There is interest in converting relational query languages (that work both over SQL databases and on local data) into data.table commands, to take advantage of data.table‘s superior performance. Obviously if one wants to use data.table it is best to learn data.table. But if we want code that can run multiple places a translation layer may be in order.

In this note we look at how this translation is commonly done.

The [dtplyr](https://cran.r-project.org/package=dtplyr) recently announced they are making changes to dtplyr to support two operation modes:

Note that there are two ways to use dtplyr:

* Eagerly [WIP]. When you use a dplyr verb directly on a data.table object, it  
  eagerly converts the dplyr code to data.table code, runs it, and returns a  
  new data.table. This is not very efficient because it can’t take advantage  
  of many of data.table’s best features.
* Lazily. In this form, trigged by using lazy\_dt(), no computation is  
  performed until you explicitly request it with as.data.table(),  
  as.data.frame() or as\_tibble(). This allows dtplyr to inspect the  
  full sequence of operations to figure out the best translation.

Reference

|  |
| --- |
| --- |
|  | title: "Translation" |
|  | output: rmarkdown::html\_vignette |
|  | vignette: > |
|  | %\VignetteIndexEntry{translation} |
|  | %\VignetteEngine{knitr::rmarkdown} |
|  | %\VignetteEncoding{UTF-8} |
|  | --- |
|  |  |
|  | ```{r, include = FALSE} |
|  | knitr::opts\_chunk$set( |
|  | collapse = TRUE, |
|  | comment = "#>" |
|  | ) |
|  | ``` |
|  |  |
|  | **# Introduction** |
|  |  |
|  | This vignette shows the details of how dtplyr translates dplyr expressions into [data.table](https://github.com/Rdatatable/data.table/wiki) code. If you see places where you think I could generate better data.table code, please let me know! |
|  |  |
|  | ```{r setup, message = FALSE} |
|  | library(dtplyr) |
|  | library(data.table) |
|  | library(dplyr) |
|  | ``` |
|  |  |
|  | Note that there are two ways to use dtplyr: |
|  |  |
|  | \* Eagerly [WIP]. When you use a dplyr verb directly on a data.table object, it |
|  | eagerly converts the dplyr code to data.table code, runs it, and returns a |
|  | new data.table. This is not very efficient because it can't take advantage |
|  | of many of data.table's best features. |
|  |  |
|  | \* Lazily. In this form, trigged by using `lazy\_dt()`, no computation is |
|  | performed until you explicitly request it with `as.data.table()`, |
|  | `as.data.frame()` or `as\_tibble()`. This allows dtplyr to inspect the |
|  | full sequence of operations to figure out the best translation. |
|  |  |
|  | **# The basics** |
|  |  |
|  | To get started, I'll create a simple lazy frame. The actual data doesn't matter here since we're just looking at the translation: |
|  |  |
|  | ```{r} |
|  | df <- data.frame(a = integer(), b = integer(), c = integer(), d = integer()) |
|  | dt <- lazy\_dt(df) |
|  | ``` |
|  |  |
|  | When we print it out, it tells us that it's a local data table with four rows. It also prints the call that dtplyr will evaluate when we execute the lazy table. In this case it's very simple: |
|  |  |
|  | ```{r} |
|  | dt |
|  | ``` |
|  |  |
|  | If we just want to see the generated code, you can use `show\_query()`. I'll use that a lot in this vignette. |
|  |  |
|  | ```{r} |
|  | dt %>% show\_query() |
|  | ``` |
|  |  |
|  |  |
|  | **# Simple verbs** |
|  |  |
|  | Many dplyr verbs have a straightforward translation to either the `i` or `j` component of `[.data.table`. |
|  |  |
|  |  |
|  | **## `i`** |
|  |  |
|  | `filter()` and `arrange()` become elements of `i`: |
|  |  |
|  | ```{r} |
|  | dt %>% arrange(a, b, c) %>% show\_query() |
|  | dt %>% filter(b == c) %>% show\_query() |
|  | dt %>% filter(b == c, c == d) %>% show\_query() |
|  | ``` |
|  |  |
|  | **## `j`** |
|  |  |
|  | `select()`, `rename()`, `summarise()` and `transmute()` all become elements of `j`: |
|  |  |
|  | ```{r} |
|  | dt %>% select(a:b) %>% show\_query() |
|  | dt %>% rename(x = a, y = b) %>% show\_query() |
|  | dt %>% summarise(a = mean(a)) %>% show\_query() |
|  | dt %>% transmute(a2 = a \* 2) %>% show\_query() |
|  | ``` |
|  |  |
|  | `mutate()` also uses the `j` component with data.table's special `:=` operator: |
|  |  |
|  | ```{r} |
|  | dt %>% mutate(a2 = a \* 2, b2 = b \* 2) %>% show\_query() |
|  | ``` |
|  |  |
|  | Note that dplyr never modifies the input data, so here it automatically `copy()`s the input data.table. |
|  |  |
|  | `mutate()` allows to refer to variables that you just created. data.tables `:=` doesn't support that out of the box, so we automatically chain together as many `[` as needed: |
|  |  |
|  | ```{r} |
|  | dt %>% mutate(a2 = a \* 2, b2 = b \* 2, a4 = a2 \* 2) %>% show\_query() |
|  | ``` |
|  |  |
|  | `transmute()` works similarly: |
|  |  |
|  | ```{r} |
|  | dt %>% transmute(a2 = a \* 2, b2 = b \* 2, a4 = a2 \* 2) %>% show\_query() |
|  | ``` |
|  |  |
|  | **## Grouping** |
|  |  |
|  | Just like in dplyr, `group\_by()` doesn't do anything by itself, but instead modifies the operation of downstream verbs. This generally just involves using the `by` argument: |
|  |  |
|  | ```{r} |
|  | dt %>% group\_by(a) %>% summarise(b = mean(b)) %>% show\_query() |
|  | ``` |
|  |  |
|  | The primary exception is grouped `filter()`, which requires the use of `.SD` () |
|  |  |
|  | ```{r} |
|  | dt %>% group\_by(a) %>% filter(b < mean(b)) %>% show\_query() |
|  | ``` |
|  |  |
|  | **# Combinations** |
|  |  |
|  | dtplyr tries to generate generate data.table code as close to what you'd write by hand as possible, as this tends to unlock data.tables tremendous speed. For example, if you `filter()` and then `select()`, dtplyr generates a single `[`: |
|  |  |
|  | ```{r} |
|  | dt %>% filter(a == 1) %>% select(-a) %>% show\_query() |
|  | ``` |
|  |  |
|  | And similarly for filtering and summarising: |
|  |  |
|  | ```{r} |
|  | dt %>% |
|  | group\_by(a) %>% |
|  | filter(b < mean(b)) %>% |
|  | summarise(c = max(c)) %>% |
|  | show\_query() |
|  | ``` |
|  |  |
|  | Note however, that `select()`ing and then `filter()`ing must generate two separate calls to `[`, because data.table evaluates `i` before `j`. |
|  |  |
|  | ```{r} |
|  | dt %>% select(-a) %>% filter(a == 1) %>% show\_query() |
|  |  |
|  | ``` |

This is a bit confusing, but we can unroll it a bit.

* The first “eager” method is how dplyr (and later dtplyr) has *always* converted dplyr pipelines into data.table realizations.  
  It is odd to mark this as “WIP” (work in progress?), as this has been dplyr‘s strategy since the first released version of dplyr.
* The second “lazy” method is the *proper* way to call data.table. Our own rqdatatable package has been calling data.table this way for over a year. It is *very* odd that dplyr didn’t use this good strategy for the data.table adaptor, as it is the strategy dplyr uses in its SQL adaptor.

