Creating, Manipulating and Querying Databases in R Using DBI and dplyr

Steven Wun

2023-08-03

Interacting with the SQLite driver in R

DBI is the library used to interact with the SQLite driver and perform various operations in manipulating the database. The library DBI is included in the package RSQLite. The package could be installed by the following code:

```
# the code is not executed as the package has already been installed
install.packages("RSQLite")
```

We can load the required library DBI by the following code:

```
# load the required library
library(DBI)
```

Connecting to Databases

Thereafter, the first step is to connect to the database. We use the function dbConnect() to create an object, conn, to connect to the SQLite driver to manipulate the database university.db. Before that, we need to check whether the database university.db exists. If the database university.db exists, the following code chunk will remove it.

```
# delete "university.db" if already exists
if (file.exists("university.db"))
  file.remove("university.db")
```

We use dbConnect() to connect to the database:

```
conn <- dbConnect(RSQLite::SQLite(), "university.db")
```

We have created the database university.db. However, it is empty now. We can list all tables in university.db using the function dbListTable() from DBI.

```
# list all table
dbListTables(conn)
```

```
## character(0)
```

Nothing is returned because we have not created any tables in the database yet.

Creating Tables

We are going to create some tables to the database `university.db`. We will create the tables using data saved in the CSV files. We first read the CSV files into `data.frame` in R:

```
# assign the CSV files on GitHub site to objects
course.csv <- "https://raw.githubusercontent.com/stevenkhwun/ST2195/main/Course%20Files/Block%203/course
student.csv <- "https://raw.githubusercontent.com/stevenkhwun/ST2195/main/Course%20Files/Block%203/stude
grade.csv <- "https://raw.githubusercontent.com/stevenkhwun/ST2195/main/Course%20Files/Block%203/grade.

# read the CSV files into data.frame in R
course <- read.csv(course.csv, header = TRUE)
student <- read.csv(student.csv, header = TRUE)
grade <- read.csv(grade.csv, header = TRUE)
```

We then copy the data frames `student`, `grade` and `course` to tables in the database `university.db` using `DBI`'s `dbWriteTable()` function:

```
dbWriteTable(conn, "Course", course)
dbWriteTable(conn, "Student", student)
dbWriteTable(conn, "Grade", grade)
```

Now we can see there are three tables in the database:

```
# list all tables
dbListTables(conn)
```

```
## [1] "Course"  "Grade"   "Student"
```

We can also browse any table in the database using the function `dbReadTable()` from `DBI`.

```
# browse the table
dbReadTable(conn, "Student")
```

```
##   student_id           name year
## 1  201921323      Ava Smith    2
## 2  201832220    Ben Johnson    3
## 3  202003219  Charlie Jones    1
## 4  202045234     Dan Norris    1
## 5  201985603     Emily Wood    1
## 6  201933222 Freddie Harris    2
## 7  201875940   Grace Clarke    2
```

Or we can see the attributes of a table (e.g. `Student`) by the function `dbListFields()`:

```
# browse the attributes of a table
dbListFields(conn, "Student")
```

```
## [1] "student_id" "name"       "year"
```

Disconnecting From the Database

After we finish creating the database, we can close the connection using the function `dbDisconnect()` from `DBI`:

```
# disconnecting from the database
dbDisconnect(conn)
```

Manipulating Databases

The simplest way to manipulate databases is to use the `dbExecute()` function. This function executes SQL statements and returns the number of rows affected.

We have just disconnected from the database. We need to reconnect to the database by the following codes in order to perform manipulating the databases.

```
# re-connect to the database
conn <- dbConnect(RSQLite::SQLite(), "university.db")
dbListTables(conn)
```

```
## [1] "Course"  "Grade"   "Student"
```

Adding a New Table

We can add a new table by using the function `dbCreateTable()`. Alternatively, you can use `dbExecute()` to run the SQL command to create a new table.

```
# add a new table using dbCreateTable() function
dbCreateTable(conn, "Teacher", c(staff_id = "TEXT", name = "TEXT"))
```

