CSC-40054

Data Analytics and Databases

Coursework Assignment Part II

SQLite vs. dplyr in R

When working with data we always needed to focus on three parameters:

- What we wanted to do with data
- How can it be programmed so that the computer can fully understand the objective
- Finally execute the program to see the corresponding output of the task

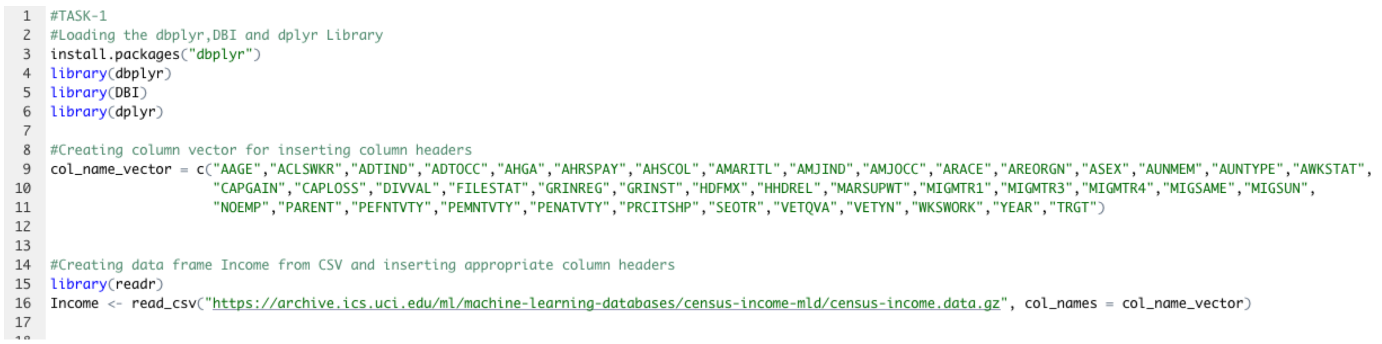
We can use the dplyr package to manifest these goals easily. Because by using dplyr we can manipulate the data efficiently. SQLite is used to manage database operations in a quite flexible manner. It is used for performing operations such as table creation, insertion, update, deletion, and selection. It also aids in transaction management. SQLite is a powerful tool but when it comes to analyzing the data beyond simple queries, it is a challenge and the queries generated will be lengthy and confusing. So, we could say that SQLite is not designed for extended data analysis, it was designed for querying and managing data operations. Whereas dplyr package in R was designed for extended data analysis and manipulation.

By comparing following tasks performed in both SQLite and dplyr we can illustrate how great is dplyr package for analyzing data than SQLite.

Critical analysis

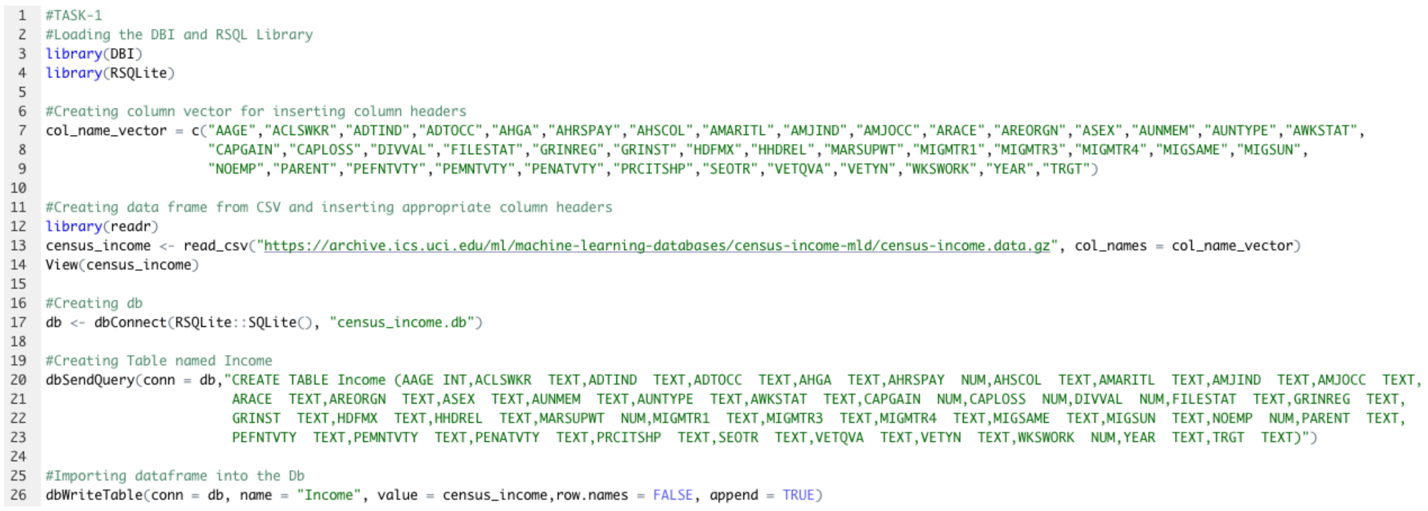
Task 1: Create an SQLite database called census_income and a table named Income with appropriate attributes provided in the appendix of the coursework document.