Let’s take a look at the current published version of dtplyr (0.0.3) and how its eager evaluation works. Consider the following 4 trivial functions: that each add one to a data.frame column multiple times.

base\_r\_fn <- function(df) {

dt <- df

for(i in seq\_len(nstep)) {

dt$x1 <- dt$x1 + 1

}

dt

}

dplyr\_fn <- function(df) {

dt <- df

for(i in seq\_len(nstep)) {

dt <- mutate(dt, x1 = x1 + 1)

}

dt

}

dtplyr\_fn <- function(df) {

dt <- as.data.table(df)

for(i in seq\_len(nstep)) {

dt <- mutate(dt, x1 = x1 + 1)

}

dt

}

data.table\_fn <- function(df) {

dt <- as.data.table(df)

for(i in seq\_len(nstep)) {

dt[, x1 := x1 + 1]

}

dt[]

}

base\_r\_fn() is idiomatic R code, dplyr\_fn() is idiomatic dplyr code, dtplyr\_fn() is a idiomatic dplyr code operating over a data.table object (hence using dtplyr), and data.table\_fn() is idiomatic data.table code.

When we time running all of these functions operating on a 100000 row by 100 column data frame for 1000 steps we see each of them takes the following time to complete the task on average:

method mean\_seconds

1: base\_r 0.8367011

2: data.table 1.5592681

3: dplyr 2.6420171

4: dtplyr 151.0217646

The “eager” dtplyr system is about 100 times slower than data.table. This trivial task is one of the *few* times that data.table isn’t by far the fastest implementation (in tasks involving grouped summaries, joins, and other non-trivial operations data.table typically has a large performance advantage).

Reference

**Code to reproduce the timings above :**

## data.table run

$ R --vanilla

require(data.table)

N=2e9; K=100

set.seed(1)

DT <- data.table(

id1 = sample(sprintf("id%03d",1:K), N, TRUE), # large groups (char)

id2 = sample(sprintf("id%03d",1:K), N, TRUE), # large groups (char)

id3 = sample(sprintf("id%010d",1:(N/K)), N, TRUE), # small groups (char)

id4 = sample(K, N, TRUE), # large groups (int)

id5 = sample(K, N, TRUE), # large groups (int)

id6 = sample(N/K, N, TRUE), # small groups (int)

v1 = sample(5, N, TRUE), # int in range [1,5]

v2 = sample(5, N, TRUE), # int in range [1,5]

v3 = sample(round(runif(100,max=100),4), N, TRUE) # numeric e.g. 23.5749

)

cat("GB =", round(sum(gc()[,2])/1024, 3), "\n")

system.time( DT[, sum(v1), keyby=id1] )

system.time( DT[, sum(v1), keyby=id1] )

system.time( DT[, sum(v1), keyby="id1,id2"] )

system.time( DT[, sum(v1), keyby="id1,id2"] )

system.time( DT[, list(sum(v1),mean(v3)), keyby=id3] )

system.time( DT[, list(sum(v1),mean(v3)), keyby=id3] )

system.time( DT[, lapply(.SD, mean), keyby=id4, .SDcols=7:9] )

system.time( DT[, lapply(.SD, mean), keyby=id4, .SDcols=7:9] )

system.time( DT[, lapply(.SD, sum), keyby=id6, .SDcols=7:9] )

system.time( DT[, lapply(.SD, sum), keyby=id6, .SDcols=7:9] )

## dplyr run

$ R --vanilla

require(dplyr)

N=2e9; K=100

set.seed(1)

DF <- data.frame(stringsAsFactors=FALSE,

id1 = sample(sprintf("id%03d",1:K), N, TRUE),

id2 = sample(sprintf("id%03d",1:K), N, TRUE),

id3 = sample(sprintf("id%010d",1:(N/K)), N, TRUE),

id4 = sample(K, N, TRUE),

id5 = sample(K, N, TRUE),

id6 = sample(N/K, N, TRUE),

v1 = sample(5, N, TRUE),

v2 = sample(5, N, TRUE),

v3 = sample(round(runif(100,max=100),4), N, TRUE)

)

cat("GB =", round(sum(gc()[,2])/1024, 3), "\n")

system.time( DF %>% group\_by(id1) %>% summarise(sum(v1)) )

system.time( DF %>% group\_by(id1) %>% summarise(sum(v1)) )

system.time( DF %>% group\_by(id1,id2) %>% summarise(sum(v1)) )

system.time( DF %>% group\_by(id1,id2) %>% summarise(sum(v1)) )

system.time( DF %>% group\_by(id3) %>% summarise(sum(v1),mean(v3)) )

system.time( DF %>% group\_by(id3) %>% summarise(sum(v1),mean(v3)) )

system.time( DF %>% group\_by(id4) %>% summarise\_each(funs(mean), 7:9) )

system.time( DF %>% group\_by(id4) %>% summarise\_each(funs(mean), 7:9) )

system.time( DF %>% group\_by(id6) %>% summarise\_each(funs(sum), 7:9) )

system.time( DF %>% group\_by(id6) %>% summarise\_each(funs(sum), 7:9) )

$ python3

import pandas as pd

import numpy as np

import timeit

# randChar is workaround for MemoryError in mtrand.RandomState.choice

# http://stackoverflow.com/questions/25627161/how-to-solve-memory-error-in-mtrand-randomstate-choice

def randChar(f, numGrp, N) :

things = [f%x for x in range(numGrp)]

return [things[x] for x in np.random.choice(numGrp, N)]

def randFloat(numGrp, N) :

things = [round(100\*np.random.random(),4) for x in range(numGrp)]

return [things[x] for x in np.random.choice(numGrp, N)]

N=int(1e9)

K=100

DF = pd.DataFrame({

'id1' : randChar("id%03d", K, N), # large groups (char)

'id2' : randChar("id%03d", K, N), # large groups (char)

'id3' : randChar("id%010d", N//K, N), # small groups (char)

'id4' : np.random.choice(K, N), # large groups (int)

'id5' : np.random.choice(K, N), # large groups (int)

'id6' : np.random.choice(N//K, N), # small groups (int)

'v1' : np.random.choice(5, N), # int in range [1,5]

'v2' : np.random.choice(5, N), # int in range [1,5]

'v3' : randFloat(100,N) # numeric e.g. 23.5749

})

timeit.Timer("DF.groupby(['id1']).agg({'v1':'sum'})" ,"from \_\_main\_\_ import DF").timeit(1)

timeit.Timer("DF.groupby(['id1']).agg({'v1':'sum'})" ,"from \_\_main\_\_ import DF").timeit(1)

timeit.Timer("DF.groupby(['id1','id2']).agg({'v1':'sum'})" ,"from \_\_main\_\_ import DF").timeit(1)

timeit.Timer("DF.groupby(['id1','id2']).agg({'v1':'sum'})" ,"from \_\_main\_\_ import DF").timeit(1)

timeit.Timer("DF.groupby(['id3']).agg({'v1':'sum', 'v3':'mean'})" ,"from \_\_main\_\_ import DF").timeit(1)

timeit.Timer("DF.groupby(['id3']).agg({'v1':'sum', 'v3':'mean'})" ,"from \_\_main\_\_ import DF").timeit(1)

timeit.Timer("DF.groupby(['id4']).agg({'v1':'mean', 'v2':'mean', 'v3':'mean'})" ,"from \_\_main\_\_ import DF").timeit(1)

timeit.Timer("DF.groupby(['id4']).agg({'v1':'mean', 'v2':'mean', 'v3':'mean'})" ,"from \_\_main\_\_ import DF").timeit(1)

timeit.Timer("DF.groupby(['id6']).agg({'v1':'sum', 'v2':'sum', 'v3':'sum'})" ,"from \_\_main\_\_ import DF").timeit(1)

timeit.Timer("DF.groupby(['id6']).agg({'v1':'sum', 'v2':'sum', 'v3':'sum'})" ,"from \_\_main\_\_ import DF").timeit(1)

