MTH 3270 Notes 10

15 Database Querying Using SQL (15)

15.1 Introduction

- Computers have two ways of storing data:
  - ***Memory*** (or ***RAM***): Only **a few Gb** of data can be stored here, but it can be quickly accessed. Actively running programs (like R) store data they're currently working on (like objects in the R Workspace) here.
  - ***Hard Disk***: **Hundreds** or **thousands** of **Gb** of data can be stored here, but accessing it is much slower. Saved files (that aren't being worked on by an actively running program) are stored here.
- Relatively small data sets can be stored in in memory. But not larger ones. For example, a data set with **100 columns** and **1 million rows** takes up about **three-quarters** of a **Gb** of memory:

```
n <- 100 * 1000000

# Generate a matrix of random numbers uniformly distributed between 0 and 1:
x <- matrix(runif(n), nrow = 1000000, ncol = 100)

print(object.size(x), units = "Mb")

## 762.9 Mb
```

  An R *data frame* with **10 million rows** would hog memory, resulting in slowed computations, and one with **100 million rows** wouldn't fit in memory.

- For **large** data sets, instead of trying to read *all* the data into R, it's preferable to store it on a **hard disk** and be allowed to *access* only *portions* of the data.
- Two commonly used forms of data storage:
  - A ***flat file*** is a data file (**.txt**, **.csv**, **.xlsx**, etc.) stored in a ***table*** format, i.e. containing just **rows** and **columns** (possibly with a **header** row).
  - A ***relational database*** (or just ***database***) is a collection of *linkable* ***tables*** (each of which contains just **rows** and **columns**).
- ***SQL*** ("structured query language") is a programming language for data stored as a ***database***.
- There are a few different **implementations** of **SQL**, also called ***database management systems*** or ***DBMSs***:
  - Oracle
  - Microsoft SQL Server
  - SQLite
  - MySQL
  - PostgreSQL

  Some of them use different **dialects** of the **SQL** language.

- **SQL** uses a ***client-server*** model – data are stored in a **database** on the **hard disk** of a ***server*** computer somewhere, and there's a ***server program*** running on that computer. You can connect to that server over the internet from a ***client*** computer using one of several ***client programs***.

  (In some cases, the **server** and **client** computers might be the same, in which case the internet isn't needed.)

- For example, recall (Class Notes 3) that the `"nycflights13"` *package* contained a `flights` data frame with 336,776 rows, each one a flight that departed New York in 2013.

  But the *full* **flights** data set is **much larger**, with more than **169 million** flights going back to 1987 – each comprising a different row in the table. The data would occupy nearly **20 Gb** as a **.csv** file on a computer.

  Instead, the data have been written (by the textbook authors) to a **table** (named **flights**) in a **database** (named **airlines**) on a **server**. We can use **SQL** to access *only* the rows that interest us.

- A ***query*** is a **request** for **data**. **Queries** in **SQL** are ***statements*** that start with the `SELECT` ***keyword*** and consist of several ***clauses*** involving *other* **keywords**, which have to be written **in this order**:

```
SELECT       # Select columns to extract from a table.  Similar to select()
             # in "dpyr".
FROM         # Specify the table where data are stored.
JOIN         # Combine together two tables based on a key.  Analogous to
             # inner_join() and left_join() from "dplyr".
WHERE        # Filter rows according to some criteria. Analogous to filter()
             # in "dplyr".
GROUP BY     # Group together rows of a table according to values of a
             # categorical variable.  Analogous to group_by() in "dplyr".
HAVING       # Like a WHERE clause that operates on the result set--not the
             # rows themselves. Analogous to applying a second filter()
             # command in "dplyr", after the rows have already been aggregated.
ORDER BY     # Specifies a condition for ordering the rows. Analogous to
             # arrange() in "dplyr".
LIMIT        # Restrict the number of rows in the output. Similar to head()
             # and slice() in R.
```

- **SQL queries** can be executed from **within R** by:

  1. **Connecting** to the **database**.
  2. **Sending** the **query**.
  3. **Disconnecting** from the **database**.

  These tasks are carried out using the following functions from the `"DBI"` package.

```
dbConnect()      # Connect to a database stored on a server.
dbGetQuery()     # Execute a query statement on a connected database.
dbDisconnect()   # Disconnect from the database.
```

  A list of the **tables** in a **database** can be obtained using the following function (also from `"DBI"`).

```
dbListTables()   # Returns a list of the tables in a connected database.
```

15.2 Connecting to a Server Using `dbConnect()` and Querying Using `dbGetQuery()`

- The first step to conducting a **database query** is to **connect** to the **server** on which the **database** is stored.

  In `dbConnect()`, you'll specify an **SQL implementation** (RSQLite, MySQL, PostgreSQL, etc.) via the `drv` (*driver*) argument, and you'll specify **authentication information** via `dbname`, `host`, `user`, `password`, and sometimes `port`.

  The **airlines database** was set up using **MySQL** (by the textbook authors) on an **AWS server**. Connecting to a **MySQL database** requires the `MySQL()` function from the `"RMySQL"` package.

```
MySQL()      #  Allows you to connect to a MySQL database.
```

Below, we open a **connection** to the **airlines database**.

```r
# For dbConnect(), dbGetQuery(), and dbListTables():
library(DBI)

# For MySQL():
library(RMySQL)

# Open a connection to the airlines database:
db_con <- dbConnect(drv = MySQL(),
                    dbname = "airlines",
                    host = "mdsr.cdc7tgkkqd0n.us-east-1.rds.amazonaws.com",
                    user = "mdsr_public",
                    password = "ImhsmflMDSwR")

class(db_con)
```

- Once we're connected, we can view a list of the **tables** in the **database** using the following:

```r
# This will return a list of the tables in the database:
dbListTables(db_con)
```

- **SQL queries** (and other **SQL statements**) can be carried out using `dbGetQuery()`, which takes arguments `conn`, an open connection to a server, and `statement`, the **SQL query** (or other **SQL statement**) to perform on the **database** or specific **table**.
- For example, `DESCRIBE` will return a *description* of the **variables** ("fields") in a **table**, similar to `str()` in R. Descriptions of the `flights` and `airports` **tables** (from the **flights database**) are obtained via:

```r
# Described the contents of the airports table:
dbGetQuery(conn = db_con,
           statement = "DESCRIBE airports;")

# Described the contents of the flights table:
dbGetQuery(conn = db_con,
           statement = "DESCRIBE flights;")
```

- It's *customary* for **keywords** in **SQL statements** to be written in *capital letters* and for each statement to *end* with a *semicolon*.
- The last step in running a **query** is to **disconnect** from the server.

