**Introduction**

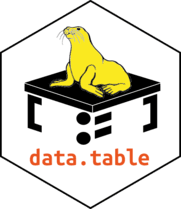
It is Christmas time again!, what better time than this to write about the great tools that are available to all interested in working with R. This post is meant as a praise to a few selected tools and packages that helped me to be more efficient and productive with R in 2019.

In this post, we will praise free tools that can help your work become more efficient, reproducible and productive, namely the data.table package, the Rocker project for R-based Docker images, the base package parallel, and the R-Hub service for package checking.

**Contents**

1. [data.table – R’s unsung powerhouse](https://jozef.io/r920-christmas-praise-2019/#data.table---rs-unsung-powerhouse)
2. [The Rocker project for R-based Docker images](https://jozef.io/r920-christmas-praise-2019/#the-rocker-project-for-r-based-docker-images)
3. [Base package Parallel](https://jozef.io/r920-christmas-praise-2019/#base-package-parallel)
4. [Rhub for fast and automated multi-platform R package testing](https://jozef.io/r920-christmas-praise-2019/#rhub-for-fast-and-automated-multi-platform-r-package-testing)
5. [Resources](https://jozef.io/r920-christmas-praise-2019/#resources)

**data.table – R’s unsung powerhouse**

One of the packages I find most under-marketed and under-appreciated in the R package ecosystem is [data.table](https://cran.r-project.org/web/packages/data.table/index.html). If it is mentioned, it is mostly for its speed and memory efficiency, which is certainly well deserved, but I feel dismissing the other benefits and features is not doing it justice. Here are a few points that I like about data.table that do not get that much exposure.

**The concise and generic syntax**

I enjoy the fact that data.table’s syntax is very concise and principle-driven. In effect, all you need for most common use cases is to learn using the [] brackets and an amazing world of opportunities will follow. Just one small example on taking 2 data tables, joining them on their common columns, filtering on rows, summarizing a variable grouped by an evaluated expression on 1 line:

# Prepare the packages and data

library(data.table)

flts <- as.data.table(nycflights13::flights)

wthr <- as.data.table(nycflights13::weather)

byCols <- intersect(names(flts), names(wthr))

# Join, filter, group by and aggregate

wthr[flts, on = byCols][origin == "JFK", mean(dep\_delay, na.rm = TRUE), precip > 0]

## precip V1

## 1: FALSE 10.92661

## 2: NA 13.66543

## 3: TRUE 29.70753

**Fully featured data wrangling toolbox**

And this is just scratching the surface as data.table also provides functions such as

* dcast() and melt() for efficient data reshaping
* rbindlist() for fast replacement of do.call("rbind", l)
* fsetdiff(), fintersect(), funion() and fsetequal() for fast and easy to use operations on data.tables
* rollup(), cube() and groupingsets() to create pivot tables, more on that [in a dedicated article](https://jozef.io/r912-datatable-grouping-sets/)

**No dependencies**

All in all, I consider data.table to be a single package that brings speed, efficiency and conciseness to all data wrangling operations. Another benefit that also often stays unmentioned is the fact that data.table has no dependencies on other non-base R packages, which is beneficial for maintenance, stability, reproducibility, size and deployment speeds.

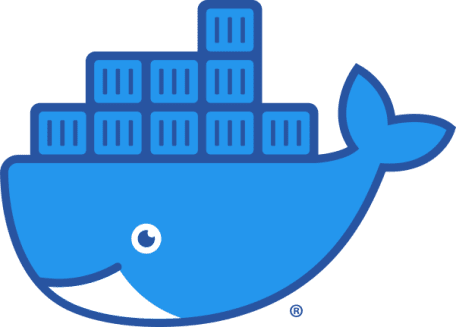
**Fast reading and writing of (compressed) csvs**

One additional feature of data.table that I use regularly is the ability to read and write data to and from text files with amazing speeds using the fread() and fwrite() functions.

Not only is it very fast and convenient, but thanks to a recently added feature, data.table now supports fwrite() directly to gzipped csvs, which saves significant space when writing large amounts of data.

For getting started with data.table, I recommend the [Introduction to data.table vignette](https://cloud.r-project.org/web/packages/data.table/vignettes/datatable-intro.html)

**The Rocker project for R-based Docker images**

Containerization is a powerful and useful tool for many purposes, one of them being reproducibility. In the R world, ensuring that our R library contains the exact versions of packages we need can be achieved by using tools such as [packrat](https://cran.r-project.org/package=packrat) or its successor [renv](https://cran.r-project.org/package=renv).

Managing the R package versions can however only get us so far, especially when relying on other system dependencies such as pandoc for rendering our R Markdown documents or Java. And when we need to test our R applications against multiple versions of R itself, things can get very tedious and messy very quickly using just one environment, especially on UNIX-based platforms.