**R info**

$ R --vanilla

# R version 3.1.1 (2014-07-10) -- "Sock it to Me"

# Copyright (C) 2014 The R Foundation for Statistical Computing

# Platform: x86\_64-pc-linux-gnu (64-bit)

> sessionInfo()

# R version 3.1.1 (2014-07-10)

# Platform: x86\_64-pc-linux-gnu (64-bit)

#

# locale:

# [1] LC\_CTYPE=en\_US.UTF-8 LC\_NUMERIC=C

# [3] LC\_TIME=en\_US.UTF-8 LC\_COLLATE=en\_US.UTF-8

# [5] LC\_MONETARY=en\_US.UTF-8 LC\_MESSAGES=en\_US.UTF-8

# [7] LC\_PAPER=en\_US.UTF-8 LC\_NAME=C

# [9] LC\_ADDRESS=C LC\_TELEPHONE=C

# [11] LC\_MEASUREMENT=en\_US.UTF-8 LC\_IDENTIFICATION=C

# attached base packages:

# [1] stats graphics grDevices utils datasets methods base

# other attached packages:

# [1] dplyr\_0.2 data.table\_1.9.2

#

# loaded via a namespace (and not attached):

# [1] assertthat\_0.1 parallel\_3.1.1 plyr\_1.8.1 Rcpp\_0.11.2 reshape2\_1.4

# [6] stringr\_0.6.2 tools\_3.1.1

**System info**

[Amazon EC2 r3.8large](https://aws.amazon.com/ec2/instance-types/)

$ lsb\_release -a

# No LSB modules are available.

# Distributor ID: Ubuntu

# Description: Ubuntu 14.04 LTS

# Release: 14.04

# Codename: trusty

$ uname -a

# Linux ip-172-31-33-222 3.13.0-29-generic #53-Ubuntu SMP

# Wed Jun 4 21:00:20 UTC 2014 x86\_64 x86\_64 x86\_64 GNU/Linux

$ lscpu

# Architecture: x86\_64

# CPU op-mode(s): 32-bit, 64-bit

# Byte Order: Little Endian

# CPU(s): 32

# On-line CPU(s) list: 0-31

# Thread(s) per core: 2

# Core(s) per socket: 8

# Socket(s): 2

# NUMA node(s): 2

# Vendor ID: GenuineIntel

# CPU family: 6

# Model: 62

# Stepping: 4

# CPU MHz: 2494.090

# BogoMIPS: 5049.01

# Hypervisor vendor: Xen

# Virtualization type: full

# L1d cache: 32K

# L1i cache: 32K

# L2 cache: 256K

# L3 cache: 25600K

# NUMA node0 CPU(s): 0-7,16-23

# NUMA node1 CPU(s): 8-15,24-31

$ free -h

# total used free shared buffers cached

# Mem: 240G 2.4G 237G 364K 60M 780M

# -/+ buffers/cache: 1.6G 238G

# Swap: 0B 0B 0B

**FAQ**

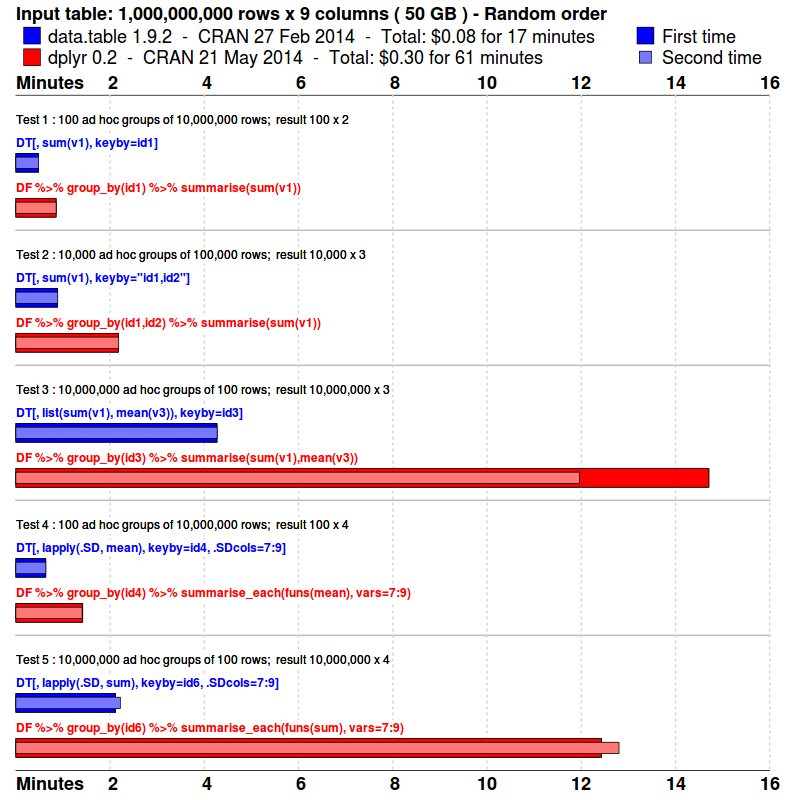
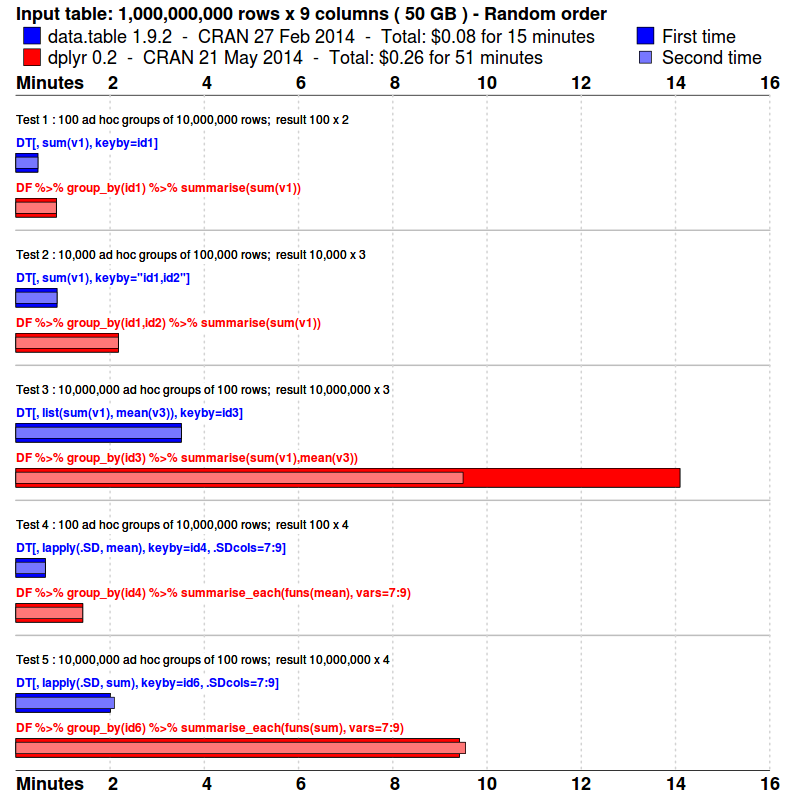
**What about compiler flags?**

We placed the following in ~/.R/Makevars before installing the packages :

CFLAGS=-O3 -mtune=native

CXXFLAGS=-O3 -mtune=native

Otherwise, R by default compiles packages with -g -O2. The R core team know of some problems with -O3 with some other CRAN packages in some circumstances so that choice of R's default is not an oversight. On the other hand we don't know which CRAN packages are affected and to what degree. So we used -O3 for these benchmarks to isolate that concern, since we're not aware of any problem with -O3.

