
   carrier flight max_delay
1:      DL   1107        -2
2:      EV   5405        -5
3:      DL    796        -8
4:      EV   5596       -10
```

Unlike **tidyverse**'s `mutate` which allows for variables to appear on both the RHS and LHS, `data.table` doesn't allow this - so instead we use this chaining.

```
dt <- data.table(a = 1:4)
dplyr::mutate(dt,
              b = a - 1,
              c = b * 2)
```

```
   a b c
1: 1 0 0
2: 2 1 2
3: 3 2 4
4: 4 3 6
```

```
dt[, .(a, b = a - 1, c = b * 2)]
```

```
Error in eval(jsub, SDenv, parent.frame()): object 'b' not found
```

```
dt[, .(a, b = a - 1)][, .(a, b, c = b * 2)]
```

```
   a b c
1: 1 0 0
2: 2 1 2
3: 3 2 4
4: 4 3 6
```

```
dt[, b := a - 1][, c := b * 2]
dt
```

```
   a b c
1: 1 0 0
2: 2 1 2
3: 3 2 4
4: 4 3 6
```

If you prefer pipes `|>` for clarity, you can use them by appending a `_` before the opening bracket:

```
nyc14[origin == "LGA" & dest == "DTW",
      .(max_delay = max(dep_delay)),
      by = .(carrier, flight)] |>
  _[, .(carrier, flight, max_delay,
        max_delay_q10 = quantile(max_delay, .1))] |>
  _[max_delay < max_delay_q10, -"max_delay_q10"]
```

```
   carrier flight max_delay
1:      DL   1107        -2
2:      EV   5405        -5
3:      DL    796        -8
4:      EV   5596       -10
```

You could do the same thing with `%>%` from **magrittr** replacing the `_` with `.`.

`.SD`

Recall that the special symbol `.N` contains the number of rows in each subset defined using `by` or `keyby`.

```r
nyc14[dest == "DTW", .N, by = carrier]
```

```
   carrier    N
1:      DL 3095
2:      EV 1584
3:      MQ 1331
```

There is another special symbol `.SD` which references the entire *subset of data* for each group. It is itself a `data.table`. Consider trying to get the mean of each column of a toy dataset.

```r
dt <- data.table(a = sample(1:100, 6),
                 b = sample(1:100, 6),
                 c = c(1, 1, 1, 2, 2, 2))
dt[, lapply(.SD, mean)]
```

```
          a    b   c
1: 50.83333 52.5 1.5
```

```r
dt[, lapply(.SD, mean), by = c]
```

```
   c        a        b
1: 1 31.33333 51.66667
2: 2 70.33333 53.33333
```

(Note the use of `lapply` since we want a `list`. What happens if we use `sapply`? Why?)

Here's a practical example. Let's get the dimensions of each sub-table defined by carrier.

```r
nyc14[dest == "DTW",
      .(rows = nrow(.SD),
        n = .N, # A second way to count the number of rows
        cols = ncol(.SD),
        class = class(.SD)[1]), # also return the class to see the `.SD` is a
                                # `data.table`
      by = carrier]
```

```
   carrier rows    n cols      class
1:      DL 3095 3095   17 data.table
2:      EV 1584 1584   17 data.table
3:      MQ 1331 1331   17 data.table
```

As a reminder, any valid R expression can be placed in `j`.

```r
nyc14[dest == "DTW", print(.SD[1:2]), by = carrier]
```

```
   year month day dep_time dep_delay arr_time arr_delay cancelled tailnum
1: 2014     1   1      901        -4     1102       -11         0  N917DL
2: 2014     1   1      555        -5      745        -7         0  N342NB
   flight origin dest air_time distance hour min delay30
1:    181    LGA  DTW       99      502    9   1   FALSE
2:    731    LGA  DTW       93      502    5  55   FALSE
   year month day dep_time dep_delay arr_time arr_delay cancelled tailnum
1: 2014     1   1     1225        10     1428         9         0  N14173
2: 2014     1   1     2055        -4     2305        11         0  N14186
   flight origin dest air_time distance hour min delay30
1:   4118    EWR  DTW      103      488   12  25   FALSE
2:   4247    EWR  DTW      102      488   20  55   FALSE
   year month day dep_time dep_delay arr_time arr_delay cancelled tailnum
1: 2014     1   1      849        -6     1058        -2         0  N818MQ
2: 2014     1   1     1920       130     2137       147         0  N833MQ
   flight origin dest air_time distance hour min delay30
1:   3478    LGA  DTW      103      502    8  49   FALSE
2:   3530    LGA  DTW      103      502   19  20    TRUE
```

```
Empty data.table (0 rows and 1 cols): carrier
```

Notice that the grouping variable `carrier` is *not* a column in `.SD`.

We can pass an additional argument `.SDcols` to the bracketing function to limit the columns in `.SD`.

