R Training Notes

Lesson 2

Stefanie Molin

April 13, 2017

I. Data Extraction

Now, we are going to build on the knowledge from the last session and learn how to use the `RJDBC` package to query Vertica for data.

A. Getting data

Every analysis starts with data, but where do we get it from and how do we get it into R?

1. Querying Vertica

The RJDBC package provides us with the framework needed to have R use a local Vertica driver (.jar file) to query Vertica. The following function will query Vertica and return a dataframe for use in R.

```
# Set .jar file location and Vertica connection string
driverLocation <- "C:\\Program Files\\Vertica Systems\\JDBC\\vertica-jdbc-7.[redacted]-0.jar"
v <- "jdbc:vertica://[redacted]:[redacted]/[redacted]?ConnectionLoadBalance=true"

#' @description Query Vertica for the specified query. Return a dataframe of the results.
#'
#' @param username Vertica login
#' @param query Vertica query (be careful with comments)
#' @param password Vertica password to access the database
#'
#' @return dataframe of results
#'
#' @note Queries are turned into strings so if you comment with "--" the result will be
#' a string with everything after that commented out. Be sure to remove those beforehand.
#'

QueryVertica <- function (username, query, password){
  msg.out <- capture.output(suppressMessages(require(RJDBC)))
  drv <- JDBC("com.vertica.jdbc.Driver", driverLocation)
  conn <- dbConnect(drv, v, username, password)
  data <- dbGetQuery(conn, query)
  dbDisconnect(conn)
  return(data)
}
```

Let's run an example. Note that I have already specified my password as the variable `password` and my username ("s.molin") as `username`.

```
# write the query
quickQuery <- "SELECT * FROM [redacted]"

# store query results in a dataframe
df <- QueryVertica(username, quickQuery, password)

# check how much data was returned
dim(df)
```

```
## [1] 35273    39
```

```
# see which columns we got back
colnames(df)
```

```
##  [1] [redacted]
##  [3]
##  [5]
##  [7]
##  [9]
## [11]
## [13]
## [15]
## [17]
## [19]
## [21]
## [23]
## [25]
## [27]
## [29]
## [31]
## [33]
## [35]
## [37]
## [39]
```

```
# inspect first 10 rows of the result for specific columns
head(df[, c("client_name", "ranking", "client_country_code")], 10)
```

```
##                 client_name ranking client_country_code
## 1  [redacted]
## 2
## 3
## 4
## 5
## 6
## 7
## 8
## 9
## 10
```

2. Reading in data from files

Another common way of getting data is through files (.csv, .txt, .xls, etc.).

```
# be sure to specify stringsAsFactors as FALSE here!
df2 <- read.csv("C:\\Users\\S.Molin\\Documents\\Tickets\\[redacted]January.csv",
                stringsAsFactors = FALSE)

# how many rows of data did we pull in?
nrow(df2)
```

```
## [1] 153702
```

```
# see which columns are available
colnames(df2)
```

```
##  [1] "day"               "campaign_name"     "hotel_code"
##  [4] "property"          "property_location" "country"
##  [7] "displays"          "clicks"            "spend"
## [10] "cpc"
```

```
# inspect select columns of the data
head(df2[, c("day", "property", "property_location")])
```

```
##        day                    property property_location
## 1 1/1/2017                        logo              NULL
## 2 1/2/2017 [redacted]  Andalusia, AL
## 3 1/3/2017 [redacted]    Absecon, NJ
## 4 1/4/2017 [redacted]   Aberdeen, MD
## 5 1/5/2017 [redacted]  Allentown, PA
## 6 1/6/2017 [redacted]  Allentown, PA
```

For other types of files, consult `?read.table` help section which contains the base function–`read.table()`–as well as the wrapper functions for specific cases, such as TSV–`read.delim()`–and CSV–`read.csv()`. Reading Excel sheets in is a little more complicated, but there are several packages out there to choose from. (I use XLConnect.)

Note that for large files, it is recommended to use ***`fread()`*** *from the* ***`data.table`*** *package for a faster import than base R. It also detects automatically certain arguments that make reading in CSV files easier.*

B. Dynamic queries

Now that we know *how* to get data from Vertica, we need to know how to make reusable queries. The query we used earlier always does the same thing, but sometimes we need our queries to be dynamic so that they are flexible and can be used for multiple situations. These "dynamic" queries will allow us to change query parameters per analysis without having to update the query each time. Here are 2 ways to do this:

1. `sprintf()`

This method will search for special replace characters (we will focus on `%s`) and replace them in the order they appear in the string with the designated replacement values. Values will be placed into the string without quotes, so be sure to surround your `%s` with quotes if that part of your query requires them.

`sprintf()` is used as `sprintf(string, replace1[, replace2, ...])`

```
# query for client names in a specific country and ranking
sampleQuery <- "
SELECT
  client_name
FROM
  [redacted]
WHERE
  ranking = '%s'
  AND client_country_code = '%s'
GROUP BY
  client_name
ORDER BY
  client_name
"

# define which clients we want to pull
client_ranking <- "TIER 1"
country <- "US"

# use sprintf() to fill in the query according to the above specifications
query <- sprintf(sampleQuery, client_ranking, country)

# query has been modified
cat(query)
```

```
##
## SELECT
##   client_name
## FROM
##   [redacted]
## WHERE
##   ranking = 'TIER 1'
##   AND client_country_code = 'US'
## GROUP BY
##   client_name
## ORDER BY
##   client_name
```

```
# query vertica and show first few results
clients <- QueryVertica(username, query, password)
head(clients)
```

```
##                                client_name
## 1 [redacted]
## 2
## 3
## 4
## 5
## 6
```

Note that % is a special character in this case, and if your query has % in it anywhere else (i.e. client_name ilike '%[redacted]%') you will have issues with this method.

