SQLite vs Dplyr for Data Analytics

Question 1

SQLite

```
# Question 1

# Create SQLite database
db <- dbConnect(SQLite(), dbname = 'census_income.sqlite')

# Create 'Income' table
dbSendQuery(conn = db,
            'CREATE TABLE Income
(AAGE INT, ACLSWKR TEXT, ADTIND TEXT, ADTOCC TEXT, AHGA TEXT,
AHRSPAY NUM, AHSCOL TEXT, AMARITL TEXT, AMJIND TEXT, AMJOCC TEXT,
ARACE TEXT, AREORGN TEXT, ASEX TEXT, AUNMEM TEXT, AUNTYPE TEXT,
AWKSTAT TEXT, CAPGAIN NUM, CAPLOSS NUM, DIVVAL NUM, FILESTAT TEXT,
GRINREG TEXT, GRINST TEXT, HDFMX TEXT, HHDREL TEXT, MARSUPWT NUM,
MIGMTR1 TEXT, MIGMTR3 TEXT, MIGMTR4 TEXT, MIGSAME TEXT, MIGSUN TEXT,
NOEMP NUM, PARENT TEXT, PEFNTVTY TEXT, PEMNTVTY TEXT, PENATVTY TEXT,
PRCITSHP TEXT, SEOTR TEXT, VETQVA TEXT, VETYN TEXT, WKSWORK NUM,
YEAR TEXT, TRGT TEXT)'
            )

# Import data frame into database
data <- read.csv('census-income.data.gz')
dbWriteTable(conn = db, name = 'Income', value = data, row.names =
FALSE, append = TRUE)
```

Dplyr

```
# Question 1

# Creating data frame
income_data <- read.csv('census-income.data.gz')
```

Analysis

One of the first tasks in the data preparation was to create tables from which the subsequent analysis would take place, this involved importing the data an[1]d manipulating the existing headers to make it more accessible. Using Dplyr, it was just a simple task of creating a data frame, using the **read.csv()** function**,** the output of this was the data frame in

[1] 13015443

which the first row contained the headers and the remaining rows contained the bulk of the data.

The corresponding task using SQLite was a lot more tedious, to begin with, a database connection needed to be established using **dbConnect().** Once the database had been created and connected, a table was created through the **CREATE TABLE** statement, although the syntax is case insensitive, it helps with readability to use SQLite's language in upper-case. Following this statement, the name of the table is given, followed by columns and their corresponding data-types. At this point the table is empty, and the data needs to imported, **dbWriteTable()** provides this functionality, allowing you to add an existing data-frame into the SQLite table, it's worth noting however, that the **append** parameter needs to be changed to allow the data to be added to an existing table.