-O3 can speed up by *up to* 30%; e.g., see tests 3 and 5 in before and after plots below. Right click and 'open image in a new tab' to see these images full size. We realise a single chart showing the percentage change would be better, but not a priority.  
  
 

For pandas 0.14-1 we installed as follows. Conveniently, it appears to use -O3 by default but if there are better options, please let us know.

wget -O- http://neuro.debian.net/lists/trusty.us-ca.libre | sudo tee /etc/apt/sources.list.d/neurodebian.sources.list

sudo apt-key adv --recv-keys --keyserver hkp://pgp.mit.edu:80 2649A5A9

sudo apt-get update

sudo apt-get install python3-pandas python3-dev

python3-config --cflags

-I/usr/include/python3.4m -I/usr/include/python3.4m -Wno-unused-result -Werror=declaration-after-statement

-g -fstack-protector --param=ssp-buffer-size=4 -Wformat -Werror=format-security -DNDEBUG -g -fwrapv

-O3 -Wall -Wstrict-prototypes

**Why not a single line chart rather than multiple bar plots for each size?**

Because we want to compare syntax too. The syntax is quite wide so a horizontal bar chart accommodates that text. The image can grow in height as more tests or packages are added whereas a line chart starts to become cluttered. The largest size is likely the most interesting; we don't expect many people to look at the smaller sizes.

**Shouldn't there be a gc() between each system.time()?**

system.time() already does that built-in. It has a gcFirst argument, by default TRUE. The time for that gc is excluded by system.time.

**Those costs are tiny. Are they the EC2 compute cost?**

Yes @ $0.30/hour as a spot instance. The machine specs are above. It was just to show how accessible and cheap large RAM on-demand machines can be.

**Does either package have a parallel option?**

Not currently. There are 32 CPU's on this machine and only one is being used, so that's an obvious thing to address.

**What if the groups aren't randomly shuffled but there's some clustering?**

All packages should be faster due to better L2/L3 cache efficiency. Some may benefit more than others and it'd be great to test that. Any volunteers? However, a feature of this benchmark is that it's a sum of 5 different queries. In real life, if groups are clustered then it can only be on one dimension (e.g. (id,time) or (time,id), but not both). We're trying to capture that with this benchmark.

**How does SQL compare, what about indexes and is dplyr faster with a database backend?**

This would be great to investigate. Any volunteers? The time and space to create indexes vs the subsequent faster grouping times should be reported separately. data.table's setkey is like a clustered index in SQL. The key is the physical row order in RAM and is very L2/L3 cache efficient when grouping those columns. dplyr would also be faster on a data.frame if the groups are contiguous in RAM for the same reason. But recall that the benchmark's 5 queries are designed to test grouping in several different columns. So just one ordering or one index would be insufficient. This is why we're working on secondary keys (set2key) analogous to CREATE INDEX in SQL. Two types of SQL database should be tested: [row-store](http://en.wikipedia.org/wiki/Relational_database_management_system) and [column-store](http://en.wikipedia.org/wiki/List_of_column-oriented_DBMSes). If they have in-memory options those should be selected since disk will always be slower than RAM. Both data.table and data.frame are in-memory column stores limited by RAM. 240GB is the biggest EC2 machine available currently which we used for these results. On the other hand, a database can of course can store and query many TB and the data can persist on disk.

**Has anything else changed?**

v2 was originally sample(1e6, N, TRUE) but the size of these ints caused sum(v2) in test 5 to overflow. R detects this and coerces to numeric. To keep things simple we changed it to sample(5, N, TRUE) just like v1.

**Why do the data.table commands use keyby= rather than by=?**

To match dplyr. by= in data.table returns the groups in the order they first appear in the dataset. This can be handy when the data is grouped in blocks and the order those blocks appear has some meaning to be preserved. As far as we know dplyr has no equivalent; it always sorts the returned groups. keyby= is a by= followed by a setkey on the group columns in the result.

**Why did you run these benchmarks?**

To know what the facts are and to demonstrate what we think a decent benchmark for large data is. To gain feedback and ideas for improvement. Single query benchmarks like this enable us to compare in a controlled way various packages on as fair and open a basis as possible using simulated data to measure how they scale in size and cardinality.

**Is 2 billion the limit?**

Yes. data.table and we believe data.frame are both currently limited to 2^31 rows. To increase this limit we need to start using long vectors internally which were new in R 3.0.0.

**Why just grouping?**

Because we had to start somewhere. Joins, updates and selecting subsets will follow.

**Is just testing sum and mean too simple?**

Yes. sum and mean benefit from advanced optimizations in both packages. Adding, say, median is expected to give different results. As might sum()+1 and other expressions that are very commonly needed in practice but are more difficult to optimize. Doing sum() by group first then adding 1 afterwards can be dramatically faster for this reason.

**Is it elapsed time that's measured?**

Strictly speaking, it's user+sys. Other unrelated tasks on the machine can affect elapsed, we believe. However, this server had so many CPUs and so much memory that elapsed==user+sys in all cases.

**How about -flto and similar extra flags?**

If there's evidence that it's worth the testing time we're happy to do so. Is there any downside to the extra flag? Why haven't the compiler developers included it in -O3 already? It's a complicated topic; e.g. [What are your compiler flags?](http://blog.httrack.com/blog/2014/03/09/what-are-your-gcc-flags/).

**Base R's mean accumulates in long double precision where the machine supports it (and most do) and then does a second pass through the entire vector to adjust mean(x) by typically a very small amount to ensure that sum(x-mean(x)) is minimized. What do these packages do?**

Both data.table and dplyr currently do a single pass sum(x)/length(x) where the sum is done in long double precision. They match base in terms of long double but don't do the second pass, currently. Feedback welcome.

**What about base R grouping functions?**

It would be great to add a comparison for 1E7 rows. We expect the scale to be hours or possibly days for 2E9.

**Why haven't the faster methods been incorporated into R itself?**

* data.table and dplyr are still developing and not yet fully mature
* as you can see from the answer about mean above, they are sometimes not precisely the same as base R. This is the same reason fread hasn't replaced read.csv, because it isn't yet a drop in replacement (ability to read from connections being the main difference).
* it isn't just speed but both packages prefer a different syntax to base R
* a lot of new code at C level would need to be added to R itself and a large amount of testing and risk of breakage (5,000 dependent packages on CRAN). To date there hasn't been the will or the time.
* packages are a fantastic way to allow progress without breaking base R and its dependants.

**Why do you use sum(gc()[,2]) rather than object.size()?**

object.size() is an estimate and quite slow (pryr::object\_size() is better) but also since each package is tested in a fresh and minimal R --vanilla session, sum(gc()[,2] is actually how much RAM that R session has consumed. It includes the increased size of R's global string cache and accounts for the efficiency of R only saving unique strings once. The size of the entire R session is the important thing for large data if you have other R sessions (or other non-R jobs) running on the same machine at the same time. This figure matches very closely to what htop displays for the R process; i.e. 100GB +/- 0.5GB.

