JD Long dplyr talk

Dplyr is fast to write, and can write SQL, generally faster than *you* can write SQL.

JD Long’s github has all the code for this lecture.

**dplyr - set up example**

library(tidyverse)

mtcars %>%

rownames\_to\_column( var = 'longname') ->

df\_cars

**dplyr - select**

Is like select in sql; just keep 3 columns from original set.

Note that head is head() fct; can skip the () with pipe when only 1 argument

df\_cars %>%

select(longname, cyl, hp) %>%

head

longname cyl hp

1 Mazda RX4 6 110

2 Mazda RX4 Wag 6 110

3 Datsun 710 4 93

4 Hornet 4 Drive 6 110

5 Hornet Sportabout 8 175

6 Valiant 6 105

**dplyr - filter**

The fieldname is implied, saving typing. Don’t have to type mtcars over and over. This is the case for all dplyr fcts!

Here, we’re selecting names, cyl, and hp for cars where cyl is 8

df\_cars %>%

select(longname, cyl, hp) %>%

filter( cyl == 8) %>%

head

longname cyl hp

1 Hornet Sportabout 8 175

2 Duster 360 8 245

3 Merc 450SE 8 180

4 Merc 450SL 8 180

5 Merc 450SLC 8 180

6 Cadillac Fleetwood 8 205

**dplyr - mutate**

Using mutate to assign shorter names to ones with long names. Great time saver renaming.

df\_cars %>%

select(longname, cyl, hp) %>%

filter( cyl == 8) %>%

mutate( sqrt\_hp = sqrt(hp),

shortname = word(longname, 1) ) %>%

head

longname cyl hp sqrt\_hp shortname

1 Hornet Sportabout 8 175 13.22876 Hornet

2 Duster 360 8 245 15.65248 Duster

3 Merc 450SE 8 180 13.41641 Merc

4 Merc 450SL 8 180 13.41641 Merc

5 Merc 450SLC 8 180 13.41641 Merc

6 Cadillac Fleetwood 8 205 14.31782 Cadillac

**dplyr - group\_by**

Note that this doesn’t change your data, this is just a rearrangement. Way to summarize.

df\_cars %>%

select(longname, cyl, hp) %>%

filter( cyl == 8) %>%

mutate( sqrt\_hp = sqrt(hp),

shortname = word(longname, 1) ) %>%

select( - longname) %>%

group\_by(shortname) %>%

head

# A tibble: 6 x 4

# Groups: shortname [4]

cyl hp sqrt\_hp shortname

<dbl> <dbl> <dbl> <chr>

1 8. 175. 13.2 Hornet

2 8. 245. 15.7 Duster

3 8. 180. 13.4 Merc

4 8. 180. 13.4 Merc

5 8. 180. 13.4 Merc

6 8. 205. 14.3 Cadillac

**clean up - note changes**

Here, getting rid of longname column since we have shortname and it’s simpler.

df\_cars %>%

select(longname, cyl, hp) %>%

mutate( shortname = word(longname, 1) ) %>%

select( - longname) ->

df\_cars\_limited

head( df\_cars\_limited )

cyl hp shortname

1 6 110 Mazda

2 6 110 Mazda

3 4 93 Datsun

4 6 110 Hornet

5 8 175 Hornet

6 6 105 Valiant

**tangent: why pipe?**

head(

select(

mutate(

select(df\_cars, longname, cyl, hp) ,

shortname = word(longname, 1)

),

-longname

)

)

cyl hp shortname

1 6 110 Mazda

2 6 110 Mazda

3 4 93 Datsun

4 6 110 Hornet

5 8 175 Hornet

6 6 105 Valiant

**tangent: why pipe? (cont)**

class: small-code

df\_cars\_3vars <-

select(df\_cars, longname, cyl, hp)

df\_cars\_3vars\_shortname <-

mutate( df\_cars\_3vars,

shortname = word(df\_cars\_3vars$longname, 1) )

df\_cars\_3vars\_shortname <-

select( df\_cars\_3vars\_shortname, - longname)

head( df\_cars\_3vars\_shortname )

cyl hp shortname

1 6 110 Mazda

2 6 110 Mazda

3 4 93 Datsun

4 6 110 Hornet

5 8 175 Hornet

6 6 105 Valiant

**Before the tangent...**

df\_cars %>%

select(longname, cyl, hp) %>%

mutate( shortname = word(longname, 1) ) %>%

select( - longname) ->

df\_cars\_limited

head( df\_cars\_limited )

cyl hp shortname

1 6 110 Mazda

2 6 110 Mazda

3 4 93 Datsun

4 6 110 Hornet

5 8 175 Hornet

6 6 105 Valiant

**dplyr - summarize**

Here, averaging horsepower in groups of name and cylinder.

df\_cars\_limited %>%

group\_by(shortname, cyl)%>%

summarize( avg\_hp = mean(hp) ) %>%

head(3)

# A tibble: 3 x 3

# Groups: shortname [3]

shortname cyl avg\_hp

<chr> <dbl> <dbl>

1 AMC 8. 150.

2 Cadillac 8. 205.

3 Camaro 8. 245.

Note the grouping change. wtf?