Dplyr method:



```r
#TASK-1
#Loading the dbplyr,DBI and dplyr Library
install.packages("dbplyr")
library(dbplyr)
library(DBI)
library(dplyr)

#Creating column vector for inserting column headers
col_name_vector = 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")


#Creating data frame Income from CSV and inserting appropriate column headers
library(readr)
Income <- read_csv("https://archive.ics.uci.edu/ml/machine-learning-databases/census-income-mld/census-income.data.gz", col_names = col_name_vector)

```

SQLite method:



```r
#TASK-1
#Loading the DBI and RSQL Library
library(DBI)
library(RSQLite)

#Creating column vector for inserting column headers
col_name_vector = 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")

#Creating data frame from CSV and inserting appropriate column headers
library(readr)
census_income <- read_csv("https://archive.ics.uci.edu/ml/machine-learning-databases/census-income-mld/census-income.data.gz", col_names = col_name_vector)
View(census_income)

#Creating db
db <- dbConnect(RSQLite::SQLite(), "census_income.db")

#Creating Table named Income
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)")

#Importing dataframe into the Db
dbWriteTable(conn = db, name = "Income", value = census_income,row.names = FALSE, append = TRUE)
```

Analysis:

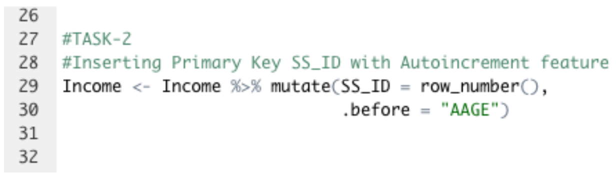
For task 1, we could see implementation by dplyr was more convenient than SQLite approach. Since the data lacks appropriate column headers, we created a column vector with appropriate header values. For SQLite we created a data frame called census_income by using read_csv() and provided data path and appended created column vector to the data.

In dplyr method, we can simply create data frame called Income directly from read_csv() and read_csv() is implemented same as SQLite approach.

But for SQLite approach, the same process is a little bit lengthy. First, we need to create and establish a database connection using dbConnect() and then we need to create table Income using CREATE TABLE query. In this query we need to provide appropriate attributes names and data types. So far, we have created an empty Income Table. For importing data to the created table Income, we need to use dbWriteTable(). This functionality helps to append the corresponding data frame to the database table.

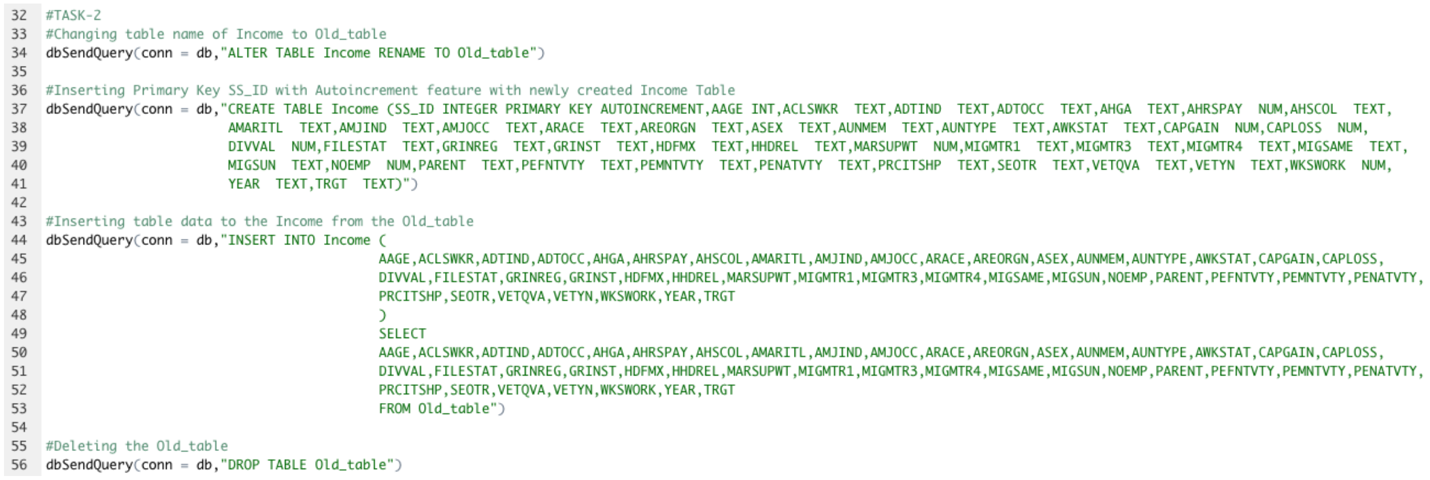
Task 2: Add a column with the name SS_ID to the Income table. Fill this column with consecutive numbers starting from 1 for the first row. Make the SS_ID attribute the primary key of the Income table.

