[jozef.io](https://jozef.io/r917-fread-comparisons/)

**How data.table's fread can save you a lot of time and memory, and take input from shell commands - Jozef's Rblog**

6-8 minutes

Recently I was involved in a task that included reading and writing quite large amounts of data, totaling more than 1 TB worth of csvs without the standard big data infrastructure. After trying multiple approaches, the one that made this possible was using data.table’s reading and writing facilities - fread() and fwrite().

This motivated me to look at benchmarking data.table’s fread() and how it compares to other packages such as tidyverse’s readr and base R for reading tabular data from text files such as csvs.

The data.table package is a bit lesser known in the R community, but if people know it, it is most likely for its speed when working with data tables themselves within R. The package however also provides functions for efficient reading and writing of tabular data from and into text files - fread() for fast reading and fwrite() for fast writing.

Another underrated property of the fread() apart from speed however is memory efficiency, which can be crucial if we need to read in a lot of data without big data infrastructure.

**The benchmarked data**

As the data for this quick benchmark, we used the [Airline on-time performance](http://stat-computing.org/dataexpo/2009/the-data.html) data from for years 2000 to 2008. This simple code chunk can be used to retrieve and extract the data. The download size is 868 MB in bz2 files. The extracted size is 5.34 GB in csv files and when combined translates to a data frame with some 59 million rows and 29 columns. This is quite limited due to the specs of the machine used, but enough to show significant differences between packages.

destDir <- path.expand("~/dataexpo")

years <- 2000:2008

baseUrl <- "http://stat-computing.org/dataexpo/2009"

bz2Names <- file.path(destDir, paste0(years, ".csv.bz2"))

dlUrls <- file.path(baseUrl, paste0(years, ".csv.bz2"))

if (!dir.exists(destDir)) {

dir.create(destDir, recursive = TRUE)

}

mapply(download.file, dlUrls, bz2Names)

system(paste0(

"cd ", destDir, "; ",

"bzip2 -d -k ", paste(bz2Names, collapse = " ")

))

**Base R code to be benchmarked**

Loading csv data from multiple files into a single data frame with base R is very simple:

dataDir <- path.expand("~/dataexpo")

dataFls <- dir(dataDir, pattern = "csv$", full.names = TRUE)

df <- do.call(rbind, lapply(dataFls, read.csv))

**data.table fread code to be benchmarked**

For data.table, we use rbindlist() for row binding instead of do.call(rbind, ...) and fread() for reading:

library(data.table)

dataDir <- path.expand("~/dataexpo")

dataFls <- dir(dataDir, pattern = "csv$", full.names = TRUE)

dt <- data.table::rbindlist(

lapply(dataFls, data.table::fread, showProgress = FALSE)

)

**readr::read\_csv code to be benchmarked**

The script for readr’s read\_csv is also simple, with the small caveat that we need to predefine the column types, as rbind\_rows does not like to coerce the data. Doing things the tidyverse way, we also use purrr::map\_dfr() to for row binding and readr::read\_csv() for reading:

library(readr)

library(purrr)

library(magrittr)

dataDir <- path.expand("~/dataexpo")

dataFiles <- dir(dataDir, pattern = "csv$", full.names = TRUE)

col\_types <- readr::cols(

.default = col\_double(),

UniqueCarrier = col\_character(),

TailNum = col\_character(),

Origin = col\_character(),

Dest = col\_character(),

CancellationCode = col\_character(),

CarrierDelay = col\_double(),

WeatherDelay = col\_double(),

NASDelay = col\_double(),

SecurityDelay = col\_double(),

LateAircraftDelay = col\_double()

)

df <- dataFiles %>%

purrr::map\_dfr(

readr::read\_csv,

col\_types = col\_types,

progress = FALSE

)

**The benchmarking method**

A simple bash script was used to measure the maximum memory needed (Maximum resident set size to be precise) and to time the run of the script 10 times:

#!/bin/bash

scriptf=$1

printf "$scriptf \n\n"

/usr/bin/time -v Rscript $scriptf \

2>&1 >/dev/null | \

grep -E 'Maximum resident'

time for i in {1..10}; do Rscript $scriptf >/dev/null; done

The results speak for themselves. Not only was fread() almost 2.5 times faster than readr’s functionality in reading and binding the data, but perhaps even more importantly, the maximum used memory was only 15.25 GB, compared to readr’s 27 GB. Interestingly, even though very slow, base R also spent less memory than the tidyverse suite.

For larger data sets, data.table’s efficiency can save not only very significant amounts of time, but also needed memory, which can have important implications with regards to the cost of the hardware needed for processing.

| **method** | **max. memory** | **avg. time** |
| --- | --- | --- |
| utils::read.csv + base::rbind | 21.70 GB | 8.13 m |
| readr::read\_csv + purrr:map\_dfr | 27.02 GB | 3.43 m |
| data.table::fread + rbindlist | 15.25 GB | 1.40 m |

And it gets better than that. Consider a scenario where we need to read the data, subset or split into groups and compute on the processed data. The classic approach would be to load the data from files into R as seen above and then do the data processing.

For scenarios like these, fread() provides an ever more powerful facility - the cmd argument with a shell command that pre-processes the file(s). If we want to filter our data used above to only look at flights operated by American Airlines the classic approach would be to read the data in and filter. With fread() we can, however, use grep first and only have fread() process output of that command:

library(data.table)

dataDir <- path.expand("~/dataexpo")

dataFiles <- dir(dataDir, pattern = "csv$", full.names = TRUE)

command <- sprintf(

"grep --text ',AA,' %s",

paste(dataFiles, collapse = " ")

)

dt <- data.table::fread(cmd = command)

Looking at our benchmarks, this approach only cost us 1.68GB of memory and about 24 seconds of runtime on average:

| **method** | **max. memory** | **avg. time** |
| --- | --- | --- |
| data.table::fread from grep | 1.68 GB | 0.40 m |

The above is of course only the beginning of potential optimizations. We could probably save a lot of time taking advantage of [GNU parallel](https://www.gnu.org/software/parallel/) to process the files with grep much faster. The key here is the flexibility of inputs that fread can process, without splitting the inputs into multiple files and other maintenance-heavy pre-processing.

In a bigger data setting, this can have a significant impact on the cost of a data science project and even investments in big data infrastructure, engineers and maintenance related to managing such a project.