SQL A Quick Introduction

Introduction

DATABASE

- There are mainly two type of database
    1. Relational:
       Tables in relational database resemble Excel spreadsheets (Rows and Columns). Tables in relational database can have relation to each others and it is done by the concept `keys`.
    2. Non-Relational: Data can be organized in any format but not a table (non-tabular form, and tends to be more flexible than SQL-based) like JSON files (It is a ollection of key-value pairs where the key must be a string type, and the value can be of any of the following types: Number, String.)
       Example of some simple JSON data

{ “course”: “DATA613”, “level”: “Graduate”, “credit”: 3 }

To utilize data in a relational database we use `SQL` and f0r non-relational we use another language called `NOSQL` (Not Only SQL). `NOSQL` databases use `JSON` (JavaScript Object Notation), XML, YAML, or binary schema, facilitating unstructured data. Our focus here is on SQL!

SQL

SQL is a database query language - a language designed specifically for interacting with a database. It offers syntax for extracting data, updating data, replacing data, creating data, etc. For our purposes, it will typically be used when accessing data off a server database. If the database isn’t too large, you can grab the entire data set and stick it in a data.frame. However, often the data are quite large so you interact with it piecemeal via SQL.

- SQL is how you download this subset. You choose which data frames ("tables" in SQL) to download, what subsets of these tables to download (by filtering rows), and whether to join tables together in this process.
- SQL allows you to do a lot that {dplr} does. It is industry standard for interacting with a relational database.
- One of the nice advantage of SQL is, you do not need to export data from the database into a file and then load it into R. And have to redo it again each time the data is updated. With SQL, we can get our code directly and get the most recent data straight out of the database.
- To work with relational database system we need special software known as a DataBase Management System (DBMS). It is a workspace for us to write SQL statement. There are different DBMS that you can use. For instance
  - MYSQL
  - Microsoft SQL Server
  - Oracle
  - Postgres SQL
  - and may more The method of connecting with each database may differ, but they support SQL (specifically they support ANSI SQL). If you get familiar with one of these DBMS then transition from one server to another is not very hard.
- We will use R Studio in this lesson. But if you get really into SQL, a popular IDE is DBeaver , with a tutorial from DuckDB here .
  - The first step in using SQL is accessing the actual data. There are a few packages that might be useful in connecting
    * DBI
    * RODBC
    * dbConnect
    * RSQLite
    * RMySQL
    * RPostgreSQL

SQL In R

- **Note:** In this class we learn
- Interface SQL with R through the `{DBI}` package.
- Write SQL code using the tidyverse and the `{dbplyr}` package.
- **Note:**

  - It's best to have SQL written in a separate file (that ends in ".sql").
  - If you want to load the results of a SQL query in R, saved in "query.sql", do

```
mydf <- DBI::dbGetQuery(conn, statement = read_file(here("query.sql")))
```

Install and Load necessary packages

The first package we load is {DBI}, it is the basic database infrastructure. It let's R to talk to lots of different kind of dadtabases and perform required tasks.

```
library(DBI)
```

{duckdb}: It tells {DBI} package translate between R and database. There are other related packages for translating between R and other database management system like {RSQLite}

```
library(duckdb)
```

{dbplyr}: If we want to interact with database directly using {dplyr} we need to load {dbplyr} (database plyr package). That will allow {dplyr} and databases to interact by using the package {DBI} and {RSQLite.}

- If we load the {dplyr} R will know to go ahead and load the {dbplyr}
  - The first argument is the deriver from the {duckdb} that helps translate between R and the database. The driver is loaded with the `duckdb()` function. The second rgument is the path to our database file. Let's download a duck database from: https://data-science-master.github.io/lectures/data/flights.duckdb

Connecting using {duckdb}

```
con <- dbConnect(duckdb(), "../data/flights.duckdb", read_only = TRUE)
```

- If you look at the "environment pane" you will see the "Formal class duckdb_connect…" This means we have connected to this database.