Question 2

SQLite

```
# Question 2

dbGetQuery(db, 'ALTER TABLE Income RENAME TO Income_old')

# Create a table with SS_ID as primary key
dbSendQuery(conn = db, 'CREATE TABLE Income
            (
SS_ID INTEGER PRIMARY KEY AUTOINCREMENT, AAGE INT, ACLSWKR TEXT,
ADTIND TEXT, ADTOCC TEXT, AHGA TEXT, AHRSPAY NUM, AHSCOL TEXT,
AMARITL TEXT, AMJIND TEXT, AMJOCC TEXT, ARACE TEXT, AREORGN TEXT,
ASEX TEXT, AUNMEM TEXT, AUNTYPE TEXT, AWKSTAT TEXT, CAPGAIN NUM,
CAPLOSS NUM, DIVVAL NUM, FILESTAT TEXT, GRINREG TEXT, GRINST TEXT,
HDFMX TEXT, HHDREL TEXT, MARSUPWT NUM, MIGMTR1 TEXT, MIGMTR3 TEXT,
MIGMTR4 TEXT, MIGSAME TEXT, MIGSUN TEXT, NOEMP NUM, PARENT TEXT,
PEFNTVTY TEXT, PEMNTVTY TEXT, PENATVTY TEXT, PRCITSHP TEXT, SEOTR
TEXT, VETQVA TEXT, VETYN TEXT, WKSWORK NUM, YEAR TEXT, TRGT TEXT)'
            )

dbGetQuery(db, 'INSERT INTO Income
            (
AAGE, ACLSWKR, ADTIND, ADTOCC, AHGA, AHRSPAY, AHSCOL, AMARITL,
AMJIND, AMJOCC, ARACE, AREORGN, ASEX, AUNMEM, AUNTYPE, AWKSTAT,
CAPGAIN, CAPLOSS, DIVVAL, FILESTAT, GRINREG, GRINST, HDFMX, HHDREL,
MARSUPWT, MIGMTR1, MIGMTR3, MIGMTR4, MIGSAME, MIGSUN, NOEMP, PARENT,
PEFNTVTY, PEMNTVTY, PENATVTY, PRCITSHP, SEOTR, VETQVA, VETYN,
WKSWORK, YEAR, TRGT
            )

SELECT
AAGE, ACLSWKR, ADTIND, ADTOCC, AHGA, AHRSPAY, AHSCOL, AMARITL,
AMJIND, AMJOCC, ARACE, AREORGN, ASEX, AUNMEM, AUNTYPE, AWKSTAT,
CAPGAIN, CAPLOSS, DIVVAL, FILESTAT, GRINREG, GRINST, HDFMX, HHDREL,
MARSUPWT, MIGMTR1, MIGMTR3, MIGMTR4, MIGSAME, MIGSUN, NOEMP, PARENT,
PEFNTVTY, PEMNTVTY, PENATVTY, PRCITSHP, SEOTR, VETQVA, VETYN,
WKSWORK, YEAR, TRGT

FROM Income_old;'
)




# Copying gave 598566 values
dbGetQuery(db, 'DELETE FROM Income
                WHERE SS_ID > 199522;')
```

Dplyr

```
# Question 2 - Add SS_ID
nrow(income_data) # Find how many rows
income_data <- mutate(income_data, SS_ID = c(1:nrow(income_data)))

# Change column names
income_data <- income_data %>% rename(AAGE = X73, ADTIND = X0,
ADTOCC = X0.1, AHGA = High.school.graduate, AHRSPAY = X0.2, AMJIND =
Not.in.universe.or.children, AMJOCC = Not.in.universe.2,
                                      AREORGN = All.other, ASEX =
Female, AWKSTAT = Not.in.labor.force, GRINREG = Not.in.universe.5,
GRINST = Not.in.universe.6, PARENT = Not.in.universe.7,
                                      PRCITSHP =
Native..Born.in.the.United.States, WKSWORK = X0.8, ARACE = White)
```

Analysis

In order to ensure SQLite database can be queried relationally, a unique identifier needed to be added, as per the requirements, this was appended to the table using ‘SS_ID’ as the column name. In order to use the columns entries as the relational identifier, it’s declaration needed to include the **PRIMARY KEY** statement, appended to the end is also **AUTOINCREMENT** which allows each row to be given its own ROWID, incremented by 1. In this case however, the table had already been created without a primary key, this required a workaround. My approach involved, renaming the existing table to a new name using **ALTER TABLE** and transferring the data from it to the new table (with the SS_ID row). Moving data between tables was relatively straight forward and implemented using an **INSERT INTO (column names) SELECT column names FROM** statement, where the columns in question are explicitly mentioned. Syntactically this is awkward, since parenthesis are required in the first set of column name declarations and not in the second.

In order to add a new column to a data-frame using Dplyr, the process is more intuitive, however the amount of code required is far less; the method that I chose meant that the number of rows needs to be determined which was done using **nrow()**, once this number had been established, the **mutate()** function was used. This function is used primarily to append columns to existing data-frames and can reference other columns through mathematical expressions, and is particular useful when different columns share features from which patterns can be drawn. Within mutate, the dataset and the column name (along with its contents) are added as parameters.

In order to add consistency to the analysis, I decided to rename the headers to match that of the SQLite table, Dplyr has a **rename()** function that allows this exact manipulation to be made this was less efficient than using SQLite's **AS** statement as a large number of columns needed to be modified each by variable declaration.

