**Introduction**

In this new expanded post, we will focus on the following:

* connect to & explore database
* read & write data
* use RStudio SQL script & knitr SQL engine
* query data using dplyr
* visualize data with dbplot
* modeling data with modeldb & tidypredict
* explore RStudio connections pane
* handling credentials

**Libraries**

Before we connect to and explore the local SQLite database, let us take a look at the R packages we will use in this post.

* DBI a database interface for R
* dbplyr a dplyr backend for databases
* dplyr for querying data
* dbplot & ggplot2 for data visualization
* modeldb & tidypredict for modeling & prediction inside database
* config for handling credentials

# install.packages(c("DBI", "dbplyr", "dplyr", "dbplot", "ggplot2", "modeldb",

# "tidypredict", "config"))

library(DBI)

library(dbplyr)

library(dplyr)

library(dbplot)

library(ggplot2)

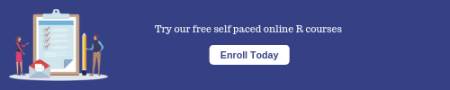
library(modeldb)

library(tidypredict)

library(config)

If you do not have all the above packages installed, go ahead and install them. In the R script we are sharing with you, we have commented out the code for installing the packages. If you are using the RStudio Cloud project, we have already installed the packages in the project and you can just load them into the R session using library().

As and when we come to the specific sections where we are using these packages, they will be reintroduced and we will look at their documentation and explore the functions we will use.

[](https://rsquared-academy.thinkific.com/)

**Connect & Explore**

The first step is to successfully connect to a database. To begin with, we will keep things simple and connect to a local **SQLite** database, mydatabase.db. We will explore the database, list the tables present and the fields/columns in those tables. In the last section of this post, we will connect to a **MySQL** database hosted on AWS using RStudio connections tab and learn how to specify the host, port, username, password etc.

**Connect**

To connect to the database, we will use dbConnect() from the DBI package which defines a common interface between R and database management systems. The first input is the database driver which in our case is SQLite and the second input is the name and location of the database. Since we are connecting to a local database, it is sufficient to specify the name and location of the database. The database connection is saved in con for further use in exploring and querying data.

con <- DBI::dbConnect(RSQLite::SQLite(), dbname = "mydatabase.db")

con

##

## Path: J:\R\Others\blogs\content\post\mydatabase.db

## Extensions: TRUE

If the database is present and the correct path to the database has been specified, R will not return any error.

**Connection Summary**

Next, let us get a quick summary of the database connection using summary(). It shows SQLiteConnection under class and we can ignore the other details for the time being. Great! We have successfully connected to the database and now it is time to list the tables present in the database.

summary(con)

## Length Class Mode

## 1 SQLiteConnection S4

**List Tables**

Now that we are connected to a database, let us list all the tables present in it using dbListTables().

DBI::dbListTables(con)

## [1] "COMPANY" "DEPARTMENT" "ecom" "trade"

There are 4 tables in the database and we will be using only the ecom and trade tables in our examples and practice questions.

**List Fields**

Let us go ahead and list all the fields/colums in the ecom table since we will be using it in the following sections. To list all the fields (columns) in a table, use dbListFields(). It takes 2 inputs:

* the database connection name (con)
* name of the table (ecom) enclosed in single/double quotes

DBI::dbListFields(con, "ecom")

## [1] "id" "referrer" "device" "bouncers" "n\_visit"

## [6] "n\_pages" "duration" "country" "purchase" "order\_items"

## [11] "order\_value"

There are 11 columns in the ecom table. Let us take a look at the data dictionary to understand what these columns stand for:

* id: row id
* referrer: referrer website/search engine
* os: operating system
* browser: browser
* device: device used to visit the website
* n\_pages: number of pages visited
* duration: time spent on the website (in seconds)
* repeat: frequency of visits
* country: country of origin
* purchase: whether visitor purchased
* order\_value: order value of visitor (in dollars)

Now that we know how to connect to a database and list the fields/columns, let us move on to the next section where we will learn how to query data from the tables present in the database.

**Querying Data**

Now that we have successfully connected to the database and explored the tables, let us look at querying data from the ecom table. In this section,  
we will learn to:

* read entire table
* read few rows
* read data in batches

**Entire Table**

We can read an entire table from a database using dbReadTable() provided the table is not very large. We will read data from the COMPANY table as it has few rows and will not fill the whole page. The first input is the database connection name and the second input is the name of the table in quotes.

DBI::dbReadTable(con, 'COMPANY')

## ID NAME AGE ADDRESS SALARY

## 1 1 Paul 32 California 20000

## 2 2 Allen 25 Texas 15000

## 3 3 Teddy 23 Norway 20000

## 4 4 Mark 25 Rich-Mond 65000

## 5 5 David 27 Texas 85000

## 6 6 Kim 22 South-Hall 45000

In some cases, we may not need the entire table but only a specific number of rows. Use dbGetQuery() and supply a SQL statement specifying the number of rows of data to be read from the table. In the below example, we read ten rows of data from the ecom table.

