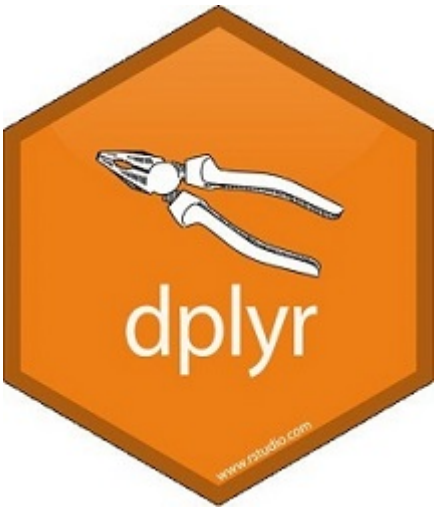
Written by Atrebas
on March 3, 2019

A data.table and dplyr tour



Introduction

About

- This document has been been inspired by this stackoverflow question and by the data.table cheat sheet published by Karlijn Willems. It has been written for my own self-teaching and may contain errors or imprecisions. Corrections and suggestions are welcome.
- Resources for data.table can be found on the data.table wiki, in the data.table vignettes, and in the package documentation.
- Reference documents for dplyr include the dplyr cheat sheet, the dplyr vignettes, and the package documentation.
- For the sake of readability, the console outputs are hidden by default. Click on the button below to show or hide the outputs. Also, the R code used in this document is independently available and can be easily reproduced.

Show the outputs

Update

- *2019-06-22*: Coding style improved (thanks to @tylerburleigh for pointing that out). Some data.table expressions with `by` reordered and indented to better highlight the similarity with dplyr. Minor comments and improvements.

---

data.table and dplyr

data.table and dplyr are two R packages that both aim at an easier and more efficient manipulation of data frames. But while they share a lot of functionalities,

their philosophies are quite different. Below is a quick overview of the main differences (from my basic user's perspective).

Syntax:
- The general data.table syntax is as follows: `DT[i, j, by, ...]` which means: "Take DT, subset rows using `i`, then calculate `j`, grouped by `by`" with possible extra options `...`. It allows to combine several operations in a very concise and consistent expression.
- The syntax of dplyr is based on five main functions (`filter()`, `arrange()`, `select()`, `mutate()`, `summarise()`) and `group_by()` + their scoped variants (suffixed with `_all`, `_at`, or `_if`) + a bunch of helper functions. It is a 'do one thing at a time' approach, chaining together functions dedicated to a specific task.

Ecosystem:
The data.table package has no dependency whereas dplyr is part of the tidyverse. So, for example, while data.table includes functions to read, write, or reshape data, dplyr delegates these tasks to companion packages like readr or tidyr.
On the other hand, data.table is focused on the processing of local in-memory data, but dplyr offers a database backend.

Memory management and performance:
In data.table, objects can be manipulated 'by reference' (using the `set*()` functions or with the `:=` symbol). It means that the data will be modified but not copied, minimizing the RAM requirements. The behaviour of dplyr is similar to the one of base R.
Memory management, parallelism, and shrewd optimization give data.table the advantage in terms of performance.

Create example data

```
library(data.table)
set.seed(1L)

## Create a data table
DT <- data.table(V1 = rep(c(1L, 2L), 5)
[-10],
                 V2 = 1:9,
                 V3 = c(0.5, 1.0, 1.5),
                 V4 = rep(LETTERS[1:3],
3))

class(DT)
DT
```

```
library(dplyr)
set.seed(1L)

## Create a data frame (tibble)
DF <- tibble(V1 = rep(c(1L, 2L), 5)
[-10],
             V2 = 1:9,
             V3 = rep(c(0.5, 1.0, 1.5),
3),
             V4 = rep(LETTERS[1:3], 3))

class(DF)
DF
```

Below, the data.table code uses `DT` and the dplyr code uses `DF`. Also, the dplyr code uses the `%>%` operator: a basic knowledge of the magrittr syntax is assumed.

Basic operations

Filter rows

Filter rows using indices

```
DT[3:4,]
DT[3:4] # same
```

```
DF[3:4,]
slice(DF, 3:4) # same
```

Discard rows using negative indices

```
DT[!3:7,]
DT[-(3:7)] # same
```

```
DF[-(3:7),]
slice(DF, -(3:7)) # same
```

Filter rows using a logical expression

```
DT[V2 > 5]
DT[V4 %chin% c("A", "C")] # fast %in%
for character
```

```
filter(DF, V2 > 5)
filter(DF, V4 %in% c("A", "C"))
```

Filter rows using multiple conditions

```
DT[V1 == 1 & V4 == "A"]
# any logical criteria can be used
```

```
filter(DF, V1 == 1, V4 == "A")
# any logical criteria can be used
```