```
nyc14[dest == "DTW",
      .(rows = nrow(.SD),
        cols = ncol(.SD)),
      by = carrier,
      .SDcols = c("origin", "dest", "flight", "dep_time")]
```

```
   carrier rows cols
1:      DL 3095    4
2:      EV 1584    4
3:      MQ 1331    4
```

```
nyc14[dest == "DTW",
      print(.SD),
      by = carrier,
      .SDcols = c("origin", "dest", "flight", "dep_time")]
```

```
      origin dest flight dep_time
   1:    LGA  DTW    181      901
   2:    LGA  DTW    731      555
   3:    LGA  DTW   1131     1302
   4:    JFK  DTW   2184     1601
   5:    LGA  DTW   2231     1628
  ---
3091:    EWR  DTW    825     1628
3092:    LGA  DTW    831      718
3093:    LGA  DTW   1131     1235
3094:    LGA  DTW   2131     1813
3095:    LGA  DTW   2231     1613
      origin dest flight dep_time
   1:    EWR  DTW   4118     1225
   2:    EWR  DTW   4247     2055
   3:    EWR  DTW   4381     1639
   4:    EWR  DTW   5078     1250
   5:    EWR  DTW   4246     1117
  ---
1580:    EWR  DTW   4297      842
1581:    EWR  DTW   4911      910
```

```
1582:    EWR  DTW   4246      731
1583:    EWR  DTW   4247     2133
1584:    EWR  DTW   4911      913
      origin dest flight dep_time
   1:    LGA  DTW   3478      849
   2:    LGA  DTW   3530     1920
   3:    LGA  DTW   3603     2037
   4:    LGA  DTW   3689     1128
   5:    LGA  DTW   3340     1350
  ---
1327:    LGA  DTW   3592     1058
1328:    LGA  DTW   3478      939
1329:    LGA  DTW   3603     1912
1330:    LGA  DTW   3631     1346
1331:    LGA  DTW   3592     1106
```

```
Empty data.table (0 rows and 1 cols): carrier
```

This can be useful in the j statement because it allows us to use `lapply` or any other function that returns a list to compute on multiple columns.

```
# What is the mean departure & arrival delay for each flight to DTW?
nyc14[dest == "DTW",
      lapply(.SD, mean),
      by = .(origin, dest, carrier, flight),
      .SDcols = c("arr_delay", "dep_delay")]
```

```
     origin dest carrier flight  arr_delay   dep_delay
  1:    LGA  DTW      DL    181   5.694118  11.8176471
  2:    LGA  DTW      DL    731  -2.637795  -0.6653543
  3:    LGA  DTW      DL   1131   5.774648   9.3450704
  4:    JFK  DTW      DL   2184  -8.000000  -4.0000000
  5:    LGA  DTW      DL   2231  10.861486  17.9054054
 ---
168:    EWR  DTW      DL   2509 -12.037037  -2.5555556
169:    EWR  DTW      EV   5283 -14.947368  -1.9473684
170:    EWR  DTW      EV   4886 -18.750000  -6.0000000
171:    LGA  DTW      EV   5596 -12.000000 -10.0000000
172:    EWR  DTW      EV   4911 -10.000000  -3.5000000
```

We could have instead defined something like `` `:=`(arr_mean = mean(arr_delay), dep_delay = mean(dep_delay)) `` but as the number of elements get larger, using `.SDcols` is cleaner.

Columns can also be specified as ranges in `.SDcols`.

```
nyc14[dest == "DTW", lapply(.SD, mean),
      by = .(origin, dest, carrier, flight),
      .SDcols = arr_delay:dep_delay
]
```

```
     origin dest carrier flight  arr_delay  arr_time   dep_delay
  1:    LGA  DTW      DL    181   5.694118 1120.9706  11.8176471
  2:    LGA  DTW      DL    731  -2.637795  754.5906  -0.6653543
  3:    LGA  DTW      DL   1131   5.774648 1450.4366   9.3450704
  4:    JFK  DTW      DL   2184  -8.000000 1815.0000  -4.0000000
  5:    LGA  DTW      DL   2231  10.861486 1872.9527  17.9054054
 ---
168:    EWR  DTW      DL   2509 -12.037037 1412.0000  -2.5555556
169:    EWR  DTW      EV   5283 -14.947368 1239.1579  -1.9473684
170:    EWR  DTW      EV   4886 -18.750000 1398.2500  -6.0000000
171:    LGA  DTW      EV   5596 -12.000000 1018.0000 -10.0000000
172:    EWR  DTW      EV   4911 -10.000000 1087.0000  -3.5000000
```

Because `.SDcols` takes a character vector it is often useful to construct it programmatically from the `names()` of the data.table object.