**Why is the 2nd run for dplyr faster (9 minutes instead of 14 minutes) unlike the other packages where the 1st and 2nd run are the same (5 mins pandas both runs and 3 mins data.table both runs)?**

We don't know. It has been suggested that dplyr uses more working memory in this test; growing R's heap to accommodate that memory takes more time on the first run. R's heap is then already big enough for the 2nd run. But that is just a theory.

**I have reproduced your timings but when I sum up the 10 system.time() calls I arrive at a total time that is ~1.5x less than the actual total time of the 10 calls.**

There are two effects to consider.  
Firstly, system.time() returns 3 figures: user, system and elapsed. On a healthy system where no other process is running (e.g. another user, or a virus scan on Windows), user+system == elapsed. Our benchmark was run on a healthy fresh EC2 instance and we checked this was the case for all calls. If you compare a proc.time() before and after the 10 calls, you should compare user+system as well as comparing elapsed to check nothing unrelated to the benchmark has increased the wall clock timing.  
Secondly, system.time() has an argument gcFirst which by default is TRUE (see earlier FAQ above). This gc() is performed first by system.time() to free up memory used by previous commands in the R session. Otherwise a gc() during the individual test might increase its variance unfairly just because of previous commands in the session. The time this up-front forced gc() takes is not included in the timings system.time() reports but it would be included when comparing proc.time() before and after the 10 calls. If the algorithm being timed is not memory inefficient and therefore incurs gc() costs within it to clean up as it is running (often many times) then those will be included which is fair. It is just the first forced extra gc() by system.time() that is excluded since if the algorithm is memory efficient and requires no gc() then it should not be penalized by a gc() that was really caused by a previous unrelated call.  
If you compare the total time of the 10 calls to the total time printed at the top of the plot then you should find it matches as that was done with a proc.time() before and after (comparing user+sys, see ealier FAQ above) and therefore includes the time of the 10 forced extra gc()s by system.time().

Here is the same data presented graphically.

This is why we don’t consider “eager” the proper way to call data.table, it *artificially* makes data.table *appear* slow. This is the negative impression of data.table that the dplyr/dtplyr adaptors have been falsely giving dplyr users for the last five years. dplyr users either felt they were getting the performance of data.table through dplyr (if they didn’t check timings), or got a (false) negative impression of data.table (if they did check timings).

Details of the timings can be found below:

# timings

library("microbenchmark")

library("dplyr")

##

## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':

##

## filter, lag

## The following objects are masked from 'package:base':

##

## intersect, setdiff, setequal, union

library("dtplyr")

library("data.table")

##

## Attaching package: 'data.table'

## The following objects are masked from 'package:dplyr':

##

## between, first, last

library("ggplot2")

library("WVPlots")

packageVersion("dplyr")

## [1] '0.8.1'

packageVersion("dtplyr")

## [1] '0.0.3'

R.version.string

## [1] "R version 3.6.0 (2019-04-26)"

mk\_data <- function(nrow, ncol) {

d <- data.frame(x1 = rep(0, nrow))

for(i in seq\_len(ncol)) {

vi <- paste0("x", i)

d[[vi]] <- rnorm(nrow)

}

d

}

base\_r\_fn <- function(df) {

dt <- df

for(i in seq\_len(nstep)) {

dt$x1 <- dt$x1 + 1

}

dt

}

dplyr\_fn <- function(df) {

dt <- df

for(i in seq\_len(nstep)) {

dt <- mutate(dt, x1 = x1 + 1)

}

dt

}

dtplyr\_fn <- function(df) {

dt <- as.data.table(df)

for(i in seq\_len(nstep)) {

dt <- mutate(dt, x1 = x1 + 1)

}

dt

}

data.table\_fn <- function(df) {

dt <- as.data.table(df)

for(i in seq\_len(nstep)) {

dt[, x1 := x1 + 1]

}

dt[]

}

df <- mk\_data(3, 2)

nstep <- 5

base\_r\_fn(df)

## x1 x2

## 1 4.264622 0.5668608

## 2 6.353611 0.8614390

## 3 6.070827 1.0268441

dplyr\_fn(df)

## x1 x2

## 1 4.264622 0.5668608

## 2 6.353611 0.8614390

## 3 6.070827 1.0268441

dtplyr\_fn(df)

## x1 x2

## 1: 4.264622 0.5668608

## 2: 6.353611 0.8614390

## 3: 6.070827 1.0268441

data.table\_fn(df)

## x1 x2

## 1: 4.264622 0.5668608

## 2: 6.353611 0.8614390

## 3: 6.070827 1.0268441

df <- mk\_data(100000, 100)

nstep <- 1000

timings <- microbenchmark(

base\_r = base\_r\_fn(df),

dplyr = dplyr\_fn(df),

dtplyr = dtplyr\_fn(df),

data.table = data.table\_fn(df),

unit = 's',

times = 10L

)

saveRDS(timings, file = "dtplyr\_timings.RDS")

tdf <- as.data.frame(timings)

tdf$seconds <- tdf$time/1e+9

tdf$method <- factor(as.character(tdf$expr),

rev(levels(tdf$expr)))

as.data.table(tdf)[

, .(mean\_seconds = mean(seconds)), by = "method" ][

order(mean\_seconds), ]

## method mean\_seconds

## 1: base\_r 0.8367011

## 2: data.table 1.5592681

## 3: dplyr 2.6420171

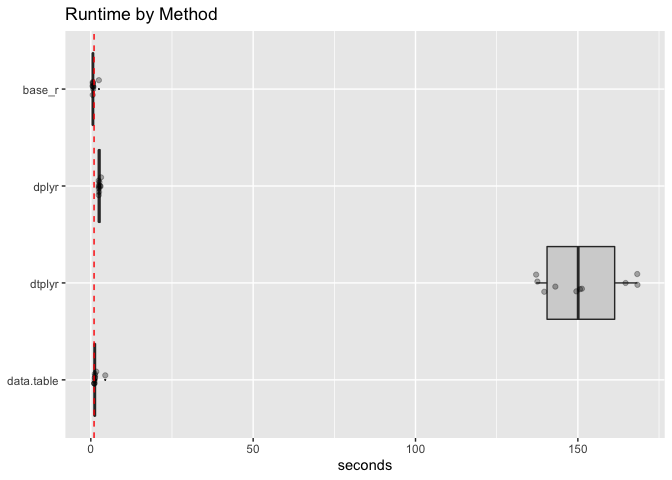
## 4: dtplyr 151.0217646

WVPlots::ScatterBoxPlotH(tdf, "seconds","method",

title="Runtime by Method") +

geom\_hline(yintercept = 1, color = "red", linetype = 2) +

xlab(NULL)

[](https://github.com/WinVector/Examples/blob/master/Time_dtplyr/time_dtplyr_files/figure-gfm/present-1.png)

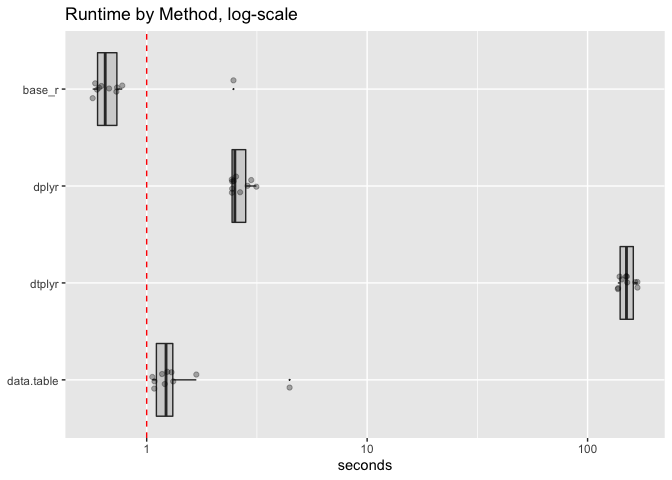
WVPlots::ScatterBoxPlotH(tdf, "seconds","method",

title="Runtime by Method, log-scale") +

scale\_y\_log10() +

geom\_hline(yintercept = 1, color = "red", linetype = 2) +

xlab(NULL)

[](https://github.com/WinVector/Examples/blob/master/Time_dtplyr/time_dtplyr_files/figure-gfm/present-2.png)

As we have said: the “don’t force so many extra copies” methodology has been in rqdatable for quite some time, and in fact works well. Notice the two rqdatatable timings have some translation overhead. This is why using data.table directly is, in general, going to be a superior methodology.