Alternatively

```
# add a new table using dbExecute() function
dbExecute(conn,
          "CREATE TABLE Other_Staff (
          staff_id TEXT PRIMARY KEY,
          name TEXT)")
```

List of tables after adding:

```
# list all tables
dbListTables(conn)
```

```
## [1] "Course"      "Grade"       "Other_Staff" "Student"     "Teacher"
```

Deleting a Table

We can remove a table by using the function `dbRemoveTable()`:

```
# delete a table using dbRemoveTable() function
dbRemoveTable(conn, "Teacher")
```

Alternatively, we can use `dbExecute()` function:

```
# delete a table using dbExecute() function
dbExecute(conn,
          "DROP TABLE Other_Staff")
```

When we list the tables, we can now see three tables.

```
# list all tables
dbListTables(conn)
```

```
## [1] "Course"  "Grade"   "Student"
```

Inserting Tuples/Rows

Below we insert the year 1 student "Harper Taylor" with student ID 202029744 to Student by using the function `dbAppendTable()`:

```
# insert tuples/rows using dbAppendTable() function
dbAppendTable(conn, "Student", data.frame(student_id = "202029744",
                                          name = "Harper Taylor",
                                          year = 1))
```

Alternatively, we can use `dbExecute()` function:

```
# insert tuples/rows using dbExecute() function
dbExecute(conn,
          "INSERT INTO Student VALUES(201989604, 'Paul Simon', 1)")
```

When we browse the table, we can see the new row has been added.

```
# browse the table
dbReadTable(conn, "Student")
```

```
##   student_id           name year
## 1  201921323      Ava Smith    2
## 2  201832220    Ben Johnson    3
## 3  202003219  Charlie Jones    1
## 4  202045234     Dan Norris    1
## 5  201985603     Emily Wood    1
## 6  201933222 Freddie Harris    2
## 7  201875940   Grace Clarke    2
## 8  202029744  Harper Taylor    1
## 9  201989604     Paul Simon    1
```

Updating Tuples/Rows

Below, we update the student ID of student Harper Taylor to 201929744 by `dbExecute()`. There is no specific function for updating a row in `DBI`.

```
# update tuples/rows
dbExecute(conn, "UPDATE Student SET student_id = '201929744'
                                 WHERE name = 'Harper Taylor'")
```

When we browse the table, we can see the new row has been added.

```
# browse the table
dbReadTable(conn, "Student")
```

```
##   student_id           name year
## 1  201921323      Ava Smith    2
## 2  201832220    Ben Johnson    3
## 3  202003219  Charlie Jones    1
## 4  202045234     Dan Norris    1
## 5  201985603     Emily Wood    1
## 6  201933222 Freddie Harris    2
## 7  201875940   Grace Clarke    2
## 8  201929744  Harper Taylor    1
## 9  201989604     Paul Simon    1
```

Deleting Tuples/Rows

Below, we delete the record for the student Harper Taylor and Never Forget from table `Student` using `dbExecute()`. There is no specific function for deleting a row in `DBI`.

```
dbExecute(conn,
          "DELETE FROM Student WHERE name = 'Harper Taylor' OR name = 'Paul Simon'")
```

```
## [1] 2
```

When we browse the table, we can see the rows have been removed.

```
# browse the table
dbReadTable(conn, "Student")
```

```
##   student_id           name year
## 1  201921323      Ava Smith    2
## 2  201832220    Ben Johnson    3
## 3  202003219  Charlie Jones    1
## 4  202045234     Dan Norris    1
## 5  201985603     Emily Wood    1
## 6  201933222 Freddie Harris    2
## 7  201875940   Grace Clarke    2
```

Basic SQL/SQLite Syntax and Queries

Once we formulate the query into an SQL SELECT statement, we can get the query result in R using the function dbGetQuery() or dbSendQuery() from DBI. The following examples show how we can run queries from within R.

Conditions: Basics

Getting Grades of a Course With course_id "ST101"

```
# getting grades of a course using dbGetQuery()
q1 <- dbGetQuery(conn, "SELECT final_mark
                        FROM Grade
                        WHERE course_id = 'ST101'")
q1
```

```
##   final_mark
## 1         78
## 2         60
## 3         47
```

The second argument in dbGetQuery() is the SQL query statement sent using DB Browser for SQLite.

