# INTRODUCTION TO SQL

jumping rivers

JUMPINGRIVERS.COM

# 1 Introduction

## What is SQL?

SQL is short for Structured Query Language and defines a standard for communicate with a relational database management system (RDBMS). RDBMS is a database management system that based on the relational model invented by Edgar Codd of IBM. SQL defines the standard but many dialects or implementations of SQL exist. Some of the more well known implementations of SQL are:[1]

- MySQL
- Oracle
- PostgreSQL

[1] SQLite is omitted here despite being arguably the most widely deployed system. This is because whilst it is a database management system that implements most of the SQL standard, it is not a client-server engine, rather embedded in the end program.

Each dialect has slight differences in the way they implement the SQL standard, however for the more simple tools that we will use for the purpose of today this should be of no concern to us. Likewise different implementations have different licensing.

## PostgresQL

For our session we will be interacting with data stored in a PostgreSQL database.[2] PostgreSQL is an open source relational data base management system that implements the SQL standard. It also extends the SQL standard through the addition of procedural programming structures such as control flow. It runs on all major operating systems and has a reputation for reliability and integrity.

[2] Often called just "postgres".

## Getting started

Each person will have been given a unique user name and password for a PostgreSQL database which is running on a Jumping Rivers server.[3] Conveniently it is possible to send SQL code to the data base from R so we can use RStudio and R programming as our front end interface. However the R packages that allow us to do this also package up some convenience functions for common tasks. During the course we will look at the SQL code for each task and where appropriate also point out the R function shortcut.

[3] In theory this should allow you all to create and manipulate data tables without causing conflicts or issues with other attendees.

## *The `RPostgreSQL` package*

The `RPostgreSQL` package enables a direct connection between a database and R.[4] The package makes use of `DBI` which gives a consistent set of functions regardless of the database type that is being used. You can install the package and it's dependencies in the usual way.

[4] There is similarly, `RMySQL`, `RSQLite` and `bigquery`.

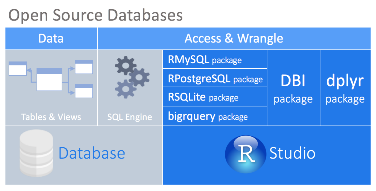

Figure 1.1: Connecting to databases.

```r
knitr::include_graphics("../graphics/sql_R.png")
```

```r
install.packages("RPostgreSQL")
```

Each database management system has it's own driver. For our connection to be made between R and the PostgreSQL management system to be made we need to let R know what driver to use.

```r
library("RPostgreSQL")
drv = dbDriver("PostgreSQL")
```

Given the correct driver we can now make a connection to our database

```
#> Loading required package:  methods
#> Loading required package:  DBI
```

```r
user = "user.name"
pass = "user.password"
dbname = "user.db"


con = dbConnect(drv, dbname = dbname,
                host = "localhost", port = 5432,
                user = user, password = pass)
```

## *Exploring the data base*

A database can have many tables of data in it. To view the list of tables that you have in your data base you can use[5]

[5] If you were at the command line interface to PostgreSQL you could use `\dt` here.

```r
dbListTables(con)
#>  [1] "test_table" "movies"     "diamonds"
#>  [4] "test"       "purchases"  "airlines"
#>  [7] "airports"   "flights"    "planes"
#> [10] "weather"
```

To see what variables (or field names) are stored within a given table, we could use[6]

[6] The corresponding command line statement would be `\d+ diamonds`.

```r
dbListFields(con, "diamonds")
#>  [1] "row.names" "carat"     "cut"
#>  [4] "color"     "clarity"   "depth"
#>  [7] "table"     "price"     "x"
#> [10] "y"         "z"
```

2

# My first data extraction with SQL

Extracting data from a table in the data base will always start with a `SELECT` *clause*. A simple select *statement* might look like,

```
SELECT column_name FROM table_name;
```

This query would fetch all data stored in the given column name from the given table name. Note that unlike R a semi-colon is required to indicate the end of a command.