- To check if the connection is closed use the `dbIsValid()` function in R. This function returns TRUE if the connection is valid (open) and FALSE if it is closed.

```
dbIsValid(con)
```

```
[1] TRUE
```

- Once we connected to database we can look at it to see wat is going on by using `dbListTables()`, where the argument is the connection object we made. It will shows the tables we have in this database. The warning message says when finishing with your work you must disconnect.

```
dbListTables(con)
```

```
[1] "airlines" "airports" "flights"  "planes"   "weather"
```

- We also can find out the details of the individual tables by `dbListFields()`, where the first argument is the connection we made and the seconf wrapped in qutation is the name of the table that we are interested in

```
dbListFields(con, "flights")
```

```
 [1] "year"           "month"          "day"            "dep_time"
 [5] "sched_dep_time" "dep_delay"      "arr_time"       "sched_arr_time"
 [9] "arr_delay"      "carrier"        "flight"         "tailnum"
[13] "origin"         "dest"           "air_time"       "distance"
[17] "hour"           "minute"         "time_hour"
```

```
dbListFields(con, "planes")
```

```
[1] "tailnum"      "year"         "type"         "manufacturer" "model"
[6] "engines"      "seats"        "speed"        "engine"
```

```
dbListFields(con, "airports")
```

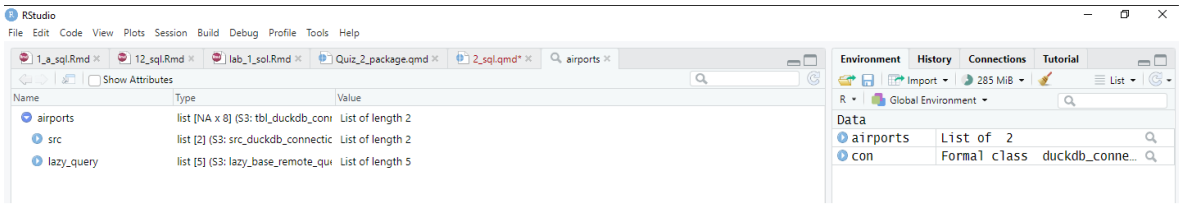
```
[1] "faa"   "name"  "lat"   "lon"   "alt"   "tz"    "dst"   "tzone"
```

_ In addition to looking at a table in a dataframe we can also make direct connections to those individual tables and we can do that using `tbl()` from {dplyr}.

- Let's assume we want to make a connect link to "airports" table. We can create a variable call it "airports" and assign the output of the `tbl()`.

```
airports <- tbl(con, "airports")
```

- However, if you click the "airports" objects in the "environment" pane you will see some confusing output.



If we want to bring this table to R we need to use `collect()` function

```
airports |>
  collect() ->
  df_airports
```

Now if you click on `df_airports` on "environment pane" you will see the exactly same thing you expect to get by `read.csv()`.

Running SQL From R

We can interact with the data in a database that we 've made a connection to.

We want to learn how to write queries in SQL. If we we want to interact with database from R based on those queries we start by writing down the query as a **string**.

- A basic SQL code chunk looks like this (put SQL code between the chunks):

```
```{sql, connection=con}

```
```

- By convention, SQL syntax is written in all UPPER CASE and variable names/database names are written in lower case.

Select Specific Columns

```
"SELECT tzone, name
 FROM airports"
```

```
[1] "SELECT tzone, name\n FROM airports"
```

There are three ways to run this query

1. Use the function `dbGetQuery()` from {DBI} package.

- Now let us run `dbGetQuery()` function

```
"SELECT tzone, name
 FROM airports" %>%
  dbGetQuery(con, .) ->
  tz_name
```

It will get run in database, select the two columns that we have requested from the table we have provided. So this is actually a dataframe that was passed back to R and if we store it , then it will be ready when we need it.

```
nrow(tz_name)
```

```
[1] 1458
```

```
tz_name %>%
  head()
```

```
             tzone                              name
1 America/New_York                Lansdowne Airport
2  America/Chicago  Moton Field Municipal Airport
3  America/Chicago              Schaumburg Regional
4 America/New_York                  Randall Airport
5 America/New_York            Jekyll Island Airport
6 America/New_York Elizabethton Municipal Airport
```