Filter unique rows

```
unique(DT)
unique(DT, by = c("V1", "V4")) # returns
all cols
```

```
distinct(DF) # distinct_all(DF)
distinct_at(DF, vars(V1, V4)) # returns
selected cols
# see also ?distinct_if
```

Discard rows with missing values

```
na.omit(DT, cols = 1:4)  # fast S3
method with cols argument
```

```
tidyr::drop_na(DF, names(DF))
```

Other filters

In addition to the main `filter()` function, dplyr also offers the `filter_all/at/if` variants as well as three helper functions to filter rows. With data.table, we can simply use a custom expression in `DT[i]` .

```
DT[sample(.N, 3)] # .N = nb of rows in
DT
DT[sample(.N, .N / 2)]
DT[frankv(-V1, ties.method = "dense") <
2]
```

```
sample_n(DF, 3)      # n random rows
sample_frac(DF, 0.5) # fraction of
random rows
top_n(DF, 1, V1)     # top n entries
(includes equals)
```

On the other hand, data.table also provides convenience functions to filter rows based on a regular expression or to find values lying in one (or several) interval(s).

```
DT[V4 %like% "^B"]                          filter(DF, grepl("^B", V4))
DT[V2 %between% c(3, 5)]                    filter(DF, dplyr::between(V2, 3, 5))
DT[data.table::between(V2, 3, 5,            filter(DF, V2 > 3 & V2 < 5)
incbounds = FALSE)]                         filter(DF, V2 >= -1:1 & V2 <= 1:3)
DT[V2 %inrange% list(-1:1, 1:3)] # see
also ?inrange
```

Below, we will see that data.table has two optimized mechanisms to filter rows efficiently (keys and indices).

Sort rows

Sort rows by column

```
DT[order(V3)]  # see also setorder         arrange(DF, V3)
```

Sort rows in decreasing order

```
DT[order(-V3)]                              arrange(DF, desc(V3))
```

Sort rows based on several columns

```
DT[order(V1, -V2)]                          arrange(DF, V1, desc(V2))
```

Select columns

Select one column using an index (not recommended)

```
DT[[3]] # returns a vector                  DF[[3]] # returns a vector
DT[, 3]  # returns a data.table            DF[3]   # returns a tibble
```

Select one column using column name

```
DT[, list(V2)] # returns a data.table       select(DF, V2) # returns a tibble
DT[, .(V2)]    # returns a data.table       pull(DF, V2)   # returns a vector
# . is an alias for list                    DF[, "V2"]       # returns a tibble
DT[, "V2"]     # returns a data.table       DF[["V2"]]       # returns a vector
DT[, V2]       # returns a vector
DT[["V2"]]     # returns a vector
```

Select several columns

```
DT[, .(V2, V3, V4)]                         select(DF, V2, V3, V4)
DT[, list(V2, V3, V4)]                      select(DF, V2:V4) # select columns
DT[, V2:V4] # select columns between V2     between V2 and V4
and V4
```

Exclude columns

```
DT[, !c("V2", "V3")]
```

```
select(DF, -V2, -V3)
```

Select/Exclude columns using a character vector

```
cols <- c("V2", "V3")
DT[, ..cols] # .. prefix means 'one-
level up'
DT[, !..cols] # or DT[, -..cols]
```

```
cols <- c("V2", "V3")
select(DF, !!cols) # unquoting
select(DF, -!!cols)
```

Other selections

As for row filtering, dplyr includes helper functions to select column. With data.table, a possible solution is to first retrieve the column names (*e.g.* using a regular expression), then select these columns. Another way (using `patterns()`) is presented in a next section.

```
cols <- paste0("V", 1:2)
cols <- union("V4", names(DT))
cols <- grep("V",   names(DT))
cols <- grep("3$",  names(DT))
cols <- grep(".2",  names(DT))
cols <- grep("^V1|X$",  names(DT))
cols <- grep("^(?!V2)", names(DT), perl
= TRUE)
DT[, ..cols]
```

```
select(DF, num_range("V", 1:2))
select(DF, V4, everything()) # reorder
columns
select(DF, contains("V"))
select(DF, ends_with("3"))
select(DF, matches(".2"))
select(DF, one_of(c("V1", "X")))


select(DF, -starts_with("V2"))
# remove variables using "-" prior to
function
```

Summarise data

Summary functions take vectors as input and return a single value (*e.g.* `min()`, `mean()`, `var()`, ...).