Dplyr method:



```r

#TASK-2
#Inserting Primary Key SS_ID with Autoincrement feature
Income <- Income %>% mutate(SS_ID = row_number(),
                            .before = "AAGE")


```

SQLite method:



```
#TASK-2
#Changing table name of Income to Old_table
dbSendQuery(conn = db,"ALTER TABLE Income RENAME TO Old_table")

#Inserting Primary Key SS_ID with Autoincrement feature with newly created Income Table
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)")

#Inserting table data to the Income from the Old_table
dbSendQuery(conn = 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 Old_table")

#Deleting the Old_table
dbSendQuery(conn = db,"DROP TABLE Old_table")
```

Analysis:

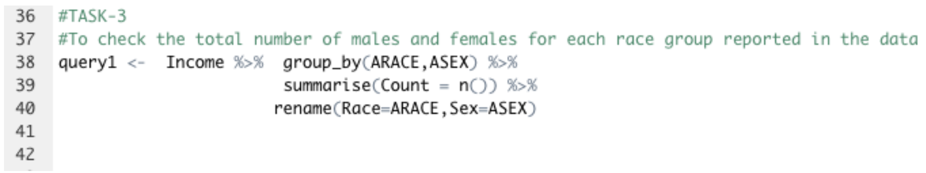
For task 2, we need to add the primary key named SS_ID to the existing table with autoincrement feature. This is done to ensure that the SQLite database can be queried relationally with a unique identifier.

In SQLite method, we cannot alter the existing table to insert a Primary Key value. We need to create a brand-new table for inserting the Primary key value. So, in this scenario, we altered the Income table name to ‘Old_table’, for that we used ALTER TABLE RENAME statement. Then we created the Income table again using the CREATE TABLE query which we used in Task 1. But this time during the declaration we additionally added the SS_ID attribute with keywords PRIMARY KEY and AUTOINCREMENT. AUTOINCREMENT helps to add unique row number each time an entry is inserted into the table. By default, it will start with 1 and increment by 1. After this, we will transfer the table data to newly created Income Table from the ‘Old_table’.It is pretty much straightforward, we used INSERT INTO (column names) SELECT column names FROM query for achieving this. Finally, we deleted Table ‘Old_table’ using DROP TABLE query.

In Dplyr method, it is a simple one-line command. It is not a cumbersome task that we did see in SQLite approach. Here we used mutate(). It is used to add columns and variables to the data frame. By default, the newly added column will append to end of the data frame. This can be changed by using .after and .before arguments in mutate(). This will help to append the column before or after the specific column. Using row_number() with mutate() will create the column of consecutive numbers. So, it helps to create identification numbers to the data frame.

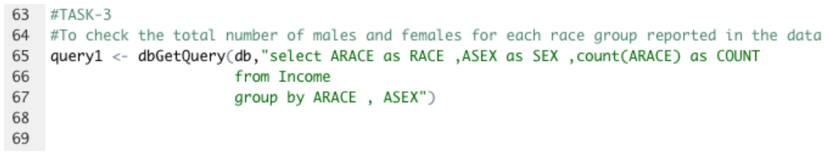
Task 3: Construct SQL queries that provide the total number of males and females for each race group reported in the data.