Question 3

SQLite

```
# Question 3

# Selecting the number of people by sex and race
dbGetQuery(db, 'SELECT ASEX AS White, COUNT(SS_ID) AS Total
                FROM Income WHERE ARACE = " White" GROUP BY ASEX')
dbGetQuery(db, 'SELECT ASEX AS Asian/PI, COUNT(SS_ID) AS Total
                FROM Income WHERE ARACE = " Asian or Pacific
Islander" GROUP BY ASEX')
dbGetQuery(db, 'SELECT ASEX AS Ind_Esk, COUNT(SS_ID) AS Total
                FROM Income WHERE ARACE = " Amer Indian Aleut or
Eskimo" GROUP BY ASEX')
dbGetQuery(db, 'SELECT ASEX AS Black, COUNT(SS_ID) AS Total
                FROM Income WHERE ARACE = " Black" GROUP BY ASEX')
dbGetQuery(db, 'SELECT ASEX AS Other, COUNT(SS_ID) AS Total
                FROM Income WHERE ARACE = " Other" GROUP BY ASEX')
```

Dplyr

```
# Question 3

# Selecting the number of people by sex and race
sex_race_numbers <- income_data %>% group_by(ARACE, ASEX) %>%
tally() %>% arrange(desc(ARACE)) %>% rename(sex_race_numbers, Race =
ARACE, Sex = ASEX, Total = n)
```

Analysis

Following the data preparation, the querying and data extraction could be started. The first requirement, was to find out how many people existed in the dataset based on two factors, race and gender. Using SQLite, this was done using a fairly lightweight solution, **COUNT** was used to count the number of rows for each pair of factors. The approach I used, was to create queries for each race and **GROUP BY** this factor, preventing the need to make separate queries for each gender. In order to separate the races within the queries, the **WHERE** statement was used to the select them independently.

When it came to querying, and extracting data, utilizing Magrittr's piping symbol (**%>%)** along with Dplyr's functions made complex analysis extremely fast and efficient, a caveat to this approach though was the code suffered in readability. Extractions begun with piping the data into multiple different functions, strung together. In this case, the data was first piped into a **group_by()** function which is analogous to the SQLite method. In this case however, both factors were grouped, which avoided the need for code repetition. The grouping was then piped into the **tally()** method which counted the row occurrences based on the previous grouping parameters, returning the relative frequencies. Following this, headers were renamed and the order in which the data was returned was modified for readability purposes.

<u>Question 4</u>

SQLite

```
# Question 4

# Create table called 'Workers' with the relevant data
dbSendQuery(db, 'CREATE TABLE Workers
                (
                SS_ID INTEGER PRIMARY KEY AUTOINCREMENT,
                AHRSPAY NUM,
                WKSWORK NUM,
                ARACE TEXT)'
            )

# Select all rows with a non-zero wage
dbGetQuery(db, 'INSERT INTO Workers
                (
                AHRSPAY,
                WKSWORK,
                ARACE
                )
                SELECT
                AHRSPAY,
                WKSWORK,
                ARACE
                FROM Income
                WHERE AHRSPAY > 0;'
          )
# Calculate average annual income for each race
dbGetQuery(db, 'SELECT ARACE AS Ethnicity, AVG(
(AHRSPAY*40)*WKSWORK) AS Average_Income
FROM Workers
GROUP BY ARACE ORDER BY AVG( (AHRSPAY*40)*WKSWORK ) DESC')
```

Dplyr

```
# Question 4

# Creating a data frame called ‘Workers’ with the relevent data
workers_data <- filter(income_data, AHRSPAY > 0)[ , c('ARACE',
'AHRSPAY', 'WKSWORK')] %>%
                rename(Race = ARACE, Wage = AHRSPAY, Weeks_worked =
WKSWORK) %>% arrange(workers_data, desc(Race))

# Calculate average annual income for each race
average_incomes <- workers_data %>% group_by(Race)
%>%summarise(Average_income = mean(Wage*Weeks_worked*40)) %>%
arrange(desc(Average_income))
```

Analysis

Following this, average annual incomes were required, using the conditions which eliminated an overwhelming amount of data from analysis. For this reason, I decided to create a new table where those who failed to meet the criteria were omitted. Creating this new table, the same protocol was used with the inclusion however of a **WHERE** statement. This made sense, as less computational resources were then needed in the subsequent querying.

Extracting the data was then carried out by using an array of statements from the SQLite language including **AVG, GROUP BY, ORDER BY and DESC**. Averaging, enabled the mathematical calculations to be conducted with a single statement, whereas the ordering produced results in a more organized manner, with the highest averages at the top.