Summarise one column

```
DT[, sum(V1)]    # returns a vector
DT[, .(sum(V1))] # returns a data.table
DT[, .(sumV1 = sum(V1))] # returns a
data.table
```

```
summarise(DF, sum(V1)) # returns a
tibble
summarise(DF, sumV1 = sum(V1)) # returns
a tibble
```

Summarise several columns

```
DT[, .(sum(V1), sd(V3))]
```

```
summarise(DF, sum(V1), sd(V3))
```

Summarise several columns and assign column names

```
DT[, .(sumv1 = sum(V1),
       sdv3  = sd(V3))]
```

```
DF %>%
  summarise(sumv1 = sum(V1),
            sdv3  = sd(V3))
```

Summarise a subset of rows

```
DT[1:4, sum(V1)]
```

```
DF %>%
  slice(1:4) %>%
  summarise(sum(V1))
```

dplyr helper functions for `summarise()` (or `summarize()`) include `first()`, `last()`, `n()`, `nth()`, and `n_distinct()`. The data.table package also include `first()`, `last()`, and `uniqueN()`.

```
DT[, data.table::first(V3)]
DT[, data.table::last(V3)]
DT[5, V3]
DT[, uniqueN(V4)]
uniqueN(DT)
```

```
summarise(DF, dplyr::first(V3))
summarise(DF, dplyr::last(V3))
summarise(DF, nth(V3, 5))
summarise(DF, n_distinct(V4))
n_distinct(DF)
```

Add/update/delete columns

In the following commands, with data.table, columns are modified by reference using the column assignment symbol `:=` (no copy performed) and the results are returned invisibly. With dplyr, we have to assign the results.

Modify a column

```
DT[, V1 := V1^2]
DT
```

```
DF <- DF %>% mutate(V1 = V1^2)
DF
```

Add one column

```
DT[, v5 := log(V1)][] # adding [] prints
the result
```

```
DF <- mutate(DF, v5 = log(V1))
```

Add several columns

```
DT[, c("v6", "v7") := .(sqrt(V1), "X")]
DT[, ':='(v6 = sqrt(V1),
          v7 = "X")]     # same,
functional form
```

```
DF <- mutate(DF, v6 = sqrt(V1), v7 =
"X")
```

Create one column and remove the others

```
DT[, .(v8 = V3 + 1)]
```

```
transmute(DF, v8 = V3 + 1)
```

Remove one column

```
DT[, v5 := NULL]
```

```
DF <- select(DF, -v5)
```

Remove several columns

```
DT[, c("v6", "v7") := NULL]
```

```
DF <- select(DF, -v6, -v7)
```

Remove columns using a vector of colnames

```
cols <- c("V3")
DT[, (cols) := NULL] # ! not DT[, cols
:= NULL]
```

```
cols <- c("V3")
DF <- select(DF, -one_of(cols))
```

Replace values for rows matching a condition

```
DT[V2 < 4, V2 := 0L]
DT
```

```
DF <- mutate(DF, V2 = base::replace(V2,
V2 < 4, 0L))
DF
```

---

by

The `dplyr::group_by()` function and the corresponding `by` and `keyby` statements in data.table allow to run manipulate each group of observations and combine the results. The sole difference between `by` and `keyby` is that `keyby` orders the results and creates a `key` that will allow faster subsetting (cf. the indexing and keys section). Below, we arbitrary use one or the other.
Again, the `all-at-if` variants of `group_by()` are available in dplyr but not presented here.
Note that it is possible to reorder the arguments in data.table: `DT[i, j, by]` <=> `DT[i, by, j]`. This is done below to better highlight the similarity with dplyr.

By group

```
# one-liner:
DT[, .(sumV2 = sum(V2)), by = "V4"]
# reordered and indented:
DT[, by = V4,
     .(sumV2 = sum(V2))]
#
```

```
DF %>%
  group_by(V4) %>%
  summarise(sumV2 = sum(V2))
```

By several groups

```
DT[, keyby = .(V4, V1),
     .(sumV2 = sum(V2))]
```

```
DF %>%
  group_by(V4, V1) %>%
  summarise(sumV2 = sum(V2))
```

Calling function in by

```
DT[, by = tolower(V4),
     .(sumV1 = sum(V1))]
```

```
DF %>%
  group_by(tolower(V4)) %>%
  summarise(sumV1 = sum(V1))
```

Assigning column name in by

```
DT[, keyby = .(abc = tolower(V4)),
     .(sumV1 = sum(V1))]
```

```
DF %>%
  group_by(abc = tolower(V4)) %>%
```

```
                                    summarise(sumV1 = sum(V1))
```

Using a condition in by

```
DT[, keyby = V4 == "A",
     sum(V1)]
```

```
DF %>%
  group_by(V4 == "A") %>%
  summarise(sum(V1))
```

By on a subset of rows

```
DT[1:5,                # i
   .(sumV1 = sum(V1)), # j
   by = V4]            # by
## complete DT[i, j, by] expression!
```

```
DF %>%
  slice(1:5) %>%
  group_by(V4) %>%
  summarise(sumV1 = sum(V1))
```

Count number of observations for each group

```
DT[, .N, by = V4]
```

```
count(DF, V4)
DF %>%
  group_by(V4) %>%
  tally()
DF %>%
  group_by(V4) %>%
  summarise(n())
DF %>%
  group_by(V4) %>%
  group_size() # returns a vector
```

