STAT 19000 Project 11 Examples

In this project we will explore another database engine called MySQL. You will find that the learning curve for basic SQL functionality is very low when switching between one of the more popular RDBMS's like MySQL, Postgresql, SQLite, Microsoft SQL Server, etc. This doesn't mean that the engines don't have some *very* different features, but it does mean that learning to navigate and pull some data will be a pretty straightforward process once you've done it on one system.

In addition, one of the primary ways to interact with any database is by using some package or interface within a given programming language. In this project we will learn to connect to both a MySQL and SQLite database from *within* R, and make queries.

Important note: This set of examples assumes you are using https://rstudio.scholar.rcac.purdue.edu/

MySQL & `RMariaDB`

One major way that MySQL differs from SQLite is that MySQL has a server running that we must connect to in order to make queries, whereas an SQLite database is represented by a single file.

To connect to a MySQL database:

1. Open a terminal in scholar
2. Execute the following command: `mysql -u <username> -h scholar-db.rcac.purdue.edu -p`

   Note: `read-only` usernames and passwords are available for class use in scholar at `/class/datamine/data/spring2020/mysql_credentials.txt`. For now, connect to MySQL using the `imdb` username.
3. Enter the password when prompted, and you will be presented with a prompt that resembles:

   `MariaDB [(none)]>`

   Note: MariaDB is a fork of MySQL. You can read about it here.

Get a list of the databases available to you:

`show databases;`

Make the imdb database your active database:

`use imdb;`

Show the tables in the active database:

`show tables;`

Show important table information:

`show columns from <table_name>;` or `describe <table_name>;`

Show any indexes a table may have:

`show index from <table_name>;`

Create a connection to the MySQL database using `RMariaDB`:

```
# if you haven't already, install RMariaDB
install.packages("RMariaDB")

library(RMariaDB)
host <- "scholar-db.rcac.purdue.edu"

# fill in the following values from the credentials provided on scholar at
# /class/datamine/data/spring2020/mysql_credentials.txt
# Note that you should not include the following two lines anywhere in your solutions.
user <- "<username>"
password <- "<password>"

connection <- dbConnect(RMariaDB::MariaDB(), host = host, user = user, password = password)
```

Connect to the proper database:

```
# executes a statement. use when you don't want a return value
dbExecute(connection, "USE imdb;")

# use dbGetQuery if you want the results (and they fit in memory) as a data.frame
dbGetQuery(connection, "SHOW tables;")
```

Connect directly to the proper database without using the USE command:

```
library(RMariaDB)
host <- "scholar-db.rcac.purdue.edu"
dbname <- "imdb"

# fill in the following values from the credentials provided on scholar at
# /class/datamine/data/spring2020/mysql_credentials.txt
# Note that you should not include the following two lines anywhere in your solutions.
user <- "<username>"
password <- "<password>"

connection <- dbConnect(RMariaDB::MariaDB(), host = host, dbname=dbname, user = user, password = password)

# this will now work right away
dbGetQuery(connection, "SHOW tables;")
```

Get results directly into a data.frame, all at once:

```
# this sends, fetches, and clears for you
result <- dbGetQuery(connection, "SELECT * FROM episodes WHERE show_title_id='tt0108778'")
```

Get a result from a query one piece at a time:

```

```

```r
res <- dbSendQuery(connection, "SELECT * FROM titles;")
some_data <- dbFetch(res, n=10)
some_data

# get the next 20 results (after the first 10 you fetched)
more_data <- dbFetch(res, n=20)
more_data

# until you try and dbFetch past the end of the data, this will be FALSE
dbHasCompleted(res)

# For example:
dbGetQuery(connection, "SELECT COUNT(*) FROM episodes;")

# This means we have 4658378 results to fetch before
dbHasCompleted(res)
# will return TRUE. Let's test it out:
# First clear the result
dbClearResult(res)
res <- dbSendQuery(connection, "SELECT * FROM episodes;")
dbHasCompleted(res) # FALSE
dat <- dbFetch(res, n=4658377)
dbHasCompleted(res) # FALSE
last <- dbFetch(res, n=1)
dbHasCompleted(res) # FALSE, what? You have to try and fetch *past* the end
one_more <- dbFetch(res, n=1)
one_more # empty
dbHasCompleted(res) # TRUE

# Last but not least, you must clear your results when using dbSendQuery,
and dbFetch
dbClearResult(res)

# And if you are done with the database, you should disconnect
dbDisconnect(con)
```

How to make comments in MySQL:

```
# this is how you make a comment, the -- notation does not work
/* This is how you make a block comment, this still works in MySQL,
    as you can see.
*/
```

How to quit MySQL:

```
/* In sqlite we can quit/exit using .exit.
In MySQL, in order to quit or exit you simply type exit and then press enter
*/
```

How to use the LIMIT clause:

```
/* In MySQL, you must not use parenthesis for the limit clause like you can in sqlite.
*/
# this is not valid:
select * from table limit(5);

# this IS valid
select * from table limit 5;
```

SQLite & `RSQLite`

Luckily, there is virtually nothing new to learn in order to use SQLite within R.