The equivalent method was carried out using Dplyr, but with the addition of a data-frame as opposed to table. In order to restrict the subset of the data from the main data-frame, the **mutate()** function was used, along with the subsequent bracket notation to extract specific columns. Although this could have been piped into the **select()** method, it cut down on code and meant one less function needed to be used. Once the data had been dissected, the **rename()** function was again used, however, since few columns needed modification, the assignment of new column names was done as variable declarations. It’s worth noting that, the new column name is placed on the left hand side of the ‘=’ sign, which is a syntactic mess. Finally, to preserve ordering, the **arrange()** method was the final function that the data was passed through, taking the column name in question as a parameter and returning the appropriate formatting that was requested.

Question 5

SQLite

```
# Question 5

# Creating the 'Person' table
dbSendQuery(db, 'CREATE TABLE Person
            (
            SS_ID INTEGER PRIMARY KEY AUTOINCREMENT, AAGE NUM, AHGA
TEXT, ASEX TEXT, PRCITSHP TEXT, PARENT TEXT, GRINST TEXT, GRINREG
TEXT, AREORGN TEXT, AWKSTAT TEXT)'
            )

dbGetQuery(db, 'INSERT INTO Person
           (
           AAGE, AHGA, ASEX , PRCITSHP, PARENT, GRINST, GRINREG,
AREORGN, AWKSTAT
           )
           SELECT
           AAGE, AHGA, ASEX , PRCITSHP, PARENT, GRINST, GRINREG,
AREORGN, AWKSTAT
           FROM Income;'
           )
```

Dplyr

```
## 5 - Creating data frames using columns from the income data
job_data <- income_data %>% select(SS_ID, ADTIND, ADTOCC, AMJOCC,
AMJIND)
```

Analysis

The next task was fairly simple and an extension of a method that had been carried out previously, individual tables needed to be created using data from the main table, and the corresponding method was carried out with data frames using Dplyr. The purpose of this was to set up the next task, which incorporates joining methods to extract data based on relational mapping.

Question 6.1

SQLite

```
# Question 6.1

# Finding the highest hourly wage
dbGetQuery(db, 'SELECT MAX(AHRSPAY) AS Highest_wage FROM Pay')

# The number in each state with the highest earning job
dbGetQuery(db, 'SELECT Person.GRINST AS State, COUNT(*) AS Total
                FROM Person
                INNER JOIN Job ON Job.SS_ID = Person.SS_ID
                WHERE AMJOCC = " Professional specialty"
                GROUP BY GRINST ORDER BY COUNT(*) DESC'
            )

# Extracting all of the data for the highest earner
dbGetQuery(db, 'SELECT Pay.AHRSPAY, Person.GRINST,
                Job.AMJOCC, Job.AMJIND
                FROM Person
                INNER JOIN Job ON Job.SS_ID = Person.SS_ID
                INNER JOIN Pay ON Pay.SS_ID = Person.SS_ID
                WHERE AHRSPAY = 9999;
           ')

# The number in the state where the highest earner is from
dbGetQuery(db, 'SELECT GRINST AS State, COUNT(*) AS Total FROM
Person WHERE GRINST = " Not in universe"')
# The number in the job type the highest earner has
dbGetQuery(db, 'SELECT AMJIND AS Job, COUNT(*) AS Total FROM Job
WHERE AMJIND = " Other professional services"')
# The number in the major industry the highest earner is in
dbGetQuery(db, 'SELECT AMJOCC AS Industry, COUNT(*) AS Total FROM
Job WHERE AMJOCC = " Professional specialty"')
```