Our database had in it a table called `diamonds`. We could use

```
SELECT * FROM diamonds;
```

to fetch all data from this table.[1] The `DBI` package has a useful function for sending raw SQL queries to the database which return the data to our R session as a data frame.

[1] * is a wildcard which says "all columns" here.

```
df = dbGetQuery(con, "SELECT * FROM diamonds;")
```

Having done this we can inspect the result in the same way as a normal data frame.

```
head(df)[,1:5]
#>   row.names carat       cut color clarity
#> 1         1  0.23     Ideal     E     SI2
#> 2         2  0.21   Premium     E     SI1
#> 3         3  0.23      Good     E     VS1
#> 4         4  0.29   Premium     I     VS2
#> 5         5  0.31      Good     J     SI2
#> 6         6  0.24 Very Good     J    VVS2
```

The `dbGetQuery()` function will send an SQL query to the database and return the result as a data frame. There is a similar `dbSendQuery()` which will send a query to the data base, but without trying to extract any results as an R object.

If we wanted to extract a single column

```
query = "
SELECT carat FROM diamonds;
"
df = dbGetQuery(con, query)
head(df)
#>   carat
#> 1  0.23
#> 2  0.21
#> 3  0.23
#> 4  0.29
#> 5  0.31
#> 6  0.24
```

or for multiple columns separate the field names by a comma

```
query = "
SELECT carat, price FROM diamonds;
"
df = dbGetQuery(con, query)
head(df)
#>   carat price
#> 1  0.23   326
#> 2  0.21   326
#> 3  0.23   327
#> 4  0.29   334
#> 5  0.31   335
#> 6  0.24   336
```

## *Data filtering*

The `DISTINCT` statement will pull out unique values in a variable

```
query = "
SELECT DISTINCT color FROM diamonds;
"
dbGetQuery(con,query)
#>   color
#> 1     F
#> 2     G
#> 3     H
#> 4     J
#> 5     E
#> 6     D
#> 7     I
```

The `WHERE` clause indicates that you wish to filter your result when a given condition is true. SQL standard supports the relation operators that you are familiar with from R programming with one difference to note. Since there is no `=` assignment operator in `SQL` the `==` R operator translates to only a single `=` in SQL. So if I wanted all diamonds that are color "D" from this data I would use

```
query = "
SELECT * FROM diamonds
  WHERE color = 'D';
"
df = dbGetQuery(con, query)
head(df)[,1:5]
#>   row.names carat       cut color clarity
#> 1        29  0.23 Very Good     D     VS2
#> 2        35  0.23 Very Good     D     VS1
#> 3        39  0.26 Very Good     D     VS2
#> 4        43  0.26      Good     D     VS2
#> 5        44  0.26      Good     D     VS1
#> 6        55  0.22   Premium     D     VS2
```

All diamonds that are at least 4 carats

```
query = "
SELECT * FROM diamonds
  WHERE carat >= 4;
"
df = dbGetQuery(con, query)
head(df)[,1:5]
```

```
#>   row.names carat       cut color clarity
#> 1     25999  4.01   Premium     I      I1
#> 2     26000  4.01   Premium     J      I1
#> 3     26445  4.00 Very Good     I      I1
#> 4     27131  4.13      Fair     H      I1
#> 5     27416  5.01      Fair     J      I1
#> 6     27631  4.50      Fair     J      I1
```

## *A note on syntax*

It is worth noting some syntactic elements about the SQL code. Unlike R, all statements in SQL must end with a `;`. The clauses are not case sensitive so `select * from diamonds;` yields the same as `SELECT * FROM diamonds;` however it is standard practice to capitalise the SQL clauses such that they are easier to distinguish from field and table names within the database. The new line is also not necessary in the above query, however again standard practice is to new line and indent for statements that have multiple clauses, much like you would in R programming.

## *Logical operators*

SQL supports the logical operators

- `AND` - conjunction - same as `&` in R,
- `OR` - disjunction - same as `|` in R,
- `NOT` - negation - same as `!`.