Note that the dbGetQuery() returns a data.frame.

```
# data type of q1
class(q1)
```

```
## [1] "data.frame"
```

Alternatively, we can use dbSendQuery() and dbFetch():

```
# getting grades of a course using dbSendQuery()
q1 <- dbSendQuery(conn, "SELECT final_mark
                         FROM Grade
                         WHERE course_id = 'ST101'")
q1
```

```
## <SQLiteResult>
##   SQL  SELECT final_mark
##                          FROM Grade
##                          WHERE course_id = 'ST101'
##   ROWS Fetched: 0 [incomplete]
##        Changed: 0
```

Note dbSendQuery() only sends and executes the SQL query to the database engine. It does not extract any records. When we run dbFetch(), the executed query result will then be fetched.

```
# fetch the result
dbFetch(q1)
```

```
##   final_mark
## 1         78
## 2         60
## 3         47
```

Conditions: The operator *

We would like to return all attributes of the students who are in ST101:

```
# getting all the attributes of the students who are in sT101
dbGetQuery(conn, "SELECT *
                  FROM Grade
                  WHERE course_id = 'ST101'")
```

```
## Warning: Closing open result set, pending rows
```

```
##   course_id student_id final_mark
## 1     ST101  201921323         78
## 2     ST101  201985603         60
## 3     ST101  202003219         47
```

Conditions: `iif()` function

In SQLite, `iif()` is a conditional function that returns the second or third argument based on the evaluation of the first argument.

It's logically equivalent to `CASE WHEN X THEN Y ELSE Z END`.

`iif()` is an abbreviation for `Immediate IF`.

```
# using iif()
dbGetQuery(conn, "SELECT *,
                  iif( final_mark > 60, 'A', 'F') AS LetterGrade
                  FROM Grade")
```

```
##   course_id student_id final_mark LetterGrade
## 1     ST101  201921323         78           A
## 2     ST101  201985603         60           F
## 3     ST101  202003219         47           F
## 4     ST115  201921323         92           A
## 5     ST115  202003219         67           A
## 6     ST115  201933222         88           A
## 7     ST207  201933222         73           A
## 8     ST207  201875940         60           F
```

Refer to this link for more information.

Conditions: `CASE` statement

The SQLite `CASE` expression evaluates a list of conditions and returns an expression based on the result of the evaluation.

The `CASE` expression is similar to the `IF-THEN-ELSE` statement in other programming languages.

```
# using iif()
dbGetQuery(conn, "SELECT *,
                  CASE
                    WHEN final_mark >= 80 THEN 'A'
                    WHEN final_mark < 80 AND final_mark >= 60 THEN 'B'
                    ELSE 'C'
                    END LetterGrade
                  FROM Grade")
```

```
##   course_id student_id final_mark LetterGrade
## 1     ST101  201921323         78           B
## 2     ST101  201985603         60           B
## 3     ST101  202003219         47           C
## 4     ST115  201921323         92           A
## 5     ST115  202003219         67           B
## 6     ST115  201933222         88           A
## 7     ST207  201933222         73           B
## 8     ST207  201875940         60           B
```

Refer to this link for more information.

Several Tables

Getting Names of Students in Alphabetical Order

Note the student name information is in the `Student` table whereas the information about which course the student took is in `Grade`. In order to perform the query we need to combine information from the `Student` and `Grade` tables.

```
# getting names of students in alphabetical order
dbGetQuery(conn, "SELECT name
                  FROM Grade, Student
                  WHERE course_id = 'ST101' AND student.student_id = Grade.student_id
                  ORDER BY name")
```

```
##            name
## 1     Ava Smith
## 2 Charlie Jones
## 3    Emily Wood
```

Note that we don't need to specify the table name for the attributes `name` and `course_id` because attribute `name` is only in the table `Student` and the attribute `course_id` is only in the table `Grade`.