Dplyr

```
# Question 6.1

# Finding the highest hourly wage and their SS_ID
max_wage <- pay_data %>% summarise(max(AHRSPAY)) %>% .$`max(AHRSPAY)

# Finding the SS_ID of the row with the highest hourly wage `
max_wage_id <- pay_data %>% filter(AHRSPAY == max(AHRSPAY)) %>%
select(SS_ID, AHRSPAY) %>% .$`SS_ID`

# The number in each state with the highest earning job
high_job <- job_data %>% filter(SS_ID == max_wage_id) %>% .$`AMJOCC`
# List of all the SS_ID's with the highest job
high_job_ids <- job_data %>% filter(AMJOCC == high_job) %>%
.$`SS_ID`

numbers_each_state_job <-
  persons_data[high_job_ids, ] %>% group_by(GRINST) %>% tally() %>%
arrange(desc(n)) %>% rename(State = GRINST, Total = n)

# The number in the state where the highest earner is from
high_state <- persons_data %>% filter(SS_ID == max_wage_id) %>%
.$`GRINST`

number_in_state <- persons_data %>% filter(GRINST == high_state) %>%
group_by(GRINST) %>% tally() %>% rename(State = GRINST, Total = n)

# The number in the job type the highest earner has
number_in_job <- job_data %>% filter(AMJOCC == high_job) %>% tally()
%>% rename('Total in job' = n)

# The number in the major industry the highest earner is in
high_industry <- job_data %>% filter(SS_ID == max_wage_id) %>%
.$`AMJIND`
number_in_industry <- job_data %>% filter(AMJIND == high_industry)
%>% tally() %>% rename('Total in industry' = n)
```

Analysis

More complex analysis was needed for this question, although the first task was simply to select the highest wage. This was relatively easy, and in SQLite required the use of **MAX** which takes the column being assessed as a parameter and returns the maximum value in it. Once this value had been found, the total number in each state within the dataset with that job needed to be extracted. Since the data that could provide this answer was distributed across three different tables, **INNER JOIN** needed to be utilized. This feature allowed the rows of different tables to be compared based on a condition that was specified. In this case, that condition was such that the rows extracted all had the same job value as that of the highest earner.

The query extracted the state, and **COUNT** for each state value in a table and was carried out using **GROUP BY**, which allowed the counting method to be applied to each state value iteratively. Following this, data about the relative numbers in the same state, job type and industry as the highest earner was needed. During my analysis, I decided to locate all of the data, from the three tables about those with the highest wage, this made the subsequent analysis much easier since only a single row contained this max value. From this result, I could simply use the column values for the state, job type and industry to locate the relative counts.

Conducting the same tasks using Dplyr proved to be more laborious than I was expecting. Extracting the maximum wage required the use of **summarise()** and the dollar sign to extract the necessary column from the resulting data frame. Once the maximum wage value had been obtained, I decided to create a vector, containing the 'SS_ID' value of the row that contained this value. Using this SS_ID value, I could then extract the relevant information from each of the tables (job type, industry and state) to conduct further analysis. Much like the SQLite method, the tables were modified using **filter(),** to flush out those that didn't match the required condition.

<u>Question 6.2</u>

SQLite

```
# Create Table 'Hispanics'
dbSendQuery(conn = db, 'CREATE TABLE Hispanics
            (
            SS_ID INTEGER PRIMARY KEY AUTOINCREMENT, AHGA TEXT,
AHRSPAY NUM, AMJIND TEXT, AREORGN TEXT, WKSWORK NUM
            )'
            )

# Insert relevant data into 'Hispanics'
dbGetQuery(db, 'INSERT INTO Hispanics
           (
           AREORGN, AHGA, AMJIND, AHRSPAY, WKSWORK
           )

           SELECT
           Person.AREORGN AS Hispanic, Person.AHGA AS Education,
Job.AMJIND AS Industry, Pay.AHRSPAY AS Wage, Pay.WKSWORK AS Weeks
           FROM Person
           INNER JOIN Job ON Job.SS_ID = Person.SS_ID
           INNER JOIN Pay ON Pay.SS_ID = Person.SS_ID
           WHERE
           (
           AHGA IN (" Bachelors degree(BA AB BS)", " Masters
degree(MA MS MEng MEd MSW MBA)", "Doctorate degree(PhD EdD)")
           )
           AND
           (
           AREORGN NOT IN (" All other", " Do not know", " NA")
           );
           ')

# List of industries
dbGetQuery(db, 'SELECT DISTINCT AMJIND AS "List of industries" FROM
Hispanics')

# Average Wage
dbGetQuery(db, 'SELECT AMJIND AS Industry, AVG(AHRSPAY) AS "Average
hourly wage" FROM Hispanics GROUP BY AMJIND ORDER BY AVG(AHRSPAY)
DESC')

WAGE
```