  A **connection** will **time out** (automatically disconnect) if it's inactive for several minutes. To **disconnect** *manually*, use:

```r
# Disconnect from the server:
dbDisconnect(conn = db_con)
```

Section 15.2 Exercises

Exercise 1 Load the following packages.

```r
library(DBI)
library(RMySQL)
```

Now open a **connection** to the **airlines database** using the command involving `dbConnect()` (from `"DBI"`) given above. Save the **connection** as, say, `db_con`.

a) What is the **class** of the object `db_con`? Use `class()`.

b) **How many tables** are in the **airlines database** and what are their **names**? Use `dbListTables()`.

c) Recall that `DESCRIBE` is used to get a description of the contents of a **table**. Using `dbGetQuery()`, with `statement = "DESCRIBE airports;"`, **how many variables** ("fields") are in the **airports table**?

d) Using `dbGetQuery()`, with `statement = "DESCRIBE flights;"`, **how many variables** ("fields") are in the **flights table**?

15.3 SELECT ... FROM

- As mentioned, a **query** is a **request** for data. We carry out **SQL queries** in R using `dbGetQuery()`, which returns the results as a *data frame*.

  Every SQL query must contain, at a minimum, `SELECT` and `FROM` clauses.

- `SELECT` is used to select **variables** (columns) to extract from a table, similar to `select()` in `"dpyr"`.

- `FROM` is used to specify the **table** where the data are stored.

- For example, to select *specific* **variables** (columns) from `flights`, specify their names after `SELECT`:

```
dbGetQuery(conn = db_con,
           statement = "SELECT year, month, day, dep_time, sched_dep_time, dep_delay, origin
                        FROM flights
                        LIMIT 0, 6;")
```

  (Output not shown.)

  Warning: `LIMIT 0, 6` limits the query to just the *first six* observations (rows). If we *didn't* specify `LIMIT 0, 6`, the command would attempt to retrieve *all* **196 million** observations (rows).

- To select *all* **variables** (columns) from `flights`, type:

```
dbGetQuery(conn = db_con,
           statement = "SELECT * FROM flights
                        LIMIT 0, 6;")
```

  (Output not shown.)

  Above, the **asterisk** indicates *all* variables should be selected.

- We can also form and select *new columns* from *existing* ones, similar to using `mutate()` in `"dplyr"`.

  For example, below, we form a *new column* containing the travel times of the flights (`arr_time` minus `dep_time`), using `AS` to give it the name `trvl_time`:

```
dbGetQuery(conn = db_con,
           statement = "SELECT arr_time, dep_time, arr_time - dep_time AS trvl_time
                        FROM flights
                        LIMIT 0, 6;")
```

  (Output not shown.)

  As another example, below, `CONCAT` *combines* (or "concatenates") two existing columns (`lat` and `lon`) from **airports** to form a *new one*, `coords`, analogous to using `unite()` in `"tidyr"`:

```r
dbGetQuery(conn = db_con,
           statement = "SELECT name, CONCAT('(', lat, ',', lon, ')') AS coords
                        FROM airports
                        LIMIT 0, 6;")
```

(Output not shown.)

Section 15.3 Exercises

Exercise 2 Use `dbGetQuery()` with `SELECT` and `FROM` to **query** the **flights table** to retrieve *just* the `carrier` and `tailnum` variables (columns).

Warning: Make sure to use `LIMIT` to limit the number of rows returned in your **query**.

Report your **R command(s)** (or just your **SQL statement**).

Exercise 3 Now use `dbGetQuery()` with `SELECT` and `FROM` to form a **query** to retrieve *all* the variables (columns) from the **carriers table**.

This time, **save** the data returned as, say, `my.carriers` in R.

(There's *no need* to `LIMIT` the number of rows returned by the **query** for the **carriers table**).

a) **Confirm** that the data set returned by `dbGetQuery()` is a *data frame* by typing:

```r
is.data.frame(my.carriers)
```

b) You can find out how much memory an object occupies in R using the `object.size()` function and its `print()` method.

   How much memory does `my.carriers` occupy? Find out by typing:

```r
print(object.size(my.carriers), units = "Kb")
```

   or

```r
print(object.size(my.carriers), units = "Mb")
```

Exercise 4 Now use `dbGetQuery()` with `SELECT` and `FROM` to form a **query** to retrieve *all* the variables (columns) from the **airports table**.

Save the data returned as, say, `my.airports` in R.

(There's *no need* to `LIMIT` the number of rows returned by the **query** for the **airports table**).

a) Each observation (row) in the `my.airports` *data frame* is an airport. **How many rows** does the data frame have? Use `str()` or `nrow()` or `dim()`.
b) **How many columns** (variables) does the data frame have?

Exercise 5 Recall that we can use `dbGetQuery()` with `SELECT` and `FROM` to form and select *new columns* from *existing* ones in a **table**, similar to using `mutate()` in `"dplyr"`.

Form a *new column* containing the travel speeds (in mph) of the flights (`distance` divided by `air_time`, then multiplied by 60), using `AS` to give it the name `trvl_speed`.

Warning: Make sure to use `LIMIT` to limit the number of rows returned in your **query**.

Report your **R command(s)** (or just your **SQL statement**).

15.4 WHERE

- `WHERE` is used to *filter* **observations** (rows) according to some criteria. It's analogous to the `filter()` function in `"dplyr"`.

  Logical operators `AND`, `OR`, and `NOT` can be used with `WHERE` to specify the condition the retrieved rows should meet.

- For example, to *filter* rows corresponding to flights on June, 26, 2013 from *Bradley International Airport* (BDL) from the **flights table**, type:

```
dbGetQuery(conn = db_con,
           statement = "SELECT * FROM flights
                        WHERE year = 2013 AND month = 6 AND day = 26 AND origin = 'BDL'
                        LIMIT 0, 6;")
```

  (Output not shown.)

- We can simultaneously *select* columns *and filter* rows in the same **query**, for example:

```
dbGetQuery(conn = db_con,
           statement = "SELECT year, month, day, dep_delay FROM flights
                        WHERE year = 2013 AND month = 6 AND day = 26 AND origin = 'BDL'
                        LIMIT 0, 6;")
```

  (Output not shown.)

- As mentioned, `AND`, `OR`, and `NOT` can be used in a `WHERE` clause.

  These operations *aren't* performed left to right. Rather, the `AND`s are performed *before* the `OR`s.

  Parentheses can be used, though, to control the order of these operations.