Dplyr method:



```
#TASK-3
#To check the total number of males and females for each race group reported in the data
query1 <-  Income %>%  group_by(ARACE,ASEX) %>%
                        summarise(Count = n()) %>%
                       rename(Race=ARACE,Sex=ASEX)
```

SQLite method:



```
#TASK-3
#To check the total number of males and females for each race group reported in the data
query1 <- dbGetQuery(db,"select ARACE as RACE ,ASEX as SEX ,count(ARACE) as COUNT
                     from Income
                     group by ARACE , ASEX")
```

Analysis:

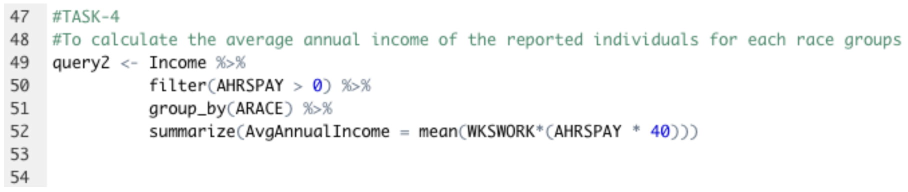
In all the above tasks, we were preparing the table or data frame for data exploration activities. In Task 3, we are going to create a query that will extract data as per the requirements of the task. Here we could observe that both SQLite and Dplyr method created light weighted queries for achieving this goal.

In SQLite, we used SELECT column names AS desired column names, COUNT (column name) FROM Table GROUP BY (column name) statement. Select query will extract the columns that are required to display, and we use AS command to rename the column with alias. Aliases are often used to make column names more readable. We use the aggregate function COUNT (column name) to return the number of rows that matches the specified criteria in the GROUP BY (column name) clause. The GROUP BY clause helps to arrange the rows of query in groups. In the task, we selected ARACE, ASEX and used Count function on ARACE attribute and grouped based on the ARACE, ASEX attributes.

One interesting feature of dplyr is its usage of pipe operator (%>%). It extracts and executes atomic blocks of code and passes the filtered data from left to right side of the pipe operator. So, it helps us to write the code in way it is easier to read and understand. In this task we piped Income data frame, then we used group_by(column names) function to group or filter the data frame based on the column names provided. Here we used ARACE, ASEX attributes, so it displays a table grouped based on ARACE, ASEX values. After that we used summarise() , it will create a new data frame and it retains the grouping variables used and we use n() inside the summarise() to get the count of current group size. Finally, we renamed the headers using rename () for the easy readability cause.

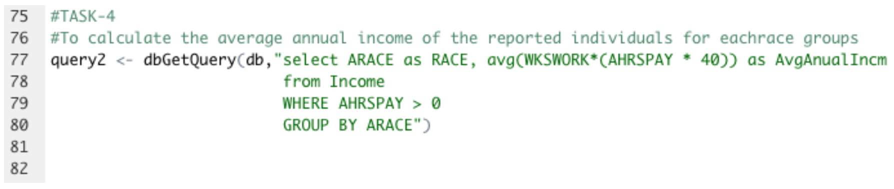
Task 4: Write queries to calculate the average annual income of the reported individuals for each race group, considering only those with non-zero wage per hour

Dplyr method:



```
#TASK-4
#To calculate the average annual income of the reported individuals for each race groups
query2 <- Income %>%
          filter(AHRSPAY > 0) %>%
          group_by(ARACE) %>%
          summarize(AvgAnnualIncome = mean(WKSWORK*(AHRSPAY * 40)))
```

SQLite method:



```
#TASK-4
#To calculate the average annual income of the reported individuals for eachrace groups
query2 <- dbGetQuery(db,"select ARACE as RACE, avg(WKSWORK*(AHRSPAY * 40)) as AvgAnualIncm
                        from Income
                        WHERE AHRSPAY > 0
                        GROUP BY ARACE")
```

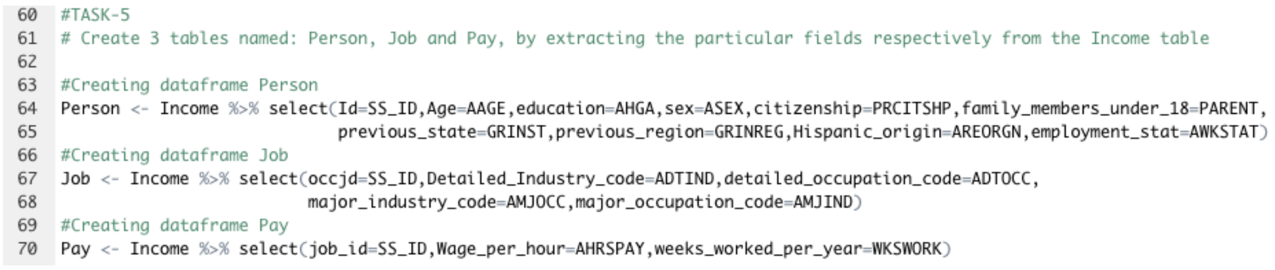
Analysis:

In SQLite approach, to achieve this task, we used a combination of SELECT, aggregate function AVG (), WHERE clause and GROUP BY clause in the query. We used SELECT to display Race and Average annual Income. Renamed column names using AS operator for readability. AVG () can be used to find the average of the given mathematical expression over column data. WHERE clause is used for satisfying the non-zero wage per hour condition. In this case we used comparison expression (AHRSPAY > 0). In the GROUP BY clause, we provided ARACE attribute since our objective was to classify the data based on Race field.

In Dplyr approach we cascaded Income data frame, then used filter (AHRSPAY > 0). This expression will filter data having AHRSPAY greater than zero, so it will consider only those with non-zero wage per hour. We will use group_by(ARACE) to group filtered data frame. So, it is now grouped by RACE. After that we used summarise() function, it will create a new data frame and it retains the grouping variables used and we used mean() inside the summarise() to calculate the average annual income. We provided the given mathematical expression to calculate the average annual income inside the mean().

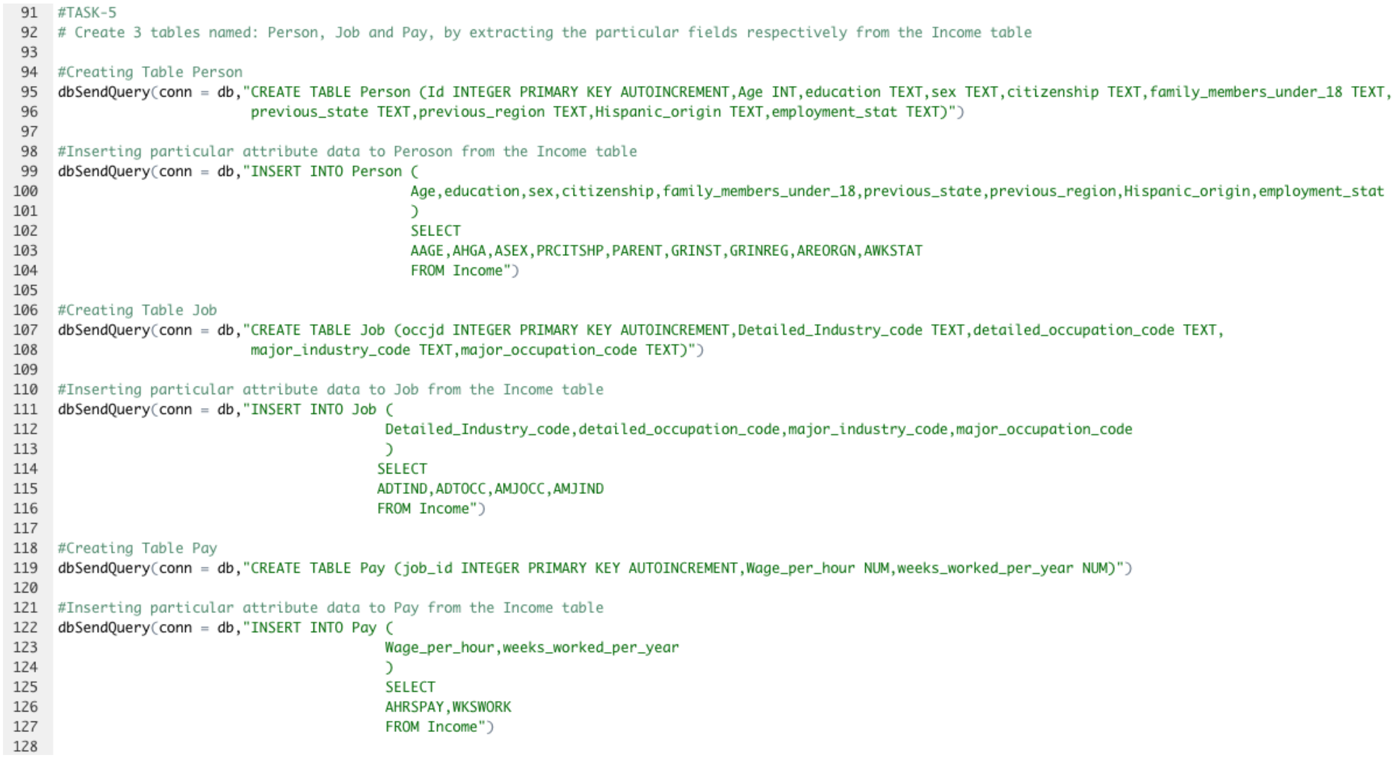
Task 5: Create 3 other tables named: Person, Job and Pay, by extracting the specific attributes from the Income table.