2. paste()

The `paste()` method is a great workaround when your query has special characters in it. Be careful to use the `sep = ""` argument to avoid extra spacing in your query or use `paste0()` instead which takes care of the empty string separator for you!

```
# query pieces for client names in a specific country and ranking
sampleQuery1 <- "
SELECT
  client_name
FROM
 [redacted]
WHERE
  ranking = '"

sampleQuery2 <- "'
  AND client_country_code = '"

sampleQuery3 <- "'
GROUP BY
  client_name
ORDER BY
  client_name
"

# define which clients we want to pull
client_ranking <- "TIER 1"
country <- "US"

# use paste0() to fill in the query according to the above specifications
query <- paste0(sampleQuery1, client_ranking, sampleQuery2, country, sampleQuery3)
```

```
# query has been modified
cat(query)
```

```
##
## SELECT
##   client_name
## FROM
##   [redacted]
## WHERE
##   ranking = 'TIER 1'
##   AND client_country_code = 'US'
## GROUP BY
##   client_name
## ORDER BY
##   client_name
```

```
# query vertica and show first few results
clients <- QueryVertica(username, query, password)
head(clients)
```

```
##                                client_name
## 1 [redacted]
## 2
## 3
## 4
## 5
## 6
```

Notice how both methods yield the same results.

C. Combining dataframes

Querying Vertica is great, but sometimes you have a very large query you can't run all at once or are running two separate parts of an analysis and want to combine the data into one dataframe without rewriting your queries or turning them into one giant query. Good news: you can take care of all of this in R! There are 2 ways of combining dataframes in base R:

1. `merge()`

Merges are joins of 2 dataframes. By default, R will `join` them on any columns that have the same name. You can specify the `ON` columns in the `by` argument if you don't want to join based on *all* matching columns and `by.x` and `by.y` if they are named differently in each dataframe.

- Inner join: `merge(df_x, df_y)`
- Left outer join: `merge(df_x, df_y, all.x = TRUE)`
- Right outer join: `merge(df_x, df_y, all.y = TRUE)`
- Full outer join: `merge(df_x, df_y, all = TRUE)`

```r
# define dataframes
ids <- data.frame(name = c("Alice", "Bob", "Carly", "Dylan"), id = 101:104,
                  stringsAsFactors = FALSE)
ages <- data.frame(age = c(24, 26, 28), id = c(101:102, 105))

# view each dataframe
ids
```

```
##    name  id
## 1 Alice 101
## 2   Bob 102
## 3 Carly 103
## 4 Dylan 104
```

```r
ages
```

```
##   age  id
## 1  24 101
## 2  26 102
## 3  28 105
```

```r
# inner join
merge(ids, ages)
```

```
##    id  name age
## 1 101 Alice  24
## 2 102   Bob  26
```

```r
# left outer join
merge(ids, ages, all.x = TRUE)
```

```
##    id  name age
## 1 101 Alice  24
## 2 102   Bob  26
## 3 103 Carly  NA
## 4 104 Dylan  NA
```

```r
# right outer join
merge(ids, ages, all.y = TRUE)
```

```
##    id  name age
## 1 101 Alice  24
## 2 102   Bob  26
## 3 105  <NA>  28
```

```r
# full outer join
merge(ids, ages, all = TRUE)
```

```
##    id  name age
## 1 101 Alice  24
## 2 102   Bob  26
## 3 103 Carly  NA
## 4 104 Dylan  NA
## 5 105  <NA>  28
```

2. Binding

Binding doesn't perform a join, but rather adds all of one dataframe to another either by column–`cbind()`–or by row–`rbind()`. Note that to use `cbind()` your dataframes must have the same number of *rows* and to use `rbind()` they must have the same number of *columns* (and be in the same order for your end result to make sense). You can think of an `rbind` as a `UNION ALL` and a `cbind` as a `UNION ALL` but on the transpose.

i. `rbind()`

Let's continue with the `ids` dataframe from above. We now have additional employees to add to the table. We can do this with `rbind()` since we have the same number of columns and column names.

```
# new employees
newEmployees <- data.frame(name = c("Eva", "Frank"), id = 106:107,
                           stringsAsFactors = FALSE)

# revise the ids dataframe to include all employees
(ids <- rbind(ids, newEmployees))
```

```
##    name  id
## 1 Alice 101
## 2   Bob 102
## 3 Carly 103
## 4 Dylan 104
## 5   Eva 106
## 6 Frank 107
```

As in the last line of code above, you can wrap variable assignment in parenthesis to print the value assigned to a variable after assigning it.

ii. `cbind()`

Suppose we want to add the office each employee works in to the `ids` dataframe, this can be accomplished with `cbind()` once we assure we have a dataframe with the proper number of rows and that the office order aligns with the order of the `ids` dataframe.

```
# create vector of offices for each employee
office <- c("NYC", "LA", "LON", "MUC", "MAD", "SF")

# show result of adding the offices to the employees table
cbind(ids, office)
```

```
##    name  id office
## 1 Alice 101    NYC
## 2   Bob 102     LA
## 3 Carly 103    LON
## 4 Dylan 104    MUC
## 5   Eva 106    MAD
## 6 Frank 107     SF
```

Note: Be careful that your dataframes' observations line up row-by-row, otherwise you will have mixed up rows. Also, since we are just adding one column, this can be done without `cbind()` *as* `ids$office <- c("NYC", "LA", "LON", "MUC", "MAD", "SF")`*.*