  For example, below `COUNT` is used to count rows. This first command (DON'T RUN IT) returns a count of 557,874:

```
# DON'T RUN THIS COMMAND.  It takes a few minutes.

# Returns a count of 557,874:
dbGetQuery(conn = db_con,
           statement = "SELECT COUNT(*) AS N FROM flights
                        WHERE year = 2013 AND month = 6 OR day = 26 AND origin = 'BDL';")
```

  It's equivalent to this:

```
# DON'T RUN THIS COMMAND.  It takes a few minutes.

# Returns a count of 557,874:
dbGetQuery(conn = db_con,
           statement = "SELECT COUNT(*) AS N FROM flights
                        WHERE (year = 2013 AND month = 6) OR (day = 26 AND origin = 'BDL');")
```

  On the other hand, the following command only returns a count of 2,542:

```
# Returns a count of 2,542:
dbGetQuery(conn = db_con,
           statement = "SELECT COUNT(*) AS N FROM flights
                        WHERE year = 2013 AND (month = 6 OR day = 26) AND origin = 'BDL';")
```

- We can use a `BETWEEN` clause or an `IN` clause to filter rows using a *range* or *set* of values, respectively. For example, the following commands (DON'T RUN THEM) *both* return flights on the 27th, 28th, and 29th of the June:

```
# DON'T RUN THIS COMMAND.  It takes a few minutes.
dbGetQuery(conn = db_con,
           statement = "SELECT year, month, day
                        FROM flights
                        WHERE year = 2013 AND month = 6 AND day BETWEEN 26 and 30
                          AND origin = 'BDL'
                        LIMIT 0, 6;")
```

```
# DON'T RUN THIS COMMAND.  It takes a few minutes.
dbGetQuery(conn = db_con,
           statement = "SELECT year, month, day
                        FROM flights
                        WHERE year = 2013 AND month = 6 AND day IN (27, 28, 29)
                          AND origin = 'BDL'
                        LIMIT 0, 6;")
```

- `WHERE` can operate on **functions** of columns (i.e. values *computed* from columns).

  For example, below, we filter on travel times of the flights (computed as `arr_time` minus `dep_time`), retrieving *only* those flights whose travel times were more than 10 hours (600 minutes):

```
dbGetQuery(conn = db_con,
           statement = "SELECT arr_time, dep_time, arr_time - dep_time AS trvl_time
                        FROM flights
                        WHERE arr_time - dep_time > 600
                        LIMIT 0, 6;")
```

Section 15.4 Exercises

Exercise 6 Report **R command(s)** (or just the **SQL statements**) involving `dbGetQuery()` and the logical operators (`AND`, `OR`, and `NOT`) with the **flights table** to retrieve flights meeting the following conditions.

Warning: Make sure to use `LIMIT` to limit the number of rows returned in your **query**.

a) Had an arrival delay of more than hours (`arr_delay` more than `120` minutes).

b) Flew to Houston (i.e. had a `dest` of `IAH` or `HOU`).

c) Were operated by United, American, or Delta (i.e. had a `carrier` of `UA`, `AA`, or `DL`).

d) Departed in summer (July, August, or September, i.e. `month` `7`, `8`, or `9`).

e) Departed between midnight and 6:00 AM, inclusive (`dep_time` at `2400` or between `0` and `600`, inclusive).

f) Were operated by United (`carrier UA`), departed in July (`month 7`), and had an arrival delay of more than two hours (`arr_delay` more than `120` minutes).

15.5 GROUP BY

- `GROUP BY` will *group* together (*aggregate*) rows of a **table** according to values of a categorical variable and then reduce each *group* of rows to a *single* row by computing a summary statistic for each *group*.

It's analogous to `group_by()` followed by `summarize()` in `"dplyr"`.

`COUNT()`, `SUM()`, and `AVG()`, `STDEV()`, `MIN()`, and `MAX()`, etc. can be used to specify the summary statistic to be computed.

- For example, we know that there were **65** flights that left *Bradley Airport* on June 26th, 2013, but how many belonged to each airline carrier?

  To answer this, we use `GROUP BY` to group the individual flights based on who the carrier was and `COUNT()` to count the flights:

```
dbGetQuery(conn = db_con,
           statement = "SELECT carrier, COUNT(*) AS numFlights
                        FROM flights
                        WHERE year = 2013 AND month = 6 AND day = 26 AND origin = 'BDL'
                        GROUP BY carrier;")
```

  (Output not shown.)

  `GROUP BY` can summarize *more than one variable* at a time. How many flights were there *and* what was the average departure delay for each carrier?

```
dbGetQuery(conn = db_con,
           statement = "SELECT carrier, COUNT(*) AS numFlights,
                          AVG(dep_delay) AS avgDepDelay
                        FROM flights
                        WHERE year = 2013 AND month = 6 AND day = 26 AND origin = 'BDL'
                        GROUP BY carrier;")
```

  (Output not shown.)

- We can group by values of *more than one* categorical variable, for example by carrier *and* destination below:

```
dbGetQuery(conn = db_con,
           statement = "SELECT carrier, dest, COUNT(*) AS numFlights,
                          AVG(dep_delay) AS avgDepDelay
                        FROM flights
                        WHERE year = 2013 AND month = 6 AND day = 26 AND origin = 'BDL'
                        GROUP BY carrier, dest
                        LIMIT 0, 6;")
```

  (Output not shown.)

Section 15.5 Exercises

Exercise 7 We know there were **65** flights that left *Bradley Airport* on June 26th, 2013, but what was the shortest departure delay for each airline carrier? What was the longest?

a) Use `dbGetQuery()` with `GROUP BY` and `MIN()` to retrieve the *shortest* departure delay for each carrier.

   Report your **R command(s)** (or just your **SQL statement**).

b) `GROUP BY` can summarize *more than one variable* at a time. Modify your command(s) from part *a* to use `MIN()` *and* `MAX()` to retrieve the *shortest and longest* departure delays for each carrier.

   Report your **R command(s)** (or just your **SQL statement**).

Exercise 8 This problem concerns the `GROUP BY` and `AVG()` functions. It uses the **flights table**.

a) Explain in words what the following command does (recall that `dest` is the *destination* of the flight):

```
dbGetQuery(conn = db_con,
           statement = "SELECT carrier, dest, AVG(arr_delay) AS meanArrDelay
                        FROM flights
                        WHERE year = 2013 AND month = 6 AND day = 26 AND origin = 'BDL'
                        GROUP BY dest;")
```

b) Recall that `GROUP BY` can summarize *more than one variable* at a time. Explain in words what the following command does:

```
dbGetQuery(conn = db_con,
           statement = "SELECT carrier, dest, AVG(arr_delay) AS meanArrDelay,
                          AVG(distance) AS meanDist
                        FROM flights
                        WHERE year = 2013 AND month = 6 AND day = 26 AND origin = 'BDL'
                        GROUP BY dest;")
```

15.6 ORDER BY

- `ORDER BY` is used to specify a condition for *ordering* the rows. It's analogous to `arrange()` in `"dplyr"`.
- For example, to *order* flights from *Bradley Airport* on June, 26, 2013 by departure delay, type:

```
dbGetQuery(conn = db_con,
           statement = "SELECT * FROM flights
                        WHERE year = 2013 AND month = 6 AND day = 26 AND origin = 'BDL'
                        ORDER BY dep_delay;")
```

(Output not shown.)

`ASC` and `DESC` can be used to specify *ascending* or *descending* order. For example, to *order* the flights *descending* by departure delay, use `DESC`:

```
dbGetQuery(conn = db_con,
           statement = "SELECT * FROM flights
                        WHERE year = 2013 AND month = 6 AND day = 26 AND origin = 'BDL'
                        ORDER BY dep_delay DESC;")
```

- Combining `GROUP BY` with an `ORDER BY` clause will bring the most interesting results to the top.

