Student ID : 21022580

CSC-40054: Data Analytics and Databases

Critical Analysis report for part 1 coursework

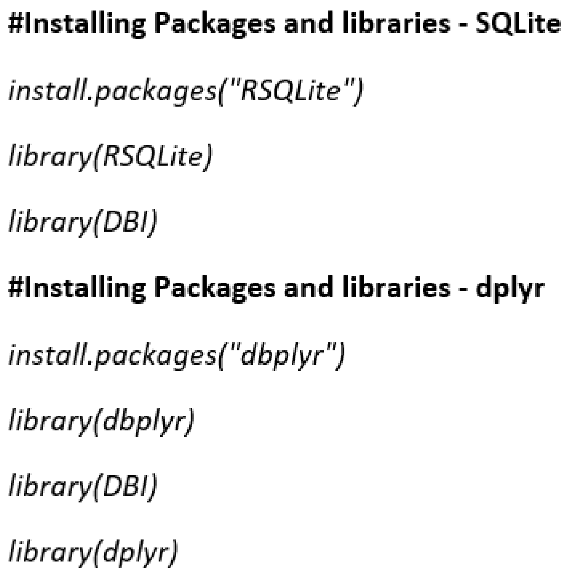
SQLite and Dplyr

SQLite is an in-process library that creates a transactional SQL database engine that is self-contained, serverless, and requires no configuration. The SQLite engine is not a stand-alone process like other databases; user application statically or dynamically depending on needs. It can directly access the storage files. All the SQLite statements are case insensitive making it easier for user. One of the main disadvantages of the SQLite system is its lack of multi-user capabilities which can be found in other RDBMS systems

While Dplyr is one of the packages in R programming languages which is used for data manipulation in user friendly way. It provides a feature to work with data stored directly in the external database. Thus, with dplyr, data can be managed natively in a relational databases, queries can be run on that database and only the results need to be returned. This solves a frequent problem with R: all operations are performed in memory, thus the quantity of data you can work with is limited by the amount of memory available.

Critical Analysis of Part 1 Questions: SQLite v/s Dplyr

Installing relevant packages and libraries



```
#Installing Packages and libraries - SQLite
install.packages("RSQLite")
library(RSQLite)
library(DBI)
#Installing Packages and libraries - dplyr
install.packages("dbplyr")
library(dbplyr)
library(DBI)
library(dplyr)
```

As part of implementation of part 1 question in SQLite, we use RSQLite library which encloses SQLite databases in R and provide an interface in compliance with DBI packages. dbplyr is the dplyr database backend. It will allow users to use remote database tables as if they are in-memory data frames by automatically converting dplyr code into SQL

1.Creating table Income

```
#Creating column header vector
col_names = c("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")
#Reading CSV data set and inserting column headers
library(readr)
census_income <- read_csv("census-income.data.gz", col_names = col_names)
```

We first create a data set which contains the data to be loaded to the new Income table. For reading the dataset we use csv reader for both SQLite and dplyr making use of library-readr. Since the dataset do not have column headers, we must create a separate column header vector. Here the column vector col_names is created which is assigned when the data frame is formed from the given dataset

Creating table in SQLite

```
#Establishing database connection
con <- dbConnect(RSQLite::SQLite(), "census_income.db")
 #Creating Table Income
dbSendQuery(conn = con, "CREATE TABLE IF NOT EXISTS 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)")
#Writing dataset to Income table
dbWriteTable(conn = con, name = "Income", value = census_income, row.names = FALSE, append = TRUE)
```

Creating table in SQLite is difficult compared to dplyr. To begin with we need to establish database connection using dB Connect ().Once dB connection is established, we can create the empty table using CREATE TABLE IF NOT EXISTS statement, only if it does not exists to avoid any conflicts. The SQLite is case insensitive. The Create table statement accepts column names along with datatype as its parameters. An empty table with specified column names and data types will be created with this create query. We can add the existing data frame to the SQLite table using the functionality dbWriteTable (). It should be noted that append parameter should be set to true while writing to the table

```
#Creating income dataset using dplyr

Income <- census_income %>% 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)
```