More Information

## 1. Introduction: Why data.table?

The data.table is an alternative to R’s default data.frame to handle tabular data. The reason it’s so popular is because of the speed of execution on larger data and the terse syntax. So, effectively you type less code and get much faster speed. It is one of the most downloaded packages in R and is preferred by Data Scientists. It is probably one of the best things that have happened to R programming language as far as speed is concerned. Though data.table provides a slightly different syntax from the regular R data.frame, it is quite intuitive. So once you get it, it feels obvious and natural that you wouldn’t want to go back the base R data.frame syntax. By the end of this guide you will understand the fundamental syntax of data.table and the structure behind it. All the core data manipulation functions of data.table, in what scenarios they are used and how to use it, with some advanced tricks and tips as well.

## 2. How to install data.table package

Installing data.table package is no different from other R packages. Its recommended to run install.packages() to get the latest version on the CRAN repository. However, if you want to use the latest development version, you can get it from github as well.

# Install from CRAN

install.packages('data.table')

# Install Dev version from Gitlab

install.packages("data.table", repos="<https://Rdatatable.gitlab.io/data.table>")

data.table::update.dev.pkg()

If you want to revert back to the CRAN version do:

# Remove and Reinstall

remove.packages("data.table")

install.packages("data.table")

## 3. Importing Data

The way you work with data.tables is quite different from how you’d work with data.frames. Let’s understand these difference first while you gain mastery over this fantastic package. The fread(), short for fast read is data.tables version of read.csv(). Like read.csv() it works for a file in your local computer as well as file hosted on the internet. Plus it is atleast 20 times faster. Let’s import the mtcars dataset stored as a csv file.

library(data.table)

mt <- fread("<https://raw.githubusercontent.com/selva86/datasets/master/mtcars.csv>")

head(mt)

class(mt)

#> [1] "data.table" "data.frame"

The imported data is stored directly as a data.table. As you see from the above output, the data.table inherits from a data.frame class and therefore is a data.frame by itself. So, functions that accept a data.frame will work just fine on data.table as well. Because the dataset we imported was small, the read.csv()‘s speed was good enough. However, the speed gain becomes evident when you import a large dataset (millions of rows). To get a flavor of how fast fread() is, run the below code. It creates a 1M rows csv file. Then reads it back again. The time taken by fread() and read.csv() functions gets printed in console.

# Create a large .csv file

set.seed(100)

m <- data.frame(matrix(runif(10000000), nrow=1000000))

write.csv(m, 'm2.csv', row.names = F)

# Time taken by read.csv to import

system.time({m\_df <- read.csv('m2.csv')})

#> user system elapsed

#> 39.798 1.326 43.003

# Time taken by fread to import

system.time({m\_dt <- fread('m2.csv')})

#> user system elapsed

#> 1.735 0.097 1.877

That’s about 20x faster. The time difference gets wider when the filesize increases.

## 4. How to convert data.frame to data.table

You can convert any `data.frame` into `data.table` using one of the approaches:

1. data.table(df) or as.data.table(df)
2. setDT(df)

The difference between the two approaches is: data.table(df) function will create a copy of df and convert it to a data.table. Whereas, setDT(df) converts it to a data.table inplace. That means, the df itself gets converted to a data.table and you don’t have to assign it to a different object. As a result, there is no copy made and no duplication of the same data. Let’s reload the mtcars dataframe from R’s default datasets pacakge.

# reload data

data("mtcars")

head(mtcars)

**Important**: The data.table() does not have any rownames. So if the data.frame has any rownames, you need to store it as a separate column before converting to data.table.

mtcars$carname <- rownames(mtcars)

mtcars\_dt <- as.data.table(mtcars)

class(mtcars\_dt)

#> [1] "data.table" "data.frame"

Alternately, use setDT() to convert it to data.table in place.

mtcars\_copy <- copy(mtcars)

setDT(mtcars\_copy)

class(mtcars\_copy)

#> [1] "data.table" "data.frame"

## 5. How to convert data.table to data.frame

Conversely, use as.data.frame(dt) or setDF(dt) to convert a data.table to a data.frame.

setDF(mtcars\_copy)

class(mtcars\_copy)

#> [1] "data.frame"

## 6. Filtering rows based on conditions

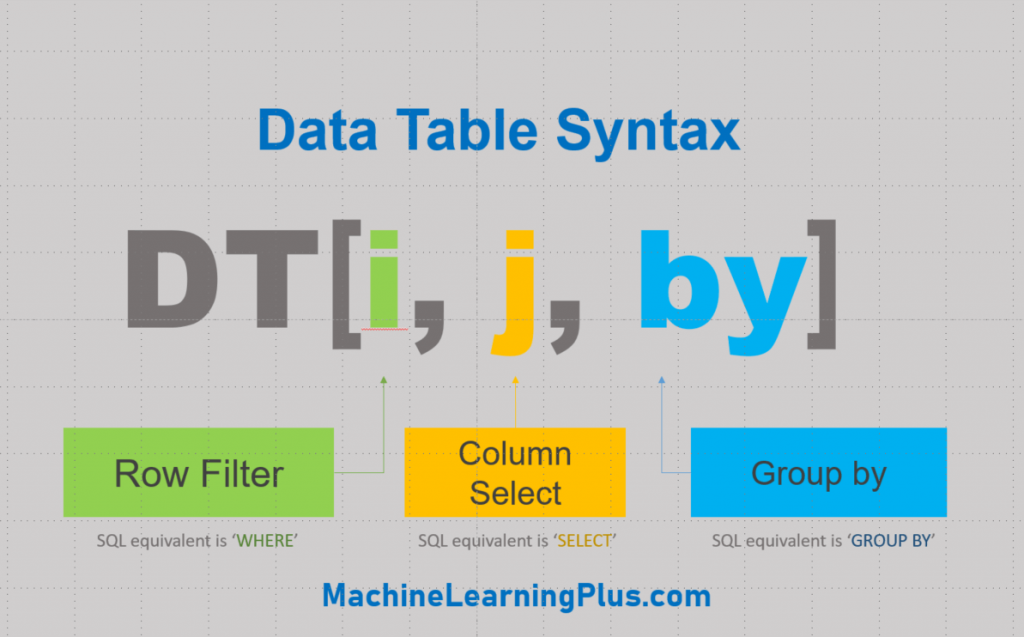
The main difference with data.frame is: data.table is aware of its column names. So while filtering, passing only the columns names inside the square brackets is sufficient.