  For example, there were 22,258 flights that left *Bradley Airport* in the year 2013. Which destinations are most common from *Bradley Airport* in 2013?

```
# THIS TAKES ABOUT 60 SECONDS TO EXECUTE:
dbGetQuery(conn = db_con,
           statement = "SELECT dest, COUNT(*) AS numFlights
                        FROM flights
                        WHERE year = 2013 AND origin = 'BDL'
                        GROUP BY dest
                        ORDER BY numFlights DESC;")
```

  Note that *derived* columns like `numFlights` above *can* be referenced in the `ORDER BY` clause because `ORDER BY` operates on the *retrieved* data, not on the rows of the original data.

  As another example, which of those destinations had the lowest average arrival delay time?

```
# THIS TAKES ABOUT 60 SECONDS TO EXECUTE:
dbGetQuery(conn = db_con,
           statement = "SELECT dest, COUNT(*) AS numFlights, AVG(arr_delay) AS avg_arr_delay
                        FROM flights
                        WHERE year = 2013 AND origin = 'BDL'
                        GROUP BY dest
                        ORDER BY avg_arr_delay ASC;")
```

The output (not shown) tells us Cleveland had the lowest average arrival delay – more than 13 minutes ahead of schedule – among destinations of flights from *Bradley* in 2013.

Section 15.6 Exercises

Exercise 9 This problem concerns `ORDER BY` combined with `GROUP BY` and `AVG()`. It uses the **flights table**.

There were 22,258 flights that left *Bradley Airport* in the year 2013.

a) Use `GROUP BY` with `AVG()` and `ORDER BY` to determine the destination `dest` for which the average travel time (`air_time`) from *Bradley Airport* was *shortest* in 2013. Report the (abbreviated) destination (`dest`) name.

b) Now use `GROUP BY` with `AVG()` and `ORDER BY` to determine the destination `dest` for which the average travel time (`air_time`) from *Bradley Airport* was *longest* in 2013. Report the (abbreviated) destination (`dest`) name.

Exercise 10 This problem concerns `ORDER BY` combined with `GROUP BY` and `COUNT()`. It uses the **flights table**.

There were 22,258 flights that left *Bradley Airport* in the year 2013.

a) Use `GROUP BY` with `COUNT()` and `ORDER BY` to determine to determine which `dest` (i.e. which **destination**) was flown to the *most* times out of *Bradley Airport* in 2013. Report the (abbreviated) destination (`dest`) name.

b) Use `GROUP BY` with `COUNT()` and `ORDER BY` to determine which `tailnum` (i.e. which individual **airplane**) flew the *most* times out of *Bradley Airport* in 2013. Report the `tailnum` value of the airplane.

15.7 HAVING

- `HAVING` is like a `WHERE` clause, but it retrieves a subset of rows from a table that's *already* been summarized **by group** (i.e. after the *full table's* rows have been *aggregated*).

  It's analogous to applying a `filter()` command in `"dplyr"` *after* the rows have *already* been grouped *and* summarized using `group_by()` and `summarize()`.

- For example, although flights to Cleveland had the lowest average arrival delay, there were only 57 flights that went to from *Bradley* to Cleveland in all of 2013.

  It probably makes more sense to consider only those destinations that had, say, at least two flights per day.

  We can filter the **result set** from the **query** made at the end of Section 15.6 using a `HAVING` clause:

```
# THIS TAKES ABOUT 60 SECONDS TO EXECUTE:
dbGetQuery(conn = db_con,
           statement = "SELECT dest, COUNT(*) AS numFlights, AVG(arr_delay) AS avg_arr_delay
                        FROM flights
                        WHERE year = 2013 AND origin = 'BDL'
                        GROUP BY dest
                        HAVING numFlights > 365 * 2
                        ORDER BY avg_arr_delay ASC;")
```

Note that above, the `HAVING` clause needs **`numFlights`**, which is is only available in the **result set** *retrieved* by the `SELECT`, `FROM`, `WHERE`, and `GROUP BY` clauses.

Section 15.7 Exercises

Exercise 11 There were 22,258 flights that left *Bradley Airport* in the year 2013.

There were 25 destinations of those flights from *Bradley*:

```
# THIS TAKES ABOUT 60 SECONDS TO EXECUTE:
dbGetQuery(conn = db_con,
           statement = "SELECT dest, COUNT(*) AS numFlights,
                          AVG(arr_delay) AS avg_arr_delay
                        FROM flights
                        WHERE year = 2013 AND origin = 'BDL'
                        GROUP BY dest;")
```

a) Alter the command above using `HAVING` so that it only returns destinations for which the average arrival delay is positive (`avg_arr_delay > 0`).

   Report your **R command(s)** (or just your **SQL statement**).

b) Now alter your command from part *a* so that it only returns destinations for which the average arrival delay is positive (`avg_arr_delay > 0`) *and* the total number of flights was more than 1,000 (`numFlights > 1000`).

   Report your **R command(s)** (or just your **SQL statement**).

Exercise 12 There were 22,258 flights that left *Bradley Airport* in the year 2013.

Those flights from *Bradley* were made by 12 airline carriers.

Use `dbGetQuery()` with `SELECT`, `FROM`, `WHERE`, `GROUP BY`, and `HAVING` to **query** the **flights table** to:

1. Group the flights from *Bradley* in 2013 by airline carrier (`carrier`).
2. Find the average departure delay for each carrier (use `AVG()` with `dep_delay`).
3. Retrieve just the carriers whose average departure delays were longer than 10 minutes.

Report your **R command(s)** (or just your **SQL statement**).

15.8 LIMIT

- We've seen that `LIMIT` is used to restrict the number of rows in the output, similar to using `head()` in R.

  But `LIMIT` can also be used to select a **specified number of rows** starting from a specific row, similar to slice() in "dplyr".

For example, this **query** will return just the **4th-7th** flight destinations returned by the previous **query**.

```
# THIS TAKES ABOUT 60 SECONDS TO EXECUTE:
dbGetQuery(conn = db_con,
           statement = "SELECT dest, COUNT(*) AS numFlights, AVG(arr_delay) AS avg_arr_delay
                        FROM flights
                        WHERE year = 2013 AND origin = 'BDL'
                        GROUP BY dest
                        HAVING numFlights > 365 * 2
                        ORDER BY avg_arr_delay ASC
                        LIMIT 3, 4;")
```

Section 15.8 Exercises

Exercise 13 This exercise concerns `LIMIT` clauses.

Recall that this **query** returns just the **4th-7th** flight destinations returned by a previous **query**:

```
# THIS TAKES ABOUT 60 SECONDS TO EXECUTE:
dbGetQuery(conn = db_con,
           statement = "SELECT dest, COUNT(*) AS numFlights, AVG(arr_delay) AS avg_arr_delay
                        FROM flights
                        WHERE year = 2013 AND origin = 'BDL'
                        GROUP BY dest
                        HAVING numFlights > 365 * 2
                        ORDER BY avg_arr_delay ASC
                        LIMIT 3, 4;")
```

How would you **modify** the **query** above so that it returns instead the **6th-9th** flight destinations?