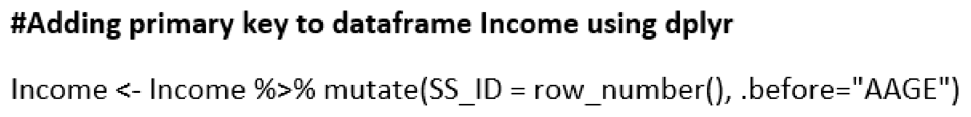
The data manipulation in R using dplyr is much easier as it uses the data frames and modifies accordingly to yield the result. Creating income table is easy in dplyr compared to SQLite. We can directly select the necessary column from the initial dataset using the statement **select().** The dplyr is case sensitive. The data types of columns are not specified in the select statement for creating the

2.Adding Primary key to Income table

SQLite

```
#Adding primary key column with auto increment

dbSendQuery(conn = con,"drop table if exists old_income")

dbSendQuery(conn = con,"ALTER TABLE Income RENAME TO old_income")

dbSendQuery(conn = con,"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)")

#Writing dataset along with primary key to new income table

dbWriteTable(conn = con, name = "Income", value = census_income, row.names = FALSE, append =
TRUE)
```

Dplyr



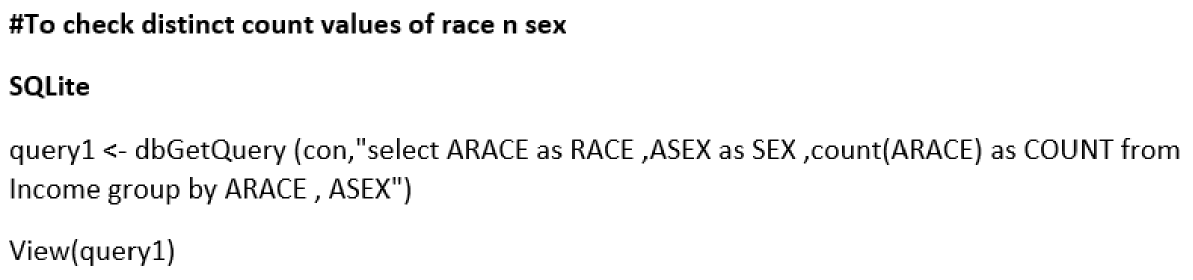
#Adding primary key to dataframe Income using dplyr

```
Income <- Income %>% mutate(SS_ID = row_number(), .before="AAGE")
```

In SQLite we cannot use Alter table statement to create or drop a primary key. We can use ALTER to add new columns to existing table but adding constraints relations are impossible. Instead, we must create a new table with the primary key and copy the data to this new table. Here in the part1, for adding the primary key, SS_ID, we must alter the name of income table and rename it to old_income. The new income table is created along with primary key with auto increment constraint using the statement SS_ID INTEGER PRIMARY KEY AUTOINCREMENT. The newly created table is empty, so user can write the date set into the new table using dbWriteTable() statement with append parameter set to true

While in order to add a new column to data frame in dplyr is much easier with less code. We can append new columns to an existing data frame using mutate()function .It can be used to reference other columns through mathematical expressions. We can use. before() function to place the newly added column to the left of the specified column in the dataset

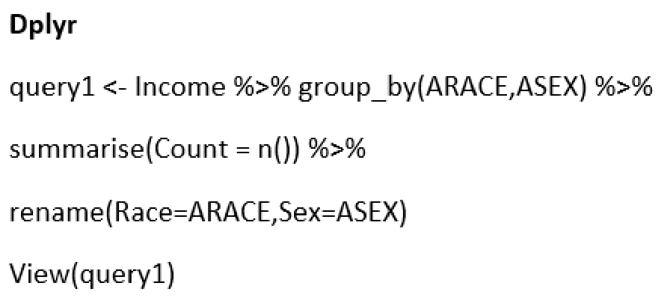
3. query to provide the total number of males and females for each race group reported in the data



#To check distinct count values of race n sex

SQLite

```
query1 <- dbGetQuery (con,"select ARACE as RACE ,ASEX as SEX ,count(ARACE) as COUNT from Income group by ARACE , ASEX")

View(query1)
```



Dplyr

```
query1 <- Income %>% group_by(ARACE,ASEX) %>%

summarise(Count = n()) %>%

rename(Race=ARACE,Sex=ASEX)

View(query1)
```

Here we do the data querying and exploration after the data preparation. In SQLite we could easily find the count values of each race based on race and sex using Count () function. This can be easily queried by grouping the data based on ARACE and ASEX using group by and then taking count(ARACE).Thus using group by we do not have to query for each separately using WHERE condition