In comes the [Rocker project – Docker Containers for the R Environment](https://www.rocker-project.org/). Thanks to the efforts of Carl Boettiger, Dirk Eddelbuettel, and Noam Ross, spinning a container with a specific version of R, RStudio or even the tidyverse packages is as easy as launching a terminal and running

docker run --rm -ti rocker/r-base

Want to test your R code using an older version of R, say some *Very, Very Secure Dishes* from 2016? As easy as

docker run --rm -ti rocker/r-ver:3.2.5

Even more usefully, all the sources to build the Docker images, so we can adapt the images for our own usage. For instance

* the series of articles on [Using Spark from R for performance with arbitrary code](https://jozef.io/tags/spark/) on this blog uses a setup adapted from the rocker/r-ver:3.6.1 image
* we have also used the images provided by the Rocker project when setting up continuous [multi-platform R package building, checking and testing](https://jozef.io/r107-multiplatform-gitlabci-rhub/#preparing-a-private-docker-image-to-use-with-r-hub) with R-Hub
* even to keep the building of this very website stable and reproducible, a Docker image based on the Rocker project is used

On a more generic note, learning Docker is beneficial to R users also when working outside R and there are many great learning resources to do so. For learning Docker I recommend the [Get started documentation](https://docs.docker.com/get-started/).

**Base package parallel**

The internals of the R language are single-threaded, meaning that when writing R code, unless optimized for multi-threaded computation under the hood such as data.table does, our code will only utilize 1 thread, which can pose a challenge to performance even in common daily tasks, especially now that even common, very portable ultrabooks come with processors with 4 or more cores and 8 or more threads.

The R ecosystem provides many ways to take advantage of the multiple threads available. In this post I would like to give more visibility to the parallelization options that come with the base R installation itself, not requiring any extra external dependencies or packages – via the package parallel.

In a very small showcase, let’s look at how much faster we can execute a brute-force-ish solution to the [Longest Collatz sequence](https://projecteuler.net/problem=14) problem for the first 10 million numbers. First, define the function that will compute the sequence length for a given integer n:

col\_len <- function(n) {

len <- 0L

while (n > 1) {

len <- len + 1L

if ((n %% 2) == 0)

n <- n / 2

else {

n <- (n \* 3 + 1) / 2

len <- len + 1L

}

}

len

}

Running the function for numbers from 1 to 9,999,999 using sapply() and measuring the time on this particular laptop shown that the process finished in around 580 seconds – almost 10 minutes:

max(sapply(seq(from = 1, to = 9999999), col\_len))

## [1] 8400511

Now we will create a simple cluster on the local machine using all available threads and send the function definition to all the created worker processes:

# Attach the parallel package

library(parallel)

# Create a cluster using all available threads

cl <- makeCluster(detectCores(), methods = FALSE)

# Send the definition of the col\_len function to the workers

clusterExport(cl, "col\_len")

Next, we execute the function in parallel using the cluster. It is as simple as just using parSapply() instead of sapply() and providing the cluster definition cl as the first argument:

# Execute in parallel using cluster cl

max(parSapply(cl, seq(from = 1, to = 9999999), col\_len))

## [1] 8400511

After the process is done, it is good practice to stop the cluster:

# Stopping the cluster

stopCluster(cl)

Using all 8 available threads the time needed to execute the code and get the same results went down to around 90 seconds or 1.5 minutes. We can therefore gain significant time savings using base R executing some of your code in parallel, adjusting the code very minimally and using very faimilar syntax.

For more information on using the parallel package, I recommend reading the package’s vignette by running vignette("parallel") or reading [online](https://stat.ethz.ch/R-manual/R-patched/library/parallel/doc/parallel.pdf). For more information on High-Performance and Parallel Computing with R, there is a dedicated [CRAN Task View](https://cran.r-project.org/web/views/HighPerformanceComputing.html).

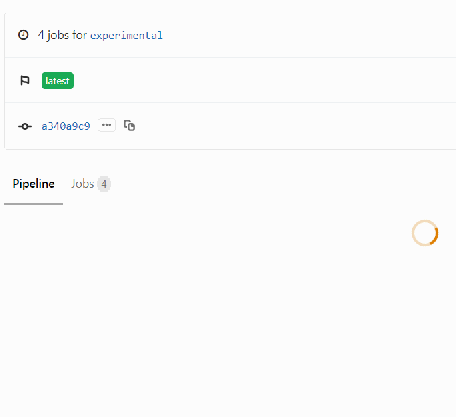
**Rhub for fast and automated multi-platform R package testing**

[R-hub](https://r-hub.github.io/rhub/index.html) offers free R CMD check as a service on different platforms. This enables R developers to quickly and efficiently check their R packages to make sure they pass all necessary checks on several platforms. As a bonus, the checks seem to be running in a very short time, which means we can have your results at hand in a few minutes.

Using R-hub interactively is as simple as installing the [rhub package](https://cran.r-project.org/package=rhub) from CRAN, validating your e-mail by running rhub::validate\_email() and running:

cr <- rhub::check()

In an interactive session, this will offer a list of platforms to choose from and check our package against them.



CI/CD running checks on multiple platforms with R-hub