15.9 JOIN

- Recall that **SQL** is a **relational** database management system – the **relations** between *linkable* **tables** allow for **queries** to tie together information from **multiple tables**.
- `JOIN` is used to **combine together** two (or more) tables based on a ***key*** variable (column) in each table, analogous to `inner_join()` and `left_join()` from "dplyr".
- There are four pieces of information you need to specify in order to **join** two **tables**:
  - The **name** of the **first table** that you want to join.
  - (Optional) The **type** of **join** that you want to use.
  - The **name** of the **second table** that you want to join.
  - The **key** columns in the two **tables**, i.e. the **condition(s)** under which you want the row in the **first table** to **match** the rows in the **second table**.
- In practice, the `JOIN` syntax varies among **SQL implementations**. In **MySQL**, `OUTER JOIN`s (aka **full joins**) are *not* available, but the following **join types** are:
  - `JOIN`: includes all of the rows that are present in **both tables** and **match**.
  - `LEFT JOIN`: includes all of the rows that are present in the **first table**. Rows in the **first table** that have **no match** in the **second** are filled with `NULL`s.
  - `RIGHT JOIN`: include all of the rows that are present in the **second table**. This is the **opposite** of a `LEFT JOIN`.

   - `CROSS JOIN`: the so-called **Cartesian product** of the two tables. Thus, **all possible combinations** of rows **matching** the **joining condition** are returned.

- For example, recall that in the `flights` table, the **origin** and **destination** of each **flight** are recorded.

```
dbGetQuery(conn = db_con,
           statement = "SELECT origin, dest, flight, carrier
                        FROM flights
                        WHERE year = 2013 AND month = 6 AND day = 26 AND origin = 'BDL'
                        LIMIT 0, 6;")
```

  (Output not shown.)

  The `flights` table contains only the FAA three-character codes for the airports – not the full airport names.

  It would be convenient to have the airport names in the `flights` table, but it would be **storage-inefficient**.

  The solution is to store information about *airports* in the `airports` table (which only has **1,458** rows, not **169 million** as `flights` has), along with the three-character codes – the ***keys*** – and to only store these **keys** in the `flights` table.

  We can then use these **keys** to **join** the two **tables** together in our **query**.

- For our **query**, we specify the **table** (`airports`) we want to **join** onto `flights` and the **condition** by which we want to **match** rows in `flights` with rows in `airports`.

  In this case, we want the *destination* `airport` code in the `dest` column of `flights` to be **matched** to the `faa` airport code in `airports`. These are the **key** columns.

  We also specify that we want to see the full *airport* name (`name` column from the `airports` table) in the **result set**.

```
dbGetQuery(conn = db_con,
           statement = "SELECT origin, dest, airports.name, flight, carrier
                        FROM flights
                        JOIN airports ON flights.dest = airports.faa
                        WHERE year = 2013 AND month = 6 AND day = 26 AND origin = 'BDL';")
```

  Note there are also codes in `flights` for the airline *carriers*. The full name of each *carrier* is stored in the `carriers` table.

  We can **join** the `carriers` table to our **result set** from above to retrieve the full name of each *carrier*.

```
dbGetQuery(conn = db_con,
           statement = "SELECT dest, airports.name AS dest_name, flights.carrier,
                          carriers.name AS carrier_name
                        FROM flights
                        JOIN airports ON flights.dest = airports.faa
                        JOIN carriers ON flights.carrier = carriers.carrier
                        WHERE year = 2013 AND month = 6 AND day = 26 AND origin = 'BDL';")
```

  (Above, the `name` columns from `airports` and `carriers` were renamed `dest_name` and `carrier_name`, otherwise there would've been two `name` columns in the **result set**.)

  Finally, to retrieve the name of the *originating airport* (in addition to the *destination airport*), we can join onto the `airports` table more than once.

  Here, so-called ***table aliases*** (`a1` and `a2` below) are necessary:

```r
dbGetQuery(conn = db_con,
           statement = "SELECT flight,
                               a1.name AS dest_name,
                               a2.name AS orig_name,
                               carriers.name AS carrier_name
                        FROM flights
                        JOIN airports AS a1 ON flights.dest = a1.faa
                        JOIN airports AS a2 ON flights.origin = a2.faa
                        JOIN carriers ON flights.carrier = carriers.carrier
                        WHERE year = 2013 AND month = 6 AND day = 26 AND origin = 'BDL';")
```

Section 15.9 Exercises

Exercise 14 This exercise involves a `JOIN` clause.

There were **65** flights that left *Bradley Airport* on June 26th, 2013.

For this **query**, you'll need to **join** the `airports` table onto the `flights` table, **matching** the *destination airport* code (`dest` column) of `flights` to the *airport* code (`faa` column) in `airports`.

You'll also need to see the *destinations*, *flight numbers*, and airline *carrier* codes (`dest`, `flight`, and `carrier` columns) from the `flights` table and the full *airport* names (`name` column) from the `airports` table in the **result set**.

Use `dbGetQuery()` to answer the following problem. List the full name of the *destination airport* of flight **EV 4714** from from *Bradley* on June 26th, 2013.

Exercise 15 This exercise involves a `JOIN` clause.

There were **65** flights that left *Bradley Airport* on June 26th, 2013.

For this **query**, you'll need to **join** the `carriers` table onto the `flights` table, **matching** the airline *carrier* code (`carrier` column) of `flights` to the *carrier* code (`carrier` column) in `carriers`.

You'll also need to see the *destinations* and *flight numbers* and (`dest` and `flight` columns) from the `flights` table and the full *carrier* names (`name` column) from the `carriers` table in the **result set**.

Use `dbGetQuery()` to answer the following problems.

a) List the full airline *carrier* name and *flight number* for all flights between *Bradley Airport* (BDL) and MSP on June 26th, 2013.

b) List the *destination airport* codes and *flight numbers* for the **three** flights from *Bradley* made by Mesa Airlines Inc. on June 26th, 2013.

15.9.1 LEFT JOIN

- `JOIN` (discussed above) returns *only* the rows that are present in **both tables** and **match**, i.e. it performs an ***inner join***, the same as `inner_join()` in `"dplyr"`.
- `LEFT JOIN` returns *all* of the rows in the **first table**, *regardless* of whether they have a match in the second. Rows in the **first table** that have **no match** in the **second** are filled with `NULL`s (converted `NA`s in R). It's equivalent to `inner_join()` in `"dplyr"`.
- `RIGHT JOIN` does the **opposite** of `LEFT JOIN`. It returns *all* of the rows in the **second table** *regardless* of whether they have a match in the first.

- For example, here's a **table** with **names** and **ages** of three people, and another **table** containing **names** and **heights** of three people, *only two of which match* the first **table**.