**Few Rows**

DBI::dbGetQuery(con, "select \* from ecom limit 10")

## id referrer device bouncers n\_visit n\_pages duration country

## 1 1 google laptop true 10 1 693 Czech Republic

## 2 2 yahoo tablet true 9 1 459 Yemen

## 3 3 direct laptop true 0 1 996 Brazil

## 4 4 bing tablet false 3 18 468 China

## 5 5 yahoo mobile true 9 1 955 Poland

## 6 6 yahoo laptop false 5 5 135 South Africa

## 7 7 yahoo mobile true 10 1 75 Bangladesh

## 8 8 direct mobile true 10 1 908 Indonesia

## 9 9 bing mobile false 3 19 209 Netherlands

## 10 10 google mobile true 6 1 208 Czech Republic

## purchase order\_items order\_value

## 1 false 0 0

## 2 false 0 0

## 3 false 0 0

## 4 true 6 434

## 5 false 0 0

## 6 false 0 0

## 7 false 0 0

## 8 false 0 0

## 9 false 0 0

## 10 false 0 0

In case of very large table, we can read data in batches using dbSendQuery() and dbFetch(). We can mention the number of rows of data to be read while fetching the data using the query generated by dbSendQuery().

**Read Data in Batches**

query <- DBI::dbSendQuery(con, 'select \* from ecom')

# first batch of 10 rows

DBI::dbFetch(query, n = 10)

## id referrer device bouncers n\_visit n\_pages duration country

## 1 1 google laptop true 10 1 693 Czech Republic

## 2 2 yahoo tablet true 9 1 459 Yemen

## 3 3 direct laptop true 0 1 996 Brazil

## 4 4 bing tablet false 3 18 468 China

## 5 5 yahoo mobile true 9 1 955 Poland

## 6 6 yahoo laptop false 5 5 135 South Africa

## 7 7 yahoo mobile true 10 1 75 Bangladesh

## 8 8 direct mobile true 10 1 908 Indonesia

## 9 9 bing mobile false 3 19 209 Netherlands

## 10 10 google mobile true 6 1 208 Czech Republic

## purchase order\_items order\_value

## 1 false 0 0

## 2 false 0 0

## 3 false 0 0

## 4 true 6 434

## 5 false 0 0

## 6 false 0 0

## 7 false 0 0

## 8 false 0 0

## 9 false 0 0

## 10 false 0 0

# second batch of 10 rows

DBI::dbFetch(query, n = 10)

## id referrer device bouncers n\_visit n\_pages duration country

## 1 11 direct laptop true 9 1 738 Jamaica

## 2 12 direct tablet false 6 12 132 Estonia

## 3 13 direct mobile false 9 14 406 Ireland

## 4 14 yahoo tablet false 5 8 80 Philippines

## 5 15 yahoo mobile false 7 1 19 France

## 6 16 bing laptop true 1 1 995 United States

## 7 17 bing tablet false 5 16 368 Peru

## 8 18 google tablet true 7 1 406 China

## 9 19 social tablet false 7 10 290 Colombia

## 10 20 social tablet false 2 1 28 Namibia

## purchase order\_items order\_value

## 1 false 0 0

## 2 false 0 0

## 3 true 3 651

## 4 false 2 362

## 5 false 7 2423

## 6 false 0 0

## 7 true 6 1049

## 8 false 0 0

## 9 true 9 1304

## 10 false 7 2077

**Your Turn**

* list fields in the trade table
* read all rows & columns from the trade table
* read first 30 rows from the trade table

**Query**

In this section, we will look at a bunch of functions that will allow us to get information about the query we sent to the database in the previous section to fetch data in batches. Before we start, let us look at the output from query.

query

##

## SQL select \* from ecom

## ROWS Fetched: 20 [incomplete]

## Changed: 0

We can see the follwing:

* SQL statement
* number of rows fetched (15)
* status of the query (incomplete)
* and number of rows changed in the table (0)

The status is incomplete as we are querying data in batches and the number of rows changed is 0 since ran a SELECT SQL statement which does not modify the table.

**Query Status**

To know the status of a query, use dbHasCompleted(). It is very useful in  
cases of queries that might take a long time to complete. It will return a logical value i.e. TRUE or FALSE. In our example, since we are querying data in batches, the output will be FALSE.

DBI::dbHasCompleted(query)

## [1] FALSE

**Query Info**

dbGetInfo() will display the following:

* SQL statement being executed
* the count of rows fetched/affected
* and the status of the query

DBI::dbGetInfo(query)

## $statement

## [1] "select \* from ecom"

##

## $row.count

## [1] 20

##

## $rows.affected

## [1] 0

##

## $has.completed

## [1] FALSE

The output is similar to what we saw when we printed query.

**Latest Query**

To view the query used in dbSendQuery() or dbSendStatement(), use dbGetStatement().

DBI::dbGetStatement(query)

## [1] "select \* from ecom"

**Rows Fetched**

dbGetRowCount() will return the total number of rows actually fetched from the table in the database.

DBI::dbGetRowCount(query)

## [1] 20

**Rows Affected**

The total number of rows added, deleted or updated by a data manipulation statement is returned by dbGetRowsAffected(). In our example, the SQL statemet did not modify the table in any way and hence the output will be 0.

DBI::dbGetRowsAffected(query)

## [1] 0

**Column Info**

dbColumnInfo() returns a data.frame that describes the output of a query. In the below example, it returns the column name and data type of the output from the query.

DBI::dbColumnInfo(query)

## name type

## 1 id integer

## 2 referrer character

## 3 device character

## 4 bouncers character

## 5 n\_visit integer

## 6 n\_pages double

## 7 duration double

## 8 country character

## 9 purchase character

## 10 order\_items double

## 11 order\_value double

**Clear Results**

To free all resources associated with a result set, use dbClearResult(). After running the below code, we will not be able to retrieve any information about query i.e. try running dbGetInfo(query) or dbGetStatement(query) and R will return an error.