Utilizing Magrittr's piping symbol (%>%) along with Dplyr function makes all the complex data analysis, querying and extracting extremely fast and efficient. In this case, the data was first piped into a group_by() function which is similar to the SQLite method and then we find the count using n() in summarise. Following this, headers were renamed as RACE and SEX for readability purposes

4. query to calculate the average annual income of individuals for each race groups, considering only those with non-zero wage per hour

# Calculate the average annual income of the individuals for each race groups

SQLite

```
query2 <- dbGetQuery(con,"select ARACE as RACE, avg(((AHRSPAY * 40))*WKSWORK) as
average_annual_income from Income WHERE AHRSPAY > 0

GROUP BY ARACE")

View(query2)
```

Dplyr

```
query2 <- Income %>%

filter(AHRSPAY > 0) %>%

group_by(ARACE) %>%

summarize(AvgAnnualIncome = mean(WKSWORK*(AHRSPAY * 40)))

View(query2)
```

In SQLite this data can be extracted in a single query with use of array of sql statements Avg(), Group by along with where condition to omit the cases which are not applicable .Here in this case the WHERE AHRSPAY > 0 condition is optional as Avg() in SQLite calculates the average of only non-null or non-zero values in the table.

An equivalent method was used in dplyr where we apply   filter AHRSPAY > 0, group_by (ARACE) into pipe and the summary statistics thus obtained on the dataset is returned using summarize().The average can be found using mean() in dplyr which is equivalent to avg() in SQLite.

5. Creating new tables Person, pay and Job by extracting fields from another table

Dplyr

```
#Creating dataframe Person

Person <- Income %>%
select(Id=SS_ID,Age=AAGE,education=AHGA,sex=ASEX,citizenship=PRCITSHP,family_members_under_18=PAR
ENT,previous_state=GRINST,previous_region=GRINREG,Hispanic_origin=AREORGN,employment_stat=AWKSTA
T)

View(Person)
```

```
#Creating dataframe Job

Job <- Income %>% select(occjd=SS_ID,Detailed_Industry_code=ADTIND,detailed_occupation_code=ADTOCC,

major_industry_code=AMJOCC,major_occupation_code=AMJIND)

View(Job)


#Creating dataframe Pay

Pay <- Income %>% select(job_id=SS_ID,Wage_per_hour=AHRSPAY,weeks_worked_per_year=WKSWORK)

View(Pay)
```

SQLite

# creating table person

```
dbSendQuery(conn = con,"drop table if exists Person")

dbSendQuery(conn = con,"CREATE TABLE Person(Id INTEGER PRIMARY KEY AUTOINCREMENT,Age
INT,education TEXT,sex TEXT,citizenship TEXT,family_members_under_18 TEXT,previous_state
TEXT,previous_region TEXT,Hispanic_origin TEXT,employment_stat TEXT)")

dbSendQuery(conn = con,"INSERT INTO Person (Age,education,sex,
citizenship,family_members_under_18,previous_state,previous_region,Hispanic_origin,employment_stat)
SELECT AAGE,AHGA,ASEX,PRCITSHP,PARENT,GRINST,GRINREG,AREORGN,AWKSTAT FROM Income")
```

#Creating Table Job

```
dbSendQuery(conn = con,"drop table Job")

dbSendQuery(conn = con,"CREATE TABLE Job (occjd INTEGER PRIMARY KEY
AUTOINCREMENT,Detailed_Industry_code TEXT,detailed_occupation_code TEXT,major_industry_code
TEXT,major_occupation_code TEXT)")

dbSendQuery(conn = con,"INSERT INTO Job (Detailed_Industry_code,detailed_occupation_code,
major_industry_code,major_occupation_code) SELECT ADTIND,ADTOCC,AMJOCC,AMJIND FROM Income")
```

#Creating Table Pay

```
dbSendQuery (conn = con,"drop table Pay")

dbSendQuery (conn = con,"CREATE TABLE Pay (job_id INTEGER PRIMARY KEY
AUTOINCREMENT,Wage_per_hour NUM,weeks_worked_per_year NUM)")

dbSendQuery(conn = con,"INSERT INTO Pay (Wage_per_hour,weeks_worked_per_year) SELECT
AHRSPAY,WKSWORK FROM Income")
```

The three new individual table was created by extracting fields from income table and inserting it into corresponding person, pay or job table. In SQLite for each case, we first check whether the table exists or not and then drops it. This was done to avoid multiple insertion of values the code is run multiple times. Then we create empty table structure with all primary key and auto increment constraints. The data values are then selected from income table and inserted to table using insert query. The table is created with primary key with autoincrement constraints