Or we can do it with `NATURAL JOIN`:

```
# using NATURAL JOIN
dbGetQuery(conn, "SELECT Student.name
                  FROM Student NATURAL JOIN Grade
                  WHERE course_id = 'ST101'
                  ORDER BY Student.name")
```

```
##            name
## 1     Ava Smith
## 2 Charlie Jones
## 3    Emily Wood
```

Multiple Conditions

Getting Courses Taken by Students Ava Smith or Freddie Harris

```
# getting courses taken by students
dbGetQuery(conn, "SELECT Course.name
                  FROM Student, Grade, Course
                  WHERE (Student.name = 'Ava Smith' OR Student.name = 'Freddie Harris')
                        AND Student.student_id = Grade.student_id
                        AND Course.course_id = Grade.course_id")
```

```
##                            name
## 1  programming for data science
## 2 Managing and Visualising Data
## 3 Managing and Visualising Data
## 4                     Databases
```

There are a few duplicate rows in the output, which we can remove using `DISTINCT`:

```
# getting courses taken by students (duplicates removed)
dbGetQuery(conn, "SELECT DISTINCT Course.name
                  FROM Student, Grade, Course
                  WHERE (Student.name = 'Ava Smith' OR Student.name = 'Freddie Harris')
                        AND Student.student_id = Grade.student_id
                        AND Course.course_id = Grade.course_id")
```

```
##                            name
## 1  programming for data science
## 2 Managing and Visualising Data
## 3                     Databases
```

Or we can do it using `JOIN`:

```
# using JOIN
dbGetQuery(conn, "SELECT DISTINCT Course.name
                  FROM (student NATURAL JOIN Grade) S JOIN Course on Course.course_id = S.course_id
                  WHERE S.name = 'Ava Smith' OR S.name = 'Freddie Harris'")
```

```
##                            name
## 1  programming for data science
## 2 Managing and Visualising Data
## 3                     Databases
```

Aggregation and `GROUP BY`

We would like to calculate the average mark for each course according to the value of `course_id`:

```
# calculate the average mark
dbGetQuery(conn, "SELECT course_id, AVG(final_mark)
                  FROM Grade
                  GROUP BY course_id")
```

```
##   course_id AVG(final_mark)
## 1     ST101        61.66667
## 2     ST115        82.33333
## 3     ST207        66.50000
```

The attribute name for the average mark looks different from other attributes. We can rename it using the `AS` clause:

```
# calculate the average mark
dbGetQuery(conn, "SELECT course_id, AVG(final_mark) AS avg_mark
                  FROM Grade
                  GROUP BY course_id")
```

```
##   course_id avg_mark
## 1     ST101 61.66667
## 2     ST115 82.33333
## 3     ST207 66.50000
```

SQL Joins

A `JOIN` clause combines rows from two or more tables based on related column(s) between them. The SQL language offers many different types of joins.

- `INNER JOIN`: Select rows that have matching values in both tables based on the given columns
- `NATURAL JOIN`: Similar to `INNER JOIN` except that there is no need to specify which columns are used for matching values
- `OUTER JOIN`: Unlike `INNER JOIN`, unmatched rows in one or both tables can be returned. There are `LEFT`, `RIGHT` and `FULL OUTER JOIN`. `SQLite` only supports `LEFT OUTER JOIN`. For `LEFT OUTER JOIN`, all the records from the left table are included in the result.
- `CROSS JOIN`: Return the Cartesian product of the two joined tables, by matching all the values from the left table with all the values from the right table.

`INNER JOIN`

Below we get the records about students who took the course with course ID ST101 and sort the student names in alphabetical order. Without using Joins, we can implement the following codes:

```
# without using join
dbGetQuery(conn, "SELECT *
                  FROM Student, Grade
                  WHERE Grade.course_id = 'ST101' AND Student.student_id = Grade.student_id
                  ORDER BY Student.name")
```

```
##   student_id          name year course_id student_id final_mark
## 1  201921323     Ava Smith    2     ST101  201921323         78
## 2  202003219 Charlie Jones    1     ST101  202003219         47
## 3  201985603    Emily Wood    1     ST101  201985603         60
```