DBI::dbClearResult(query)

**Tables**

In this section, we will learn to:

* check if a table exists
* create table
* overwrite table
* append data to a table
* insert rows into a table
* append one table to another
* remove a table

**Check Table Name**

It is a good practice to check whether a table of the same name exists before trying to create a new one in the database. In the below example, we use dbExistsTable() to check if the database already has a table by the name trial\_db. dbExistsTable() always returns a logical value.

DBI::dbExistsTable(con, "trial\_db")

## [1] FALSE

Since there is no table by the name trial\_db, let us go ahead and create a new table of the same name.

**Create Table**

To create a new table, use dbWriteTable(). It takes the following 3 arguments:

* connection name
* name of the new table
* data for the new table

Let us first create a new dataset trial\_db. It has 2 columns, x and y, and 10 rows of data. Next, we create a new table of the same name in the database. In dbWriteTable(), we specify the following:

* con: database connection
* "trial\_db": name of the table in database
* trial\_data: name of the dataset used to create the table in the database

Ensure that the name of the table in the database is always enclosed in single/double quotes.

# sample data

x <- 1:10

y <- letters[1:10]

trial\_data <- tibble::tibble(x, y)

# write table to database

DBI::dbWriteTable(con, "trial\_db", trial\_data)

Let us check if the table has been created.

DBI::dbListTables(con)

## [1] "COMPANY" "DEPARTMENT" "ecom" "trade" "trial\_db"

DBI::dbExistsTable(con, "trial\_db")

## [1] TRUE

**Overwrite Table**

Sometimes instead of creating a new table, you may want to overwrite the data in an existing table. In such cases, use the overwrite argument in dbWriteTable() and set it to TRUE. In the below example, we overwrite the trial\_db table in the database using the data set trial2\_data.

# sample data

x <- sample(100, 10)

y <- letters[11:20]

trial2\_data <- tibble::tibble(x, y)

# overwrite table trial

DBI::dbWriteTable(con, "trial\_db", trial2\_data, overwrite = TRUE)

Let us query sone data from trial\_db table to ensure that it has been overwritten.

DBI::dbGetQuery(con, "select \* from trial\_db")

## x y

## 1 24 k

## 2 57 l

## 3 29 m

## 4 46 n

## 5 58 o

## 6 13 p

## 7 93 q

## 8 90 r

## 9 25 s

## 10 92 t

**Append Data**

Alright! Now let us say instead of overwriting the data in the table, you want to append the data. In such cases, you can append data to an existing table by setting the append argument in dbWriteTable() to TRUE. In the below example, we append the data set trial3\_data to the trial\_db table in the database.

# sample data

x <- sample(100, 10)

y <- letters[5:14]

trial3\_data <- tibble::tibble(x, y)

# append data

DBI::dbWriteTable(con, "trial\_db", trial3\_data, append = TRUE)

Let us check if the data has been appended successfully by querying data from the trial\_db table.

DBI::dbGetQuery(con, "select \* from trial\_db")

## x y

## 1 24 k

## 2 57 l

## 3 29 m

## 4 46 n

## 5 58 o

## 6 13 p

## 7 93 q

## 8 90 r

## 9 25 s

## 10 92 t

## 11 45 e

## 12 5 f

## 13 47 g

## 14 20 h

## 15 78 i

## 16 27 j

## 17 1 k

## 18 18 l

## 19 48 m

## 20 32 n

**Insert Rows**

We can insert new rows into existing tables using:

* dbExecute()
* dbSendStatement()

Both the function take 2 arguments:

* connection name
* sql statement

In the below example, we insert a new row of data into the trial-db table in the database using `dbExecute().

DBI::dbExecute(con,

"INSERT into trial\_db (x, y) VALUES (32, 'c'), (45, 'k'), (61, 'h')"

)

## [1] 3

Let us check if the new row of data has been inserted into the trial\_db table by querying data from the same table.

DBI::dbGetQuery(con, "select \* from trial\_db")

## x y

## 1 24 k

## 2 57 l

## 3 29 m

## 4 46 n

## 5 58 o

## 6 13 p

## 7 93 q

## 8 90 r

## 9 25 s

## 10 92 t

## 11 45 e

## 12 5 f

## 13 47 g

## 14 20 h

## 15 78 i

## 16 27 j

## 17 1 k

## 18 18 l

## 19 48 m

## 20 32 n

## 21 32 c

## 22 45 k

## 23 61 h

In the next example, we insert another row of data into the trial\_db table in the database using dbSendStatement().

DBI::dbSendStatement(con,

"INSERT into trial\_db (x, y) VALUES (25, 'm'), (54, 'l'), (16, 'y')"

)

##

## SQL INSERT into trial\_db (x, y) VALUES (25, 'm'), (54, 'l'), (16, 'y')

## ROWS Fetched: 0 [complete]

## Changed: 3

Let us check if the new row of data has been inserted into the trial\_db table by querying data from the same table.