In dplyr, creating a new table from an existing table is much more easier using select statement. The columns names to be extracted are specified in the select statement. We do not have to insert values like in SQLite as it a sub dataset formed

6. select the highest hourly wage, the number of people residing in each state (GRINST) employed in this job, the state, the job type and major industry

SQLite

```
query3 <- dbGetQuery(con,"select Pay.Wage_per_hour,Count(Person.previous_state) as
State_Count,Person.previous_state,job.major_occupation_code, job.major_industry_code from Person inner
join Job on Person.Id = job.occjd inner join Pay on Person.Id = Pay.job_id where Pay.Wage_per_hour = (select
max(Wage_per_hour) from Pay) ")
```

Dplyr

```
query3 <- Person %>% inner_join(Job, by = c("Id" = "occjd")) %>%

inner_join(Pay, by = c("Id" = "job_id")) %>%

filter(Wage_per_hour == max(Wage_per_hour)) %>%

group_by(previous_state) %>%

ungroup() %>%

summarise(Wage_per_hour,stateCount = n(),previous_state,major_occupation_code,major_industry_code)

View(query3)
```

This data interpretation needed much more complex analysis, although the task was to simply calculate the highest hourly wage. SQLite uses MAX function which takes the column to be evaluated as a parameter and returns the maximum value. In order to get highest hourly wage, number of people in each state, corresponding job id, three tables need to be joined. SQLite can use Inner join to join these table based on SS_ID, along with WHERE statement to filter out results based on Max wage per hour. This query will extract all the matching rows based on the condition. The count for each state can be applied using COUNT() method.

In dplyr, the data interpretation was much more complex, but using concepts analogous to SQLite. Here also we join all the required table using inner_join statement based on the SS_ID. The joined dataset is filter out based on the highest wage per hour. The highest wage per hour is calculated same as that in SQLite using max().Using summarise()we can extract the required columns from the filtered data set

7. Query to determine the employment of people of Hispanic origin with BSc (Bachelors degree), MSc (Masters degree), and PhD (Doctorate degree) showing the type of industry they are employed in, their average hourly wage and average number of weeks worked per year for each industry.

SQLite

```
query4 <- dbGetQuery(con,"select job.major_occupation_code as Industries, avg(Pay.Wage_per_hour) as AvgWage_per_hour, avg(Pay.weeks_worked_per_year) as AvgWeeks_worked_per_year from Person inner join Job on Person.Id = job.occjd inner join Pay on Person.Id = Pay.job_id where (Person.education='Doctorate degree(PhD EdD)' or Person.education='Bachelors degree(BA AB BS)' or Person.education='Masters degree(MA MS MEng MEd MSW MBA)' ) AND (Hispanic_origin NOT IN ('All other', 'Do not know', 'NA')) group by job.major_occupation_code")
```

Dplyr

```
query4 <- Person %>% inner_join(Job, by = c("Id" = "occjd")) %>% inner_join(Pay, by = c("Id" = "job_id")) %>%

filter(education == 'Doctorate degree(PhD EdD)' |education == 'Bachelors degree(BA AB BS)' |

education == 'Masters degree(MA MS MEng MEd MSW MBA)',

Hispanic_origin! = "All other",

Hispanic_origin != "Do not know",

Hispanic_origin != "NA") %>%

select(Hispanic_origin, education,major_occupation_code,Wage_per_hour, weeks_worked_per_year) %>%

group_by(major_occupation_code) %>%

summarise (AvgWage_per_hour = mean (Wage_per_hour),

AvgWeeks_worked_per_year = mean(weeks_worked_per_year)) %>%

rename(Industry = major_occupation_code)
```

Much like previous question, here also in Dplyr , inner_join is used to join all the required table based on the common column SS_ID .The joined dataset is filtered on the basis of education specified and also the Hispanic origin. All the irrelevant Hispanic origin values are filtered. The filtered dataset is again grouped using group_by based on industries and average wage per hour is calculated using mean ().

The data imported here need to satisfy multiple conditions. So, SQLite here uses Boolean statements such as AND, OR and NOT IN. Basically, we use inner join, where and group by to extract the data in SQLite for this case. All the necessary tables are joined together using inner join based on SS_ID and on WHERE condition satisfying the education required and Hispanic origin. The filtered data is grouped based on the industries and average wage per hour is returned which is determined by avg() in SQLite