We can rewrite the above query using `INNER JOIN`:

```
# using INNER JOIN
dbGetQuery(conn, "SELECT *
                  FROM Student JOIN Grade ON Student.student_id = Grade.student_id
                  WHERE course_id = 'ST101'
                  ORDER BY Student.name")
```

```
##   student_id          name year course_id student_id final_mark
## 1  201921323     Ava Smith    2     ST101  201921323         78
## 2  202003219 Charlie Jones    1     ST101  202003219         47
## 3  201985603    Emily Wood    1     ST101  201985603         60
```

Note that:

- We write `JOIN` instead of `INNER JOIN` as by default `INNER JOIN` is used when you do not specify the join type.
- The `ON` keyword specifies on what condition you want to join the tables.
- If you look at the result, `student_id` appears twice - this is because all the columns from both tables are returned.
- By using the `JOIN` clause, we separate the logic of combining the tables (`Student.student_id = Grade.student_id`) and the other condition (`course_id = 'ST101'`), which makes the SQL query more readable.

Instead of joining using `ON`, we can use `USING` with `JOIN` if the columns that we are joining have the same name. For example:

```
# 'JOIN' clause using 'USING'
dbGetQuery(conn, "SELECT *
                  FROM Student JOIN Grade USING (student_id)
                  WHERE course_id = 'ST101'
                  ORDER BY Student.name")
```

```
##   student_id          name year course_id final_mark
## 1  201921323     Ava Smith    2     ST101         78
## 2  202003219 Charlie Jones    1     ST101         47
## 3  201985603    Emily Wood    1     ST101         60
```

Note that:

- The `USING` keyword specifies which column is used to select rows that have matching values in both tables.
- If you look at the result, `student_id` appears only once now.

NATURAL JOIN

```
# using 'NATURAL JOIN'
dbGetQuery(conn, "SELECT *
                  FROM Student NATURAL JOIN Grade
                  WHERE course_id = 'ST101'
                  ORDER BY Student.name")
```

```
##   student_id          name year course_id final_mark
## 1  201921323     Ava Smith    2     ST101         78
## 2  202003219 Charlie Jones    1     ST101         47
## 3  201985603    Emily Wood    1     ST101         60
```

Note that:

- We do not specify how to join the two tables. The join condition is automatically identified.
- If you look at the result, `student_id` appears only once.

LEFT JOIN

When we run the following `SQL` commands to use `INNER JOIN` to combine the tables `Student` and `Grade`

```
# use 'INNER JOIN' to join tables
dbGetQuery(conn, "SELECT *
                  FROM Student INNER JOIN Grade USING (student_id)
                  ORDER BY Student.name")
```

```
##   student_id           name year course_id final_mark
## 1  201921323      Ava Smith    2     ST101         78
## 2  201921323      Ava Smith    2     ST115         92
## 3  202003219  Charlie Jones    1     ST101         47
## 4  202003219  Charlie Jones    1     ST115         67
## 5  201985603     Emily Wood    1     ST101         60
## 6  201933222 Freddie Harris    2     ST115         88
## 7  201933222 Freddie Harris    2     ST207         73
## 8  201875940   Grace Clarke    2     ST207         60
```

the record of students Ben Johnson and Dan Norris are not shown, because there are not corresponding records for these two students in the table `Grade`. If we instead use `LEFT OUTER JOIN` to combine the tables `Student` and `Grade`:

```
# use 'LEFT OUTER JOIN' to join tables
dbGetQuery(conn, "SELECT *
                  FROM Student LEFT JOIN Grade USING (student_id)
                  ORDER BY Student.name")
```

```
##    student_id           name year course_id final_mark
## 1   201921323      Ava Smith    2     ST101         78
## 2   201921323      Ava Smith    2     ST115         92
## 3   201832220    Ben Johnson    3      <NA>         NA
## 4   202003219  Charlie Jones    1     ST101         47
## 5   202003219  Charlie Jones    1     ST115         67
## 6   202045234     Dan Norris    1      <NA>         NA
## 7   201985603     Emily Wood    1     ST101         60
## 8   201933222 Freddie Harris    2     ST115         88
## 9   201933222 Freddie Harris    2     ST207         73
## 10  201875940   Grace Clarke    2     ST207         60
```

we get a result where:

- All the students from the left table `Student` are included (including Ben Johnson and Dan Norris)
- The students with no corresponding record in the right table `Grade`, have `NULL` value in attributes `course_id` and `final_mark` from `Grade`.