2. We can use `tbl()` from {dplyr} tbl(con, sql(count_airports))

```
  "SELECT tzone, name
   FROM airports" %>%
    sql() %>%
    tbl(con, .)
```

```
# Source:   SQL [?? x 2]
# Database: DuckDB v0.9.2 [semiyari@Windows 10 x64:R 4.3.2/../data/flights.duckdb]
   tzone               name
   <chr>               <chr>
 1 America/New_York    Lansdowne Airport
 2 America/Chicago     Moton Field Municipal Airport
 3 America/Chicago     Schaumburg Regional
 4 America/New_York    Randall Airport
 5 America/New_York    Jekyll Island Airport
 6 America/New_York    Elizabethton Municipal Airport
 7 America/New_York    Williams County Airport
 8 America/New_York    Finger Lakes Regional Airport
 9 America/New_York    Shoestring Aviation Airfield
10 America/Los_Angeles Jefferson County Intl
# i more rows
```

This will create a table similar to one we had with `dbGetQuery()`. The difference is the `dbGetQuery()` returns a dataframe back to R. `tbl()` function actually leaves the result table still in the database. And we can see it R does not know how many rows there. on the top of the table you will see something similar to

Source:SQL [?? x 2]

If we want to get the result of that query back to R we need to use `collect()` function

```
  "SELECT tzone, name
   FROM airports" %>%
    sql() %>%
    tbl(con, .) %>%
    collect()
```

```
# A tibble: 1,458 x 2
   tzone               name
   <chr>               <chr>
 1 America/New_York    Lansdowne Airport
 2 America/Chicago     Moton Field Municipal Airport
 3 America/Chicago     Schaumburg Regional
```

```
 4 America/New_York    Randall Airport
 5 America/New_York    Jekyll Island Airport
 6 America/New_York    Elizabethton Municipal Airport
 7 America/New_York    Williams County Airport
 8 America/New_York    Finger Lakes Regional Airport
 9 America/New_York    Shoestring Aviation Airfield
10 America/Los_Angeles Jefferson County Intl
# i 1,448 more rows
```

Now it will return the data as a datafram into R.

3. We can use he curly braces {} to denote code chunks. Within these code chunks, you can specify the language of the code using the syntax

- `{sql, connection=con}` indicates a SQL code chunk with additional parameters, such as specifying a connection object (con in this case).

```
SELECT tzone, name
 FROM airports
```

Table 1: Displaying records 1 - 10

tzone	name
America/New_York	Lansdowne Airport
America/Chicago	Moton Field Municipal Airport
America/Chicago	Schaumburg Regional
America/New_York	Randall Airport
America/New_York	Jekyll Island Airport
America/New_York	Elizabethton Municipal Airport
America/New_York	Williams County Airport
America/New_York	Finger Lakes Regional Airport
America/New_York	Shoestring Aviation Airfield
America/Los_Angeles	Jefferson County Intl

- We select every column by using * (Wild card):

```
SELECT *
FROM airports;
```

Table 2: Displaying records 1 - 10

faa	name	lat	lon	alt	tz	dst	tzone
04G	Lansdowne Airport	41.13047	-80.61958	1044	-5	A	America/New_York
06A	Moton Field Municipal Airport	32.46057	-85.68003	264	-6	A	America/Chicago
06C	Schaumburg Regional	41.98934	-88.10124	801	-6	A	America/Chicago
06N	Randall Airport	41.43191	-74.39156	523	-5	A	America/New_York
09J	Jekyll Island Airport	31.07447	-81.42778	11	-5	A	America/New_York
0A9	Elizabethton Municipal Airport	36.37122	-82.17342	1593	-5	A	America/New_York
0G6	Williams County Airport	41.46731	-84.50678	730	-5	A	America/New_York
0G7	Finger Lakes Regional Airport	42.88356	-76.78123	492	-5	A	America/New_York
0P2	Shoestring Aviation Airfield	39.79482	-76.64719	1000	-5	U	America/New_York
0S9	Jefferson County Intl	48.05381	-122.81064	108	-8	A	America/Los_Angeles

