# Notes 5 - Databases and SQL

Rick Brown
Southern Utah University

Math 3190

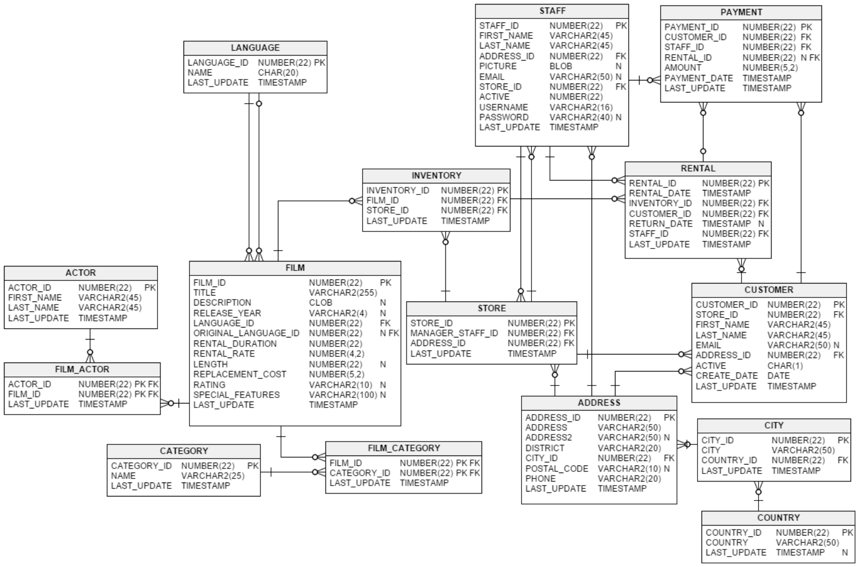
Section 1

## Databases and SQL

## Intro to Database Design

In many businesses and government agencies, data files are not directly accessible. Instead, data are stored in databases that are usually in the cloud. The most common type of database is a *relational database*. These types of databases are made up of **tables**, which are very similar to data frames in **R**. These tables usually have at least one variable in common with at least one other table that can be used to connect them.

## Relational Database Example



## SQL Intro

The most common way to access information stored in relational databases is with SQL.

SQL stands for structured query language and is ubiquitous in business, government, and academia where large data sets are used. SQL is usually either pronounced like "sequel" or the letters are individually said "S-Q-L".

We are going to go over the common SQL commands and then look at how to access SQL databases and use SQL within **R**.

SQL is one of the most basic "coding languages" there is because of how similar the statements are to spoken English.

## SELECT, FROM, and WHERE Statements

The backbone of SQL are the SELECT, FROM, and WHERE statements.

- SELECT is very much like the select() function in the tidyverse (dplyr package). It selects the columns or variables from the database. You can use "*" to select all columns.
- FROM is the statement that specifies the table in the database from which we are taking the data.
- WHERE is similar to the filter() function in the tidyverse. This will subset the table based on some criterion or criteria.

Note: these words do not *need* to be written in all capital letters, but it is the convention to type them in all caps for readability.

## SQL Syntax

The basic syntax for SQL statements is as follows:

```sql
SELECT column1, column2, ....
FROM table_name
WHERE condition;
```

## SQL Example

Suppose we have a database named `sakila` that models a DVD rental store, featuring things like films, actors, film-actor relationships. In this database are many tables, one of which is “film”. Here is some information about the film table:

| film | |
|---|---|
| film_id | smallint |
| title | varchar |
| description | text |
| release_year | year |
| language_id | tinyint |
| original_language_id | tinyint |
| rental_duration | tinyint |
| rental_rate | decimal |
| length | smallint |
| replacement_cost | decimal |
| rating | enum |
| special_features | set |
| last_update | timestamp |

Let’s run some SQL queries with this table.