```
delay_cols <- names(nyc14)[ grep("delay", names(nyc14)) ]
delay_stats <-
  nyc14[dest == "DTW",
        c(lapply(.SD, mean),
          lapply(.SD, sd)),
        keyby = .(carrier),
        .SDcols = delay_cols]
delay_stats
```

```
   carrier dep_delay arr_delay   delay30 dep_delay arr_delay   delay30
1:      DL  9.406785  3.074960 0.1350565  37.47738  40.83548 0.3418392
2:      EV 17.391414 13.052399 0.2468434  43.13941  46.21437 0.4313110
3:      MQ  4.903080  4.510143 0.1367393  26.47499  30.55465 0.3437011
```

```
new_names <- c(key(delay_stats),
               paste(delay_cols, "mean", sep = "_"),
               paste(delay_cols, "sd", sep = "_"))
setnames(delay_stats, new_names)
delay_stats
```

```
   carrier dep_delay_mean arr_delay_mean delay30_mean dep_delay_sd arr_delay_sd
1:      DL       9.406785       3.074960    0.1350565     37.47738     40.83548
2:      EV      17.391414      13.052399    0.2468434     43.13941     46.21437
3:      MQ       4.903080       4.510143    0.1367393     26.47499     30.55465
   delay30_sd
1:  0.3418392
2:  0.4313110
3:  0.3437011
```

In this example, note that `setnames()` like all `set*` functions in data.table updates in place by reference.

Copies

One of the goals of the `data.table` package is to use memory efficiently. This is achieved in part by preferring "shallow" copies by reference over "deep copies" by value when appropriate. When an object is copied by *reference* it shares physical memory address with the object it is copied from. This is more efficient, but *may* lead to confusion as changing the value in memory also changes what is pointed to by both objects.

In the example below, we create a `data.table` `DT1` and then assign it to `DT2`. Typical R objects would be copied by value using "copy on modify" semantics, but `DT2` is copied by reference. We can ask for a copy by value explicitly using `copy()`.

```
DT1 <- data.table(a = 5:1, b = letters[5:1])
DT2 <- DT1          # Copy by reference
DT3 <- copy(DT1)    # Copy by value
rbind(address(DT1),
      address(DT2),
      address(DT3))
```

```
     [,1]
[1,] "0x1072ed600"
[2,] "0x1072ed600"
[3,] "0x1072f1a00"
```

```r
DT1[, c := 2 * a]     # Create a new column
DT1
```

```
   a b  c
1: 5 e 10
2: 4 d  8
3: 3 c  6
4: 2 b  4
5: 1 a  2
```

```r
DT2
```

```
   a b  c
1: 5 e 10
2: 4 d  8
3: 3 c  6
4: 2 b  4
5: 1 a  2
```

```r
DT3
```

```
   a b
1: 5 e
2: 4 d
3: 3 c
4: 2 b
5: 1 a
```

```r
rbind(address(DT1),
      address(DT2),
      address(DT3))
```

```
     [,1]
[1,] "0x1072ed600"
[2,] "0x1072ed600"
[3,] "0x1072f1a00"
```

After updating `DT1` to include a new column, `C`, the column appears in `DT2` as well because `DT1` and `DT2` refer to the same object. This is in stark constrast to the majority of R which does lazy evaluation - it references by copy until an object is modified, then creates a copy by value.

```
df <- data.frame(a = 5:1, b = letters[5:1])
df2 <- df
rbind(address(df),
      address(df2))
```

```
     [,1]
[1,] "0x14117fc88"
[2,] "0x14117fc88"
```

```
df$c <- 2*df$a
df
```

```
  a b  c
1 5 e 10
2 4 d  8
3 3 c  6
4 2 b  4
5 1 a  2
```

```
df2
```

```
  a b
1 5 e
2 4 d
3 3 c
4 2 b
5 1 a
```

```
rbind(address(df),
      address(df2))
```

```
     [,1]
[1,] "0x1072b0878"
[2,] "0x14117fc88"
```

Reference Semantics Redux

In the last example above we used reference semantics to create a new column in `DT1` without copying the other other columns and reassigning to a new `DT1` object.

One way in which this is useful is to modify subsets of a `data.table` without re-allocating the entire thing. As an example, let's truncate all `arr_delay` below 0.

