Stat 240 - Lab 03

Dr. Lloyd T. Elliott

January 27, 2022

The goal of this document is to walk you through more advanced *RSQLite* code in *R*. This document will show you how to handle *SQL* in a more realistic setting, and also there is a question about kernel density estimates. Provide the solutions to the questions through the quiz tab on canvas.

Reading the Olympic Dataset into R

Make sure you've downloaded the file `lab03.sqlite` and that you can connect to it in *R*. This database extends the data you worked with for the previous lab with new tables. Let's get some info about those new tables! What we did last week actually loads an entire table into *R* and runs code on that big `data.frame`. This week we are using more sophisticated code that performs a query but doesn't load the entire database into local memory. When the datafile is realistically large, you will not be able to load the entire table into *R* and will need to use a smarter workflow. We'll use `PRAGMA` to get some table info but note that unlike most other queries we've used, `PRAGMA` is specific to *SQLite*. The database software *MySQL* and other such database software have different ways of doing this. Let's start with exploring the tables:

```
dbcon = dbConnect(SQLite(), dbname='lab03.sqlite')
dbListTables(dbcon)
```

The command `PRAGMA` returns a data frame with one row for each column of the database table. The data frams includes the column id, the column name, the type of data (integer, text,...), and the `notnull` column states if the column has any `NULL` values. The column `dflt_value` gives a default value, if there is one. Use the `PRAGMA` command to find information about the columns of the new tables:

```
query = "PRAGMA table_info('WinterO')"
dbGetQuery(dbcon, query)
```

The `DISTINCT` Operation

In a table, a column may contain many duplicate values and sometimes you only want to list the different/distinct values. The *DISTINCT* operation can be used to return only distinct/unique values from a *SQL* database. The *RSQLite* software shares much syntax with *SQL*, which is very well documented. You can get extensive help for *SQL* commands here: `https://www.w3schools.com/`

sql/sql_distinct.asp and here: https://dev.mysql.com/doc/. **Question 1a, (3 points)**: How many distinct years are present among the medals listed in the WinterO table in lab03.sqlite? **Question 1b, (3 points)**: Provide code that can calculate the value you found in the previous section, making use of the DISTINCT *SQL* command.

Returning records in order

As described in the lecture for this week, we can use the *ORDER BY* command to sort the returned records by one or more columns. *ORDER BY variablename* sorts the records in ascending order by default. To sort the records in a descending order, you can use the option *DESC*. The syntax then becomes: *ORDER BY variablename DESC* or *ORDER BY variablename1, variablename2,... DESC*. In this second syntax, ties in variablename1 are broken by examining variablename2. **Question 2a, (4 points)**: Use an SQL command to provide the unique set of distinct of heights of pokemon from the Height_m column of the Pokem table in the database, in descending order, providing your result with one height per line.

dbSendQuery and dbFetch

The function *dbSendQuery* submits and executes the SQL query to the database engine, but it does not extract any records. For that you need to use the function *dbFetch*, and then you must call *dbClearResult* to clear the results when you finish fetching the records you need. On the other hand, *dbFetch* fetches **the next n elements** (rows) from the result set and returns them as a *data.frame*. This makes it easier for you to extract results if all you want to do is print the first 'n' results because it avoids the need for loading all of the results into *R* and then doing *head* or *tail*. You should know these types of tricks for getting results without loading the entire table into *R* - especially when working in real life. From the population and CA tables if we wanted to select all populations of regions in Saskatchewan in 2011 this would become:

```
query = "SELECT Population__2011, Region FROM CA INNER JOIN
POP2011 ON CA.Geographic_name=POP2011.Geographic_name WHERE
province == 'Saskatchewan'"
QueryOut = dbSendQuery(dbcon, query)
QueryOut
```

Notice that *QueryOut* describes results but doesn't actually give results. That's where we need to use *dbFetch*.

```
dbFetch(QueryOut, 5)
```

Running the last command again will give the next rows in the set:

```
dbFetch(QueryOut, 5)
```

To get all remaining rows set the number to -1:

```
dbFetch(QueryOut, -1)
```

Use *dbClearResult* to terminate a partially fetched query. This will provide more memory efficient code.

```
dbClearResult(QueryOut)
```

Let's compare the time that it takes to perform `dbSendQuery` versus `dbFetch`. You can use the syntax `system.time({ ... })` to find the amount of time it takes R to run the code inside the curly braces. **Question 3a, (3 points)**: Write a query string that selects all rows from the CA table of the database provided with this lab. How long does it take to `dbSendQuery` this query 10,000 times? **Question 3b, (3 points)**: Similarly, write code to call `dbFetch` on your query string, and then retrieve one row of the streamed query. How long does it take to run this code 10,000 times? **Question 3c, (2 points)**: Which method was faster, or did they take approximately the same amount of time? Why do you think one was faster than the other (or why do you think they were the same speed)?

SQL Query INSERT INTO

The `INSERT INTO` statement is used to insert new records in a table. Assume that new regions called Statsville and Statsville240 have recently been founded on top of a mountain in British Columbia and that we wish to add it to our CA table. We need to fill in all the values for the row.

```
sql_ins ="INSERT INTO CA
   (Country, Geographic_name, Region, Province, Prov_acr,
     Latitude, Longitude)
   VALUES
     ('CA', 'V5A','Statsville', 'British Columbia', 'BC',
       '49.278417', '-122.916454'),
     ('CA', 'V5A','Statsville240', 'British Columbia', 'BC',
       '49.278417', '-122.916454')"
dbSendQuery(dbcon, sql_ins)
```

Note that the output will tell you how many rows have been changed. To see them you will need to use the extraction queries from before. Note that we didn't input values into all the columns. Those columns are filled with the default value (as per whatever we found when using *PRAGMA* last.)

```
sql_ins3 = "SELECT * FROM CA WHERE Region LIKE 'Statsville%'"
dbGetQuery(dbcon, sql_ins3)
```

SQL Query DELETE

The `DELETE` statement is used to delete records in a table. Let's say I want to delete the Statsville entry I inserted to the table.

```
sql_del = "DELETE FROM CA WHERE Region == 'Statsville'"
dbSendQuery(dbcon, sql_del)
```

Now when you retrieve the results you will see that Statsville240 remains:

```
sql_ins3 = "SELECT * FROM CA WHERE Region LIKE 'Statsville%'"
dbGetQuery(dbcon, sql_ins3)
```

Question 4a, (4 points): In at most a paragraph, explain what the `LIKE` operator does in SQL (you may need to search the resources provided on page 1). Give some example code demonstrating the `LIKE` operator with example queries on the CA table. Show the code, and the output of the code (design the queries so that the output is less than half a page), and summarize why we get this output.