## SQL Query 1

```sql
SELECT title, release_year, rating
FROM film
WHERE rating = 'PG';
```

Table 1: My Caption

| title | release_year | rating |
|---|---|---|
| ACADEMY DINOSAUR | 2006 | PG |
| AGENT TRUMAN | 2006 | PG |
| ALASKA PHANTOM | 2006 | PG |
| ALI FOREVER | 2006 | PG |
| AMADEUS HOLY | 2006 | PG |

## WHERE Statement Commands

The following can be used in a `WHERE` statement:

| Operator | Description |
|---|---|
| $=$ | Equal |
| $>$ | Greater than |
| $<$ | Less than |
| $>=$ | Greater than or equal |
| $<=$ | Less than or equal |
| $<>$ or $!=$ | Not equal |
| AND | Restricts the output based on another criterion. |
| OR | Expands the output based on another criterion. |
| NOT | Gives the opposite of the criterion. |
| BETWEEN | Between a certain range. |
| IN | To specify multiple possible values for a column. |
| LIKE | Search for a pattern. |

## SQL Query 2: BETWEEN

The syntax for the BETWEEN statement is

```
WHERE column_name BETWEEN value1 AND value2
```

```
SELECT film_id, title, length, rental_duration
FROM film
WHERE film_id BETWEEN 13 AND 14
  OR film_id BETWEEN 21 AND 23;
```

Table 2: Displaying records 1 - 5

| film_id | title | length | rental_duration |
|---:|---|---:|---:|
| 13 | ALI FOREVER | 150 | 4 |
| 14 | ALICE FANTASIA | 94 | 6 |
| 21 | AMERICAN CIRCUS | 129 | 3 |
| 22 | AMISTAD MIDSUMMER | 85 | 6 |
| 23 | ANACONDA CONFESSIONS | 92 | 3 |

## SQL Query 3: IN

The IN operator is a shorthand for multiple OR conditions.

```
SELECT film_id, title, length, rental_duration
FROM film
WHERE film_id IN ("1", "50", "100", "613");
```

Table 3: Displaying records 1 - 4

| film_id | title | length | rental_duration |
|---:|---|---:|---:|
| 1 | ACADEMY DINOSAUR | 86 | 6 |
| 50 | BAKED CLEOPATRA | 182 | 3 |
| 100 | BROOKLYN DESERT | 161 | 7 |
| 613 | MYSTIC TRUMAN | 92 | 5 |

Note: the quotes aren't needed for numbers, but can still be used.

## SQL Query 4: LIKE

There are two wildcards often used in conjunction with the LIKE operator:

- The percent sign % represents zero, one, or multiple characters.
- The underscore sign _ represents one single character. This can be repeated to stand for multiple characters.

```sql
SELECT film_id, title, length, rental_duration
FROM film
WHERE title LIKE "b%" AND length >= 175;
```

Table 4: Displaying records 1 - 4

| film_id | title | length | rental_duration |
|---:|---|---:|---:|
| 50 | BAKED CLEOPATRA | 182 | 3 |
| 61 | BEAUTY GREASE | 175 | 5 |
| 88 | BORN SPINAL | 179 | 7 |
| 94 | BRAVEHEART HUMAN | 176 | 7 |

## SQL Query 5: LIKE

```sql
SELECT film_id, title, length, rental_duration
FROM film
WHERE title LIKE "__ab%";
```

Table 5: 3 records

| film_id | title | length | rental_duration |
|---:|---|---:|---:|
| 9 | ALABAMA DEVIL | 114 | 3 |
| 34 | ARABIA DOGMA | 62 | 6 |
| 773 | SEABISCUIT PUNK | 112 | 6 |

## More on SELECT Statments

In the `SELECT` statement, instead of simply putting column names, we can also use the following functions:

- `MIN()` returns the minimum value of a numeric column.
- `MAX()` returns the maximum value of a numeric column.
- `COUNT()` returns the number of rows that matches a specified criterion.
- `SUM()` returns the total sum of a numeric column.
- `AVG()` returns the average value of a numeric column.

Additionally, we can use aliases to rename columns with the `AS` statement.

## SQL Query 6: SELECT Functions

```
SELECT AVG(length) AS "Average Length"
FROM film;
```

Table 6: 1 records

| Average Length |
|---:|
| 115.272 |

## SQL Query 7: SELECT Functions

```sql
SELECT COUNT(title) AS "Number of A or B Movies"
FROM film
WHERE title LIKE "a%" OR title LIKE "b%";
```

Table 7: 1 records

| Number of A or B Movies |
|---:|
| 109 |

## SQL Query 8: SELECT Functions

The following will return all the films with a length higher than the average length.

```
SELECT title, length
FROM film
WHERE length > (SELECT AVG(length) FROM film);
```

Table 8: Displaying records 1 - 5

| title | length |
|---|---|
| AFFAIR PREJUDICE | 117 |
| AFRICAN EGG | 130 |
| AGENT TRUMAN | 169 |
| ALAMO VIDEOTAPE | 126 |
| ALASKA PHANTOM | 136 |

## ORDER BY Statement

We can also use the ORDER BY statement to change the order in which the data table is given.