Create a connection to an sqlite database:

```
# if you haven't already, install RMariaDB
install.packages("RSQLite")

library(RSQLite)

connection <- dbConnect(RSQLite::SQLite(), "/class/datamine/data/spring2020/imdb.db")
```

Get a list of the tables:

```
# this won't work
dbGetQuery(connection, ".tables")

# instead use this
dbListTables(connection)
```

And that is the extent of the differences (at least that we will cover).

Examples

So when are we supposed to use ON vs WHERE?

This is a great question. You want to use ON to *join* data, and WHERE to *filter* data. Here is a great (and easy to read) article that should help: https://dataschool.com/how-to-teach-people-sql/difference-between-where-and-on-in-sql/

So let's say we want to join the episodes and title tables from out imdb.db database found on scholar:

`/class/datamine/data/spring2020/imdb.db`

In addition, we want to just do this for FRIENDS. In this case we want to join the tables where the episode_title_id from the episodes table and the title_id from the titles table are the same. Then, we'd want to filter out all rows where the show_title_id from the episodes table (or alternatively the title_id from the titles table) is not 'tt0108778'.

So how about this?

```sql
SELECT *
FROM episodes AS e
WHERE e.show_title_id = 'tt0108778'
INNER JOIN titles AS t
  ON e.episode_title_id = t.title_id;
```

This won't work, the WHERE clause must come *after* the join. How about this?

```sql
SELECT *
FROM episodes AS e
INNER JOIN titles AS t
  ON e.episode_title_id = t.title_id
  WHERE e.show_title_id = 'tt0108778';
```

Yes, this will work. Now some may say, why don't we just move the show_title_id='tt0108778' part to the ON clause?

```sql
SELECT *
FROM episodes AS e
INNER JOIN titles AS t
  ON e.episode_title_id = t.title_id
  AND e.show_title_id = 'tt0108778';
```

In fact, you *can* do this and you will get the same result. However, let's look at if you do this and a LEFT JOIN:

```sql
SELECT *
FROM episodes AS e
LEFT JOIN titles AS t
  ON e.episode_title_id = t.title_id
  AND e.show_title_id = 'tt0108778';
```

Here, the result would be every row in the episodes table on the left side. On the right side, we'd have appended title information for only FRIENDS episodes. If instead you had done:

```sql
SELECT *
FROM episodes AS e
LEFT JOIN titles AS t
  ON e.episode_title_id = t.title_id
  WHERE e.show_title_id = 'tt0108778';
```

The interim table (the table that is created *before* the WHERE filtering), would contain every row in the episodes table and every matching piece of title info appended. Then it would be filtered using the WHERE clause and only episodes of FRIENDS would remain.

How about using the BETWEEN clause with the GROUP BY, is this possible? Yes. Here we demonstrate using the chinook database which can be found on scholar:

`/class/datamine/data/spring2020/chinook.db`

```
--Get the SUM(Total) for invoices in 2009 and invoices in 2013
SELECT InvoiceDate, SUM(Total) FROM invoices GROUP BY InvoiceDate BETWEEN
'2009-01-01' AND '2009-12-31', InvoiceDate BETWEEN '2013-01-01' AND '2013-12-
31';
```

The result:

InvoiceDate|SUM(Total)
2010-01-08 00:00:00|1428.56
2013-01-02 00:00:00|450.58
2009-01-01 00:00:00|449.46

You can see that in 2009 the total was 449.46 and in 2013 the total was 450 and for the rest of the dates the total was 1428. You can confirm these results with the following queries:

```
SELECT SUM(Total) FROM invoices WHERE InvoiceDate BETWEEN '2013-01-01' AND
'2013-12-31';
SELECT SUM(Total) FROM invoices WHERE InvoiceDate BETWEEN '2009-01-01' AND
'2009-12-31';
SELECT SUM(Total) FROM invoices;
```

If at this point you find you would like some more MySQL examples, you can find a good tutorial here.