Add a column with number of observations for each group

```
DT[, n := .N, by = V1][]
DT[, n := NULL] # rm column for
consistency
```

```
add_count(DF, V1)
DF %>%
  group_by(V1) %>%
  add_tally()
```

Retrieve the first/last/nth observation for each group

```
DT[, data.table::first(V2), by = V4]
DT[, data.table::last(V2), by = V4]
DT[, V2[2], by = V4]
```

```
DF %>%
  group_by(V4) %>%
  summarise(dplyr::first(V2))
DF %>%
  group_by(V4) %>%
  summarise(dplyr::last(V2))
DF %>%
  group_by(V4) %>%
  summarise(dplyr::nth(V2, 2))
```

Going further

Advanced columns manipulation

To further manipulate columns, dplyr includes nine functions: the `_all`, `_at`, and `_if` versions of `summarise()`, `mutate()`, and `transmute()`.
With data.table, we use `.SD`, which is a `data.table` containing the Subset of Data for each group, excluding the column(s) used in `by`. So, `DT[, .SD]` is `DT` itself and in the expression `DT[, .SD, by = V4]`, `.SD` contains all the DT columns (except V4) for each values in V4 (see `DT[, print(.SD), by = V4]`). `.SDcols` allows to select the columns included in `.SD`.

Summarise all the columns

```
DT[, lapply(.SD, max)]
```

```
summarise_all(DF, max)
```

Summarise several columns

```
DT[, lapply(.SD, mean),
     .SDcols = c("V1", "V2")]
# .SDcols is like "_at"
```

```
summarise_at(DF, c("V1", "V2"), mean)
```

Summarise several columns by group

```
DT[, by = V4,
     lapply(.SD, mean),
     .SDcols = c("V1", "V2")]
## using patterns (regex)
DT[, by = V4,
     lapply(.SD, mean),
     .SDcols = patterns("V1|V2")]
```

```
DF %>%
  group_by(V4) %>%
  summarise_at(c("V1", "V2"), mean)
## using select helpers
DF %>%
  group_by(V4) %>%
  summarise_at(vars(one_of("V1", "V2")),
mean)
```

Summarise with more than one function by group

```
DT[, by = V4,
     c(lapply(.SD, sum),
       lapply(.SD, mean))]
```

```
DF %>%
  group_by(V4) %>%
  summarise_all(list(sum, mean))
# columns named automatically
```

Summarise using a condition

```
cols <- names(DT)[sapply(DT,
is.numeric)]
DT[, lapply(.SD, mean),
     .SDcols = cols]
```

```
summarise_if(DF, is.numeric, mean)
```

Modify all the columns

```
DT[, lapply(.SD, rev)]
```

```
mutate_all(DF, rev)
# transmute_all(DF, rev)
```

Modify several columns (dropping the others)

```
DT[, lapply(.SD, sqrt),
     .SDcols = V1:V2]
DT[, lapply(.SD, exp),
     .SDcols = !"V4"]
```

```
transmute_at(DF, c("V1", "V2"), sqrt)
transmute_at(DF, vars(-V4), exp)
```

Modify several columns (keeping the others)

```
DT[, c("V1", "V2") := lapply(.SD, sqrt),
     .SDcols = c("V1", "V2")]
```

```
DF <- mutate_at(DF, c("V1", "V2"), sqrt)
DF <- mutate_at(DF, vars(-V4), "^", 2L)
```

```
cols <- setdiff(names(DT), "V4")
DT[, (cols) := lapply(.SD, "^", 2L),
     .SDcols = cols]
```

Modify columns using a condition (dropping the others)

```
cols <- names(DT)[sapply(DT,
is.numeric)]
DT[, .SD - 1,
     .SDcols = cols]
```

```
transmute_if(DF, is.numeric, list(~ '-
'(., 1L)))
```

Modify columns using a condition (keeping the others)

```
DT[, (cols) := lapply(.SD, as.integer),
     .SDcols = cols]
```

```
DF <- mutate_if(DF, is.numeric,
as.integer)
```

The use of `DT[,j]` is very flexible, allowing to pass complex expressions in a straightforward way, or combine expressions with multiple outputs.

Use a complex expression

```
DT[, by = V4,
     .(V1[1:2], "X")]
```

```
DF %>%
  group_by(V4) %>%
  slice(1:2) %>%
  transmute(V1 = V1,
            V2 = "X")
```

Use multiple expressions (with `DT[,{j}]`)

```
DT[, {print(V1) #  comments here!
      print(summary(V1))
      x <- V1 + sum(V2)
     .(A = 1:.N, B = x) # last list
returned as a data.table
     }]
```

```
#
```

Chain expressions

The dplyr workflow relies on the magrittr pipe operator (`%>%`). The magrittr package can also be used with `data.table` objects, but data.table comes with its own chaining system: `DT[ ... ][ ... ][ ... ]` .