Dplyr method:



```
#TASK-5
# Create 3 tables named: Person, Job and Pay, by extracting the particular fields respectively from the Income table

#Creating dataframe Person
Person <- Income %>% select(Id=SS_ID,Age=AAGE,education=AHGA,sex=ASEX,citizenship=PRCITSHP,family_members_under_18=PARENT,
                            previous_state=GRINST,previous_region=GRINREG,Hispanic_origin=AREORGN,employment_stat=AWKSTAT)
#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)
#Creating dataframe Pay
Pay <- Income %>% select(job_id=SS_ID,Wage_per_hour=AHRSPAY,weeks_worked_per_year=WKSWORK)
```

SQLite method:



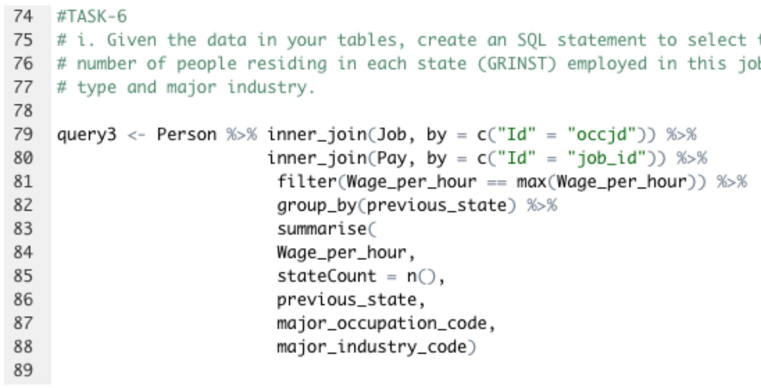
```
#TASK-5
# Create 3 tables named: Person, Job and Pay, by extracting the particular fields respectively from the Income table

#Creating Table Person
dbSendQuery(conn = db,"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)")

#Inserting particular attribute data to Peroson from the Income table
dbSendQuery(conn = db,"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 = db,"CREATE TABLE Job (occjd INTEGER PRIMARY KEY AUTOINCREMENT,Detailed_Industry_code TEXT,detailed_occupation_code TEXT,
                       major_industry_code TEXT,major_occupation_code TEXT)")

#Inserting particular attribute data to Job from the Income table
dbSendQuery(conn = db,"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 = db,"CREATE TABLE Pay (job_id INTEGER PRIMARY KEY AUTOINCREMENT,Wage_per_hour NUM,weeks_worked_per_year NUM)")

#Inserting particular attribute data to Pay from the Income table
dbSendQuery(conn = db,"INSERT INTO Pay (
                                       Wage_per_hour,weeks_worked_per_year
                                       )
                                       SELECT
                                       AHRSPAY,WKSWORK
                                       FROM Income")

```

Analysis:

In Task 5, we could see how easy it is for the dplyr package to create 3 data frames as per the course work's requirement. We are not going to dig into critical analysis since we already described how to create a table with primary key attribute with autoincrement feature for SQLite and creating its corresponding data frame using Dplyr package in Task 2. This task is a steppingstone for future tasks to perform join operations and analyze complex data