```
range(nyc14$arr_delay)
```

```
[1] -112 1494
```

The `tracemem` function tracks an object and prints a message whenever it is copied. Additionally, it tells us the memory location of an object.

```
tracemem(nyc14$arr_delay)
```

```
[1] "<0x1393f0000>"
```

```
nyc14[arr_delay < 0, arr_delay := 0]
range(nyc14$arr_delay)
```

```
[1]    0 1494
```

```
untracemem(nyc14$arr_delay)
# Ending memory location
address(nyc14$arr_delay)
```

```
[1] "0x1393f0000"
```

So what happened here - we modified a `data.table` object in place, without copying it. Let's see what would happen on a `data.frame`.

```
nyc14df <- as.data.frame(nyc14)
tracemem(nyc14df$arr_delay)
```

```
[1] "<0x13b138000>"
```

```r
nyc14df$arr_delay[nyc14df$arr_delay < 0] <- 0
```

```
tracemem[0x13b138000 -> 0x13c2a8000]: eval eval eval_with_user_handlers withVisible withCalli
tracemem[0x13c2a8000 -> 0x13c590000]: eval eval eval_with_user_handlers withVisible withCalli
```

```r
untracemem(nyc14$arr_delay)
address(nyc14df$arr_delay)
```

```
[1] "0x13c590000"
```

There are two copies of the memory in this simple operation.

We can also delete columns by reference using `NULL`:

```r
nyc14[, "month" := NULL]
# i.e. nyc14$month = NULL
```

It turns out that this is a substantially faster operation than using negative indexing.

```r
microbenchmark(
  copy = nyc14b <- copy(nyc14),
  null = {
    nyc14b <- copy(nyc14)
    nyc14b[, "year" := NULL]
  },
  negindex = {
    nyc14b <- copy(nyc14)
    nyc14b[, -"year"]
  }
)
```

```
Unit: microseconds
     expr      min       lq     mean   median       uq      max neval
     copy  656.123 1330.348 2968.947 1508.985 3151.055 34330.98   100
     null  856.941 1430.490 3892.736 1685.695 3639.796 49195.33   100
 negindex 4629.187 5755.170 8404.631 7295.171 8567.524 36768.18   100
```

It's hard to demonstrate here because of the need to include the `copy()` inside, but according to the documentation, setting a column to `NULL` actually takes identically 0 time. Try running this a few times on your own - you'll see that most times, the `copy` and `null` have nearly identical timings

We can use this with `by` to accomplish tasks such as adding a column showing the maximum departure delay by flight.

```
nyc14[, max_dep_delay := max(dep_delay), by = .(carrier, flight)][]
```

```
        year day dep_time dep_delay arr_time arr_delay cancelled carrier
     1: 2014   1      914        14     1238        13         0      AA
     2: 2014   1     1157        -3     1523        13         0      AA
     3: 2014   1     1902         2     2224         9         0      AA
     4: 2014   1      722        -8     1014         0         0      AA
     5: 2014   1     1347         2     1706         1         0      AA
    ---
253312: 2014  31     1459         1     1747         0         0      UA
253313: 2014  31      854        -5     1147         0         0      UA
253314: 2014  31     1102        -8     1311        16         0      MQ
253315: 2014  31     1106        -4     1325        15         0      MQ
253316: 2014  31      824        -5     1045         1         0      MQ
        tailnum flight origin dest air_time distance hour min delay30
     1:  N338AA      1    JFK  LAX      359     2475    9  14   FALSE
     2:  N335AA      3    JFK  LAX      363     2475   11  57   FALSE
     3:  N327AA     21    JFK  LAX      351     2475   19   2   FALSE
     4:  N3EHAA     29    LGA  PBI      157     1035    7  22   FALSE
     5:  N319AA    117    JFK  LAX      350     2475   13  47   FALSE
    ---
253312:  N23708   1744    LGA  IAH      201     1416   14  59   FALSE
253313:  N33132   1758    EWR  IAH      189     1400    8  54   FALSE
253314:  N827MQ   3591    LGA  RDU       83      431   11   2   FALSE
253315:  N511MQ   3592    LGA  DTW       75      502   11   6   FALSE
253316:  N813MQ   3599    LGA  SDF      110      659    8  24   FALSE
        max_dep_delay
     1:           156
     2:           284
     3:           848
     4:            89
     5:           248
    ---
253312:           385
253313:            42
253314:           240
```

```
253315:             68
253316:            121
```

The last set of empty brackets above is a short-hand for a subsequent call to `print(nyc14)`.

Keys

Above we used "indexing" in a generic sense to mean "subsetting". What we mean by "indexing" here is more specific and technical: we create an *indexed* data table by designating specific columns as *keys* and sorting the table by these keys to create more efficient look-ups and aggregations. This is similar to keys in SQL, or how Stata stored what variables a dataset was sorted on.

We saw earlier the `keyby=` argument for grouping `i` and `j` operations. The "key" we generate performs a similar role to row names in a `data.frame` - a way to refer to a row by name rather than position.

Earlier we saw a key being added by `keyby=`. The more explicit way to add a key is with the `setkey()` function.