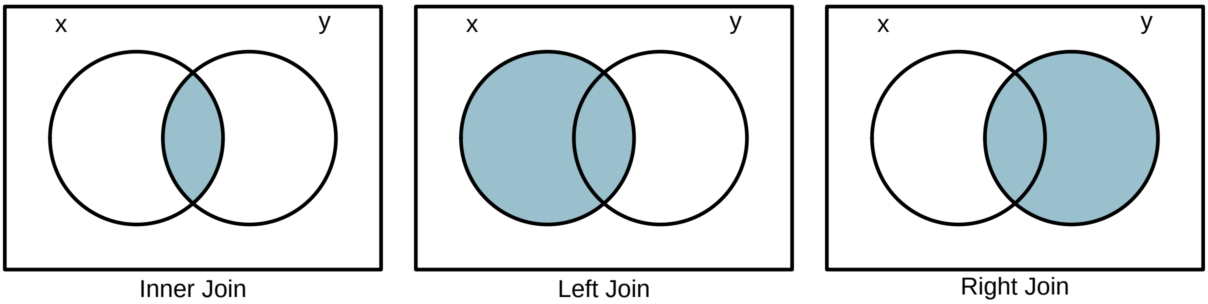
  The **Name** variable is **common** to **both tables**, and is used as the `key` to match rows:

Table X

Name	Age
John	23
Karen	**27**
Ann	**19**

Table Y

Name	Height
Karen	**63**
Ann	**65**
Karl	68



`JOIN` **X, Y**

Name	Age	Height
Karen	**27**	**63**
Ann	**19**	**65**

`LEFT JOIN` **X, Y**

Name	Age	Height
John	23	`NA`
Karen	**27**	**63**
Ann	**19**	**65**

`RIGHT JOIN` **X, Y**

Name	Age	Height
Karen	**27**	**63**
Ann	**19**	**65**
Karl	`NA`	68

(Note the `NA`s produced by `LEFT JOIN` and by `RIGHT JOIN`.)

- As another example, there are flights in the **flights table** whose destination airport (SJU) has no corresponding entry in **airports**.

  `LEFT JOIN` returns **all 65** flights from *Bradley Airport*, *including* the flight to SJU, but with `NA` as its *destination airport name* for that flight:

```
dbGetQuery(conn = db_con,
           statement = "SELECT origin, dest, airports.name AS dest_name, flight, carrier
                        FROM flights
                        LEFT JOIN airports ON flights.dest = airports.faa
                        WHERE year = 2013 AND month = 6 AND day = 26 AND origin = 'BDL';")
```

  `JOIN` *doesn't include* the flight to SJU in the **result set** *at all*:

```
dbGetQuery(conn = db_con,
           statement = "SELECT origin, dest, airports.name AS dest_name, flight, carrier
                        FROM flights
                        JOIN airports ON flights.dest = airports.faa
                        WHERE year = 2013 AND month = 6 AND day = 26 AND origin = 'BDL';")
```

Section 15.9 Exercises

Exercise 16 This exercise concerns `LEFT JOIN`.

There were **50** flights that left *Palm Beach International Airport* ('PBI') on June 26th, 2013.

Explain in words why, below, `LEFT JOIN` returns **all 50** flights (with `NA` as one of the *destination airport names*), but `JOIN` only returns **49**.

```
dbGetQuery(conn = db_con,
           statement = "SELECT origin, dest, name AS dest_name, flight, carrier
                        FROM flights
                        LEFT JOIN airports ON flights.dest = airports.faa
                        WHERE year = 2013 AND month = 6 AND day = 26 AND origin = 'PBI';")
```

```
dbGetQuery(conn = db_con,
           statement = "SELECT origin, dest, name AS dest_name, flight, carrier
                        FROM flights
                        JOIN airports ON flights.dest = airports.faa
                        WHERE year = 2013 AND month = 6 AND day = 26 AND origin = 'PBI';")
```

15.10 UNION

- **Two** separate **SQL queries** can be **combined** using a `UNION` clause. This is analogous to `rbind()` in R and `bind_rows()` in `"dplyr"`.
- Each **query** should be enclosed in **parentheses**.

  For example, in the **query** below, flights from *Bradley* (BDL) to *Minneapolis St Paul* (MSP) are **combined** with ones from *John F Kennedy* (JFK) to *Chicago Ohare* (ORD) to comprise the **result set**:

```
dbGetQuery(conn = db_con,
           statement = "(SELECT year, month, day, origin, dest, flight, carrier
                        FROM flights
                        WHERE year = 2013 AND month = 6 AND day = 26
                          AND origin = 'BDL' AND dest = 'MSP')
                        UNION
                        (SELECT year, month, day, origin, dest, flight, carrier
                        FROM flights
                        WHERE year = 2013 AND month = 6 AND day = 26
                          AND origin = 'JFK' AND dest = 'ORD');")
```

Section 15.10 Exercises

Exercise 17 This exercise concerns the use of `UNION`.

Describe **in words** which sets of flights will be **combined** to comprise the **result set** of the following **query**, then check your answer.

```
dbGetQuery(conn = db_con,
           statement = "(SELECT year, month, day, origin, dest, flight, carrier
                        FROM flights
                        WHERE year = 2013 AND month = 6 AND day = 26
                          AND origin = 'BDL' AND dest = 'ORD')
                        UNION
                        (SELECT year, month, day, origin, dest, flight, carrier
                        FROM flights
                        WHERE year = 2013 AND month = 6 AND day = 26
                          AND origin = 'MSP' AND dest = 'JFK');")
```

15.11 Subqueries

- It's possible to **query** the **result set** of another **query** as if it were a **table**. The *initial* **query** is called a ***subquery***.
- For example, does *Bradley Airport* have any flights going to Alaska or Hawaii?

  We can retrieve a list of the **258** airports in Alaska and Hawaii by filtering the **airports table** using the time zone (`tz`) column:

```
#This retrieves only the airports in Alaska and Hawaii:
dbGetQuery(conn = db_con,
           statement = "SELECT faa, name, tz, city
                        FROM airports
                        WHERE tz < -8;")
```

  Now we'll use the **result set** (list of Alaska and Hawaii airports) generated by the **query** above as a **subquery** in a `WHERE` clause to retrieve the flights from *Bradley* in 2013 whose destinations were in Alaska or Hawaii:

```
#THIS TAKES ABOUT 60 SECONDS TO EXECUTE:
dbGetQuery(conn = db_con,
           statement = "SELECT dest
                        FROM flights
                        WHERE year = 2013
                          AND origin = 'BDL'
                          AND dest IN
                            (SELECT faa                 # <- The subquery is en-
                             FROM airports              #    closed in parentheses
                             WHERE tz < -8);")
```

  The output (not shown) indicates *no* flights were identified by the **query**, meaning there *weren't any* to Alaska or Hawaii.

  Note (above) that the **subquery** needed to be enclosed in **parentheses**.