CROSS JOIN

When we use `CROSS JOIN`:

```
# use 'CROSS JOIN'
dbGetQuery(conn, "SELECT *
                  FROM Student CROSS JOIN Grade
                  ORDER BY Student.name")
```

```
##    student_id           name year course_id student_id final_mark
## 1   201921323      Ava Smith    2     ST101  201921323         78
## 2   201921323      Ava Smith    2     ST101  201985603         60
## 3   201921323      Ava Smith    2     ST101  202003219         47
## 4   201921323      Ava Smith    2     ST115  201921323         92
## 5   201921323      Ava Smith    2     ST115  202003219         67
## 6   201921323      Ava Smith    2     ST115  201933222         88
## 7   201921323      Ava Smith    2     ST207  201933222         73
## 8   201921323      Ava Smith    2     ST207  201875940         60
## 9   201832220    Ben Johnson    3     ST101  201921323         78
## 10  201832220    Ben Johnson    3     ST101  201985603         60
## 11  201832220    Ben Johnson    3     ST101  202003219         47
## 12  201832220    Ben Johnson    3     ST115  201921323         92
## 13  201832220    Ben Johnson    3     ST115  202003219         67
## 14  201832220    Ben Johnson    3     ST115  201933222         88
## 15  201832220    Ben Johnson    3     ST207  201933222         73
## 16  201832220    Ben Johnson    3     ST207  201875940         60
## 17  202003219  Charlie Jones    1     ST101  201921323         78
## 18  202003219  Charlie Jones    1     ST101  201985603         60
## 19  202003219  Charlie Jones    1     ST101  202003219         47
## 20  202003219  Charlie Jones    1     ST115  201921323         92
## 21  202003219  Charlie Jones    1     ST115  202003219         67
## 22  202003219  Charlie Jones    1     ST115  201933222         88
## 23  202003219  Charlie Jones    1     ST207  201933222         73
## 24  202003219  Charlie Jones    1     ST207  201875940         60
## 25  202045234     Dan Norris    1     ST101  201921323         78
## 26  202045234     Dan Norris    1     ST101  201985603         60
## 27  202045234     Dan Norris    1     ST101  202003219         47
## 28  202045234     Dan Norris    1     ST115  201921323         92
## 29  202045234     Dan Norris    1     ST115  202003219         67
## 30  202045234     Dan Norris    1     ST115  201933222         88
## 31  202045234     Dan Norris    1     ST207  201933222         73
## 32  202045234     Dan Norris    1     ST207  201875940         60
## 33  201985603     Emily Wood    1     ST101  201921323         78
## 34  201985603     Emily Wood    1     ST101  201985603         60
## 35  201985603     Emily Wood    1     ST101  202003219         47
## 36  201985603     Emily Wood    1     ST115  201921323         92
## 37  201985603     Emily Wood    1     ST115  202003219         67
## 38  201985603     Emily Wood    1     ST115  201933222         88
## 39  201985603     Emily Wood    1     ST207  201933222         73
## 40  201985603     Emily Wood    1     ST207  201875940         60
## 41  201933222 Freddie Harris    2     ST101  201921323         78
## 42  201933222 Freddie Harris    2     ST101  201985603         60
## 43  201933222 Freddie Harris    2     ST101  202003219         47
## 44  201933222 Freddie Harris    2     ST115  201921323         92
```

```
## 45  201933222 Freddie Harris    2     ST115  202003219         67
## 46  201933222 Freddie Harris    2     ST115  201933222         88
## 47  201933222 Freddie Harris    2     ST207  201933222         73
## 48  201933222 Freddie Harris    2     ST207  201875940         60
## 49  201875940   Grace Clarke    2     ST101  201921323         78
## 50  201875940   Grace Clarke    2     ST101  201985603         60
## 51  201875940   Grace Clarke    2     ST101  202003219         47
## 52  201875940   Grace Clarke    2     ST115  201921323         92
## 53  201875940   Grace Clarke    2     ST115  202003219         67
## 54  201875940   Grace Clarke    2     ST115  201933222         88
## 55  201875940   Grace Clarke    2     ST207  201933222         73
## 56  201875940   Grace Clarke    2     ST207  201875940         60
```

we get 56 rows, which is number of rows in `Student` (7) times number of rows in `Grade` (8).