Dplyr

```
# Question 6.2

# Create Table 'Hispanics' with all of the relevent data from the
three tables
hispanics_grads_data <- data.frame(persons_data %>% select(AREORGN,
AHGA), job_data %>% select(AMJIND), pay_data %>% select(AHRSPAY,
WKSWORK)) %>%
filter(AREORGN != " All other", AREORGN != " Do not know", AREORGN
!= " NA", AHGA %in% c(" Bachelors degree(BA AB BS)", " Masters
degree(MA MS MEng MEd MSW MBA)", " Doctorate degree(PhD EdD)")) %>%
rename(Hispanic = AREORGN, Education = AHGA, Industries = AMJIND,
Wage = AHRSPAY,  Weeks = WKSWORK)

# List of industries
hispanic_grads_industries <- hispanics_grads_data %>%
distinct(Industries)

# Average wage
hispanic_grad_average_wages <- hispanics_grads_data %>%
group_by(Industries) %>% summarise(avg_wage = mean(Wage)) %>%
arrange(desc(avg_wage)) %>% rename("Average wage" = avg_wage,
'Industries of Hispanics' = Industries)

# Average weeks
hispanic_grads_average_weeks <- hispanics_grads_data %>%
group_by(Industries) %>% summarise(avg_weeks = mean(Weeks)) %>%
arrange(desc(avg_weeks)) %>% rename("Average weeks worked" =
avg_weeks, 'Industries of Hispanics' = Industries)
```

<u>Analysis</u>

Much like the previous question, the focus was shifted on utilizing relational concepts to derive results from the data. In this case, rows with Hispanic origin were in question.

Applying the same principle as I did for the previous 'average income' query, I decided to create a table with the relevant conditions met. The data import in this case required the use of different conditional statements. In the previous question, those without a wage could be characterized as failing to meet a single condition. However, in this case, the rows needed, had to satisfy multiple conditions, therefore Boolean statements such as **OR, AND** and **NOT** were used. To further segregate the data, Hispanics with specific degrees were required, such that the column corresponding to these values needed to match more than one value, thankfully SQLite allows multiple values to be matched via **IN** statements, where the column value can match any values specified. The converse of this could then be used when values needed to be excluded through **NOT IN** statements

Having created the table that consisted of ‘Hispanics with higher education degrees’, the industries that they work in needed to be queried. Using the **DISTINCT** statement, unique column values could be extracted, removing all duplicated, providing the correct output.

Following this, the average wage and weeks were extracted for each state, using the **AVG** method just like before, with **GROUP BY** used to ensure the function was implemented on each state iteratively.

Carrying out the same analysis with Dplyr, a data frame was created containing the same data, in this case **select()** functions were piped with each of the different data-frames and their corresponding column names. The result of this piping was analogous to the inner joining in SQLite. In order to remove unnecessary data, filtering was used through the **filter()** method, where instead of **NOT IN** the not equal (**!=**) mathematical operator was used. However, a similar feature exists in the Dplyr package with regards to matching multiple values, this is in the form of **%in%** which is followed by a list of values that are tested against. After filtering out the information, the first query could be performed.

Finding the list of states that Hispanic graduates work in was possible with a single function call, **distinct()** with the column name as a parameter and performs just like its SQLite counterpart. Averaging queries were then extracted using the same methods as before such as, **summarise()** to calculate the mean, and **group_by()** to apply the function to each industry.

<u>Bibliography</u>

Anderson, S. (2014). dplyr and pipes: the basics. [Blog].

Cran.rstudio.com. (2016). *Introduction to dplyr*. [online] Available at: https://cran.rstudio.com/web/packages/dplyr/vignettes/introduction.html [Accessed 13 Dec. 2016].

Grolemund, G. (n.d.). *Hands-on programming with R*. 1st ed.

R-tutor.com. (2016). *Data Import | R Tutorial*. [online] Available at: http://www.r-tutor.com/r-introduction/data-frame/data-import [Accessed 13 Dec. 2016].

www.tutorialspoint.com. (2016). *SQLite Tutorial*. [online] Available at: http://www.tutorialspoint.com/sqlite/ [Accessed 13 Dec. 2016].