# dataframe syntax

mtcars[mtcars$cyl == 6 & mtcars$gear == 4, ]

# datatable syntax

mtcars\_dt[cyl==6 & gear==4, ]

This saves a significant amount of keystrokes in the long run. This is a major advantage. [](https://www.machinelearningplus.com/wp-content/uploads/2019/11/data_table_syntax.png)

## 7. How to select given columns

Now, let see how to subset columns. The most unexpected thing you will notice with data.table is you cant select a column by its numbered position in a data.table. For example, you can expect the following to work in a data.frame.

mtcars[, 1]

# <returns first column>

But this would just return ‘1’ in a data.table

mtcars\_dt[, 1]

#> 1

If you want to get that column by position alone, you should add an additional argument, with=FALSE.

mtcars\_dt[, 1, with=F]

# <returns first column>

The returned output is a 1-column data.table. An alternate way and a better practice is to pass in the actual column name.

mtcars\_dt[, mpg]

Notice here that the ‘mpg’ is not a string as it’s not written within quotes.

## 8. How to select multiple columns using a character vector

What if the column name is present as a string in another variable (vector)? In that case, you can’t use mpg directly. You need to additionally pass with=FALSE.

myvar <- "mpg"

mtcars\_dt[, myvar, with=F]

# <returns mpg column>

The same principle applies if you have multiple columns to be selected.

columns <- c('mpg', 'cyl', 'disp')

mtcars\_dt[, columns]

#> [1] "mpg" "cyl" "disp"

To make the above command work, you need to pass with=FALSE inside the square brackets

mtcars\_dt[, columns, with=FALSE]

Clear? If you want to select multiple columns directly, then enclose all the required column names within list.

# syntax 1:

mtcars\_dt[1:4, list(mpg, cyl, gear)]

# syntax 2: most used

mtcars\_dt[, .(mpg, cyl, gear)]

## 9. How to drop columns

How to drop the mpg, cyl and gear columns alone? Place them in a vector and use the ! in front to drop them. This effectively returns all columns except those present in the vector.

drop\_cols <- c("mpg", "cyl", "gear")

mtcars\_dt[, !drop\_cols, with=FALSE]

## 10. How to rename columns

The setnames() function is used for renaming columns. It takes the data.table (or data.frame), current name and new name as arguments and changes the column names in place without any copying of data. Always recommended!

setnames(mtcars\_dt, 'vs', 'engine\_type')

colnames(mtcars\_dt)

#> <'vs' column is renamed to 'engine\_type'.

DT <- data.table(A=1:5)

DT[ , X := shift(A, 1, type="lag")]

DT[ , Y := shift(A, 1, type="lead")]

Before moving on, let’s try out a mini challenge in R console. **Q: Convert the in-built airquality dataset to a data.table. Then select “Solar.R”, “Wind” and “Temp” for those rows where “Ozone” is not missing.** Show Solution

## 11. Creating a new column from existing columns

You can always create a new column as you do with a data.frame, but, data.table lets you create column from within square brackets. This saves key strokes.

# data.frame syntax (works on data.table)

mtcars\_dt$cyl\_gear <- mtcars\_dt$cyl + mtcars\_dt$gear

# data.table syntax

mtcars\_dt[, cyl\_gear2 := cyl + gear]

mtcars\_dt

To create multiple new columns at once, use the special assignment symbol as a function.

mtcars\_dt[, `:=`(cyl\_gear3 = cyl \* gear,

cyl\_gear4 = cyl - gear)]

mtcars\_dt

To select only specific columns, use the list or dot symbol instead.

mtcars\_dt[, .(cyl\_gear3 = cyl \* gear,

cyl\_gear4 = cyl - gear)]

Now let’s see a special but frequently used case. Let’s suppose you have the column names in a character vector and want to select those columns alone from the data.table. Passing it inside the square brackets don’t work.

columns <- c('mpg', 'cyl', 'disp')

mtcars\_dt[, columns]

#> [1] "mpg" "cyl" "disp"

To make the above command work, you need to pass with=FALSE inside the square brackets

mtcars\_dt[1:3, columns, with=FALSE]

## 12. How to create new columns using character vector

Suppose you want to create a new column but you have the name of that new column in another character vector. How to create the new column without using the actual column name? For example, you have the new column name in the myvar vector. And, you want to assign some value, say the value of 1 to this column. Doing this will create a new column named ‘myvar’. And not var1 as intended.

myvar <- c('var1')

mtcars\_dt[, myvar:=1]

# <creates a column named myvar -- Bad! >

To create a column named ‘var1’ instead, keep myvar inside a vector.

# Create column named 'var1'

# Syntax 1

mtcars\_dt[, c(myvar):=1]

# Syntax 2

mtcars\_dt[, (myvar):=2]

Finally, if you want to delete a columns, assign it to NULL.

mtcars\_dt[, c("myvar", "var1") := NULL]

All the above column names are now deleted. Before moving on, try solving this exercise in your R console. **Question: Create a new column called ‘mileage\_type’ that has the value ‘high’ if mpg > 20 else has value low.** Show Solution

## 13. Grouping

Now, let’s move on to the second major and awesome feature of R data.table: grouping using by. In base R, grouping is accomplished using the aggregate() function. It’s a bit cumbersome and hard to remember the syntax. All the functionalities can be accomplished easily using the ‘by’ argument within square brackets. For example, in mtcars data, how to get the mean mileage for each cylinder type? Answer: Since you want to see the mileage by cyl column, set by = 'cyl' inside the square brackets.

# Mean mileage by `cyl`

mtcars\_dt[, .(mean\_mileage=mean(mpg)), by=cyl]

Thats really useful isnt it? You can even add multiple columns to the ‘by’ argument.

mtcars\_dt[, .(mean\_mileage=mean(mpg)), by=.(cyl, gear)]

## 14. A slightly complex groupby problem

Now, lets see some really special cases. How to select the first occurring value of mpg for each unique cyl value That is, instead of taking the mean of mileage for every cylinder, you want to select the first occurring value of mileage. How to do that?

mtcars\_dt[, .(first\_mileage=mpg[1]), by=cyl]

What to do if you want the second value? Just replace the 1 with 2.

mtcars\_dt[, .(second\_mileage=mpg[2]), by=cyl]

And what if you want the last value? You can either use length(mpg) or .N:

# Option 1

mtcars\_dt[, .(first\_mileage=mpg[length(mpg)]), by=cyl]

# Option 2

mtcars\_dt[, .(first\_mileage=mpg[.N]), by=cyl]

## 15. What does .N and .I do

.N contains the number of rows present. So the following will get the number of rows for each unique value of cyl.

mtcars\_dt[, .N, by=cyl]

# < returns the number of rows for each unique value of cyl>

Now, how to create row numbers of items? It can be done using .I variable, short for ‘index’ (I guess). Lets first understand what .I returns.

mtcars\_dt[, .I]

# <returns numbers from 1,2,3..>

It returns all the row numbers. Now, how to return the row numbers where cyl=6 ? This can get confusing in the beginning so pay close attention. If you want to get the row numbers of items that satisfy a given condition, you might tend to write like this:

mtcars\_dt[cyl==6, .I]

# <again returns numbers from 1,2,3..>

But this returns the wrong answer because, `data.table` has already filtered the rows that contain cyl value of 6. So, what you want to do instead is to write that condition to subset .I alone instead of the whole `data.table`.

mtcars\_dt[, .I[cyl==6]]

The result is same as using the `which()` function, which we used in `data.frames`.

mtcars\_dt[, which(cyl==6)]

Great! Let’s solve another challenge before moving on.   Q: Compute the number of cars and the mean mileage for each gear type. After a few seconds I will show the answer.   Show Solution

## 16. Chaining

Data.Table offers unique features there makes it even more powerful and truly a swiss army knife for data manipulation. First lets understand what chaining is. Using chaining, you can do multiple datatable operatations one after the other without having to store intermediate results. For example, instead of writing two statements you can do it on one. Below code sorts after grouping by cyl:

dt1 <- mtcars\_dt[, .(mean\_mpg=mean(mpg),

mean\_disp=mean(disp),

mean\_wt=mean(wt),

mean\_qsec=mean(qsec)), by=cyl]

output <- dt1[order(cyl), ]

output

With chaining, that is, by attaching the square brackets at the end, it’s done in one step.

output <- mtcars\_dt[, .(mean\_mpg=mean(mpg),

mean\_disp=mean(disp),

mean\_wt=mean(wt),

mean\_qsec=mean(qsec)), by=cyl][order(cyl), ]

Actually, chaining is available in dataframes as well, but with features like by, it becomes convenient to use on a data.table.