Querying Databases in `R` Using `dplyr`

Introduction to dplyr

`dplyr` is an `R` package for data manipulation. It provides a set of functions, named as verbs, that can be used to carry out a wide range of data manipulation operations:

- The `mutate()` function adds new variables that are transformations of existing variables.
- The `select()` function picks variables based on their names.
- The `filter()` function picks cases based on their values.
- The `summarize()` function reduces multiple values down to a single summary.
- The `arrange()` function changes the ordering of the rows.

`dplyr` verbs operate on data frames, but they also, almost seamlessly apply to database tables! In particular `dplyr` allows you to use database tables as if they are data frames, by internally converting `dplyr` code into `SQL` commands using `dbplyr`. The following table compares the syntax used in `SQL` with `dplyr` syntax:

Action	`SQL`	`dplyr`
select a column	SELECT	select
select a row	WHERE	filter
sort	ORDER BY	arrange
group	GROUP BY	group_by
aggregation	aggregation functions (e.g. `AVG()`) in SELECT	summarize

See the SQL Transalation article in `dbplyr`'s pages for more information.

Pipe Operator

All of the `dplyr` functions take a data frame (or a `tibble`) as the first argument. In this way, the `%>%` operator from the `magrittr` `R` package can be used, so that user does not need to save intermediate objects or nest verbs. For example, the statement `x %>% f(y)` is equivalent to `f(x, y)`, and the result from one step is "piped" into the next step. You can think of the pipe operator as "then."

Querying Databases Using `dplyr`

Connecting to the Database

We should have connected to the database `university.db`. If not, connect to the database by the method as mentioned above. In the meantime, let us see all the three tables `student`, `course` and `grade` in the database `university.db`.

```
# list all the tables in university.db
dbListTables(conn)
```

```
## [1] "Course"  "Grade"   "Student"
```

Creating a Reference to Table

```
# creating a reference to table
library(dplyr)
student_db <- tbl(conn, "Student")
grade_db <- tbl(conn, "Grade")
course_db <- tbl(conn, "Course")
```

By creating references to the tables as done above, we can treat `student_db`, `grade_db`, and `course_db` as data frames, and use `dplyr` functionality to query the database.

Getting Grades of the Course using `filter()`

```
# getting grades of the course using filter()
q1 <- grade_db %>% filter(course_id == "ST101")
q1
```

```
## # Source:   lazy query [?? x 3]
## # Database: sqlite 3.41.2 [D:\Documents\GitHub\ST2195\university.db]
##   course_id student_id final_mark
##   <chr>          <int>      <int>
## 1 ST101      201921323         78
## 2 ST101      201985603         60
## 3 ST101      202003219         47
```

The function `filter()` selects the rows in the `grade_db` which satisfy the condition `course_id == "ST101"`.

We can use the function `show_query()` to show the `SQL` query that `dbplyr` produced, when we run the code above:

```
# show the SQL query produced
show_query(q1)
```

```
## <SQL>
## SELECT *
## FROM `Grade`
## WHERE (`course_id` = 'ST101')
```

Getting Names of Stuents in Alphabetic Order

If we want to work on more than one table in `dplyr`, we can use join or set operations. In this example we show how to use `inner_join()`. `arrange()` is then used to order the query result.

```
# using 'inner_join()' and 'arrange()'
q2 <- inner_join(student_db, grade_db) %>%
  filter(course_id == "ST101") %>%
  select(name) %>%
  arrange(name)
q2
```

```
## # Source:     lazy query [?? x 1]
## # Database:   sqlite 3.41.2 [D:\Documents\GitHub\ST2195\university.db]
## # Ordered by: name
##   name
##   <chr>
## 1 Ava Smith
## 2 Charlie Jones
## 3 Emily Wood
```