Result is grouped by shortname, not by cylinder as well. Why? Bc summarize drops the last group\_by variable! It’s dropping grouping by cylinder.

Grouping by only 1 row is non-useful, so auto-drops this. That’s why summarize does this.

**dplyr - join**

Note that don’t have to tell it what column to join by! It can figure it out and just automatically join the two data frames (on cyl, the correct way to do so).

cyl\_map <- data.frame(

cyl = c(4,6,8),

cyl\_name = c('four','six','eight'))

df\_cars\_limited %>%

left\_join( cyl\_map ) %>%

head(3)

cyl hp shortname cyl\_name

1 6 110 Mazda six

2 6 110 Mazda six

3 4 93 Datsun four

**dplyr - case\_when (+ sort)**

Creates a factor variable “Pow”, which is Strong when car has hp > 200 and cyl >=8, and Weak otherwise.

df\_cars\_limited %>%

mutate(Pow= case\_when( hp>200 & cyl>=8 ~ 'Strong',

TRUE ~ 'Weak') ) %>%

arrange(-hp)

cyl hp shortname Pow

1 8 335 Maserati Strong

2 8 264 Ford Strong

3 8 245 Duster Strong

4 8 245 Camaro Strong

5 8 230 Chrysler Strong

6 8 215 Lincoln Strong

7 8 205 Cadillac Strong

8 8 180 Merc Weak

9 8 180 Merc Weak

10 8 180 Merc Weak

11 8 175 Hornet Weak

12 8 175 Pontiac Weak

13 6 175 Ferrari Weak

14 8 150 Dodge Weak

15 8 150 AMC Weak

16 6 123 Merc Weak

17 6 123 Merc Weak

18 4 113 Lotus Weak

19 6 110 Mazda Weak

20 6 110 Mazda Weak

21 6 110 Hornet Weak

22 4 109 Volvo Weak

23 6 105 Valiant Weak

24 4 97 Toyota Weak

25 4 95 Merc Weak

26 4 93 Datsun Weak

27 4 91 Porsche Weak

28 4 66 Fiat Weak

29 4 66 Fiat Weak

30 4 65 Toyota Weak

31 4 62 Merc Weak

32 4 52 Honda Weak

**mutate vs. summarize**

mutate adds a column but keeps same # rows

summarize adds a column but reduces the rows down to one row per unique group based on group\_by

**database time!**

Above is all local stuff, here is how to do database stuff.

con <- DBI::dbConnect(RSQLite::SQLite(),

path = ":memory:")

## could use memdb\_frame

This is connecting to data frame:

copy\_to(con, df\_cars, "df\_cars",

temporary = FALSE)

Can now act on database like it was a local data frame:

df\_cars\_db <- tbl(con, "df\_cars")

**what are these things?**

Tbl is tibble

class(df\_cars)

[1] "data.frame"

class(as.tibble(df\_cars))

[1] "tbl\_df" "tbl" "data.frame"

class(df\_cars\_db)

[1] "tbl\_dbi" "tbl\_sql" "tbl\_lazy" "tbl"

**so this looks sane...**

So, acting on our database:

df\_cars\_db %>%

group\_by( cyl) %>%

summarize( avg = mean(hp),

std = sd(hp) ) ->

mean\_hp\_by\_cyl

head( mean\_hp\_by\_cyl )

# Source: lazy query [?? x 3]

# Database: sqlite 3.22.0 []

cyl avg std

<dbl> <dbl> <dbl>

1 4. 82.6 20.9

2 6. 122. 24.3

3 8. 209. 51.0

This is done by an SQL query that dplyr makes behind the scenes!

**what's in mean\_hp\_by\_cyl?**

df\_cars\_db %>%

group\_by( cyl) %>%

summarize( avg = mean(hp),

std = sd(hp) ) %>%

show\_query

<SQL>

SELECT

`cyl`,

AVG(`hp`) AS `avg`,

STDEV(`hp`) AS `std`

FROM `df\_cars`

GROUP BY `cyl`

That’s the SQL dplyr wrote behind the scenes.

**automagic subqueries**

df\_cars\_db %>%

group\_by( cyl) %>%

summarize( avg = mean(hp),

std = sd(hp) ) %>%

filter( avg > 100) %>%

show\_query

<SQL>

SELECT \*

FROM ( SELECT `cyl`,

AVG(`hp`) AS `avg`,

STDEV(`hp`) AS `std`

FROM `df\_cars`

GROUP BY `cyl`

)

WHERE (`avg` > 100.0)

Nested queries in SQL are often a pain to write, but v useful for Redshift databases. Dplyr is a lifesaver for queries nested like 7 or more deep.