## 17. What is .SD and How to write functions inside data.table

Next, lets see how to write functions within a data.table square brackets. Let’s suppose, you want to compute the mean of all the variables, grouped by ‘cyl’. How to do that? You can create the columns one by one by writing by hand. Or, you can use the lapply() function to do it all in one go. But `lapply()` takes the data.frame as the first argument. Then, how to use `lapply() inside a data.table? You can use the .SD object as the first argument for lapply(). But, what is the .SD object? It is nothing but a data.table that contains all the columns of the original datatable except the column specified in ‘by’ argument. So, here is what it looks like.

mtcars\_dt[, .SD, by=cyl]

# <returns the mtcars\_dt table grouped by cyl. Try it>

So, now you can pass this as the first argument in `lapply()`. The 11th column in `.SD` is rownames, so let’s include only the first 10.

output <- mtcars\_dt[, lapply(.SD[, 1:10, with=F], mean), by=cyl]

output

Optionally, Instead of subsetting .SD like this, You can specify the columns that should be part of .SD using the .SDCols object

output <- mtcars\_dt[, lapply(.SD, mean), by=cyl, .SDcols=c("mpg", "disp", "hp", "drat", "wt", "qsec")]

output

The output now contains only the specified columns.

## 18. Keys

Now, we have come to the ‘key’ concept for data.tables: Keys Let’s understand why keys can be useful and how to set it. Setting one or more keys on a data.table enables it to perform [binary search](https://en.wikipedia.org/wiki/Binary_search_algorithm), which is many order of magnitudes faster than linear search, especially for large data. As a result, the filtering operations are super fast after setting the keys. There is a side effect though. By setting a key, the `data.table` gets sorted by that key. So how to set a key? Just use the setkey function.

setkey(mtcars\_dt, carname)

It’s so fast making it look like nothing happened. But it internally sorted data.table with ‘carname’ as the key.

mtcars\_dt

# <mtcars\_dt is sorted by carname column which we created in beginning>

If you notice, this table is sorted by ‘carname’ variable. To check the keys for a data table, you can use the key() function.

key(mtcars\_dt)

Once the key is set, merging data.tables is very direct. I have distributed few columns of mtcars in the following data.tables.

dt1 <- mtcars\_dt[,.(carname, mpg, cyl)]

dt2 <- mtcars\_dt[1:10, .(carname, gear)]

You can join these two datatables:

dt1[dt2]

This returns dt1‘s rows using dt2 based on the key of these data.tables. You can also set multiple keys if you wish.

setkey(mtcars\_dt, cyl, mpg)

mtcars\_dt

Now, how to remove the keys? Use setkey() and set it to NULL. But the datatable will not go back to it original row arrangement.

setkey(mtcars\_dt, NULL)

Another aspect of setting keys is the ‘keyby’ argument. Using keyby you can do grouping and set the by column as a key in one go. For example, in this example we saw earlier, you can skip the chaining by using keyby instead of just by.

# Group and sort using chaining

output <- mtcars\_dt[, .(mean\_mpg=mean(mpg),

mean\_disp=mean(disp),

mean\_wt=mean(wt),

mean\_qsec=mean(qsec)), by=cyl][order(cyl), ]

# Group and sort using keyby

output <- mtcars\_dt[, .(mean\_mpg=mean(mpg),

mean\_disp=mean(disp),

mean\_wt=mean(wt),

mean\_qsec=mean(qsec)), keyby=cyl]

key(output)

As a result, the output has the key as cyl.

## 19. How to join two or more datatables

The data.table package provides a faster implementation of the merge() function. The syntax is pretty much the same as base R’s merge().

dt1 <- mtcars\_dt[5:25,.(carname, mpg, cyl)]

dt2 <- mtcars\_dt[1:10, .(carname, gear)]

dt3 <- mtcars\_dt[2:12, .(carname, disp)]

# Inner Join

merge(dt1, dt2, by='carname')

#> <returns 6 rows>

# Left Join

merge(dt1, dt2, by='carname', all.x = T)

#> <returns 21 rows>

# Outer Join

merge(dt1, dt2, by='carname', all = T)

#> <returns 25 rows>

## 20. How to merge multiple data.tables in one shot

This is bit of a hack by using the Reduce() function to repeatedly merge multiple data.tables stored in a list. Reduce() takes in a function that has to be applied consequtively (which is merge\_func in this case) and a list that stores the arguments for function.

# Merge multiple data.tables

dt\_list <- list(dt1, dt2, dt3)

merge\_func <- function(...) merge(..., all = TRUE, by='carname')

dt\_merged <- Reduce(merge\_func, dt\_list)

#> <returns 25 rows from outer join>

## 21. Pivot Table operations

The dcast.data.table() is the function used for doing pivot table like operations as seen in spreadsheet softwares like Microsoft Office Excel or Google spreadsheets. The good thing is dcast.data.table() works equally well on data.frame object as well. Let’s create a pivot table showing the mean mileage(mpg) for Cylinders vs Carburetter (Carb)

dcast.data.table(mtcars\_dt, cyl ~ carb, fun.aggregate = mean, value.var = 'mpg')

So how to understand the syntax? There are 4 primary arguments:

1. data.table
2. formula: Rows of the pivot table on the left of ‘~’ and columns of the pivot on the right
3. value.var: column whose values should be used to fill up the pivot table
4. fun.aggregate: the function used to aggregate the value.var column.

dcast.data.table() is versatile in allowing multiple columns to be passed to the value.var and allows multiple functions to fun.aggregate as well. Let’s solve a quick exercise based on pivot table. **Question 1: Create a pivot table to show the maximum and minimum mileage observed for Carburetters vs Cylinders combination?** **Question 2: Which carb value has the highest difference between max – min?** Show Solution

## 22. set() – A magic function for fast assignment operations

The set() command is an incredibly fast way to assign values to a new column. The syntax is: set(dt, i, j, value), where i is the row number and j is the column number. As a best practice, always explicitly use integers for i and j, that is, use 10L instead of 10. It is usually used in for-loops and is literally thousands of times faster. Yes, it is so fast even when used within a for-loop, which is proof that for-loop is not really a bottleneck for speed. It is the underlying data structure related overhead that causes for-loop to be slow, which is exactly what set() avoids. Besides, it works on a data.frame object as well. It works like magic! Below is an example to illustrate the power of set() taken from official documentation itself. The speed benchmark may be outdated, but, run and check the speed by yourself to believe it.

m = matrix(1,nrow=100000,ncol=100)

DF = as.data.frame(m)

DT = as.data.table(m)

system.time(for (i in 1:10000) DF[i,1] <- i)

#> 591 seconds

system.time(for (i in 1:10000) DT[i,V1:=i])

#> 2.4 seconds ( 246 times faster, 2.4 is overhead in [.data.table )

system.time(for (i in 1:10000) set(DT,i,1L,i))

#> 0.03 seconds ( 19700 times faster, overhead of [.data.table is avoided )