```
setkey(nyc14, origin) #also, setkeyv(nyc14, "origin") if character is preferred.
key(nyc14)
```

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

After a key has been set, we can subset in the `i`-statement using lists:

```
nyc14[.("LGA")]
```

```
       year day dep_time dep_delay arr_time arr_delay cancelled carrier tailnum
    1: 2014   1      722        -8     1014         0         0      AA  N3EHAA
    2: 2014   1      553        -7      739         0         0      AA  N3KHAA
    3: 2014   1      623        -7      815         0         0      AA  N3BSAA
    4: 2014   1      652        -8      833         0         0      AA  N560AA
    5: 2014   1      738        -2      940        15         0      AA  N3GMAA
   ---
84429: 2014  31      609        24      843         0         0      UA  N16709
84430: 2014  31     1459         1     1747         0         0      UA  N23708
84431: 2014  31     1102        -8     1311        16         0      MQ  N827MQ
84432: 2014  31     1106        -4     1325        15         0      MQ  N511MQ
84433: 2014  31      824        -5     1045         1         0      MQ  N813MQ
```

```
       flight origin dest air_time distance hour min delay30 max_dep_delay
    1:     29    LGA  PBI      157     1035    7  22   FALSE            89
    2:    301    LGA  ORD      142      733    5  53   FALSE            65
    3:    303    LGA  ORD      143      733    6  23   FALSE           126
    4:    305    LGA  ORD      139      733    6  52   FALSE           263
    5:    307    LGA  ORD      145      733    7  38   FALSE           140
   ---
84429:   1714    LGA  IAH      198     1416    6   9   FALSE            53
84430:   1744    LGA  IAH      201     1416   14  59   FALSE           385
84431:   3591    LGA  RDU       83      431   11   2   FALSE           240
84432:   3592    LGA  DTW       75      502   11   6   FALSE            68
84433:   3599    LGA  SDF      110      659    8  24   FALSE           121
```

rather than having to specifically refer to the column `origin`, e.g. `origin == "LGA"`, as we've been doing.

We can have more than one column contribute to the order used to form the key.

```
# key by origin and destination
setkey(nyc14, origin, dest)
key(nyc14)
```

```
[1] "origin" "dest"
```

```
nyc14[.("LGA", "ATL")]
```

```
      year day dep_time dep_delay arr_time arr_delay cancelled carrier tailnum
   1: 2014   1     1810        10     2054        10         0      DL  N930DL
   2: 2014   1     1657        -3     1940         0         0      DL  N965DL
   3: 2014   1     1255        -5     1521         0         0      DL  N994DL
   4: 2014   1     1558        -1     1835         0         0      DL  N955DL
   5: 2014   1      603         3      815         0         0      DL  N392DA
  ---
6921: 2014  31     1708        -2     1933         0         0      WN  N434WN
6922: 2014  31     1533        -2     1748         0         0      WN  N797MX
6923: 2014  31     1259         4     1538         0         0      WN  N298WN
6924: 2014  31      929        -1     1158         0         0      WN  N243WN
6925: 2014  31     2025        -5     2252         0         0      WN  N913WN
      flight origin dest air_time distance hour min delay30 max_dep_delay
   1:     61    LGA  ATL      126      762   18  10   FALSE           325
   2:    221    LGA  ATL      129      762   16  57   FALSE           375
```

```
   3:    781    LGA   ATL       121       762    12   55    FALSE             341
   4:    847    LGA   ATL       130       762    15   58    FALSE             416
   5:    904    LGA   ATL       116       762     6    3    FALSE             164
  ---
6921:    397    LGA   ATL       114       762    17    8    FALSE             156
6922:    419    LGA   ATL       115       762    15   33    FALSE             168
6923:    538    LGA   ATL       116       762    12   59    FALSE               4
6924:    706    LGA   ATL       112       762     9   29    FALSE              43
6925:   2969    LGA   ATL       112       762    20   25    FALSE             189
```

Note that each element of the `list` defined by `.()` corresponds to a key - in this example, it is equivalent to `origin == "LGA" & dest == "ATL"`. You can combine the `list` with `c()` as well.