Expression chaining using `DT[][]` (recommended)

```
DT[, by = V4,
     .(V1sum = sum(V1)) ][
     V1sum > 5]
```

```
DF %>%
  group_by(V4) %>%
```

```
  summarise(V1sum = sum(V1)) %>%
  filter(V1sum > 5)
```

Expression chaining using `%>%`

```
DT[, by = V4,
     .(V1sum = sum(V1))] %>%
  .[order(-V1sum)]
```

```
DF %>%
  group_by(V4) %>%
  summarise(V1sum = sum(V1)) %>%
  arrange(desc(V1sum))
```

Indexing and Keys

Row subsetting in dplyr relies on the `filter()` and `slice()` functions, as shown in the first section. With data.table, in addition to the above-mentioned approach, two systems are available to make row filtering and join operations more convenient and blazingly fast (~170x speed-up): *keys* (primary ordered index) and *indices* (automatic secondary indexing).

The main differences between keys and indices are:
- When using keys, data are physically reordered in memory. When using indices, the order is stored as an attribute.
- Only one key is possible but several indices can coexist.
- Keys are defined explicitly. Indices can be created manually but are also created on-the-fly (and stored when using `==` or `%in%`).
- Indices are used with the `on` argument. It is optional when using keys, but recommended (and used below) for better readability.
Note: in the following code, we set both a key and an index to demonstrate their usage, but internally, indices are not used when a key already exists for the same columns(s).

Set the key/index

```
setkey(DT, V4)
setindex(DT, V4)
```

```
DF <- arrange(DF, V4) # ordered just for
consistency
```

Select the matching rows

```
DT["A", on = "V4"]
DT[c("A", "C"), on = .(V4)] # same as on
= "V4"
```

```
filter(DF, V4 == "A")
filter(DF, V4 %in% c("A", "C"))
```

Select the first matching row

```
DT["B", on = "V4", mult = "first"]
DT[c("B", "C"), on = "V4", mult =
"first"]
```

```
DF %>%
  filter(V4 == "B") %>%
  slice(1)
# ?
```

Select the last matching row

```
DT["A", on = "V4", mult = "last"]
```

```
DF %>%
  filter(V4 == "A") %>%
  slice(n())
```

Nomatch argument

```
# (default) returns a row with "D" even
if not found
DT[c("A", "D"), on = "V4", nomatch = NA]
# no rows for unmatched values
DT[c("A", "D"), on = "V4", nomatch = 0]
```

```
#
filter(DF, V4 %in% c("A", "D"))
```

Apply a function on the matching rows

```
DT[c("A", "C"), sum(V1), on = "V4"]
```

```
DF %>%
  filter(V4 %in% c("A", "C")) %>%
  summarise(sum(V1))
```

Modify values for matching rows

```
DT["A", V1 := 0, on = "V4"]
DT
```

```
DF <- DF %>%
  mutate(V1 = base::replace(V1, V4 ==
"A", 0L)) %>%
  arrange(V4)
DF
```

Use keys in by

```
DT[!"B", sum(V1), on = "V4", by =
.EACHI]
DT[V4 != "B",
   by = V4,
   sum(V1)]   # same
```

```
DF %>%
  filter(V4 != "B") %>%
  group_by(V4) %>%
  summarise(sum(V1))
```

Set keys/indices for multiple columns

```
setkey(DT, V4, V1) # or setkeyv(DT,
c("V4", "V1"))
setindex(DT, V4, V1) # setindexv(DT,
c("V4", "V1"))
```

```
DF <- arrange(DF, V4, V1) # ordered just
for consistency
```

Subset using multiple keys/indices

```
DT[.("C", 1), on = .(V4, V1)]
DT[.(c("B", "C"), 1), on = .(V4, V1)]
# using which = TRUE only returns the
matching rows indices
DT[.(c("B", "C"), 1), on = .(V4, V1),
which = TRUE]
```

```
filter(DF, V1 == 1, V4 == "C")
filter(DF, V1 == 1, V4 %in% c("B", "C"))
# ?
```

Remove keys/indices

```
setkey(DT, NULL)
setindex(DT, NULL)
```

```
#
```

`set*()` modifications

In data.table, `set*()` functions modify objects by reference, making these operations fast and memory-efficient. In case this is not a desired behaviour, users can use `copy()`. The corresponding expressions in dplyr will be less memory-efficient.