The corresponding `SQL` query is:

```
# show the SQL query produced
show_query(q2)
```

```
## <SQL>
## SELECT `name`
## FROM (SELECT `LHS`.`student_id` AS `student_id`, `name`, `year`, `course_id`, `final_mark`
## FROM `Student` AS `LHS`
## INNER JOIN `Grade` AS `RHS`
## ON (`LHS`.`student_id` = `RHS`.`student_id`)
## )
## WHERE (`course_id` = 'ST101')
## ORDER BY `name`
```

Getting Courses Taken by Ava Smith or Freddie Harris

Here we use `inner_join` to specify which attribute should be used to join by, in this case `course_id`. As both `student_db` and the `course_db` have the attribute `name`, we use the argument `suffix` to rename the attribute `name` to `name.student` and `name.course`, correspondingly. In this way we eliminate ambiguity.

```
# use the argument 'suffix'
q3 <- inner_join(student_db, grade_db, by = "student_id") %>%
  inner_join(course_db, by = "course_id", suffix = c(".student", ".course")) %>%
  filter(name.student == 'Ava Smith' | name.student == 'Freddie Harris') %>%
  select(name.course) %>%
  distinct()
q3
```

```
## # Source:   lazy query [?? x 1]
```

```
## # Database: sqlite 3.41.2 [D:\Documents\GitHub\ST2195\university.db]
##   name.course
##   <chr>
## 1 programming for data science
## 2 Managing and Visualising Data
## 3 Databases
```

The corresponding SQL query is:

```
# show the SQL query produced
show_query(q3)
```

```
## <SQL>
## SELECT DISTINCT `name.course`
## FROM (SELECT `student_id`, `LHS`.`name` AS `name.student`, `year`, `LHS`.`course_id` AS `course_id`,
## FROM (SELECT `LHS`.`student_id` AS `student_id`, `name`, `year`, `course_id`, `final_mark`
## FROM `Student` AS `LHS`
## INNER JOIN `Grade` AS `RHS`
## ON (`LHS`.`student_id` = `RHS`.`student_id`)
## ) AS `LHS`
## INNER JOIN `Course` AS `RHS`
## ON (`LHS`.`course_id` = `RHS`.`course_id`)
## )
## WHERE (`name.student` = 'Ava Smith' OR `name.student` = 'Freddie Harris')
```

Note the computer-generated SQL code that dplyr created internally is more complicated than the SQL code we wrote for the same query.

Calculating Average Mark for Each Course

The combination of the verbs group_by() and summarize() are used to calculate the average final_mark for each course_id.

```
# use of 'group_by()' and 'summarize()'
q4 <- grade_db %>%
  group_by(course_id) %>%
  summarize(avg_mark = mean(final_mark, na.rm = TRUE))
q4
```

```
## # Source:   lazy query [?? x 2]
## # Database: sqlite 3.41.2 [D:\Documents\GitHub\ST2195\university.db]
##   course_id avg_mark
##   <chr>        <dbl>
## 1 ST101         61.7
## 2 ST115         82.3
## 3 ST207         66.5
```

The corresponding SQL query is:

```
# show the SQL query produced
show_query(q4)
```

```
## <SQL>
## SELECT `course_id`, AVG(`final_mark`) AS `avg_mark`
## FROM `Grade`
## GROUP BY `course_id`
```

Disconnecting From the Database

After we finish manipulating the database, we can close the connection using the function `dbDisconnect()` from `DBI`:

```
# disconnect from the database
dbDisconnect(conn)
```

Useful Links and Resources

- `DBI`
  - Using **DBI**
  - **DBI** reference manual
  - **RSQLite** vignettes
- `SQL` Joins
  - SQLite join to learn more about how to join tables via SQLite
  - SQLite select to learn more about how to query via SQLite
  - SQL from Wikipedia
- `dplyr`
  - Using `dplyr` with databases: A guide on how to use `dplyr` with database from RStudio
  - `dplyr` vignettes: A introduction to `dplyr`
  - `dbplyr` SQL Translation