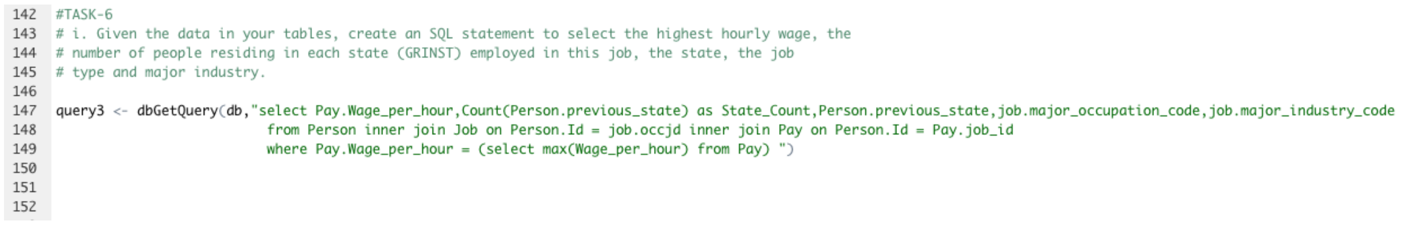
Task 6.1: Given the data in your tables, create an SQL statement to select the highest hourly wage, the number of people residing in each state (GRINST) employed in this job, the state, the job type, and major industry.

Dplyr method:



```
#TASK-6
# i. Given the data in your tables, create an SQL statement to 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.

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

```

SQLite method:



```
#TASK-6
# i. Given the data in your tables, create an SQL statement to 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.

query3 <- dbGetQuery(db,"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) ")
```

Analysis:

Complex analysis is required for carrying out this task. Since the data required for extraction is scattered across 3 different tables or data frames that we created in Task 5, we need to INNER JOIN these tables or data frames.

In SQLite method, to use the `INNER JOIN` we use the following statement

SELECT column names FROM Table1 `INNER JOIN Table2 ON`

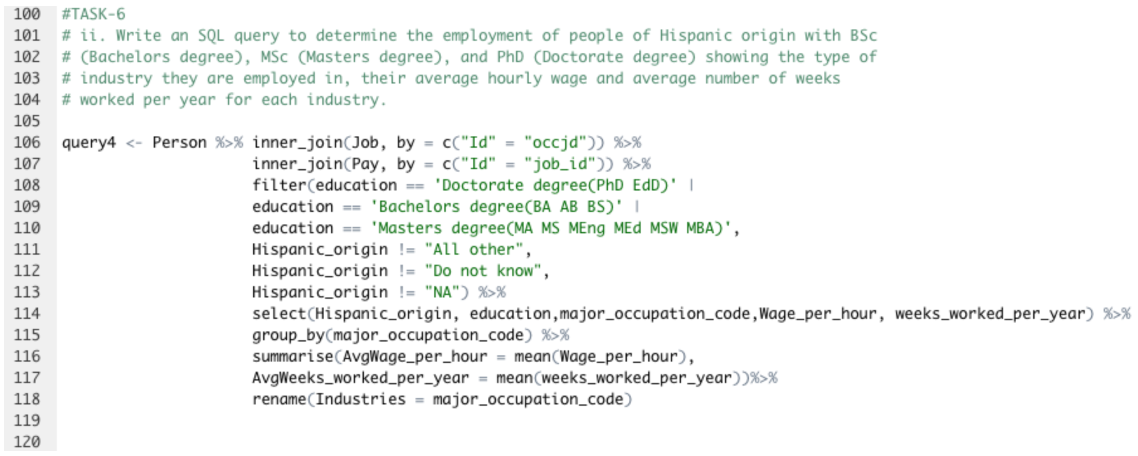
`Table1.column name = Table2.column name`

So, we inner joined tables Person, Job and Pay based on equating Primary Key column values. In this task we need to select maximum wage per hour, its corresponding state count, state, major occupation, and industry information. In the SELECT we used COUNT () aggregate function on pervious_state attribute to get the state count and we appended it with WHERE Clause condition. Here with the aid of subquery we managed to return the maximum wage per hour value using MAX ()

In Dplyr method, initially we used inner_join() on data frame Person and Job. Later we cascaded the data frame using pipe operator and inner joined data frame Pay as well. Now we have all the data frames required for extraction. Then we use filter() to get the maximum wage per hour. Used Max(column name) inside filter() and assigned it to Wage_per_hour attribute. We will use group_by(previous_state) function to group filtered data frame. So, it is now grouped by the State. After that we used summarise() function, it will create a new data frame and it retains the grouping variables used and we use n() inside the summarise() to get the count of corresponding group i.e.; previous_state attribute. Additionally, we provided Wage_per_hour, previous_state, major_occupation_code and major_industry_code attributes inside summarise() to display corresponding values associated to it with the grouping attribute.