Filter rows

```
dbListFields(con, "flights")
```

```
 [1] "year"           "month"          "day"            "dep_time"
 [5] "sched_dep_time" "dep_delay"      "arr_time"       "sched_arr_time"
 [9] "arr_delay"      "carrier"        "flight"         "tailnum"
[13] "origin"         "dest"           "air_time"       "distance"
[17] "hour"           "minute"         "time_hour"
```

- You use the `WHERE` command in SQL to filter by rows.

```
SELECT "flight", "distance", "origin", "dest"
FROM flights
WHERE "distance" < 50;
```

Table 3: 1 records

flight	distance	origin	dest
1632	17	EWR	LGA

- To test for equality, you just use one equals sign.

```
SELECT "flight", "month"
FROM flights
WHERE "month" = 12;
```

Table 4: Displaying records 1 - 10

flight	month
745	12
839	12
1895	12
1487	12
2243	12
939	12
3819	12
1441	12
2167	12
605	12

- For characters you **must use single quotes**, not double.

```
SELECT "flight", "origin"
FROM flights
WHERE "origin" = 'JFK';
```

Table 5: Displaying records 1 - 10

flight	origin
1141	JFK
725	JFK
79	JFK
49	JFK
71	JFK
194	JFK
1806	JFK
1743	JFK
303	JFK
135	JFK

- You can select multiple criteria using the `AND` command

SQL

```
SELECT "flight", "origin", "dest"
FROM flights
WHERE "origin" = 'JFK' AND "dest" = 'CMH';
```

Table 6: Displaying records 1 - 10

flight	origin	dest
4146	JFK	CMH
3783	JFK	CMH
4146	JFK	CMH
3783	JFK	CMH
4146	JFK	CMH
3783	JFK	CMH
4146	JFK	CMH
3783	JFK	CMH
4146	JFK	CMH
3650	JFK	CMH

- You can use the `OR` logical operator too. Just put parentheses around your desired order of operations.

  SQL

```
SELECT "flight", "origin", "dest"
FROM flights
WHERE ("origin" = 'JFK' OR "origin" = 'LGA') AND dest = 'CMH';
```

Table 7: Displaying records 1 - 10

flight	origin	dest
4146	JFK	CMH
3783	JFK	CMH
4146	JFK	CMH
3783	JFK	CMH
4490	LGA	CMH
4485	LGA	CMH
4426	LGA	CMH
4429	LGA	CMH
4626	LGA	CMH

flight	origin	dest
4555	LGA	CMH

- Missing data is `NULL` in SQL (instead of `NA`). We can remove them by the special command:

  SQL

```
SELECT "flight", "dep_delay"
FROM flights
WHERE "dep_delay" IS NOT NULL;
```

Table 8: Displaying records 1 - 10

flight	dep_delay
1545	2
1714	4
1141	2
725	-1
461	-6
1696	-4
507	-5
5708	-3
79	-3
301	-2

- Just use `IS` if you want only the missing data observations

  SQL

```
SELECT "flight", "dep_delay"
FROM flights
WHERE "dep_delay" IS NULL;
```

Table 9: Displaying records 1 - 10

flight	dep_delay
4308	NA
791	NA
1925	NA

flight	dep_delay
125	NA
4352	NA
4406	NA
4434	NA
4935	NA
3849	NA
133	NA

- When you are building a query, you often want to subset the rows while you are finishing it (you don't want to return the whole table each time you are trouble shooting a query). Use `LIMIT` to show only the top subset.