Replace values

```
set(DT, i = 1L, j = 2L, value = 3L)
```

```
DF[1, 2] <- 3L
```

Reorder rows

```
setorder(DT, V4, -V1)
setorderv(DT, c("V4", "V1"), c(1, -1))
```

```
DF <- arrange(DF, V4, desc(V1))
```

Modify colnames

```
setnames(DT, old = "V2", new = "v2")
setnames(DT, old = -(c(1, 3)), new =
"V2")
```

```
DF <- rename(DF, v2 = V2)
DF <- rename(DF, V2 = v2) # reset upper
```

Reorder columns

```
setcolorder(DT, c("V4", "V1", "V2"))
```

```
DF <- select(DF, V4, V1, V2)
```

Convert data

```
?setDT # data.frame or list to
data.table
?setDF # data.table to data.frame
?setattr # modify attributes
```

```
#
```

Advanced use of by

Select first/last/... row by group

```
DT[, .SD[1], by = V4]
DT[, .SD[c(1, .N)], by = V4]
DT[, tail(.SD, 2), by = V4]
```

```
DF %>%
  group_by(V4) %>%
  slice(1)
DF %>%
  group_by(V4) %>%
  slice(1, n())
DF %>%
  group_by(V4) %>%
  group_map(~ tail(.x, 2))
```

Select rows using a nested query

```
DT[, .SD[which.min(V2)], by = V4]
```

```
DF %>%
  group_by(V4) %>%
  arrange(V2) %>%
  slice(1)
```

Add a group counter column

```
DT[, Grp := .GRP, by = .(V4, V1)][]
DT[, Grp := NULL] # delete for
consistency
```

```
DF %>% mutate(Grp = group_indices(., V4,
V1))
```

Get row number of first (and last) observation by group

```
DT[, .I, by = V4] # returns a data.table
DT[, .I[1], by = V4]
DT[, .I[c(1, .N)], by = V4]
```

```
DF %>%
  group_by(V4) %>%
  group_data() %>%
  tidyr::unnest(.rows)
# DF %>% group_by(V4) %>% group_rows() #
returns a list
#
#
```

Handle list-columns by group

```
DT[, .(.(V1)),  by = V4]  # return V1 as
a list
DT[, .(.(.SD)), by = V4] # subsets of
the data
```

```
DF %>%
  group_by(V4) %>%
  summarise(list(V1))
DF %>%
  group_by(V4) %>%
  group_nest()
```

Grouping sets (multiple by at once)

```
rollup(DT,
       .(SumV2 = sum(V2)),
       by = c("V1", "V4"))

rollup(DT,
       .(SumV2 = sum(V2), .N),
       by = c("V1", "V4"),
       id = TRUE)

cube(DT,
     .(SumV2 = sum(V2), .N),
     by = c("V1", "V4"),
     id = TRUE)

groupingsets(DT,
             .(SumV2 = sum(V2), .N),
             by   = c("V1", "V4"),
             sets = list("V1", c("V1",
"V4")),
             id   = TRUE)
```

```
#
```

Miscellaneous

Read / Write data

`fread()` and `fwrite()` are among the most powerful functions of data.table. They are not only incredibly fast (see benchmarks), they are also extremely robust. The few commands below only scratch the surface and there are a lot of awesome features. For example, `fread()` accepts `http` and `https` URLs directly as well as operating system commands such as `sed` and `awk` output. Make sure to check the docs.
Here again, `fread()` and `fwrite()` are very versatile and allow to handle different file formats while dplyr delegates file reading and writing to the readr package with several specific functions (csv, tsv, delim, ...).

Write data to a csv file

```
fwrite(DT, "DT.csv")
```

```
readr::write_csv(DF, "DF.csv")
# see also vroom
```

Write data to a tab-delimited file

```
fwrite(DT, "DT.txt", sep = "\t")
```

```
readr::write_delim(DF, "DF.txt", delim =
"\t")
```

Write list-column data to a csv file

```
fwrite(setDT(list(0, list(1:5))),
"DT2.csv")
```

```
#
```

Read a csv / tab-delimited file

```
fread("DT.csv")
# fread("DT.csv", verbose = TRUE) # full
details
fread("DT.txt", sep = "\t")
```

```
readr::read_csv("DF.csv")


readr::read_delim("DF.txt", delim =
"\t")
```

Read a csv file selecting / droping columns

```
fread("DT.csv", select = c("V1", "V4"))
fread("DT.csv", drop = "V4")
```

```
# NA
```

Read and rbind several files

```
rbindlist(lapply(c("DT.csv", "DT.csv"),
fread))
# c("DT.csv", "DT.csv") %>%
lapply(fread) %>% rbindlist
```

```
c("DF.csv", "DF.csv") %>%
  purrr::map_dfr(readr::read_csv)
```

Reshape data

This part is still a bit clunky. I need to find better examples. See here and here for more details.