The syntax for ORDER BY is

```
ORDER BY column1, column2, .. columnN
```

We can put a a ASC for ascending or DESC for descending values after any one of the columns we are ordering by.

## SQL Query 9: ORDER BY Statement

```
SELECT title, length
FROM film
WHERE length > (SELECT AVG(length) FROM film)
ORDER BY length ASC, title DESC;
```

Table 9: Displaying records 1 - 5

| title | length |
|---|---|
| WORDS HUNTER | 116 |
| MADIGAN DORADO | 116 |
| INSTINCT AIRPORT | 116 |
| DUCK RACER | 116 |
| RESURRECTION SILVERADO | 117 |

## GROUP BY Statement

The `GROUP BY` statement groups rows that have the same values into summary rows. It is often used with aggregate functions (`COUNT()`, `MAX()`, `MIN()`, `SUM()`, `AVG()`) to group the result-set by one or more columns.

Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
ORDER BY column_name(s);
```

## SQL Query 10: GROUP BY Statement

```
SELECT AVG(length), rating
FROM film
WHERE release_year = 2006
GROUP BY rating
ORDER BY rating;
```

Table 10: Displaying records 1 - 5

| AVG(length) | rating |
|---|---|
| 111.0506 | G |
| 112.0052 | PG |
| 120.4439 | PG-13 |
| 118.6615 | R |
| 113.2286 | NC-17 |

## SELECT DISTINCT Statement

The `SELECT DISTINCT` statement is used to return only distinct (different) values.

Syntax:

```
SELECT DISTINCT column1, column2, ...
FROM table_name;
```

## SQL Query 11: DISTINCT Statement

```
SELECT DISTINCT release_year
FROM film;
```

Table 11: 1 records

| release_year |
|---|
| 2006 |

So this database only has films from the year 2006.

## UNION Statement

The `UNION` operator is used to combine the result-set of two or more `SELECT` statements.

Syntax:

```
SELECT column_name(s) FROM table1
UNION
SELECT column_name(s) FROM table2;
```

## SQL Query 12: UNION Statement

```
SELECT AVG(length) AS "Average Length and Payment Amount"
FROM film
UNION
SELECT AVG(amount)
FROM payment;
```

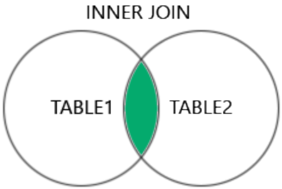
Table 12: 2 records

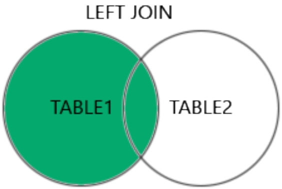
| Average Length and Payment Amount |
|---:|
| 115.272000 |
| 4.201356 |

## JOIN Statements

A `JOIN` clause is used to combine rows from two or more tables, based on a related column between them. There are four types of `JOIN` statements:

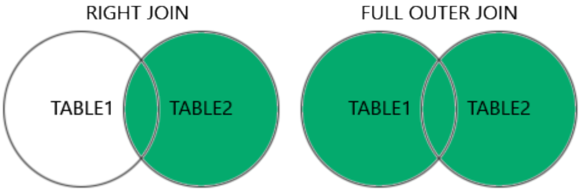
- `(INNER) JOIN`: Returns records that have matching values in both tables.
- `LEFT (OUTER) JOIN`: Returns all records from the left table, and the matched records from the right table.





## JOIN Statements

- `RIGHT (OUTER) JOIN`: Returns all records from the right table, and the matched records from the left table.
- `FULL (OUTER) JOIN`: Returns all records when there is a match in either left or right table.



## JOIN Statements

The syntax for JOIN statements is as follows:

```sql
SELECT column_name(s)
FROM table1
(JOIN STATEMENT) table2
  ON table1.column_name = table2.column_name;
```

Replace the `(JOIN STATEMENT)` with `INNER JOIN`, `LEFT JOIN`, `RIGHT JOIN`, or `FULL JOIN`.

You can select columns from either table, so it is common to denote which table we are selecting from by using `table.column_name`.

## JOIN Examples

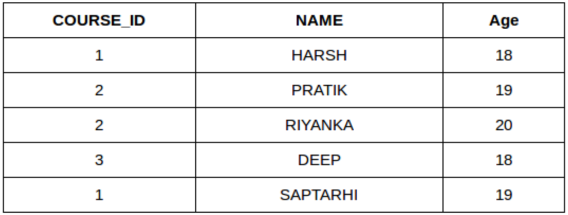
Consider the following two tables called Student and StudentCourse, respectively.

| ROLL_NO | NAME | ADDRESS | PHONE | Age |
|---|---|---|---|---|
| 1 | HARSH | DELHI | XXXXXXXXXX | 18 |
| 2 | PRATIK | BIHAR | XXXXXXXXXX | 19 |
| 3 | RIYANKA | SILIGURI | XXXXXXXXXX | 20 |
| 4 | DEEP | RAMNAGAR | XXXXXXXXXX | 18 |
| 5 | SAPTARHI | KOLKATA | XXXXXXXXXX | 19 |
| 6 | DHANRAJ | BARABAJAR | XXXXXXXXXX | 20 |
| 7 | ROHIT | BALURGHAT | XXXXXXXXXX | 18 |
| 8 | NIRAJ | ALIPUR | XXXXXXXXXX | 19 |

| COURSE_ID | ROLL_NO |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 2 | 3 |
| 3 | 4 |
| 1 | 5 |
| 4 | 9 |
| 5 | 10 |
| 4 | 11 |

## JOIN Example: INNER JOIN

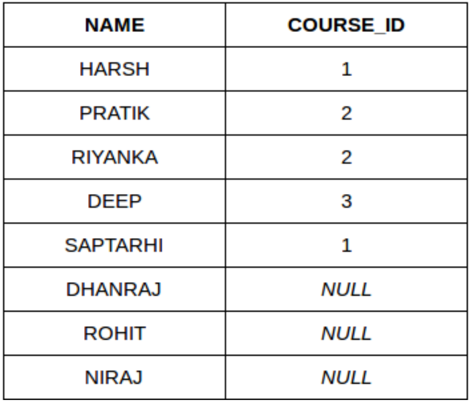
```sql
SELECT StudentCourse.COURSE_ID, Student.NAME, Student.AGE
FROM Student
INNER JOIN StudentCourse
  ON Student.ROLL_NO = StudentCourse.ROLL_NO;
```



| COURSE_ID | NAME | Age |
|---|---|---|
| 1 | HARSH | 18 |
| 2 | PRATIK | 19 |
| 2 | RIYANKA | 20 |
| 3 | DEEP | 18 |
| 1 | SAPTARHI | 19 |

## JOIN Example: LEFT JOIN

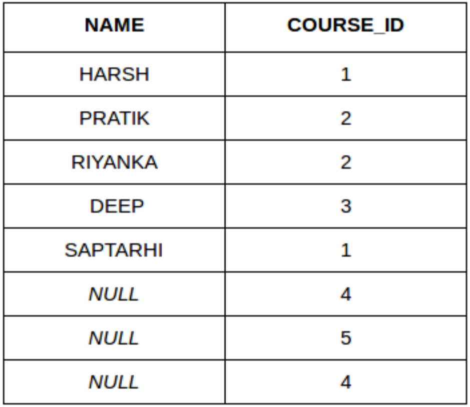
```sql
SELECT Student.NAME, StudentCourse.COURSE_ID
FROM Student
LEFT JOIN StudentCourse
ON StudentCourse.ROLL_NO = Student.ROLL_NO;
```



| NAME | COURSE_ID |
|---|---|
| HARSH | 1 |
| PRATIK | 2 |
| RIYANKA | 2 |
| DEEP | 3 |
| SAPTARHI | 1 |
| DHANRAJ | *NULL* |
| ROHIT | *NULL* |
| NIRAJ | *NULL* |

## JOIN Example: RIGHT JOIN

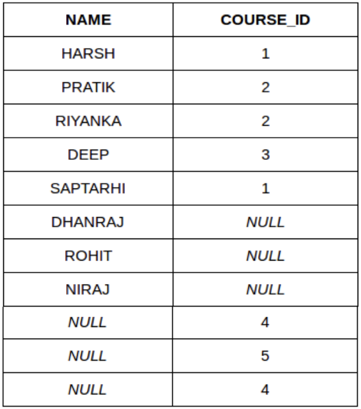
```
SELECT Student.NAME,StudentCourse.COURSE_ID
FROM Student
RIGHT JOIN StudentCourse
ON StudentCourse.ROLL_NO = Student.ROLL_NO;
```



| NAME | COURSE_ID |
|---|---|
| HARSH | 1 |
| PRATIK | 2 |
| RIYANKA | 2 |
| DEEP | 3 |
| SAPTARHI | 1 |
| *NULL* | 4 |
| *NULL* | 5 |
| *NULL* | 4 |

## JOIN Example: FULL JOIN

