Queries that take significant processing time or have large result sets do not play nicely with the provided ODBC and JDBC drivers. This means “hung” R sessions and severe frustration, especially when you can login to the AWS Athena console and see that the results are *right there*!!

I’ve been crafting SQL by hand or using sql\_render() by hand to avoid this (when I remember to) but finally felt sufficient frustration to craft a better way, provided you can install and run rJava-based code (it’s 2018 and that still is not an easy given on many systems unfortunately).

There are two functions below:

* collect\_async(), and
* gather\_results()

The collect\_async() function is designed to be used like collect() but uses Athena components from the AWS SDK for Java to execute the SQL query behind the dplyr chain *asynchronously*. The companion function gather\_results() takes the object created by collect\_async() and checks to see if the results are ready. If if they are, it will use the aws.s3 package to download them. Personally, I’d just aws s3 sync ... from the command line vs use the aws.s3 package but that’s not everyone’s cup of tea.

Once I figure out the best package API for this I’ll add it to the metis package. There are many AWS idiosyncrasies that need to be accounted for and I’d rather ship this current set of functions via the blog so folks can use it (and tweak it to their needs) before waiting for perfection.

Here’s the code:

library(rJava)

library(awsjavasdk)

library(aws.signature)

library(aws.s3)

library(odbc)

library(tidyverse)

library(dbplyr)

#' Collect Amazon Athena query results asynchronously

#'

#' Long running Athena queries and Athena queries with large result

#' sets can seriously stall a `dplyr` processing chain due to poorly

#' implemented ODBC and JDBC drivers. The AWS SDK for Athena has

#' methods that support submitting a query asynchronously for "batch"

#' processing. All Athena resutls are stored in CSV files in S3 and it's

#' easy to use the R `aws.s3` package to grab these or perform an

#' `aws s3 sync ...` operation on the command line.

#'

#' [@md](http://twitter.com/md)

#' [@param](http://twitter.com/param) obj the `dplyr` chain

#' [@param](http://twitter.com/param) schema Athena schema (usually matches the `Schema` parameter to the

#' Simba ODBC connection)

#' [@param](http://twitter.com/param) region Your AWS region. All lower case with dashes (usually matches

#' the `AwsRegion` parameter to the Simba ODBC connection)

#' [@param](http://twitter.com/param) results\_bucket the S3 results bucket where query results are stored

#' (usually matches the `S3OutputLocation` parameter to the Simba ODBC

#' connection)

#' [@return](http://twitter.com/return) a `list` with the query execution ID and the S3 bucket. This object

#' is designed to be passed to the companion `gather\_results()` if you

#' want to use the `aws.s3` package to retrieve the results. Otherwise,

#' sync the file however you want using the query execution id.

#' [@note](http://twitter.com/note) You may need to change up the authentication provider depending on how

#' you use credentials with Athena

collect\_async <- function(obj, schema, region, results\_bucket) {

ugly\_query <- as.character(sql\_render(obj))

region <- toupper(region)

region <- gsub("-", "\_", region, fixed=TRUE)

regions <- J("com.amazonaws.regions.Regions")

available\_regions <- grep("^[[:upper:][:digit:]\_]+$", names(regions), value=TRUE)

if (!region %in% available\_regions) stop("Invalid region.", call.=FALSE)

switch(

region,

"GovCloud" = regions$GovCloud,

"US\_EAST\_1" = regions$US\_EAST\_1,

"US\_EAST\_2" = regions$US\_EAST\_2,

"US\_WEST\_1" = regions$US\_WEST\_1,

"US\_WEST\_2" = regions$US\_WEST\_2,

"EU\_WEST\_1" = regions$EU\_WEST\_1,

"EU\_WEST\_2" = regions$EU\_WEST\_2,

"EU\_WEST\_3" = regions$EU\_WEST\_3,

"EU\_CENTRAL\_1" = regions$EU\_CENTRAL\_1,

"AP\_SOUTH\_1" = regions$AP\_SOUTH\_1,

"AP\_SOUTHEAST\_1" = regions$AP\_SOUTHEAST\_1,

"AP\_SOUTHEAST\_2" = regions$AP\_SOUTHEAST\_2,

"AP\_NORTHEAST\_1" = regions$AP\_NORTHEAST\_1,

"AP\_NORTHEAST\_2" = regions$AP\_NORTHEAST\_2,

"SA\_EAST\_1" = regions$SA\_EAST\_1,

"CN\_NORTH\_1" = regions$CN\_NORTH\_1,

"CN\_NORTHWEST\_1" = regions$CN\_NORTHWEST\_1,

"CA\_CENTRAL\_1" = regions$CA\_CENTRAL\_1,

"DEFAULT\_REGION" = regions$DEFAULT\_REGION

) -> region

provider <- J("com.amazonaws.auth.DefaultAWSCredentialsProviderChain")