Melt data (from wide to long)

```
melt(DT, id.vars = "V4")
mDT <- melt(DT,
            id.vars       = "V4",
            measure.vars  = c("V1",
"V2"),
            variable.name = "Variable",
            value.name    = "Value")
```

```
tidyr::gather(DF, variable, value, -V4)
mDF <- tidyr::gather(DF,
                     key = Variable,
                     value = Value,
                     -V4)
# pivot_longer todo
```

Cast data (from long to wide)

```
dcast(mDT, V4 ~ Variable) # aggregate by
count


dcast(mDT, V4 ~ Variable, fun.aggregate
= sum)


dcast(mDT, V4 ~ Value > 5)


# see ?dcast: multiple values /
fun.aggregate
```

```
tidyr::spread(data  = count(mDF, V4,
Variable),
              key   = Variable,
              value = n,
              fill  = 0)
# pivot_wider todo
```

Split

```
split(DT, by = "V4") # S3 method
```

```
group_split(DF, V4)
```

Split and transpose a vector/column

```
vec <- c("A:a", "B:b", "C:c")
tstrsplit(vec, split = ":", keep = 2L) #
works on vector
setDT(tstrsplit(vec, split = ":"))[]
```

```
vec <- c("A:a", "B:b", "C:c")
# vector not handled
tidyr::separate(tibble(vec), vec,
c("V1", "V2"))
```

Other

Check package installation

```
# test.data.table()
# There's more lines of test code in
data.table than there is code!
```

```
#
```

List data.tables/tibbles

```
tables()
```

```
#
```

Get/Set number of threads when parallelized

```
getDTthreads() # setDTthreads()            #
```

Lead/Lag

```
shift(1:10, n = 1,   fill = NA, type =      lag(1:10, n = 1, default = NA)
"lag")                                      purrr::map(1:2, ~lag(1:10, n = .x))
shift(1:10, n = 1:2, fill = NA, type =      lead(1:10, n = 1, default = NA)
"lag") # multiple
shift(1:10, n = 1,   fill = NA, type =
"lead")
```

Generate run-length ids

```
rleid(rep(c("a", "b", "a"), each = 3)) #    #
see also ?rleidv
rleid(rep(c("a", "b", "a"), each = 3),
prefix = "G")
```

Vectorised `ifelse` statements

```
#                                           x <- 1:10
                                            case_when(
                                              x %% 6 == 0 ~ "fizz buzz",
                                              x %% 2 == 0 ~ "fizz",
                                              x %% 3 == 0 ~ "buzz",
                                              TRUE ~ as.character(x)
                                            )
```

Rolling functions

```
# todo                                      #
```

Join/Bind data sets

Join

Joining data in data.table works like the fast subsetting approach described above. It can be performed using keys, using the *ad hoc* `on` argument, or using the `merge.data.table` method. For the sake of completeness, the three methods are presented below. As previously mentioned, the `on` and `by` (in `merge`) arguments are optional with keyed data.tables, but recommended to make the code more explicit.
In the examples below, the `x`, `y`, and `z` data.tables are also used with dplyr.

```
x <- data.table(Id  = c("A", "B", "C", "C"),
                X1  = c(1L, 3L, 5L, 7L),
                XY  = c("x2", "x4", "x6", "x8"),
                key = "Id")

y <- data.table(Id  = c("A", "B", "B", "D"),
                Y1  = c(1L, 3L, 5L, 7L),
                XY  = c("y1", "y3", "y5", "y7"),
                key = "Id")
```

Join matching rows from y to x

```
y[x, on = "Id"]                          left_join(x, y, by = "Id")
merge(x, y, all.x = TRUE, by = "Id")
y[x] # requires keys
```

Join matching rows from x to y

```
x[y, on = "Id"]                          right_join(x, y, by = "Id")
merge(x, y, all.y = TRUE, by = "Id")
x[y] # requires keys
```

Join matching rows from both x and y

```
x[y, on = "Id", nomatch = 0]             inner_join(x, y, by = "Id")
merge(x, y)
x[y, nomatch = 0] # requires keys
```

Join keeping all the rows

```
merge(x, y, all = TRUE, by = "Id")       full_join(x, y, by = "Id")
```

Return rows from x matching y

```
unique(x[y$Id, on = "Id", nomatch = 0])  semi_join(x, y, by = "Id")
unique(x[y$Id, nomatch = 0]) # requires
keys
```

Return rows from x not matching y

```
x[!y, on = "Id"]                         anti_join(x, y, by = "Id")
x[!y] # requires keys
```

More joins

data.table excels at joining data, and offers additional functions and features.

Select columns while joining

```
x[y, .(Id, X1, i.XY)]   # i. prefix      right_join(select(x, Id, X1),
refers to cols in y                                 select(y, Id, XY),
                                                    by = "Id")
```

```
x[y, .(Id, x.XY, i.XY)] # x. prefix refers to cols in x
```

```
right_join(select(x, Id, XY),
           select(y, Id, XY),
           by = "Id")
```

Aggregate columns while joining

```
y[x, .(X1Y1 = sum(Y1) * X1), by = .EACHI]
```

```
y %>%
 group_by(Id) %>%
 summarise(SumY1 = sum(Y1)) %>%
 right_join(x) %>%
 mutate(X1Y1 = SumY1 * X1) %>%
 select(Id, X1Y1)
```