```sql
SELECT Student.NAME,StudentCourse.COURSE_ID
FROM Student
FULL JOIN StudentCourse
ON StudentCourse.ROLL_NO = Student.ROLL_NO;
```



| NAME | COURSE_ID |
|---|---|
| HARSH | 1 |
| PRATIK | 2 |
| RIYANKA | 2 |
| DEEP | 3 |
| SAPTARHI | 1 |
| DHANRAJ | *NULL* |
| ROHIT | *NULL* |
| NIRAJ | *NULL* |
| *NULL* | 4 |
| *NULL* | 5 |
| *NULL* | 4 |

Section 2

## SQL in R Markdown

## SQL in R Markdown

In R Markdown, you can insert an SQL code chunk.

~~~
```{sql connection=}
```
~~~

We will need to establish a connection before using it, though. This connection must be to a database, either remote or local.

## Define an SQL Connection in **R**

To connect to a remote database, we can use the following code chunk in R Markdown. We will need a few SQL and database packages, though. Some common ones are `DBI`, `RMySQL`, and `RSQL`, and `RSQLite`. This database is a MySQL database, so we will need `RMySQL`.

~~~
```{r, echo = T, eval = T}
library(DBI)
library(RMySQL)
con <- dbConnect(RMySQL::MySQL(),
                 dbname = "sakila",
                 host = "relational.fit.cvut.cz",
                 port = 3306,
                 user = "guest",
                 password = "relational")
```
~~~

## Define an SQL Connection in **R**

Note: this website for the sakilla database has since gone down since these notes were created. A remote database I found that is still up (as of January 31, 2024) can be accessed using the code on the following slide:

~~~
```{r, echo = T, eval = T}
con = dbConnect(RMySQL::MySQL(),
                dbname='Rfam',
                host='mysql-rfam-public.ebi.ac.uk',
                port=4497,
                user='rfamro',
                password='')
```
~~~

The following slides will still use the sakila database as an example.

## Define an SQL Connection in **R**

Then we can use that connection in our SQL code chunk.

~~~
```{sql connection = con, echo = T, eval = T, max.print = 5}
SELECT title, release_year, rating
FROM film
WHERE rating = 'PG';
```
~~~

Table 13: Displaying records 1 - 5

| title | release_year | rating |
|---|---|---|
| ACADEMY DINOSAUR | 2006 | PG |
| AGENT TRUMAN | 2006 | PG |
| ALASKA PHANTOM | 2006 | PG |
| ALI FOREVER | 2006 | PG |
| AMADEUS HOLY | 2006 | PG |

## Creating a Data Frame in **R** from SQL

Instead of printing the table, we can store the table we created from our SQL command as a data frame using the `output.var` argument in the SQL code chunk:

~~~
```{sql connection = con, output.var = "pg_movies"}
SELECT title, release_year, rating
FROM film
WHERE rating = 'PG';
```
~~~

## Using SQL Table in **R**

Now we can access the pg_movies variable in **R**.

```
pg_movies <- tibble(pg_movies)
head(pg_movies, 5)
```

```
## # A tibble: 5 x 3
##   title            release_year rating
##   <chr>            <chr>        <chr>
## 1 ACADEMY DINOSAUR 2006         PG
## 2 AGENT TRUMAN     2006         PG
## 3 ALASKA PHANTOM   2006         PG
## 4 ALI FOREVER      2006         PG
## 5 AMADEUS HOLY     2006         PG
```

```
dim(pg_movies)
```

```
## [1] 194   3
```

## Using Tidyverse Functions

The same thing can be accomplished if we had read in the film table as a data frame or tibble.