DBI::dbGetQuery(con, "select \* from trial\_db")

## Warning: Closing open result set, pending rows

## x y

## 1 24 k

## 2 57 l

## 3 29 m

## 4 46 n

## 5 58 o

## 6 13 p

## 7 93 q

## 8 90 r

## 9 25 s

## 10 92 t

## 11 45 e

## 12 5 f

## 13 47 g

## 14 20 h

## 15 78 i

## 16 27 j

## 17 1 k

## 18 18 l

## 19 48 m

## 20 32 n

## 21 32 c

## 22 45 k

## 23 61 h

## 24 25 m

## 25 54 l

## 26 16 y

**Remove Table**

To delete/remove a table from the database, use dbRemoveTable().

DBI::dbRemoveTable(con, "trial\_db")

**Your Turn**

* check if mytable exists in the database
* create new table mytable using the first 3 rows of mtcars data set
* list all tables to check if the new table has been created
* overwrite mytable with the first 10 rows of mtcars data set
* append the 20th row of mtcars data set to mytable
* create a new table using the last 5 rows of mtcars and append it to mytable
* remove mytable

**Data Type**

We know of the different data types in R such as integer, numeric/double, logical, factor etc. How do databases treat these data types? To know the data type of a particular value in a database, use dbDataType(). The first input is the database driver and the next is the value whose data type we are seeking. In the below example, we look at the data type of 3 different values in SQLite.

DBI::dbDataType(RSQLite::SQLite(), "a")

## [1] "TEXT"

DBI::dbDataType(RSQLite::SQLite(), 1:5)

## [1] "INTEGER"

DBI::dbDataType(RSQLite::SQLite(), 1.5)

## [1] "REAL"

**Generate SQL Query**

sqlCreateTable() will generate the SQL statement for simple CREATE TABLE operations. In the below example, it generates the SQL statement for creating table new with two fields x and y.

DBI::sqlCreateTable(con, "new", c(x = "integer", y = "text"))

## Warning: Do not rely on the default value of the row.names argument for

## sqlCreateTable(), it will change in the future.

## CREATE TABLE `new` (

## `x` integer,

## `y` text

## )

sqlAppendTable() will generate the SQL statement for simple INSERT operations. In the below example, it generates the SQL statement for inserting a new row of data into the trial\_db table.

trial\_new <- data.frame(x = 30, y = 'k')

DBI::sqlAppendTable(con, "trial\_db", trial\_new)

## Warning: Do not rely on the default value of the row.names argument for

## sqlAppendTable(), it will change in the future.

## INSERT INTO `trial\_db`

## (`x`, `y`)

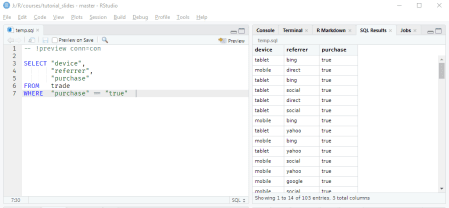
## VALUES

## (30, 'k')

[](https://www.youtube.com/user/rsquaredin/)

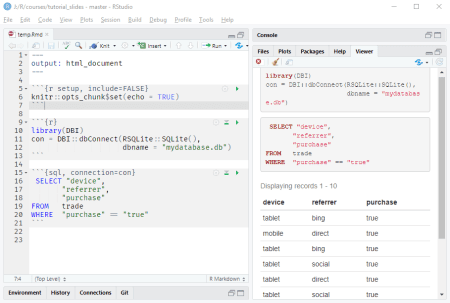
**Running SQL Scripts**

Once you are connected to a database, you may want to run some SQL queries. So far, we have run the SQL queries in R using function from the DBI package. Using RStudio SQL scripts, we can execute plain SQL queries as shown below. In the first line, we specify the database connection -- !preview conn=con followed by SQL queries. The output can be viewed by clicking on the preview button. We have included a sample SQL script (dbi.sql) which you can open and execute in RStudio.



**knitr SQL Engine**

In addition to R, the knitr package can execute code chunks in a variety of languages including SQL. In the below image, we show how to execute SQL queries. First, we establish a DBIconnection to a database in a R code chunk which is then used in a SQL chunk via the connection option (connection = con). Check out the dbi.Rmd file in the resources section.



**Your Turn**

* check the data type of "NULL"
* use SQL script to select duration, n\_visit from trade table  
  where device has the value tablet
* create a html report for the above sql query using the knitr SQL engine

**Data Transformation**

In this section, we will learn to query data from a database using dplyr. We will learn to:

* reference data
* query data using dplyr
* display query
* collect data
* simulate

**Reference Data**

The first step is to reference the table in the database using tbl(). Since we want to use the ecom table from the database, we reference it as ecom2 using tbl().

ecom2 <- dplyr::tbl(con, "ecom")

ecom2

## # Source: table [?? x 11]

## # Database: sqlite 3.22.0 [J:\R\Others\blogs\content\post\mydatabase.db]

## id referrer device bouncers n\_visit n\_pages duration country purchase

##

## 1 1 google laptop true 10 1 693 Czech ~ false

## 2 2 yahoo tablet true 9 1 459 Yemen false

## 3 3 direct laptop true 0 1 996 Brazil false

## 4 4 bing tablet false 3 18 468 China true

## 5 5 yahoo mobile true 9 1 955 Poland false

## 6 6 yahoo laptop false 5 5 135 South ~ false

## 7 7 yahoo mobile true 10 1 75 Bangla~ false

## 8 8 direct mobile true 10 1 908 Indone~ false

## 9 9 bing mobile false 3 19 209 Nether~ false

## 10 10 google mobile true 6 1 208 Czech ~ false

## # ... with more rows, and 2 more variables: order\_items ,

## # order\_value

If you look at the output, ecom2 displays a tibble but in the second line it also shows the database information as well. Let us now move on and calculate the average time on site by device type.

