Bill Gold data.table talk

“Life in the Fast Line: data.table an introduction and best practices”

Partly a reply to JD Long’s talk on dplyr

Outline

1. Why use data.table?
2. DT[i,j,by][c] syntax
3. Data exploration
4. Something unexpected
5. Next steps

Bill background: management consultant and data scientist, background in electrical engineering. Did lots of model training often requiring building a whole environment, currently with a company called Ataeva that’s hiring for fintech.

Data.table history: released 2006 by Matt Dowle et. Al

Advantages of it: fast, concise, integrates well with R.

Can pivot 10 years of financial history very rapidly. Saves a lot of time for finding new insights bc don’t have to wait for stuff to execute.

Speed: data.table on a benchmark with 1B rows, 9 columns, 50 GB. Faster than dplyr, competitive with Spark. Also, more cost-effective in hosting costs than everything except Spark.

Works in memory, by reference rather than by value (so can update data in place, rather than making copies like data.frame does). Some parallel processing, algos leveraging R’s efficient internal strs like globacl character cache. Radix sorting, setkey continguous than use efficient contiguous methods.

Concise. Gives example comparing to sql and data.frame operations

**SQL**: SELECT cyl, mean (mpg) FROM mtcars WHERE am=1 GROUP BY cyl

**Data frame**: aggregate(mtcars$mpg[mtcars$am==1], by=list(cyl=mtcars$cyl[mtcars$am==1]), FUN=mean)

**Data.table**: mtcars[am==1, mean(mpg), cyl]

Requires a lot fewer characters and less repetition.

Any R fct from any package can be used in data table.

**Data table syntax**

**DT[i, j, by ] [c ]**

Compare to sql:

j is like SELECT select-list

DT is like FROM table

i is like WHERE row filters

by is like GROUP BY aggregates

c is like ORDER BY or HAVING (c is for *chaining*)

**setup example:**

library(data.table

dt.mtcars <- data.table(mtcars, keep.rownames=T) #makes the row names into a column

Filter multiple conditions:

dt.mtcars[cyl==8 &

wt < 4 &

rn %like% ‘Merc’]

The rn %like% cause adds rownames that are like or contain “Merc”

Abbreviation for Mercedes

Essentially creates a vector that pass in! V powerful

Filter row numbers:

dt.mtcars[1:5]

Select clause vector output:

dt.mtcars[[“rn”]]

Select clause data table output:

dt.mtcars[ 1:5, list(rn, cyl, hp)]

Rows 1:5 from those columns

4 different selects that give same output:

Dt.mtcars[ , list (rn, cyl, hp) ]

Dt.mtcars[ , . (rn, cyl, hp) ] # .() is equivalent to list()

Dt.mtcars[ , c(‘rn’, ‘cyl’, ‘hp’), with=F] #good for writing functions that need to generalize

Dt.mtcars [ , c(1,3,5) ] #column numbers

You can choose which columns you choose to manipulate using .SD token:

Dt.mtcars[ 1:5, .SD, .SDcols = rn:cyl]

In this case it tells it to take the columns from “rowname” column through “cyl” column

Variable column names:

Double dot notation lets you make column names generic, execute on any column

Variable.col.name <- ‘rn’

Dt.mtcars [ 1:5 , ..variable.col.name ]

Double dot tells data table to look for a variable named variable.col.name and use it

Group by:

Dt.mtcars[ , . (mean(mpg)), by=cyl]

Recall that .() signifies list, so could use this to take summary stats of multiple columns by cyl easily. And can also aggregate by multiple vars rather than just by cyl – by can take a vector

Dt.mtcars[ , . (mpg=mean(mpg)), by=cyl]

Renaming output column to mpg, just like with dplyr summarize() or mutate()

Dt.mtcars[ , lapply(.SD, mean)

, .SDcols = mpg:carb

, by=cyl ]

Data.table is good for taking summary stats of a large data set rapidly. This gets the mean of every column from mpg to carb, grouped by cylinder

Here, there are 4 parameters cause of the SD and SD column; dt usually only has 3 params

Chaining:

Take basic blocks of data table fcts, append them onto the next. Like SQL HAVING

Dt.mtcars [ , . (mpg=mean(mpg)), by=cyl ] [mpg > 16 ]

Like with pipe, no need to create intermediate object, can just chain and get the subset you want

Dt.mtcars [ , . (mpg=mean(mpg)), by=cyl ] [order(-mpg) ]

Same concept, with order by mpg in descending order

Vectors and %in%

Intutition first:

1:2 %in% 1:6 gives TRUE TRUE

1:6 %in% 1:2 gives TRUE TRUE FALSE FALSE FALSE FALSE

Dt.mtcars [ , cyl ] %in% c(4, 6)

Vector of 32 TRUE/ FALSEs

Very powerful at scale bc can compare different data tables’ intersections very quickly

Filters vectors %in%

Dt.mtcars [ cyl %in% c(4, 6) ] [1:5] [order(cyl)]

Take rows where cyl is 4 or 6, take the first 5 of them, order by cyl

Can demystify relationships between badly sorted tables, figure out what keys exist where, very quickly

Join:

Here, let’s create a summary table by cylinders and join that back into mtcars

Dt.mtcars.cyl.aggr <- dt.mtcars [ , (mpg.mean.cyl=mean(mpg)

, mpg.sd.cyl=sd(mpg)