```
movies <- read.csv("film.csv")
pg_movies <- movies |>
  filter(rating == "PG") |>
  select(title, release_year, rating) |> tibble()
head(pg_movies, 5)
```

```
## # A tibble: 5 x 3
##   title            release_year rating
##   <chr>                   <int> <chr>
## 1 ACADEMY DINOSAUR         2006 PG
## 2 AGENT TRUMAN             2006 PG
## 3 ALASKA PHANTOM           2006 PG
## 4 ALI FOREVER              2006 PG
## 5 AMADEUS HOLY             2006 PG
```

## Using Tidyverse Functions

The following statements and functions are equivalent:

| SQL | **R** Tidyverse |
|---|---|
| SELECT | select() |
| WHERE | filter() |
| ORDER BY | arrange() |
| GROUP BY | group_by() |
| INNER JOIN | inner_join() |
| LEFT JOIN | left_join() |
| RIGHT JOIN | right_join() |
| FULL (OUTER) JOIN | full_join() |

## Using **R** Objects in SQL Chunks

If you need to use the values of an **R** variables into SQL queries, you can do so by prefacing **R** variable references with a ?. For example:

~~~
```{r}
movie_length = 130
```


```{sql, connection=con, max.print = 5}
SELECT title, length
FROM film
WHERE length >= ?movie_length
```
~~~

## Using **R** Objects in SQL Chunks

Table 14: Displaying records 1 - 5

| title | length |
|---|---:|
| AFRICAN EGG | 130 |
| AGENT TRUMAN | 169 |
| ALASKA PHANTOM | 136 |
| ALI FOREVER | 150 |
| ALLEY EVOLUTION | 180 |

## Close Session

When we are finished with connecting to the database, it is best to close the connection using the following line:

```
dbDisconnect(con) # Disconnects from server.
```

```
## [1] TRUE
```

## Other SQL Uses

There are lots of other things we can do using SQL. We can create tables, create databases, drop tables, insert rows and columns into tables, and more.

For this class, however, we will leave it there since most of the time, we will simply be accessing databases for data analysis rather than editing them.

Section 3

## Local Databases

## Simulating SQL Database with Local Files in Memory

To simulate a connection to a database, we can use the following:

```r
con <- dbConnect(RSQLite::SQLite(), ":memory:")
sakila_files <- list.files(paste0("~/My Drive/Math 3190 - Fundamentals ",
                                  "of Data Science/Data/Sakila DB Data"))
for (file in sakila_files) {
  data_file <- read.csv(paste0("~/My Drive/Math 3190 - Fundamentals ",
                               "of Data Science/Data/Sakila DB Data/",
                               file))
  # Add data frame as a table
  dbWriteTable(con, substr(file, 1, nchar(file) - 4), data_file)
  rm(data_file)
}
dbListTables(con)
```

```
##  [1] "actor"         "address"       "category"      "city"
##  [5] "country"       "customer"      "film"          "film_actor"
##  [9] "film_category" "film_text"     "inventory"     "language"
## [13] "payment"       "rental"        "staff"         "store"
```

These tables are not stored in **R**'s memory, though!

## Simulating SQL Database with Local Files

```
SELECT title, length
FROM film
WHERE length > (SELECT AVG(length) FROM film);
```

Table 15: Displaying records 1 - 5

| title | length |
|---|---|
| AFFAIR PREJUDICE | 117 |
| AFRICAN EGG | 130 |
| AGENT TRUMAN | 169 |
| ALAMO VIDEOTAPE | 126 |
| ALASKA PHANTOM | 136 |

## Connect to Local Data Databases

If we have an SQL-type file (like one that ends in `.sqlite`), then we can connect to it without reading it into **R**'s memory.

For example, I have a database called "portal_mammals.sqlite" (downloadable from here). If it is in my working directory, I can connect to this database using

```
mammals_con <- DBI::dbConnect(RSQLite::SQLite(),
                              "portal_mammals.sqlite")
DBI::dbListTables(mammals_con) # List all database tables
```