**Query Data**

Let us compute the average time on site for different referrer groups when the visitor browses the site using a laptop. Now, instead of using SQL statement to extract the above information, we will use dplyr. This is especially useful if the user is not well versed in SQL. While dplyr can be used to query data, it is still advisable to learn the basics of SQL.

ecom2 %>%

dplyr::select(referrer, device, duration) %>%

dplyr::filter(device == "laptop") %>%

dplyr::group\_by(referrer) %>%

dplyr::summarise(avg\_tos = mean(duration)) %>%

dplyr::arrange(avg\_tos)

## Warning: Missing values are always removed in SQL.

## Use `AVG(x, na.rm = TRUE)` to silence this warning

## # Source: lazy query [?? x 2]

## # Database: sqlite 3.22.0 [J:\R\Others\blogs\content\post\mydatabase.db]

## # Ordered by: avg\_tos

## referrer avg\_tos

##

## 1 direct 326.

## 2 yahoo 331.

## 3 social 362.

## 4 bing 434.

## 5 google 439.

**Display Query**

If you want to view the SQL translation of the dplyr code used in the previous example, use show\_query().

tos\_query <-

ecom2 %>%

dplyr::select(referrer, device, duration) %>%

dplyr::filter(device == "laptop") %>%

dplyr::group\_by(referrer) %>%

dplyr::summarise(avg\_tos = mean(duration)) %>%

dplyr::arrange(avg\_tos)

dplyr::show\_query(tos\_query)

## Warning: Missing values are always removed in SQL.

## Use `AVG(x, na.rm = TRUE)` to silence this warning

##

## SELECT `referrer`, AVG(`duration`) AS `avg\_tos`

## FROM (SELECT `referrer`, `device`, `duration`

## FROM `ecom`)

## WHERE (`device` = 'laptop')

## GROUP BY `referrer`

## ORDER BY `avg\_tos`

**Collect Data**

Now, some interesting facts. We will understand this using a different simple example. Let us read the referrer and device column from the ecom table in the database and store it in result.

result <-

ecom2 %>%

dplyr::select(referrer, device)

result

## # Source: lazy query [?? x 2]

## # Database: sqlite 3.22.0 [J:\R\Others\blogs\content\post\mydatabase.db]

## referrer device

##

## 1 google laptop

## 2 yahoo tablet

## 3 direct laptop

## 4 bing tablet

## 5 yahoo mobile

## 6 yahoo laptop

## 7 yahoo mobile

## 8 direct mobile

## 9 bing mobile

## 10 google mobile

## # ... with more rows

When we print result, it displays the first 10 rows. In addition it shows the database information at the beginning as well as ... with more rows at the bottom of the table but it does not exactly say how many more rows are there.  
Let us use nrow() to find the total number of rows in result.

nrow(result)

## [1] NA

No luck with nrow() either as it returns NA instead of the number of rows in result. Now, why does this happen? When working with databases, **dplyr** never pulls data into R unless you explicitly ask for it. In the previous example, it just displays the first 10 rows and has not read the entire table. The ecom table in the database has 1000 rows of data and ideally dplyr should have read all the rows of data. But it does not work like that and the reason is this statement at the beginning of the output: Source: lazy query [?? x 2]. It does display the number of columns, 2. In place of the number of columns there is ??because it has not read the entire data from the ecom table.

What do we do if we need the entire data? In such cases, we can use collect() as shown in the below example.

result %>%

dplyr::collect()

## # A tibble: 1,000 x 2

## referrer device

##

## 1 google laptop

## 2 yahoo tablet

## 3 direct laptop

## 4 bing tablet

## 5 yahoo mobile

## 6 yahoo laptop

## 7 yahoo mobile

## 8 direct mobile

## 9 bing mobile

## 10 google mobile

## # ... with 990 more rows

In the above output, dplyr has read the entire data from ecom. It show the number of rows and columns at the top and the number of rows not displayed (990) at the bottom. More importantly, it does not show any information about the database as the entire data from ecom has been read and is available in the R session.

result %>%

dplyr::collect() %>%

nrow()

## [1] 1000

Even nrow() returns 1000 as the entire data has been read from the database. Unless and until required or explicitly asked for, the data is not pulled from the database. When you are playing around with or iterating or experimenting with R code, do not use collect(). Only when you have finalized the code for the information being extracted from the database, use collect() to read the complete output into the R session.

**Simulate**

simulate\_\*() functions from dbplyr are useful for testing SQL generation. In the below example, we want to generate the SQL for computing average time on site by referrer type for a MySQL database connection. The SQL generated is rendered to a SQL string by sql\_render(). You can test SQL generation for a wide variety of databases using dbplyr.

ecom2 %>%

dplyr::group\_by(referrer) %>%

dplyr::summarise(avg\_tos = mean(duration)) %>%

dbplyr::sql\_render(dbplyr::simulate\_mysql())

## Warning: Missing values are always removed in SQL.

## Use `AVG(x, na.rm = TRUE)` to silence this warning

## SELECT `referrer`, AVG(`duration`) AS `avg\_tos`

## FROM `ecom`

## GROUP BY `referrer`

**Your Turn**

* use tbl() to reference trade table as trade2
* use dplyr verbs to compute average duration for device from the trade table
* store the above query in a variable tos\_device
* use show\_query() to display the underlying SQL query of tos\_device
* use collect() to retrieve data from tos\_device
* use explain() to display the underlying computation logic of tos\_device

**Data Visualization**

dbplot leverages dplyr to process the underlying data computations of a plot inside a database. It uses ggplot2 to generate the following plots:

* box plot
* bar plot
* histogram
* line chart
* raster plot

Some of the plots work with only Hive or Sparklyr connections. You can refere to the documentation for more details. Since we are dealing with a SQLite database, we will be able to generate the following plots.

