Like more posts than I care to admit, this one starts innocently enough with a tweet by [@gshotwell](http://twitter.com/gshotwell):

Is there a reference document somewhere of which dplyr commands work on various database backends? [#rstats](https://twitter.com/hashtag/rstats?src=hash&ref_src=twsrc%5Etfw)

— Gordon Shotwell ([@gshotwell](http://twitter.com/gshotwell)) [April 9, 2019](https://twitter.com/gshotwell/status/1115653121269796865?ref_src=twsrc%5Etfw)

Since I use at least 4 different d[b]plyr backends every week, this same question surfaces in my own noggin on occasion and I couldn’t resist going all Columbo on this mystery.

I should note that if you only really care about the backends that come with dbplyr [@paleolimbot](http://twitter.com/paleolimbot) has you covered [with this post](https://apps.fishandwhistle.net/archives/1503), which also shows you the translated SQL!

**Executing The Plan**

There are at least 24 separate backends for dbplyr. Most folks won’t need more than one if their databases all have a decent ODBC or JDBC driver. To be able to use dplyr idioms with databases there needs to be a way to translate R code (e.g. function calls) into SQL. [A ton of functions are pre-mapped](https://github.com/tidyverse/dbplyr/blob/master/R/backend-.R) in dbplyr already and most backend implementations start by relying on these defaults. Furthermore, since SQL is not nearly as “standard” across installations as one might think, some common tasks — such as string manipulation — [have a default noop translation](https://github.com/tidyverse/dbplyr/blob/master/R/backend-.R#L236-L267).

If you do have to switch across backends with any frequency, knowing which backend provides support for which functions might be nice, but there hasn’t been a reference for this until Dewey & I accepted Gordon’s challenge. What makes this a “challenge” is that you first have to figure out what packages provide a d[b]plyr backend interface then figure out what SQL translations they offer (they don’t necessarily have to inherit from the ones provided by dbplyr and may add other ones to account for SQL clauses that aren’t in functional form). So the first step was just a look through CRAN for which packages import dbplyr and also adding in some I knew were on GitHub:

library(stringi)

library(hrbrthemes)

library(tidyverse)

# All the pkgs from the home CRAN mirror that import 'dbplyr'

c(

"arkdb", "bigrquery", "childesr", "chunked", "civis", "corrr", "cytominer", "dbplot",

"dbplyr", "dexter", "dexterMST", "dlookr", "dplyr", "dplyr.teradata", "etl",

"healthcareai", "hydrolinks", "implyr", "infuser", "ipumsr", "macleish", "mdsr",

"mlbgameday", "modeldb", "MonetDBLite", "mudata2", "parsemsf", "pivot", "pleiades",

"pool", "poplite", "RClickhouse", "replyr", "RPresto", "sergeant", "sparklyr",

"sqlscore", "srvyr", "taxizedb", "valr", "wordbankr", "metis.tidy"

) -> pkgs

I ended up doing install.pkgs(pkgs) which was easy since I have a home CRAN mirror and use macOS (so binary package installs).

The presence of a dbplyr import does not mean a package implements a backend, so we have to load their namespaces and see if they have the core “tell” (i.e. they implement sql\_translate()):

(map\_df(pkgs, ~{

tibble(

pkg = .x,

trans = loadNamespace(.x) %>%

names() %>%

keep(stri\_detect\_fixed, "sql\_translate")

)

}) -> xdf)

## # A tibble: 28 x 2

## pkg trans

##

## 1 bigrquery sql\_translate\_env.BigQueryConnection

## 2 civis sql\_translate\_env.CivisConnection

## 3 dbplyr sql\_translate\_env.ACCESS

## 4 dbplyr sql\_translate\_env.Oracle

## 5 dbplyr sql\_translate\_env.SQLiteConnection

## 6 dbplyr sql\_translate\_env.Impala

## 7 dbplyr sql\_translate\_env.OdbcConnection

## 8 dbplyr sql\_translate\_env.MySQLConnection

## 9 dbplyr sql\_translate\_env.PqConnection

## 10 dbplyr sql\_translate\_env.PostgreSQLConnection

## # … with 18 more rows

Now we know the types of connections that package has SQL translation support for. But, we’re looking for the actual functions they provide. To discover that, we’re going to make dummy classed connection objects and get the translations they offer.