client <- J("com.amazonaws.services.athena.AmazonAthenaAsyncClientBuilder")

my\_client <- client$standard()

my\_client <- my\_client$withRegion(region)

my\_client <- my\_client$withCredentials(provider$getInstance())

my\_client <- my\_client$build()

queryExecutionContext <- .jnew("com.amazonaws.services.athena.model.QueryExecutionContext")

context <- queryExecutionContext$withDatabase(schema)

result <- .jnew("com.amazonaws.services.athena.model.ResultConfiguration")

result$setOutputLocation(results\_bucket)

startQueryExecutionRequest <- .jnew("com.amazonaws.services.athena.model.StartQueryExecutionRequest")

startQueryExecutionRequest$setQueryString(ugly\_query)

startQueryExecutionRequest$setQueryExecutionContext(context)

startQueryExecutionRequest$setResultConfiguration(result)

res <- my\_client$startQueryExecutionAsync(startQueryExecutionRequest)

r <- res$get()

qex\_id <- r$getQueryExecutionId()

list(

qex\_id = qex\_id,

results\_bucket = results\_bucket

)

}

#' Gather the results of an asynchronous query

#'

#' [@md](http://twitter.com/md)

#' [@param](http://twitter.com/param) async\_result the result of a call to `collect\_async()`

#' [@return](http://twitter.com/return) a data frame (tibble) or `NULL` if the query results are not ready yet

gather\_results <- function(async\_result) {

if (bucket\_exists(sprintf("%s/%s", async\_result$results\_bucket, async\_result$qex\_id))) {

readr::read\_csv(

get\_object(sprintf("%s/%s.csv", async\_result$results\_bucket, async\_result$qex\_id))

)

} else {

message("Results are not in the designated bucket.")

return(NULL)

}

}

Now, we give it a go:

# Setup the credentials you're using

use\_credentials("personal")

# load the AWS Java SDK classes

awsjavasdk::load\_sdk()

# necessary for Simba ODBC and the async query ops

aws\_region <- "us-east-1"

athena\_schema <- "sampledb"

athena\_results\_bucket <- "s3://aws-athena-query-results-redacted"

# connect to Athena and the sample database

DBI::dbConnect(

odbc::odbc(),

driver = "/Library/simba/athenaodbc/lib/libathenaodbc\_sbu.dylib",

Schema = athena\_schema,

AwsRegion = aws\_region,

AuthenticationType = "IAM Profile",

AwsProfile = "personal",

S3OutputLocation = athena\_results\_bucket

) -> con

# the sample table in the sample db/schema

elb\_logs <- tbl(con, "elb\_logs")

# create your dplyr chain. This one is small so I don't incur charges

# collect\_async() MUST be the LAST item in the dplyr chain.

elb\_logs %>%

filter(requestip == "253.89.30.138") %>%

collect\_async(

schema = athena\_schema,

region = aws\_region,

results\_bucket = athena\_results\_bucket

) -> async\_result

async\_result

## $qex\_id

## [1] "d5fe7754-919b-47c5-bd7d-3ccdb1a3a414"

##

## $results\_bucket

## [1] "s3://aws-athena-query-results-redacted"

# For long queries we can wait a bit but the function will tell us if the results

# are there or not.

gather\_results(async\_result)

## Parsed with column specification:

## cols(

## timestamp = col\_datetime(format = ""),

## elbname = col\_character(),

## requestip = col\_character(),

## requestport = col\_integer(),

## backendip = col\_character(),

## backendport = col\_integer(),

## requestprocessingtime = col\_double(),

## backendprocessingtime = col\_double(),

## clientresponsetime = col\_double(),

## elbresponsecode = col\_integer(),

## backendresponsecode = col\_integer(),

## receivedbytes = col\_integer(),

## sentbytes = col\_integer(),

## requestverb = col\_character(),

## url = col\_character(),

## protocol = col\_character()

## )

## # A tibble: 1 x 16

## timestamp elbname requestip requestport backendip backendport

##

## 1 2014-09-29 03:24:38 lb-demo 253.89.30.138 20159 253.89.30.138 8888

## # ... with 10 more variables: requestprocessingtime , backendprocessingtime ,

## # clientresponsetime , elbresponsecode , backendresponsecode ,

## # receivedbytes , sentbytes , requestverb , url , protocol