```
## [1] "plots"   "species" "surveys"
```

## Connect to Local Data Databases

Then I can run SQL queries using the `dbGetQuery()` function:

```
dbGetQuery(mammals_con, "SELECT * FROM surveys;") |> head()
```

```
##   record_id month day year plot_id species_id sex hindfoot_
## 1         1     7  16 1977       2         NL   M
## 2         2     7  16 1977       3         NL   M
## 3         3     7  16 1977       2         DM   F
## 4         4     7  16 1977       7         DM   M
## 5         5     7  16 1977       3         DM   M
## 6         6     7  16 1977       1         PF   M
```

or I can connect to the database in R Markdown using

~~~
```{sql connection=mammals_con, max.print = 5}

```
~~~

## Connect to Local Data Databases

Select from December 2000:

```sql
SELECT record_id, month, day, year, species_id, sex, weight
FROM surveys
WHERE month = 12 AND year = 2000;
```

Table 16: Displaying records 1 - 5

| record_id | month | day | year | species_id | sex | weight |
|---|---|---|---|---|---|---|
| 31616 | 12 | 22 | 2000 | DO | M | 52 |
| 31617 | 12 | 22 | 2000 | PB | F | 26 |
| 31618 | 12 | 22 | 2000 | OT | M | 24 |
| 31619 | 12 | 22 | 2000 | PB | F | 25 |
| 31620 | 12 | 22 | 2000 | DM | M | 38 |

## Create a Local Database

We can create a .sqlite file from a .csv using the `csv_to_sqlite()` function from the `inborutils` package. This is not available on the CRAN, but can be installed using `devtools::install_github("inbo/inborutils")`.

This function reads a csv in smaller chucks (50,000 lines at a time by default) and writes it to a .sqlite file so we can connect to it as a database.

Example:

```
library(inborutils)
csv_to_sqlite(csv_file = "combined.csv",
              sqlite_file = "combined.sqlite",
              table_name = "combined")

combined_con <- DBI::dbConnect(RSQLite::SQLite(),
                               "combined.sqlite")
dbGetQuery(combined_con, "SELECT * FROM combined") |> head()
```

## Create a Local Database

```
##   record_id month day year plot_id species_id  sex
## 1         1     7  16 1977       2         NL    M
## 2        72     8  19 1977       2         NL    M
## 3       224     9  13 1977       2         NL <NA>
## 4       266    10  16 1977       2         NL <NA>
## 5       349    11  12 1977       2         NL <NA>
## 6       363    11  12 1977       2         NL <NA>
##   hindfoot_length weight   genus  species   taxa plot_type
## 1              32     NA Neotoma albigula Rodent   Control
## 2              31     NA Neotoma albigula Rodent   Control
## 3              NA     NA Neotoma albigula Rodent   Control
## 4              NA     NA Neotoma albigula Rodent   Control
## 5              NA     NA Neotoma albigula Rodent   Control
## 6              NA     NA Neotoma albigula Rodent   Control
```

## Create a Local Database

You can add another table to the same .sqlite file using `csv_to_sqlite()` as long as a different table name is specified.

```
csv_to_sqlite(csv_file = "actor.csv",
              sqlite_file = "sakila.sqlite",
              table_name = "actor")
csv_to_sqlite(csv_file = "film.csv",
              sqlite_file = "sakila.sqlite",
              table_name = "film")
```

The above code will create the `sakila.sqlite` file in the first function call with the table “actor”. The second function call will then add the “film” table to that same file.

## Convert from dplyr Functions to SQL

The `dbplyr` package is an add-on to the `dplyr` package that allows you to use remote database tables as if they are in-memory data frames by automatically converting dplyr code into SQL. We can install it in the normal way: `install.packages("dbplyr")`. If you don't install it, you will be prompted to when you do something that requires it.

Example:

```r
# Access the SQL table "surveys" from the mammals_con connection
mammal_surveys <- tbl(mammals_con, "surveys")
mammal_surveys
```

```
## # Source:   table<surveys> [?? x 9]
## # Database: sqlite 3.45.0 [/Users/rickbrown/My Drive/Math 3190 -
##    record_id month   day  year plot_id species_id sex
##        <int> <int> <int> <int>   <int> <chr>      <chr>
##  1         1     7    16  1977       2 NL         M
##  2         2     7    16  1977       3 NL         M
##  3         3     7    16  1977       2 DM         F
```

## Convert from dplyr Functions to SQL