**Bar Plot**

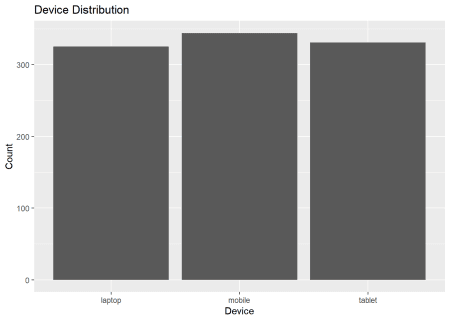
ecom2 %>%

dbplot::dbplot\_bar(device) +

ggplot2::xlab("Device") +

ggplot2::ylab("Count") +

ggplot2::ggtitle("Device Distribution")



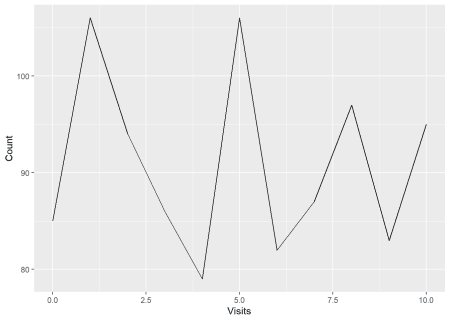
**Line Chart**

ecom2 %>%

dbplot::dbplot\_line(n\_visit) +

ggplot2::xlab("Visits") +

ggplot2::ylab("Count")



**Your Turn**

* create bar plot of referrer column from the trade table
* create line chart of n\_visit column from the trade table

**Data Modeling**

In this section, we will explore fitting models and running predictions inside the database using the following packages:

* modeldb
* tidypredict

Let us start with fitting models inside database. The modeldb package fits models inside database by using dplyr and dbplyr for SQL translation of the algorithms and currently supports linear regression and k-means clustering.

**Simple Regression**

Let us begin with a simple linear regression model. From the ecom table in the database, we want to regress duration on n\_visit. As shown below, we first select the required fields using select() and pass the resulting data to linear\_regression\_db() from modeldb. We need to specify the dependent variable which in our case is duration.

ecom2 %>%

dplyr::select(duration, n\_visit) %>%

modeldb::linear\_regression\_db(duration)

## # A tibble: 1 x 2

## `(Intercept)` n\_visit

##

## 1 364. -1.72

Let us move on to a multiple regression example. In the below example, we want to regress duration on n\_visit (number of visit) and n\_pages (number of pages browsed).

**Multiple Regression**

ecom2 %>%

dplyr::select(duration, n\_visit, n\_pages) %>%

modeldb::linear\_regression\_db(duration)

## # A tibble: 1 x 3

## `(Intercept)` n\_visit n\_pages

##

## 1 415. -2.02 -8.37

**Categorical Variables**

So how do we handle categorical variables? To handle categorical variables, use add\_dummy\_variables(). We need to specify the categorical variable and its values. It will create the dummy variables.

ecom2 %>%

dplyr::select(duration, device) %>%

modeldb::add\_dummy\_variables(device, values = c("laptop", "mobile", "tablet")) %>%

modeldb::linear\_regression\_db(duration)

## # A tibble: 1 x 3

## `(Intercept)` device\_mobile device\_tablet

##

## 1 376. -39.2 -22.1

**Full Example**

Below is a full example, where we have both continuous and categorical predictors. Whenever you have 3 or more predictors, use the sample\_size or auto\_count arguments.

# use sample size

ecom2 %>%

dplyr::select(duration, n\_visit, n\_pages, device) %>%

modeldb::add\_dummy\_variables(device, values = c("laptop", "mobile", "tablet")) %>%

modeldb::linear\_regression\_db(duration, sample\_size = 1000)

## # A tibble: 1 x 5

## `(Intercept)` n\_visit n\_pages device\_mobile device\_tablet

##

## 1 427. -1.52 -8.27 -31.1 -14.4

# use auto\_count

ecom2 %>%

dplyr::select(duration, n\_visit, n\_pages, device) %>%

modeldb::add\_dummy\_variables(device, values = c("laptop", "mobile", "tablet")) %>%

modeldb::linear\_regression\_db(duration, auto\_count = TRUE)

## # A tibble: 1 x 5

## `(Intercept)` n\_visit n\_pages device\_mobile device\_tablet

##

## 1 427. -1.52 -8.27 -31.1 -14.4

**Your Turn**

* regress duration on n\_pages
* regress duration on referrer
* and finally regress duration on n\_pages, n\_visit and referrer