# lazy af...

dplyr doesn’t go get the data until needed. E.g., to answer query you made. Also, collect function goes and fetches from the database.

mean\_hp\_by\_cyl %>%

collect ->

local\_mean\_hp\_by\_cyl

head(local\_mean\_hp\_by\_cyl)

# A tibble: 3 x 3

cyl avg std

<dbl> <dbl> <dbl>

1 4. 82.6 20.9

2 6. 122. 24.3

3 8. 209. 51.0

class(local\_mean\_hp\_by\_cyl)

[1] "tbl\_df" "tbl" "data.frame"

# R Mapped to SQL

Can’t easily translate statistics functions into SQL – tend to be pretty disgusting. The right thing to do is bring a subset or sample of your data from the SQL database into R and do R stats on it in-memory in R.

math operators: +, -, \*, /, %%, ^

math functions: abs, acos, acosh, asin, asinh, atan, atan2, atanh, ceiling, cos, cosh, cot, coth, exp, floor, log, log10, round, sign, sin, sinh, sqrt, tan, tanh

logical comparisons: <, <=, !=, >=, >, ==, %in%

boolean operations: &, &&, |, ||, !, xor

basic aggregations: mean, sum, min, max, sd, var

string functions: tolower, toupper, trimws, nchar, substr

coerce types: as.numeric, as.integer, as.character

# KNEAD MOAR?

Dbplyr is behind-the-scenes package that does the SQL translation.

vignette("sql-translation")

# tbl\_lazy & simulate\_dbi

This is good way to debug; make a simulation of an SQLite database inside R, so that you can figure out what SQL you’d be writing. Great for making dummy examples to find out what you should do; and to figure out what’s happening to report bugs for other people.

library(dbplyr)

data.frame(x = 1, y = 2) %>%

tbl\_lazy(src = simulate\_sqlite()) ->

df\_sqlite ## this is a lazy table

df\_sqlite %>%

summarise(x = sd(x)) %>%

show\_query()

<SQL> SELECT STDEV(`x`) AS `x`

FROM `df`

# simulate functions

dbi sqlite

postgres mysql

odbc impala

mssql oracle

hive odbc\_postgresql

teradata odbc\_access

# Common Snags (for me)

1. Search Path: Easy to set up wrong.

dbSendQuery(con,

build\_sql("set search\_path to

'$user', public, reports, sandbox"))

1. Name Clashes on Join. If two tables both have a *bob* field, the automatic selection can screw that up. (Also, recall auto-renaming of duplicate fields in a single dataframe, e.g. bob, bob.x. That can also cause trouble).

## drop the duplicate field or rename

df\_cars\_db <- tbl(con, "df\_cars") %>% select(-wt)

# workflow with redshift

count(\*)

349,523,453,670

* pre define joins
* pass around a "big table"
* set up criteria
* transform
* collect

# WARNING!!!

dplyr can stomp on Core R functions:

head( lag(ldeaths, 12) )

[1] 3035 2552 2704 2554 2014 1655

head( lag(ldeaths, k=12) ==

head( lag(ldeaths, 12) ) )

[1] TRUE TRUE TRUE TRUE TRUE TRUE

lag() fct in base is *not* lag() fct in dplyr. Here, it’s bc timeseries object that dplyr doesn’t recognize.

# WARNING!!!

Now load dplyr

library(dplyr)

head( lag(ldeaths, 12) )

Error: `x` must be a vector, not a ts object, do you want `stats::lag()`?

head( lag(ldeaths, k=12) )

Error: `x` must be a vector, not a ts object, do you want `stats::lag()`?

# WARNING!!!

must clarify in your code:

library(dplyr)

head( stats::lag(ldeaths, 12) )

[1] 3035 2552 2704 2554 2014 1655

head( stats::lag(ldeaths, k=12) )

[1] 3035 2552 2704 2554 2014 1655

# WTF?!?! Hidden Failure!

vec\_dbl <- as.double(ldeaths)

head( lag(vec\_dbl, 12) )

[1] 3035 2552 2704 2554 2014 1655

library(dplyr)

head( lag(vec\_dbl, 12) )

NA NA NA NA NA

Those NAs can be catastrophic if you don’t know that’s going to happen. Have to test things in your existing codebase out before swapping it all over to tidyverse.

# row-wise design pattern

If a function isn’t vectorized, it won’t work with dplyr. This is approach to use to fix.

fun <- function(a, b, c){

sum(seq(a,b,c))

}

df <- data.frame(mn=c(1,2,3),

mx=c(8,13,18),

rng=c(1,2,3))

df

mn mx rng

1 1 8 1

2 2 13 2

3 3 18 3

@JennyBryan: <http://bit.ly/rowwise>

# row-wise

df %>%

mutate( output =

pmap(list(a=mn, b=mx, c=rng), fun)

)

mn mx rng output

1 1 8 1 36

2 2 13 2 42

3 3 18 3 63

# vector in a df cell

fun2 <- function(a, b, c){

seq(a,b,c)

}

df %>%

mutate( output =

pmap(list(a=mn, b=mx, c=rng), fun2)

)

mn mx rng output

1 1 8 1 1, 2, 3, 4, 5, 6, 7, 8

2 2 13 2 2, 4, 6, 8, 10, 12

3 3 18 3 3, 6, 9, 12, 15, 18