```
nyc14[.(c("LGA", "EWR"), "ATL")]
```

```
       year day dep_time dep_delay arr_time arr_delay cancelled carrier tailnum
    1: 2014   1     1810        10     2054        10         0      DL  N930DL
    2: 2014   1     1657        -3     1940         0         0      DL  N965DL
    3: 2014   1     1255        -5     1521         0         0      DL  N994DL
    4: 2014   1     1558        -1     1835         0         0      DL  N955DL
    5: 2014   1      603         3      815         0         0      DL  N392DA
   ---
11103: 2014  31      741        -4      958         0         0      DL  N926AT
11104: 2014  31     1521        50     1752        43         0      UA  N15712
11105: 2014  31      555        -5      805         0         0      UA  N14731
11106: 2014  31     1159        -9     1426         0         0      UA  N817UA
11107: 2014  31      811        11     1027         0         0      UA  N33203
       flight origin dest air_time distance hour min delay30 max_dep_delay
    1:     61    LGA  ATL      126      762   18  10   FALSE           325
    2:    221    LGA  ATL      129      762   16  57   FALSE           375
    3:    781    LGA  ATL      121      762   12  55   FALSE           341
    4:    847    LGA  ATL      130      762   15  58   FALSE           416
    5:    904    LGA  ATL      116      762    6   3   FALSE           164
   ---
11103:    807    EWR  ATL      108      746    7  41   FALSE           197
11104:   1554    EWR  ATL      113      746   15  21    TRUE           344
11105:   1614    EWR  ATL      111      746    5  55   FALSE             4
11106:    606    EWR  ATL      119      746   11  59   FALSE           224
11107:   1162    EWR  ATL      109      746    8  11   FALSE           275
```

This is all flights with `origin` LGA or EWR, and destination ATL.

To only refer to the first key, pass a single argument to the `list`:

```
nyc14[.("LGA")]
```

```
       year day dep_time dep_delay arr_time arr_delay cancelled carrier tailnum
    1: 2014   6     1059        -6     1332         0         0      EV  N760EV
    2: 2014   7     1122         2     1352         1         0      EV  N197PQ
    3: 2014  11     1033         0     1245         0         0      EV  N391CA
    4: 2014   1     1810        10     2054        10         0      DL  N930DL
    5: 2014   1     1657        -3     1940         0         0      DL  N965DL
   ---
84429: 2014  29      630         0      849         9         0      MQ  N530MQ
84430: 2014  29     1454        -5     1658         0         0      MQ  N517MQ
84431: 2014  30      626        -4      817         0         0      MQ  N542MQ
84432: 2014  30     1452        -7     1703         0         0      MQ  N514MQ
84433: 2014  31      625        -5      829         0         0      MQ  N501MQ
       flight origin dest air_time distance hour min delay30 max_dep_delay
    1:   5624    LGA  AGS      110      678   10  59   FALSE            19
    2:   5625    LGA  AGS      111      678   11  22   FALSE             2
    3:   5632    LGA  AGS      102      678   10  33   FALSE             0
    4:     61    LGA  ATL      126      762   18  10   FALSE           325
    5:    221    LGA  ATL      129      762   16  57   FALSE           375
   ---
84429:   3547    LGA  XNA      174     1147    6  30   FALSE           105
84430:   3553    LGA  XNA      162     1147   14  54   FALSE           163
84431:   3547    LGA  XNA      154     1147    6  26   FALSE           105
84432:   3553    LGA  XNA      157     1147   14  52   FALSE           163
84433:   3547    LGA  XNA      165     1147    6  25   FALSE           105
```

To refer only to the second key, you need to get all the first keys.

```
nyc14[.(unique(origin), "LGA")]
```

```
   year day dep_time dep_delay arr_time arr_delay cancelled carrier tailnum
1:   NA  NA       NA        NA       NA        NA        NA    <NA>    <NA>
2:   NA  NA       NA        NA       NA        NA        NA    <NA>    <NA>
3:   NA  NA       NA        NA       NA        NA        NA    <NA>    <NA>
   flight origin dest air_time distance hour min delay30 max_dep_delay
1:     NA    EWR  LGA       NA       NA   NA  NA      NA            NA
2:     NA    JFK  LGA       NA       NA   NA  NA      NA            NA
3:     NA    LGA  LGA       NA       NA   NA  NA      NA            NA
```

`unique(origin)` returns a vector of all origins, so it is essentially matching the `origin` key to any input.

We can combine this with `j` and `by` statements.