**Predict Inside Database**

tidypredict can return SQL statement that can be run inside the database. Let us first create a linear model in R using lm()

model <- lm(duration ~ device + referrer + n\_visit + n\_pages, data = ecom2)

model

##

## Call:

## lm(formula = duration ~ device + referrer + n\_visit + n\_pages,

## data = ecom2)

##

## Coefficients:

## (Intercept) devicemobile devicetablet referrerdirect

## 441.450 -30.952 -14.497 -8.980

## referrergoogle referrersocial referreryahoo n\_visit

## -10.038 -19.841 -32.097 -1.433

## n\_pages

## -8.298

**Fit**

To add the fitted values in a new column, use tidypredict\_to\_column(). In the below example, we use model to compute the fitted values and add it as a new column.

ecom2 %>%

tidypredict::tidypredict\_to\_column(model) %>%

dplyr::select(duration, fit)

## # Source: lazy query [?? x 2]

## # Database: sqlite 3.22.0 [J:\R\Others\blogs\content\post\mydatabase.db]

## duration fit

##

## 1 693 409.

## 2 459 374.

## 3 996 424.

## 4 468 273.

## 5 955 357.

## 6 135 361.

## 7 75 356.

## 8 908 379.

## 9 209 249.

## 10 208 384.

## # ... with more rows

tidypredict\_fit() returns a Tidy Eval formula that can be used inside a dplyr command.

tidypredict::tidypredict\_fit(model)

## 441.450192491919 + (ifelse(device == "mobile", 1, 0) \* -30.9522074131866) +

## (ifelse(device == "tablet", 1, 0) \* -14.4972018107797) +

## (ifelse(referrer == "direct", 1, 0) \* -8.98035001912995) +

## (ifelse(referrer == "google", 1, 0) \* -10.038005625893) +

## (ifelse(referrer == "social", 1, 0) \* -19.8411767075006) +

## (ifelse(referrer == "yahoo", 1, 0) \* -32.0969778768984) +

## (n\_visit \* -1.4325653130794) + (n\_pages \* -8.29825840984566)

Let us use the above R code to calculate fitted values using mutate() from dplyr.

ecom2 %>%

dplyr::mutate(

fit = 441.450192491919 + (ifelse(device == "mobile", 1, 0) \*

-30.9522074131866) + (ifelse(device == "tablet", 1,

0) \* -14.4972018107797) + (ifelse(referrer == "direct",

1, 0) \* -8.98035001912995) + (ifelse(referrer == "google",

1, 0) \* -10.038005625893) + (ifelse(referrer == "social",

1, 0) \* -19.8411767075006) + (ifelse(referrer == "yahoo",

1, 0) \* -32.0969778768984) + (n\_visit \* -1.4325653130794) +

(n\_pages \* -8.29825840984566)

) %>%

dplyr::select(duration, fit)

## # Source: lazy query [?? x 2]

## # Database: sqlite 3.22.0 [J:\R\Others\blogs\content\post\mydatabase.db]

## duration fit

##

## 1 693 409.

## 2 459 374.

## 3 996 424.

## 4 468 273.

## 5 955 357.

## 6 135 361.

## 7 75 356.

## 8 908 379.

## 9 209 249.

## 10 208 384.

## # ... with more rows

The SQL translation of the above step can be viewed using tidypredict\_sql().

tidypredict::tidypredict\_sql(model, con)

## 441.450192491919 + (CASE WHEN (`device` = 'mobile') THEN (1.0) WHEN NOT(`device` = 'mobile') THEN (0.0) END \* -30.9522074131866) + (CASE WHEN (`device` = 'tablet') THEN (1.0) WHEN NOT(`device` = 'tablet') THEN (0.0) END \* -14.4972018107797) + (CASE WHEN (`referrer` = 'direct') THEN (1.0) WHEN NOT(`referrer` = 'direct') THEN (0.0) END \* -8.98035001912995) + (CASE WHEN (`referrer` = 'google') THEN (1.0) WHEN NOT(`referrer` = 'google') THEN (0.0) END \* -10.038005625893) + (CASE WHEN (`referrer` = 'social') THEN (1.0) WHEN NOT(`referrer` = 'social') THEN (0.0) END \* -19.8411767075006) + (CASE WHEN (`referrer` = 'yahoo') THEN (1.0) WHEN NOT(`referrer` = 'yahoo') THEN (0.0) END \* -32.0969778768984) + (`n\_visit` \* -1.4325653130794) + (`n\_pages` \* -8.29825840984566)

**Close Connection**

It is a good practice to close connection to a database when you no longer need to read/write data from/to it. Use dbDisconnect() to close the database connection.

DBI::dbDisconnect(con)

[](https://pkgs.rsquaredacademy.com/)

**RStudio Connections Pane**

In this section, we will learn to connect and explore databases using RStudio connections pane. We will connect to a MySQL database hosted on AWS. For security reasons, the database will be deleted after this post has been published and you will not be able to reproduce the results from this section onwards. Now, in the below images we show how to add and explore a new connection. **The Connections Pane is available only in RStudio 1.1 and later.**

**Step 1: Click on New Connection**

In the Connections Pane, click on the New Connection button.



**Step 2: Connect to a Data Source**

Once you click on New Connection, RStudio will display the exisiting data sources. If you do not see the driver for the database you want to connect to, install the driver and check again. Visit <https://db.rstudio.com/best-practices/drivers/> for more information about setting up ODBC drivers.



**Step 3: Supply Database Connection Parameters**

If the database driver is already present, click on it to create a new  
connection. Specify the database parameters in the text box as shown in the below image. Visit <https://www.connectionstrings.com/> to learn how to specify the connection strings for different databases.