*However*, some may take the defaults from dbplyr and not override them so we also need to test if they use the sql\_not\_supported() noop, which we can do by seeing if the function body [has a call to stop()](https://github.com/tidyverse/dbplyr/blob/e867e8b64be285bf7fc137f5d5add9ff7e1c4b1f/R/translate-sql-helpers.R#L217) in it. We’re also going to ignore maths operators along the way:

(filter(xdf, stri\_detect\_fixed(trans, ".")) %>%

filter(trans != "sql\_translate\_env.NULL") %>% # ignore NULL

filter(trans != "sql\_translate\_env.Pool") %>% # ignore db connection pooling

filter(trans != "sql\_translate\_env.PrestoConnection") %>% # this one errored out

mutate(ƒ = map(trans, ~{

# get the sql translate functions

con <- NA

cls <- stri\_replace\_first\_fixed(.x, "sql\_translate\_env.", "")

class(con) <- cls

env <- sql\_translate\_env(con)

# but ^^ rly isn't a nice, tidy object, it's a list of environments

# with functions in it so we have to iterate through it to extract

# the function names.

map\_df(env, ~{

part <- .x

fs <- names(part)

# but it's not just good enough to do that b/c a given function name

# might just implement the "sql\_not\_supported()" pass through. So we have

# to actually look to see if the function body has a "stop()" call in it

# and ignore it if it does.

map\_df(fs, ~{

tibble(ƒ = .x, src = paste0(as.character(body(part[[.x]])), collapse = "; ")) %>% # this gets the body of the function

filter(!stri\_detect\_fixed(src, "stop(")) %>%

filter(stri\_detect\_regex(ƒ, "[[:alpha:]]")) %>% # and we rly don't care about maths

select(-src)

})

})

})) %>%

unnest(ƒ) %>%

mutate(trans = stri\_replace\_first\_fixed(trans, "sql\_translate\_env.", "")) -> xdf)

## # A tibble: 1,318 x 3

## pkg trans ƒ

##

## 1 bigrquery BigQueryConnection median

## 2 bigrquery BigQueryConnection gsub

## 3 bigrquery BigQueryConnection as.logical

## 4 bigrquery BigQueryConnection is.null

## 5 bigrquery BigQueryConnection case\_when

## 6 bigrquery BigQueryConnection is.na

## 7 bigrquery BigQueryConnection if\_else

## 8 bigrquery BigQueryConnection str\_replace\_all

## 9 bigrquery BigQueryConnection as.integer

## 10 bigrquery BigQueryConnection as.character

## # … with 1,308 more rows

The rest is all just ggplot2 basics:

mutate(xdf, db = glue::glue("{pkg}\n{trans}")) %>% # make something useful to display for the DB/conn

mutate(n = 1) %>% # heatmap block on

complete(db, ƒ) %>% # complete the heatmap

arrange(ƒ) %>%

mutate(ƒ = factor(ƒ, levels=rev(unique(ƒ)))) %>% # arrange the Y axis in the proper order

ggplot(aes(db, ƒ)) +

geom\_tile(aes(fill = n), color="#2b2b2b", size=0.125, show.legend=FALSE) +

scale\_x\_discrete(expand=c(0,0.1), position = "top") +

scale\_fill\_continuous(na.value="white") +

labs(

x = NULL, y = NULL,

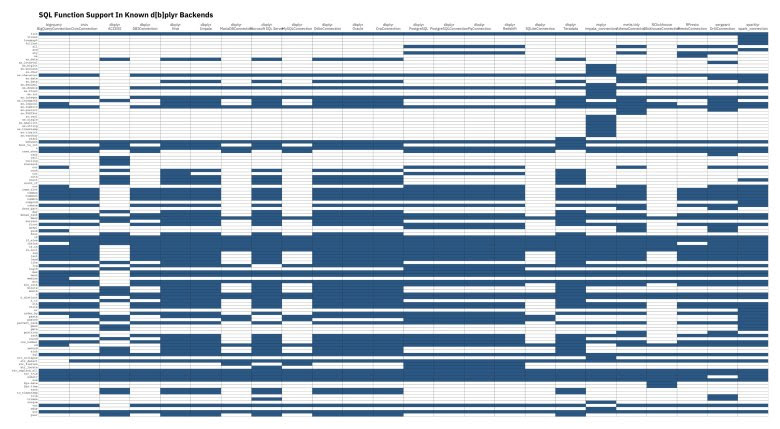
title = "SQL Function Support In Known d[b]plyr Backends"

) +

theme\_ipsum\_ps(grid="", axis\_text\_size = 9) + # you'll need to use the dev version of hrbrthemes for this function; just sub out a diff theme if you already have hrbrthemes loaded

theme(axis.text.y = element\_text(family = "mono", size = 7))

Which makes:

[](https://i2.wp.com/rud.is/b/wp-content/uploads/2019/04/backend-heatmap.jpg?ssl=1)

(WP wouldn’t make the featured image linkable so I had to stick it in again to enabled the link so folks can make it full size which is absolutely necessary to see it).