Section 15.11 Exercises

Exercise 18 In this section, we used a **subquery** to determine whether *Bradley Airport* had any flights to Alaska or Hawaii (time zone, `tz`, less than **-8**) in 2013.

Did *Bradley* have any flights to airports in the *Pacific time zone* (`tz` less than **-7**) in 2013? If so, **which airports** (three-letter codes) in the *Pacific time zone* were the flights' destinations?

Exercise 19 In this section, we used a **subquery** to determine whether *Bradley Airport* had any flights to Alaska or Hawaii (time zone, `tz`, less than **-8**) in 2013.

Did *John F Kennedy Airport* (JFK) have any flights to Alaska or Hawaii (time zone, `tz`, less than **-8**) in 2013? If so, **which airport(s)** (three-letter codes) in Alaska or Hawaii were the flights' destinations?

15.12 Database Querying With "dplyr" and "dbplyr"

- The `"dplyr"` package allows you to use **tables** in a remote **database** *as if* they're ***in-memory*** (on-your-computer) *data frames* by automatically translating `"dplyr"` code to **SQL**.
- `"dplyr"` automatically calls functions from the `"dbplyr"` package, which then translate `"dplyr"` commands into **SQL queries** .
- We'll use the following functions from `"dplyr"` and `"dbplyr"`.

```
tbl()               # Maps a table in a remote database to an object in R
show_query()        # Show the SQL statement corresponding to an R query.
translate_sql()     # Translates an R expression to SQL.
collect()           # Saves data retrieved by a "dplyr" database query as a
                    # data frame.
```

- The first step is again to **connect** to the remote **server** using `dbConnect()` (from the `"DBI"` package).

  Here, we connect to the **airlines database** on the remote **MySQL server** (set up by the textbook authors):

```r
# For functions in "dplyr" and "dbplyr":
library(dplyr)

# For dbConnect(), dbGetQuery(), and dbListTables():
library(DBI)

# For MySQL():
library(RMySQL)

# Connect to the database:
db_con <- dbConnect(drv = MySQL(), dbname = "airlines",
                         host = "mdsr.cdc7tgkkqd0n.us-east-1.rds.amazonaws.com",
                         user = "mdsr_public",
                         password = "ImhsmflMDSwR")
```

- Now we can map to two **tables** using `tbl()`:

```r
flights <- tbl(db_con, "flights")
carriers <- tbl(db_con, "carriers")
```

  Above, the `tbl()` function maps the **flights table** in the **airlines database** to an object in R, in this case also called `flights`. The same is done for the **carriers table**.

  Now we can use `flights` and `carriers` *as if* they were *data frames*:

```r
head(flights)
head(carriers)
```

  (Output not shown.)

- As another example, this command runs a **query** to retrieve the **65** flights from *Bradley Airport* on June 6, 2013:

```r
filter(.data = flights, year == 2013 & month == 6 & day == 6 & origin == "BDL")
```

  (Output not shown.)

- A `tbl` is a special kind of object created by `"dbplyr"` that *behaves* like a data frame, but isn't, in fact, a data frame. Rather, they belong to the `tbl_MySQLConnection` class, and more generally, `tbl_sql` and `tbl`:

```
class(flights)
class(carriers)
```

(Output not shown.)

Because `flights` is a `tbl_sql` and *not* a *data frame*, it **resides** on the **remote server**, *not* on our computer, and the computations are actually being done by **MySQL** on the **server**, not here on our computer.

The `"dplyr"` commands are simply translated into **SQL** and submitted to the **server**.

We can see the translation by passing our `"dplyr"` **query** through `show_query()`:

```
my.query <- filter(.data = flights, year == 2013 & month == 6 & day == 6 & origin == "BDL")

show_query(my.query)

## <SQL>
## SELECT *
## FROM `flights`
## WHERE (`year` = 2013.0 AND `month` = 6.0 AND `day` = 6.0 AND `origin` = 'BDL')
```

Even though we *wrote* our **query** in R, it was **translated** to **SQL** by `"dbplyr"`.

`"dbplyr"` does this automatically any time it encounters an object of class `tbl_sql` (such as `flights` in the **query** above).

- How is the **translation** done? `"dplyr"` calls the `translate_sql()` function (from `"dbplyr"`). For example:

```
library(dbplyr)

translate_sql(mean(arr_delay, na.rm = TRUE))

## <SQL> AVG(`arr_delay`) OVER ()
```

- Note that `translate_sql()` *cannot* translate *every* R command to **SQL**. (See the textbook for examples.)
- Note also that running a **query** using `"dplyr"` *doesn't* actually *retrieve* the data onto our computer.

Instead, it returns an object of class `tbl_MySQLConnection`, and more generally, `tbl_sql` and `tbl`:

```
my.query <- filter(.data = flights, year == 2013 & month == 6 & day == 6 & origin == "BDL")

class(my.query)

## [1] "tbl_MySQLConnection" "tbl_dbi"             "tbl_sql"
## [4] "tbl_lazy"            "tbl"
```

This means the **result set** of the `"dplyr"` **query** still **resides** on the **remote server**, *not* on our computer.

If we want the results of the `"dplyr"` **query** pulled into a *data frame* here in R, we can use the `collect()` function. For example:

```
my.query <- filter(.data = flights, year == 2013 & month == 6 & day == 6 & origin == "BDL")

my.query.df <- collect(my.query)

class(my.query.df)

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

The `collect()` function breaks the connection to the **MySQL server** and returns a *data frame* (which is also a `tbl_df`).

Warning: Above, `collect()` retrieved a **query result set** that only had **65** rows. **Do not** attempt to use `collect()` to retrieve *all* **196 million** rows of the `flights` **table**.

```
## [1] TRUE
```

Section 15.12 Exercises

Exercise 20 Make sure the the following packages are loaded.

```r
library(dplyr)
library(DBI)
library(RMySQL)
```

Now open a **connection** to the **airlines database** using `dbConnect()`, and save the **connection** as, say, `db_con`:

```r
# Connect to the database:
db_con <- dbConnect(drv = MySQL(),
                    dbname = "airlines",
                    host = "mdsr.cdc7tgkkqd0n.us-east-1.rds.amazonaws.com",
                    user = "mdsr_public",
                    password = "ImhsmflMDSwR")
```

Next, map to the `flights` and `carriers` **tables** from the **airlines database** using `tbl()`:

```r
flights <- tbl(db_con, "flights")
carriers <- tbl(db_con, "carriers")
```

In Exercise 2, `SELECT` and `FROM` were used to **query** the **flights table** to retrieve *just* the `carrier` and `tailnum` variables (columns).

Write a command using `"dplyr"` that performs this same **query**. **Do not** use `dbGetQuery()`.

Report your **R command(s)**.

Exercise 21 If you're not still connected to the **airlines database** and mapped to the `flights` and `carriers` **tables**, reconnect and re-map.

In Excercise 6, the logical operators (`AND`, `OR`, and `NOT`) were used with the **flights table** to retrieve flights meeting the conditions listed below.

Report **R command(s)** involving `"dplyr"` and the logical operators `&` (**"and"**), `|` (**"or"**), and `!` (**"not"**) with the **flights table** to map to flights meeting the conditions:

a) Had an arrival delay of more than hours (`arr_delay` more than `120` minutes).

b) Flew to Houston (i.e. had a `dest` of `IAH` or `HOU`).

c) Were operated by United, American, or Delta (i.e. had a `carrier` of `UA`, `AA`, or `DL`).

d) Departed in summer (July, August, or September, i.e. `month` `7`, `8`, or `9`).

e) Departed between midnight and 6:00 AM, inclusive (`dep_time` at `2400` or between `0` and `600`, inclusive).

f) Were operated by United (`carrier UA`), departed in July (`month 7`), and had an arrival delay of more than two hours (`arr_delay` more than `120` minutes).

15.13 Practicing SQL In-Memory (Without Connecting to a Server)

- Up to now, we've only **retrieved** data from a **SQL database** by running **queries**.

  Modifying a **database** on a remote *server* requires **permission** from the **manager** of the **database**.

- For **practicing modifying** a **SQL database**, it's useful to be able to be able to create *your own* **database** on your **computer's memory**, *not* on a remote server.

  This can be done by specifying `dbname = ":memory:"` in `dbConnect()`.

  Note that for this, we need to use **SQLite**. MySQL *doesn't* work.

```
library(DBI)            # For dbConnect(), dbWriteTable(), etc.
library(RSQLite)        # For SQLite()

# Connect to the computer's memory using dbConnect():
db_con <- dbConnect(drv = SQLite(),
                    dbname = ":memory:")
```

  This causes **SQLite** to create a temporary **in-memory database**.

  Initially, there are **no tables** in the **database**:

```
dbListTables(db_con)

## character(0)
```

- We can use `dbWriteTable()` (from the `"DBI"` package) to *write* an R *data frame* into a **table** in the **database** stored on the *computer's memory* (or a *remote server*).

  Other functions (from `"DBI"`) can be used to *modify* or *remove* tables.

```
dbWriteTable()     # Copy a data frame to a table in a database.
dbListTables()     # List the tables in a database.
dbListFields()     # List field (column) names of a table in a database.
dbAppendTable()    # Insert rows into a table in a database.
dbRemoveTable()    # Remove a table from a database.
```

- `dbWriteTable()` takes arguments `conn` (a connection to *computer's memory* or a *server*), `name`, the name to be assigned to the newly created **table** in the **database**, and `value`, the *data frame* used to create the **table**.

Data Set: `who` and `population`

The `who` and `population` data sets (in `"tidyr"`) contain a subset of data from the World Health Organization Global Tuberculosis (TB) Report, and accompanying global populations.

`who` has 7,240 rows and the 60 variables:

`country`	Country name.
`iso2`, `iso3`	2 & 3 letter ISO country codes.
`year`	Year.
`new_sp_m014` - `new_rel_f65`	Counts of new TB cases recorded by group. Column names encode three variables that describe the group (see details in the help page, `?who`).

`population` has 4,060 rows and three variables:

`country`	Country name.
`year`	Year.
`population`	Population.

The help page (`?who`) has more information about these data sets.

- For example, below, we create an *in-memory* **database** containing **two tables**, the `who` and `population` *data frames* (from the `"tidyr"` package):

```
#Load the who table:
dbWriteTable(conn = db_con,
             name = "whoTable",
             value = who)

#Load the population table:
dbWriteTable(conn = db_con,
             name = "popTable",
             value = population)
```

Now we can confirm that the **database** contains the **two tables**:

```
# Now the database contains the population and who tables:
dbListTables(db_con)

## [1] "popTable" "whoTable"
```

Note that the **two tables** *aren't* stored in the R Workspace:

```
whoTable

## Error in eval(expr, envir, enclos):  object 'whoTable' not found

popTable

## Error in eval(expr, envir, enclos):  object 'popTable' not found
```

Rather, they're in a separate **database** elsewhere in the computer's *memory*.

Once the *in-memory* **database** contains one or more **tables**, we can run **queries** using `dbGetQuery()`.

For example,

```
my.query <- dbGetQuery(conn = db_con,
                       statement = "SELECT whoTable.country, whoTable.year,
                                      whoTable.new_sp_m014,
                                      popTable.population
                                    FROM popTable, whoTable
                                    WHERE whoTable.country = 'Afghanistan' AND
                                      whoTable.year > 1995 AND
                                      whoTable.country = popTable.country AND
                                      whoTable.year = popTable.year")

head(my.query)
```

(Output not shown.)

Recall that `dbGetQuery()` returns the **result set** as a *data frame*:

```
is.data.frame(my.query)

## [1] TRUE
```

- The function `dbAppendTable()` can be used to **insert rows** into a **table** in a **database**.

  `dbAppendTable()` takes arguments `conn`, an open connection to a **database**, `name`, the name of the **table** in the **database**, and `value`, a *data frame* containing the **rows** to be **appended** to the **table**.

  For example, below, we create a *data frame* containing a *single* **row** and **append** it to the `popTable` **table**:

```
newData <- data.frame(country = "Imaginary Country"    ,
                      year = 2010,
                      population = 30033440)
newData

##             country year population
## 1 Imaginary Country 2010   30033440
```

```
dbAppendTable(conn = db_con,
              name = "popTable",
              value = newData)
```

Section 15.13 Exercises

Exercise 22 Create an *in-memory* **database**, then add **two tables**, the `who` and `population` *data frames* (from the `"dplyr"` package):

```r
library(DBI)            # For dbConnect(), dbWriteTable(), etc.
library(RSQLite)        # For SQLite()

# Connect to the computer's memory using dbConnect():
db_con <- dbConnect(drv = SQLite(),
                    dbname = ":memory:")

#Load the who table:
dbWriteTable(conn = db_con,
             name = "whoTable",
             value = who)

#Load the population table:
dbWriteTable(conn = db_con,
             name = "popTable",
             value = population)
```

a) Now use `dbGetQuery()` with `SELECT` and `FROM` to **query** the **who table** to retrieve *just* the `country`, `year`, and `new_sp_m014` variables (columns).

   Report your **R command(s)** (or just your **SQL statement**).

b) Now modify your **query** from part *a* using a `WHERE` clause so that it *only* retrieves rows corresponding to the U.S. (`"United States of America"`) more recently than 2009 (`year >= 2010`) .

   Report your **R command(s)** (or just your **SQL statement**).

Exercise 23 If you're not still connected to the *in-memory* **database** containing the **two tables**, `who` and `population`, reconnect and reinsert the **tables**.

Use `GROUP BY` with `AVG()` and `ORDER BY` to determine the `country` for which the average number of new TB cases for the (`new_sp_m014`) group was *lowest* in the `year` 2013.

Note: It's okay if you end up with `NA`s.

Report your **R command(s)** (or just your **SQL statement**).