Once you specify the database parameters, the R code will be automatically updated by RStudio as shown below.



**Step 4: Test Connection**

After specifying the database connection parameters, we can test if the connection works by clicking on Test.



If RStudio is able to connect to the database, it will show a success message as shown below.



**Step 5: Connect Options**

After testing the connection, you can choose to connect from

* the console
* R script
* or a notebook.

You can copy the R code to the clipboard as well. Depending on where you intend to use the connection i.e. interactive session, R script or notebook, choose the appropriate option.



**Step 6: Open New Connection**

Click on OK button to open a new connection to the database.



**Step 7: Explore Database**

You can explore the database from the Connections tab. View the tables in the database, explore the fields in a table, open a SQL script to run queries or close the connection if you don’t need it any longer.



**Handling Credentials**

Handling database credentials is one of the most important part of working with  
databases in R. In this section, we will look at the different options for  
securely storing and accessing credentials. After connecting to the database, we  
will list the tables in the database (just to check that the connection is  
working) and then disconnect.

**rstudioapi**

You can prompt the user to enter the database credentials using RStudio IDE. askForPassword() will show a popup box that masks what is typed.

db\_con <- DBI::dbConnect(drv = RMySQL::MySQL(),

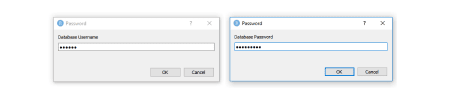
username = rstudioapi::askForPassword("Database Username"),

password = rstudioapi::askForPassword("Database Password"),

host = "[mysql-ecom.cowqoftkc0gy.us-east-2.rds.amazonaws.com](http://mysql-ecom.cowqoftkc0gy.us-east-2.rds.amazonaws.com)",

port = 3306,

dbname = "mysql\_test")



**.Renviron**

The second method is store the credentials as environment variables. This can  
be achieved using Sys.setenv() or using .Renviron file. The credentials can then be retrieved using Sys.getenv() as shown in the below example:

db\_con <- DBI::dbConnect(drv = RMySQL::MySQL(),

username = Sys.getenv("db\_uid"),

password = Sys.getenv("db\_pwd"),

host = "[mysql-ecom.cowqoftkc0gy.us-east-2.rds.amazonaws.com](http://mysql-ecom.cowqoftkc0gy.us-east-2.rds.amazonaws.com)",

port = 3306,

dbname = "mysql\_test")

# list tables in the database

DBI::dbListTables(db\_con)

## [1] "mtcars"

DBI::dbDisconnect(db\_con)

## [1] TRUE

In RStudio, create a new file and save it as .Renviron. In this file, define the  
credentials as shown below:

userid = "username"

pwd = "password"

Save the file in the home directory of your project and restart R. Why should you restart R? .Renviron is processed only at the beginning of an R session. If you try to access the credentials using Sys.getenv() without restarting R, the credentials will not be retrieved and you will see an error if you try to connect to the database. After restarting R, use Sys.getenv() to retrieve the  
credentials while opening a new connection to the database. We have added the .Renviron file used to store credentials in the resources section of the learning management system as well as in the GitHub repo.

**options**

The database credentials can be recorded as a global option in R. There are two ways to do this:

* use options()
* use an R file

Below is the code that records credentials using options():

options(db\_userid = "user\_id")

options(db\_password = "pass\_word")

The above code can be stored in a R file which can then be sourced before opening a new connection to the database. The credentials can be retrieved using getOptions(). We have added the options.R file used to store credentials to the database in the resources section of the learning management system as well as in the GitHub repo.

source("options.R")

db\_con <- DBI::dbConnect(drv = RMySQL::MySQL(),

username = getOption("db\_userid"),

password = getOption("db\_password"),

host = "[mysql-ecom.cowqoftkc0gy.us-east-2.rds.amazonaws.com](http://mysql-ecom.cowqoftkc0gy.us-east-2.rds.amazonaws.com)",

port = 3306,

dbname = "mysql\_test")

# list tables in the database

DBI::dbListTables(db\_con)

## [1] "mtcars"

DBI::dbDisconnect(db\_con)

## [1] TRUE

**config**

The config package allows you to manage environment specific configuration values. Configurations are defined using a YAML text file and are read by default from a file named config.yml in the current working directory. Store the database connection details such as driver, username, password, host, port, database name etc. in a YAML file and read it using get(). We have added the config.yml file used to store the credentials in the resources section of the learning management system as well as in the GitHub repo.

# read configurations

md <- config::get("mysql-dev")

## Warning in readLines(con): incomplete final line found on 'J:

## \R\Others\blogs\content\post\config.yml'

# test

md$port

## [1] 3306

md$dbname

## [1] "mysql\_test"

# connect

db\_con <- DBI::dbConnect(drv = RMySQL::MySQL(),

username = md$username,

password = md$password,

host = md$host,

port = md$port,

dbname = md$dbname)

# list tables in the database

DBI::dbListTables(db\_con)

## [1] "mtcars"

DBI::dbDisconnect(db\_con)

## [1] TRUE

**keyring**

The keyring package provides platform independent API to access the operating systems credential store. We leave it to the reader to explore the keyring package for storing and accessing credentials safely.

**dbx**

dbx is another interesting package built on top  
of DBI for both research and production environments and we hope to explore it  
in a separate post in the coming days.