  SQL

```
SELECT "flight", "origin", "dest"
FROM flights
LIMIT 5;
```

Table 10: 5 records

flight	origin	dest
1545	EWR	IAH
1714	LGA	IAH
1141	JFK	MIA
725	JFK	BQN
461	LGA	ATL

- You can also randomly sample rows via `USING SAMPLE`:

  SQL

```
SELECT "flight", "origin", "dest"
FROM flights
USING SAMPLE 5 ROWS;
```

Table 11: 5 records

flight	origin	dest
87	JFK	SLC
1129	JFK	RSW

flight	origin	dest
1903	LGA	SRQ
4667	EWR	MSP
563	EWR	IAH

Arranging Rows

- Use `ORDER BY` to rearrange the rows (let's remove missing values so we can see the ordering)

  SQL

```
SELECT "flight", "dep_delay"
FROM flights
WHERE "dep_delay" IS NOT NULL
ORDER BY "dep_delay";
```

Table 12: Displaying records 1 - 10

flight	dep_delay
97	-43
1715	-33
5713	-32
1435	-30
837	-27
3478	-26
4573	-25
4361	-25
2223	-24
3318	-24

- Use `DESC` after the variable to arrange in descending order

  SQL

```
SELECT "flight", "dep_delay"
FROM flights
WHERE "dep_delay" IS NOT NULL
ORDER BY "dep_delay" DESC;
```

Table 13: Displaying records 1 - 10

flight	dep_delay
51	1301
3535	1137
3695	1126
177	1014
3075	1005
2391	960
2119	911
2007	899
2047	898
172	896

- You break ties by adding more variables in the `ORDER BY` statement

  SQL

```
SELECT "flight", "origin", "dep_delay"
FROM flights
WHERE "dep_delay" IS NOT NULL
ORDER BY "origin" DESC, "dep_delay";
```

Table 14: Displaying records 1 - 10

flight	origin	dep_delay
1715	LGA	-33
5713	LGA	-32
1435	LGA	-30
837	LGA	-27
3478	LGA	-26
4573	LGA	-25
375	LGA	-24
4065	LGA	-24
2223	LGA	-24
5956	LGA	-23

Mutate

- In SQL, you mutate variables while you SELECT. You use `AS` to specify what the new variable is called (choosing a variable name is called "aliasing" in SQL).

SQL

```
SELECT <expression> AS <myvariable>
FROM <mytable>;
```

- Let's calculate average speed from the `flights` table. We'll also keep the flight number, distance, and air time variables.

  SQL

```
SELECT "flight", "distance" / "air_time" AS "speed", "distance", "air_time"
FROM flights;
```

Table 15: Displaying records 1 - 10

flight	speed	distance	air_time
1545	6.167401	1400	227
1714	6.237885	1416	227
1141	6.806250	1089	160
725	8.612022	1576	183
461	6.568966	762	116
1696	4.793333	719	150
507	6.740506	1065	158
5708	4.320755	229	53
79	6.742857	944	140
301	5.311594	733	138

Joining

- For joining, in the `SELECT` call, you write out all of the columns in **both** tables that you are joining.
- If there are shared column names, you need to distinguish between the two via `table1."var"` or `table2."var"` etc…
- Use `LEFT JOIN` to declare a left join, and `ON` to declare the keys.

  SQL

```
-- flight is from the flights table
-- type is from the planes table
-- both tables have a tailnum column, so we need to tell them apart
```

```
-- if you list both tailnums in SELECT, you'll get two tailnum columns
SELECT "flight", flights."tailnum", "type"
FROM flights
JOIN planes
ON flights."tailnum" = planes."tailnum";
```

Table 16: Displaying records 1 - 10

flight	tailnum	type
569	N846UA	Fixed wing multi engine
4424	N19966	Fixed wing multi engine
684	N809UA	Fixed wing multi engine
1279	N328NB	Fixed wing multi engine
1691	N34137	Fixed wing multi engine
1447	N117UW	Fixed wing multi engine
583	N632JB	Fixed wing multi engine
3574	N790SW	Fixed wing multi engine
3351	N711MQ	Fixed wing multi engine
303	N502UA	Fixed wing multi engine

Example: Count the number of time zones from `airports` table.