```r
mammal_summary <- mammal_surveys |>
  filter(month == 7 & year == 1992) |>
  group_by(species_id) |>
  summarize(avg_weight = mean(weight, na.rm = TRUE)) |>
  arrange(desc(avg_weight))
head(mammal_summary)
```

```
## # Source:     SQL [6 x 2]
## # Database:   sqlite 3.45.0 [/Users/rickbrown/My Drive/Math
## # Ordered by: desc(avg_weight)
##   species_id avg_weight
##   <chr>           <dbl>
## 1 NL              131.
## 2 DS              120
## 3 SO               57.5
## 4 DO               44.6
## 5 DM               43.4
```

## Convert from dplyr Functions to SQL

The show_query() function will show an SQL query that will accomplish the same thing the dplyr functions did to create the object. Sometimes these queries look a bit strange, but they work!

```
show_query(mammal_summary)
```

```
## <SQL>
## SELECT `species_id`, AVG(`weight`) AS `avg_weight`
## FROM (
##   SELECT `surveys`.*
##   FROM `surveys`
##   WHERE (`month` = 7.0 AND `year` = 1992.0)
## ) AS `q01`
## GROUP BY `species_id`
## ORDER BY `avg_weight` DESC
```

## Same thing with SQL

```
SELECT `species_id`, AVG(`weight`) AS `avg_weight`
FROM (
  SELECT `surveys`.*
  FROM `surveys`
  WHERE (`month` = 7.0 AND `year` = 1992.0)
) AS `q01`
GROUP BY `species_id`
ORDER BY `avg_weight` DESC
```

Table 17: Displaying records 1 - 10

| species_id | avg_weight |
|---|---:|
| NL | 130.66667 |
| DS | 120.00000 |
| SO | 57.50000 |
| DO | 44.60000 |

## Object Sizes

What is especially amazing about this is even though we are working with an object called `mammal_surveys`, it is actually not read into memory. We can see that here using the `obj_size()` function in the `lobstr` package:

```
lobstr::obj_size(mammal_surveys)
```

```
## 6.86 kB
```

```
# collect function brings the object into memory
mammal_surveys2 <- collect(mammal_surveys)
lobstr::obj_size(mammal_surveys2)
```

```
## 1.57 MB
```

So, being an **R** wizard also makes you an SQL wizard (by proxy)!

## Close Connection

As always, we should not forget to disconnect when we are done.

```
dbDisconnect(con)
dbDisconnect(combined_con)
dbDisconnect(mammals_con)
```

## Conclusion

- Knowing SQL is a great skill to not only access databases remotely, but also to be able to work with large files locally without reading them into **R**.
- Even though tools exist to convert **R** code to SQL queries, it is still important to know both to be able to handle working with any database or data set.

## Session info

```r
sessionInfo()
```

```
## R version 4.3.2 (2023-10-31)
## Platform: aarch64-apple-darwin20 (64-bit)
## Running under: macOS Sonoma 14.2.1
##
## Matrix products: default
## BLAS:   /Library/Frameworks/R.framework/Versions/4.3-arm64/Resources/lib/libRblas.0.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/4.3-arm64/Resources/lib/libRlapack.dylib;  LAPACK version 3
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## time zone: America/Denver
## tzcode source: internal
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets
## [6] methods   base
##
## other attached packages:
##  [1] inborutils_0.4.0 lubridate_1.9.3  forcats_1.0.0
##  [4] stringr_1.5.1    dplyr_1.1.4      purrr_1.0.2
##  [7] readr_2.1.5      tidyr_1.3.1      tibble_3.2.1
## [10] ggplot2_3.4.4    tidyverse_2.0.0  RMySQL_0.10.27
## [13] DBI_1.2.1
##
## loaded via a namespace (and not attached):
##  [1] gtable_0.3.4       xfun_0.41
##  [3] htmlwidgets_1.6.4  tzdb_0.4.0
##  [5] vctrs_0.6.5        tools_4.3.2
##  [7] crosstalk_1.2.1    generics_0.1.3
##  [9] parallel_4.3.2     curl_5.2.0
```