, hp.mean.cyl=mean(hp)

, hp.sd.cyl=sd(hp))

, by = cyl ]

To join, first we set a key with setkeyv() fct

Setkeyv(dt.mtcars, c(‘cyl’) )

Setkeyv(dt.mtcars.cyl.aggr, c(‘cyl’) )

DT <- dt.mtcars [ dt.mtcars.cyl.aggr ]

DT[1:5] #now has columns with the summary stats

The above is the fastest join syntax, but treating one table you want to join as a subset of the other and explicitly setting keys first is a bit unintuitive. More intuitive is the data.table::merge() syntax.

|  |  |  |
| --- | --- | --- |
| JOIN type | DT syntax | data.table::merge() syntax |
| INNER | X[Y, nomatch=0] | merge(X, Y, all=FALSE) |
| LEFT OUTER | Y[X] | merge(X, Y, all.x=TRUE) |
| RIGHT OUTER | X[Y] | merge(X, Y, all.y=TRUE) |
| FULL OUTER |  | merge(X, Y, all=TRUE) |
| FULL OUTER WHERE NULL (NOT INNER) |  | merge(X, Y, all=TRUE), subset NA |

Key shortcuts:

. list

.. variable to select

X[Y] right outer join

:= update

Updating syntax:

In SQL, SELECT and UPDATE syntax are similar. Same in DT.

Update add a new column:

Dt.mtcars[ , N := 1 ] #adds a new column that is all 1s.

Let’s add a column for manufacturer, which we can get from first word in rownames column:

v.manufacturer <- gsub(“([A-Za-z]+).\*”, [\\1](file:///\\1), dt.mtcars[ , rn ] )

dt.mtcars[ , manufacturer := v.manufacturer ]

dt.mtcars[ 1:5]

Create a new indicator variable, here a Mercedes indicator:

Dt.mtcars[ manufacturer == ‘Merc’, is.merc := 1 ]

Dt.mtcars [ , .N, by = is.merc ] #now this gives us a count of how many have is.merc = 1 vs is.merc = NA. 25 NA, 7 1, so there are 7 Mercedes and 25 other cars.

Other functions:

Fread

Fwrite

readRDS

saveRDS

setnames

setcolorder

dcast & melt

.N .SD .I .GRP .BY

Data exploration examples:

Example: 450 CSV tables handed to them by client with no documentation.

Created an fct called edd, enhanced data dictionary, to summarize data row by row in a very concise manner.

Display.data(edd(

Dt = data.table(iris)

, path\_out = ‘/home/bill’

, file\_out = ‘iris.edd’

, return\_edd = T ) )

Output: each row is one var. Gives you its best guess at datatype; how many are unique; how many are NA; how many are blank, and fill rates. For continuous vars, gives center/ mean and spread; for categorical vars, gives top ten values by frequency. Great with a Shiny package.

So, what he did was run this for each of the 450 tables, stacked all the output into one data.table, and used that to figure out key-value pairs between tables; whether number of unique values was the same; and could pass to plotting fcts to understand each table better.

Great for aggregating data in time series. Example:

Cutoffs <- c(0:10)/10

By.vars <- c(‘factorID.01’, factorID.02’)

Setkeyv(time.series, by.vars)

Quantile.time.series <- time.series[ Period >= ‘2009-01-01’

, . [Variable.01 = quantile(Variable.01, cutoffs, na.rm=T)

, . [Variable.02 = quantile(Variable.02, cutoffs, na.rm=T)

, . [Variable.03 = quantile(Variable.03, cutoffs, na.rm=T)

, . [Variable.04 = quantile(Variable.04, cutoffs, na.rm=T)

, . [Variable.05 = quantile(Variable.05, cutoffs, na.rm=T)

, . [Variable.06 = quantile(Variable.06, cutoffs, na.rm=T)

, by=by.vars ]

Says that finds getting filters right and being transparent about what they do is very helpful in client communication, and how transparent this code is is helpful for that.

Scoring a probit model: example with a time series, can make dependent variable and coefficients very transparent.

Fct for intersection of multiple data tables:

Venn.diagram(dt.mtcars[ , carb ]

, dt.mtcars[ , gear ]

, ‘carb’

, ‘gear’ )

Union, intersection, and right and left sides

Plotting:

Dt.mtcars[ , plot ( x = mpg

, y = cyl

, main = manufacturer )

, keyby = manufacturer ]

Automatically produces one plot per manufacturer.

Also works with ggplot like so:

Plot.All.XY.by.Z <- function (dt, x, y, z) {

#numerics only

Dt[, (y) := lapply( .SD, function(x) {as.numeric(as.character(x))}), .SDcols = y ]

Dts <- melt(dt, id = c(x, z), measure=y)

P <- ggplot(dts, aes\_string(x = colnames(dt)[x], y = “value”, colors = colnames(dt)[z])) +

Geom\_line() +

Facet\_wrap(~ variable)

Print(p)

}

Plot.All.XY.by.Z(dt.mtcars, x=2, y=4:11, z=2)

Facet by z

Good for visualizating a lot of data quickly.

Something unexpected:

He’s working with a server that has a terabyte of RAM. $21k. Working with data.table in memory on it is his favorite way to handle lots of data.

Next steps: GPUs! 43x performance increase based on GPU. Got better speed than with Python.

Computer vision: nice results using data.table to store one-hot vectors.