We want to count the number of individual flights associated with time zones. 1. Select the `tzone` column
2. count each row (`*`) associated with `tzone`
3. We get this from the `airports` table 4. We need to group them by `tzone`

Everything must be inside of the quotation marks.

```
count_airports <- "SELECT tzone,
                  COUNT(*)
                  From airports
                  GROUP BY tzone"
```

- as you may remember there are three ways to run this query

1. Use the function `dbGetQuery()` from {DBI} package.

- Now let us run `dbGetQuery()` function

count_tzone

```
dbGetQuery(con, count_airports) ->
  count_tzone
```

It will get run in database, the count for each time zone are totaled and it will get returned to R as a dataframe. So this is actually a dataframe that was passed back to R and if we store it , then it will be ready when we need it.

2. We can use `tbl()` from {dplyr}

```
tbl(con, sql(count_airports))
```

```
# Source:   SQL [10 x 2]
# Database: DuckDB v0.9.2 [semiyari@Windows 10 x64:R 4.3.2/../data/flights.duckdb]
   tzone               `count_star()`
   <chr>                        <dbl>
 1 America/Chicago                342
 2 America/Los_Angeles            176
 3 America/Phoenix                 38
 4 America/Vancouver                2
 5 America/Anchorage              239
 6 America/Denver                 119
 7 Asia/Chongqing                   2
 8 <NA>                             3
 9 America/New_York               519
10 Pacific/Honolulu                18
```

This will create a table similar to one we had in method 1. The difference is the `dbGetQuery()` returns a dataframe back to R. `tbl()` function actually leaves the result table still in the database. And we can see it R doesnot know how many rows there. on the top of the table you will see somthing similar to

Source:SQL [?? x 2]

If we want to get the resul of that query back to R we need to use `collect()` function

```
tbl(con, sql(count_airports)) %>%
  collect()
```

```
# A tibble: 10 x 2
   tzone               `count_star()`
   <chr>                        <dbl>
 1 America/Phoenix                 38
 2 America/Chicago                342
 3 America/New_York               519
 4 America/Los_Angeles            176
 5 America/Vancouver                2
 6 America/Anchorage              239
 7 America/Denver                 119
 8 Asia/Chongqing                   2
 9 <NA>                             3
10 Pacific/Honolulu                18
```

Now it will return the data as a datafram into R.

SQL Using {dplyr}

If you are familiar with dplyr and not very comfortable with SQL, you may want using database connection with dplyr commands inside of the database.

Let us we want to count the time zone in flights data frame.

- We start with creating a connection to an individal table (or multiple tables) using `tbl()`
- We create a linked version of table in R. The first argument is the connection we made and the second is the name of the table we want to connect to.

```
airports_table <- tbl(con, "airports")
```

- Now we have the connection we can do exactly what we did in {dplyr}
- Remeber in {dplyr} we gorup by things first

```
airports_table %>%
  group_by(tzone) %>%
  summarise(count = n())
```

```
# Source:   SQL [10 x 2]
# Database: DuckDB v0.9.2 [semiyari@Windows 10 x64:R 4.3.2/../data/flights.duckdb]
   tzone               count
```

```
   <chr>               <dbl>
 1 America/Phoenix        38
 2 America/Los_Angeles   176
 3 America/New_York      519
 4 America/Vancouver       2
 5 America/Anchorage     239
 6 America/Denver        119
 7 Asia/Chongqing          2
 8 <NA>                    3
 9 Pacific/Honolulu       18
10 America/Chicago       342
```

- We see the same output that we had before. If you look closely at the table, you will see the number of rows are not known.
- All of the work was done in the database. That happens because the {dbplyr} package helps translate our {dplyr} pipeline into SQL and then run everything in the database.

_ The benefit of working in the database caused the faster performance (without using R memory) Also without knowing SQL, we just use {dplyr} functions.

- As you can see the number of rows is unknown. The reason is, R just has the first few observations and it does not know how many is there. To bring back data to R we need to do one additional step , which was applying the `collect()` function

```
airports_table %>%
  group_by(tzone) %>%
  summarise(count = n()) %>%
  collect()
```

```
# A tibble: 10 x 2
   tzone               count
   <chr>               <dbl>
 1 America/Vancouver       2
 2 America/Anchorage     239
 3 America/Denver        119
 4 America/Los_Angeles   176
 5 America/Chicago       342
 6 Pacific/Honolulu       18
 7 America/Phoenix        38
 8 Asia/Chongqing          2
 9 <NA>                    3
10 America/New_York      519
```

Copying data from R to the database

So far we have been reade,create data, retrieve data from database. Now we want the table that we created and add it to database. We can do this by using `copy_to()` function.