```
# Find the median departure delay for all flights to DTW
nyc14[.(unique(origin), "DTW"),
      .(med_dep_delay = as.numeric(median(dep_delay)), n = .N),
      by = .(origin, dest, flight)] |>
  _[order(origin, med_dep_delay, -n)]
```

```
     origin dest flight med_dep_delay n
  1:    EWR  DTW   3810         -12.0 1
  2:    EWR  DTW   5823         -10.0 1
  3:    EWR  DTW    355          -8.0 1
  4:    EWR  DTW   4132          -8.0 1
  5:    EWR  DTW   4886          -7.5 4
 ---
168:    LGA  DTW   2352          32.0 1
169:    LGA  DTW   2801          36.0 1
170:    LGA  DTW   2458          76.0 4
171:    LGA  DTW   5437         148.0 1
172:    LGA  DTW   2099         243.0 1
```

In `data.table` when we designate columns as keys, the rows are re-ordered by *reference* in *increasing* order. This physically reorders the rows but uses the same locations in memory for the columns.

Timing Comparisons

A toy comparison

First, let's do a toy example. Let's generate a large data set, and time subsetting the data with and without a key.

```
n <- 2e8
DT <- data.table(group = sample(1:26, n, replace = TRUE),
                 x = rnorm(n))
print(object.size(DT), units = "GB", digits = 2)
```

```
2.24 Gb
```

Let's subset a single group.

```
system.time(DT[group == 9])
```

```
   user  system elapsed 
  0.881   0.223   1.105 
```

Next, set a key and repeat subsetting.

```
setkey(DT, group)
system.time(DT[.(9), ])
```

```
   user  system elapsed 
  0.036   0.005   0.041 
```

Note that the speed-up here is due to the key, not the way we index:

```
system.time(DT[group == 9])
```

```
   user  system elapsed 
  0.024   0.004   0.028 
```

Here, even with the non-key approach to indexing, because we do have `group` keyed, we still see the speed improvement.

Setting the key does add some time:

```
setkey(DT, NULL)
system.time(setkey(DT, group))
```

```
   user  system elapsed 
  0.930   0.070   0.999 
```

A more advanced example

```
n <- 2e8
DT <- data.table(group = sample(1:26, n, replace = TRUE))
DT <- DT[ , .(count = rpois(.N, group)), by = group]
```

```r
DT[1:2,]
```

```
   group count
1:    10    10
2:    10     7
```

```r
print(object.size(DT), units = "GB", digits = 2)
```

```
1.49 Gb
```

`group` identifies group membership, and `count` is a Poisson random variable associated with each observation.

To test, we'll calculate the average `count` within each `group` to obtain $\hat{\lambda}$, focusing only on the first and last group.

First, an approach without keys.

```r
## Unkeyed approach
key(DT)
```

```
NULL
```

```r
tm1 <- system.time({
  ans1 <- DT[group == 1 | group == 26,
             .(lambda_hat = mean(count)),
             by = group]
})
```

Next, we'll set the key and use the keyed approach.

```r
tm_key <- system.time({
  setkey(DT, group)
})
key(DT)
```

```
[1] "group"
```

```r
# keyed approach
tm2 <- system.time({
  ans2 <- DT[.(c(1, 26)),
             .(lambda_hat = mean(count)),
             by = group]
})
```

```r
ans1
```

```
   group lambda_hat
1:     1  0.9999601
2:    26 26.0018974
```

```r
ans2
```

```
   group lambda_hat
1:     1  0.9999601
2:    26 26.0018974
```

```r
rbind(naive = tm1, addkey = tm_key, keyed = tm2)[, "elapsed"]
```

```
 naive addkey  keyed
 1.228  1.454  0.120
```

So while adding the key was slow; the actual processing time of the oepration is an order of magnitude smaller. For a single operation, perhaps keying isn't worth it, but the more operations you do, the more you gain.

Let's compare against base R.

```r
tm3 <- system.time({
  DTsmall <- DT[DT$group == 1 | DT$group == 26]
  ans3 <- aggregate(DTsmall$count, by = list(DTsmall$group), FUN = mean)
})
ans3
```

```
  Group.1          x
1       1  0.9999601
2      26 26.0018974
```

And the **tidyverse**.