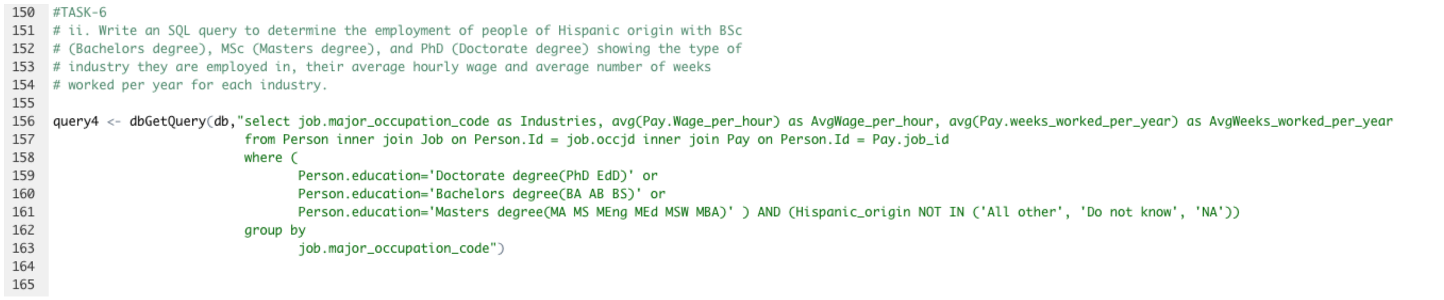
Task 6.2: Write an SQL query to determine the employment of people of Hispanic origin with BSc (Bachelor's degree), MSc (Master's degree), and PhD (Doctorate degree) showing the type of industry they are employed in, their average hourly wage and the average number of weeks worked per year for each industry.

Dplyr method:



```
#TASK-6
# ii. Write an SQL 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.

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(Industries = major_occupation_code)
```

SQLite method:



```
#TASK-6
# ii. Write an SQL 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.

query4 <- dbGetQuery(db,"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")
```

Analysis:

In SQLite, we inner joined three tables Person, Job and Pay just like we did in the previous task and used SELECT query to display attributes as major_occupation_code, Average of Wage_per_hour and Average of weeks_worked_per_year. In the SELECT, used AVG(column name) aggregate function to return the average row values that match the specified criteria in the GROUP BY (column names) clause. We use column corresponding to Industry attribute in the GROUP BY(major_occupation_code) clause. As per the SQLite syntax we need to place GROUP BY after WHERE clause. So, before GROUP BY clause, we will use WHERE clause for satisfying the degree condition and for unwanted data cleansing operations. Since we are using multiple conditions, we need to use Logical OR and AND wisely. Here Logical OR(I) operator is to consider mentioned education qualification in the task. And in Logical AND we used NOT IN operator to replace the group of arguments that do not have a specific Hispanic origin.

Whereas in Dplyr it is not complex querying like SQLite. We use the power of the pipe operator to break the query and carry out efficient execution. We inner joined corresponding data frames just like we did in the previous task. We will use filter function next to cleanse the data and filter the data frame with specific educational qualifications. Unlike NOT IN used in SQLite here we use mathematical operator != to perform the same course of action. After cascading pipe operator, now we have filtered data. We will use group_by(major_occupation_code) function to group filtered data frame. So, it is now grouped by Industries. After that we used summarise(), it will create a new data frame and it retains the grouping variables used and we used mean() inside the summarise() to get the average of Wage_per_hour and weeks_worked_per_year attributes of current group. Finally, we renamed the column name major_occupation_code to Industries using rename(). We did the same in SQLite too for readability purposes.

References

[1]K. Nishida, "Why SQL is not for Analysis, but dplyr is," *learn data science*, Aug. 05, 2016. Accessed: Jan. 05, 2022. [Online]. Available: https://blog.exploratory.io/why-sql-is-not-for-analysis-but-dplyr-is-5e180fef6aa7

[2]Y. W. Huynh, *7.3 row_number()*. Accessed: Jan. 05, 2022. [Online]. Available: https://bookdown.org/yih_huynh/Guide-to-R-Book/row-number.html

[3]"SQL Server GROUP BY," *SQL Server Tutorial*, Apr. 23, 2018. https://www.sqlservertutorial.net/sql-server-basics/sql-server-group-by/ (accessed Jan. 06, 2022).

[4]"SQL Aliases." https://www.w3schools.com/sql/sql_alias.asp (accessed Jan. 06, 2022).

[5]"Summarise each group to fewer rows — summarise." https://dplyr.tidyverse.org/reference/summarise.html (accessed Jan. 06, 2022).

[6]"Subset rows using column values — filter." https://dplyr.tidyverse.org/reference/filter.html (accessed Jan. 06, 2022).