Update columns while joining

```
y[x, SqX1 := i.X1^2]
y[, SqX1 := x[.BY, X1^2, on = "Id"], by = Id] # more memory-efficient
y[, SqX1 := NULL] # rm column for consistency
```

```
x %>%
 select(Id, X1) %>%
 mutate(SqX1 = X1^2) %>%
 right_join(y, by = "Id") %>%
 select(names(y), SqX1)
```

Adds a list column with rows from y matching x (nest-join)

```
x[, y := .(.(y[.BY, on = "Id"])), by = Id]
x[, y := NULL] # rm column for consistency
```

```
nest_join(x, y, by = "Id")
```

Update columns while joining (using vectors of colnames)

```
cols  <- c("NewXY", "NewX1")
icols <- paste0("i.", c("XY", "X1"))

y[x, (cols) := mget(icols)]

y[, (cols) := NULL] # rm columns for consistency
```

```
# ?
```

Join passing columns to match in the `on` argument

```
z <- data.table(ID = "C", Z1 = 5:9, Z2 = paste0("z", 5:9))
x[, X2 := paste0("x", X1)] # used to track the results
z; x
```

```
x[z, on = "X1 == Z1"]
x[z, on = .(X1 == Z1)] # same
x[z, on = .(Id == ID, X1 == Z1)] # using two columns
```

```
right_join(x, z, by = c("X1" = "Z1"))
right_join(x, z, by = c("Id" = "ID", "X1" = "Z1"))
```

Non-equi joins

```
x[z, on = .(Id == ID, X1 <= Z1)]
x[z, on = .(Id == ID, X1 > Z1)]
x[z, on = .(X1 < Z1), allow.cartesian =
```

```
#
```

```
TRUE] # allows 'numerous' matching
values
```

Rolling joins/subsets (performed on the last numeric column)

```
# Nearest                                    #
x[z, on = .(Id == ID, X1 == Z1), roll =
"nearest"]
## below, simplified examples with ad
hoc subsets on a keyed data.table
setkey(x, Id, X1)
x[.("C", 5:9), roll = "nearest"]


# Last Observation Carried Forward           #
x[.("C", 5:9), roll = Inf]
x[.("C", 5:9), roll = 0.5]  # bounded
x[.("C", 5:9), roll = Inf, rollends =
c(FALSE, TRUE)]  # default
x[.("C", 5:9), roll = Inf, rollends =
c(FALSE, FALSE)] # ends not rolled


# Next Observation Carried Backward          #
x[.("C", 5:9), roll = -Inf]
x[.("C", 5:9), roll = -0.5] # bounded
x[.("C", 5:9), roll = -Inf, rollends =
c(TRUE, FALSE)]
x[.("C", 5:9), roll = -Inf, rollends =
c(TRUE, TRUE)]  # roll both ends
```

Cross join (`CJ ~ expand.grid`)

```
CJ(c(2, 1, 1), 3:2)                        # base::expand.grid(c(2, 1, 1), 3:2)
CJ(c(2, 1, 1), 3:2, sorted = FALSE,        #
unique = TRUE)
```

Overlap join

It is important to mention `foverlaps()` from data.table that allows to perform 'overlap joins'. This is a very powerful function, but a bit out of scope for this document. See these slides for more details.

Bind

```
x <- data.table(1:3)
y <- data.table(4:6)
z <- data.table(7:9, 0L)
```

Bind rows

```
rbind(x, y)                                bind_rows(x, y)
rbind(x, z, fill = TRUE)                   bind_rows(x, z) # always fills
```

Bind rows using a list

```
rbindlist(list(x, y), idcol = TRUE)
```

```
bind_rows(list(x, y), .id = "id")
```

Bind columns

```
base::cbind(x, y)
```

```
bind_cols(x, y)
```

Set operations

```
x <- data.table(c(1, 2, 2, 3, 3))
y <- data.table(c(2, 2, 3, 4, 4))
```

Intersection

```
fintersect(x, y)
fintersect(x, y, all = TRUE)
```

```
dplyr::intersect(x, y)
# no all option
```

Difference

```
fsetdiff(x, y)
fsetdiff(x, y, all = TRUE)
```

```
dplyr::setdiff(x, y)
# no all option
```

Union

```
funion(x, y)
funion(x, y, all = TRUE)
```

```
dplyr::union(x, y)
union_all(x, y)
```

Equality

```
fsetequal(x, x[order(-V1),])
all.equal(x, x) # S3 method
```

```
setequal(x, x[order(-V1),])
all_equal(x, x)
```

Summary

This article presented the most important features of both data.table and dplyr, two packages that are now essential tools for data manipulation in R.
There are still a lot of features not covered in this document, in particular, data.table functions to deal with time-series or dplyr vectorized functions have not been discussed, but 'done is better than perfect'...
Hopefully, this comparison is not too biased, but I must admit that my preference is for data.table. So, I hope this post will encourage some readers to give it a try!