```
dtibble <- tibble::as_tibble(DT)
tm4 <- system.time({
  dtibble |>
    dplyr::filter(group == 1 | group == 26) |>
    dplyr::group_by(group) |>
    dplyr::summarize(lambda_hat = mean(count)) |>
    dplyr::ungroup() -> ans4
})
ans4
```

```
# A tibble: 2 x 2
  group lambda_hat
  <int>      <dbl>
1     1       1.00
2    26      26.0
```

```
rbind(naive = tm1, addkey = tm_key, keyed = tm2,
      baseR = tm3, tidy = tm4)[, "elapsed"]
```

```
 naive addkey  keyed  baseR   tidy
 1.228  1.454  0.120  3.072  1.738
```

```
rm(DT) # clean up the large object
```

Joining `data.table`'s

There is often a need to join information stored across two or more data frames. In *R* we have previously used `dplyr::left_join()` or similar `*_join()` functions for this. In base *R* two data.frames can be joined using the S3 generic `merge()` which dispatches the `merge.data.frame()` method.

`merge()`

The `data.table` package defines a `merge.data.table()` method. In addition to the tables to join, there are two key parameters: `by` and `all`.

We use `by` to specify which columns to join on.

```r
nyc14[ origin == "JFK",
      .N,
      .(carrier)] |>
  merge(x=_, nycflights13::airlines,
        by = "carrier", all = TRUE) |>
  _[order(-N)]
```

```
    carrier     N                        name
 1:      B6 34220             JetBlue Airways
 2:      DL 18860        Delta Air Lines Inc.
 3:      AA 11923      American Airlines Inc.
 4:      MQ  5444                   Envoy Air
 5:      UA  3924       United Air Lines Inc.
 6:      VX  3138              Virgin America
 7:      US  2645             US Airways Inc.
 8:      EV  1069    ExpressJet Airlines Inc.
 9:      HA   260      Hawaiian Airlines Inc.
10:      9E    NA           Endeavor Air Inc.
11:      AS    NA        Alaska Airlines Inc.
12:      F9    NA      Frontier Airlines Inc.
13:      FL    NA AirTran Airways Corporation
14:      OO    NA        SkyWest Airlines Inc.
15:      WN    NA      Southwest Airlines Co.
16:      YV    NA          Mesa Airlines Inc.
```

The `by` variables must exist in both tables. If not, use `by.x` and `by.y` instead.

We can specify inner (`all=FALSE`), left (`all.x=TRUE`), right (`all.y=TRUE`), or full (`all=TRUE`) joins using the `all*` parameters.

The resulting merge contains all columns from both tables with duplicate names not used in `by` renamed using a suffix, i.e. `col.x` or `col.y`.

```r
x <- data.table(id = 0:4, letter = letters[26 - 0:4])
y <- data.table(id = 1:5, letter = LETTERS[26 - 1:5])
merge(x, y, by = "id", all = TRUE)
```

```
   id letter.x letter.y
1:  0        z     <NA>
2:  1        y        Y
3:  2        x        X
```

```
4:  3      w      W
5:  4      v      V
6:  5   <NA>      U
```

Joining with []

There may be times where you wish to perform some computation using columns from two tables without the need for an explicit merge first.

In these cases you can use the DT1[DT2,] syntax for joins.

```r
x[y, , on = "id"]
```

```
   id letter i.letter
1:  1      y        Y
2:  2      x        X
3:  3      w        W
4:  4      v        V
5:  5   <NA>        U
```

```r
y[x, , on = "id"]
```

```
   id letter i.letter
1:  0   <NA>        z
2:  1      Y        y
3:  2      X        x
4:  3      W        w
5:  4      V        v
```

```r
x[y, .(id, letter), on = "id"]
```

```
   id letter
1:  1      y
2:  2      x
3:  3      w
4:  4      v
5:  5   <NA>
```

```
x[y, .(id, letter), on = "id", nomatch = 0L]
```

```
   id letter
1:  1      y
2:  2      x
3:  3      w
4:  4      v
```

If we are matching on set keys, we do not need to provide these as `on`.

```
setkey(x, id)
x[y]
y[x] ## Fails because we have no key set for y
setkey(y, id)
y[x]
```

Here is the previous flights example using this syntax.

```
airlines <- data.table(nycflights13::airlines)
nyc14[origin == "JFK",
      .N,
      keyby = carrier] |>
  _[airlines, nomatch = 0L] |>
  _[order(-N)]
```

```
   carrier     N                     name
1:      B6 34220          JetBlue Airways
2:      DL 18860     Delta Air Lines Inc.
3:      AA 11923   American Airlines Inc.
4:      MQ  5444                Envoy Air
5:      UA  3924    United Air Lines Inc.
6:      VX  3138           Virgin America
7:      US  2645          US Airways Inc.
8:      EV  1069 ExpressJet Airlines Inc.
9:      HA   260   Hawaiian Airlines Inc.
```