- Suppose we want to store the table `count_tzone` from R back to database as a permanent table.
- Let us look at the tables in our database just to make sure that table isn't there.

```
dbListTables(con)
```

```
[1] "airlines" "airports" "flights"  "planes"   "weather"
```

- Let's give a name to table that we have created by SQLin R.

To store the table we use `copy_to()` function, where the first argument is the connection we made, the second argument is the table that we want to copy to database and the third argument is the name of the table on the database. By default table will be added temporary if you want it permanent you need to type `temporary = FALSE`, to store data permanently.

- Now let us add the `count_tzone` which we were created by {dbplyr} to database

```
copy_to(con, count_tzone, name = "zone_count", overwrite = TRUE)
```

check to see if the table was added to database

```
dbListTables(con)
```

```
[1] "airlines"   "airports"   "flights"    "planes"     "weather"
[6] "zone_count"
```

tzone_c

```
airports_table %>%
  group_by(tzone) %>%
  summarise(count = n()) %>%
  collect() ->
  tzone_c
```

- Now let us add the `tzone_c` which we were created by {dbplyr} to database

```
copy_to(con, tzone_c, name = "zone_t", overwrite = TRUE)
```

check to see if the table was added to database

```
dbListTables(con)
```

```
[1] "airlines"   "airports"   "flights"    "planes"     "weather"
[6] "zone_count" "zone_t"
```

Remove a table from database

We can Delete table by `dbRemoveTable()` function from {DBI} package.

- Let us remove the file we just added

```
dbRemoveTable(con, "zone_t")
```

check to see if the table was removed from database

```
dbListTables(con)
```

```
[1] "airlines"   "airports"   "flights"    "planes"     "weather"
[6] "zone_count"
```

- Now we want to remove `count_tzone` from database

```
dbRemoveTable(con, "zone_count")
```

- Let us to check if it was deleted

```
dbListTables(con)
```

```
[1] "airlines" "airports" "flights"  "planes"   "weather"
```

Close the connection

```
dbDisconnect(con)
```

__ To check if the connection is closed use the `dbIsValid()` function in R. This function returns TRUE if the connection is valid (open) and FALSE if it is closed.

```
dbIsValid(con)
```

```
[1] FALSE
```

- Or

```
# Check if the connection is closed
if (!dbIsValid(con)) {
  print("Connection is closed.")
} else {
  print("Connection is open.")
}
```

```
[1] "Connection is closed."
```

How about if we want to Use `{RSQLite}` instead of `{duckdb}`

```
library(RSQLite)
```

- To collect the file go to SQLITE TUTORIAL
- Scroll down to Download SQLite sample database
- You get a zipfile. Double click to open it and then extract the file. I put the file `chinook.db` in my data folder

```
con_sqlite <- dbConnect(SQLite(), "../data/chinook.db", synchronous = NULL)
```

- We just created the connection

In the following we will get the list of tables in our database

```
dbListTables(con_sqlite)
```

```
 [1] "albums"          "artists"         "customers"       "employees"
 [5] "genres"          "invoice_items"   "invoices"        "media_types"
 [9] "playlist_track"  "playlists"       "sqlite_sequence" "sqlite_stat1"
[13] "tracks"
```

Here is the detail of table that called `artists`

```
dbListFields(con_sqlite, "artists")
```

```
[1] "ArtistId" "Name"
```

```
dbListFields(con_sqlite, "customers")
```

```
 [1] "CustomerId"   "FirstName"    "LastName"     "Company"      "Address"
 [6] "City"         "State"        "Country"      "PostalCode"   "Phone"
[11] "Fax"          "Email"        "SupportRepId"
```

```
dbDisconnect(con_sqlite)
```

```
dbIsValid(con_sqlite)
```

```
[1] FALSE
```

Source

Source