So we could pull out all rows of the data that are bigger than 1 carat in weight and of D colour

```
query = "
SELECT * FROM diamonds
  WHERE carat > 1
    AND color ='D';
"
```

```
df = dbGetQuery(con, query)
head(df)[,1:5]
#>   row.names carat       cut color clarity
#> 1       845  1.08   Premium     D      I1
#> 2      1555  1.01      Fair     D     SI2
#> 3      3172  1.01   Premium     D     SI2
#> 4      4296  1.11   Premium     D      I1
#> 5      4361  1.01   Premium     D     SI2
#> 6      4366  1.01 Very Good     D     SI2
```

From this point on, I will stop including the R code to run the query to save space, showing only the SQL query that was used to generate the results.

## *Pattern matching*

SQL has a special operator `LIKE` for use with a `WHERE` clause for searching a specific pattern. There is also a movies table in this data base which we could search for all film names matching certain string patterns.

```
dbListFields(con,"movies")
#>  [1] "row.names"   "title"
#>  [3] "year"        "length"
#>  [5] "budget"      "rating"
#>  [7] "votes"       "r1"
#>  [9] "r2"          "r3"
#> [11] "r4"          "r5"
#> [13] "r6"          "r7"
#> [15] "r8"          "r9"
#> [17] "r10"         "mpaa"
#> [19] "Action"      "Animation"
#> [21] "Comedy"      "Drama"
#> [23] "Documentary" "Romance"
#> [25] "Short"
```

A _ (underscore) is a wildcard character which can represent any *single* character that does not break the pattern.

```
SELECT * FROM movies
  WHERE title LIKE 'Se_en';
```

```
#>   row.names title year length   budget
#> 1     45331 Se7en 1995    127 30000000
#> 2     45892 Seven 1979    100       NA
```

`%` allows matching on multiple (or none) characters.

- `A%` will match all names that start with "A"
- `%a` will match all names ending in "a".
- You can use `%` both before and after a pattern.
- `LIKE` is not case sensitive

To find all films containing the word man we could

```
SELECT * FROM movies
  WHERE title LIKE '%man%';
```

```
#>   row.names                     title year
#> 1       131 1000 Convicts and a Woman 1971
#> 2       151     11 commandements, Les 2004
#> 3       216 18 anni tra una settimana 1991
#>   length budget
#> 1     92     NA
#> 2     85     NA
#> 3     98     NA
```

## *Organising results*

When extracting data from a table in the database, often we want to arrange the results in a certain way.[2]

[2] Of course we should know how to do this in R relatively trivially once we have the result as a data frame but that is beside the point.

The `ORDER BY` clause is one which allows you to arrange a result set by a column in increasing order.

```
SELECT title, rating, length FROM movies
  ORDER BY rating;
```

shows the films in order from worst to best according to their ratings

```
#>                  title rating length
#> 1         20/20 Vision      1     20
#> 2 Pizzaiolo et Mozzarel      1     85
#> 3  American Flatulators      1      3
```

We can switch the order if descending order is required by using `DESC` after the field by which we are sorting.

```
SELECT title, rating, length FROM movies
  ORDER BY rating DESC;
```

```
#>                                     title
#> 1 Dimensia Minds Trilogy: The Hope Factor
#> 2                       Summer Sonata, A
#> 3                       Fishing for Love
#>   rating length
#> 1     10     10
#> 2     10     30
#> 3     10      7
```

It is trivial to extend the result set sorting to multiple columns with extra fields separated by commas. So to sort the movies result set by best to worst film but in increasing length within that we would use:

```
SELECT title, rating, length FROM movies
  ORDER BY rating DESC, length;
```

```
#>                                     title
#> 1                        Fishing for Love
#> 2 Dimensia Minds Trilogy: The Hope Factor
#> 3                        Summer Sonata, A
#>   rating length
#> 1     10      7
#> 2     10     10
#> 3     10     30
```

We can also limit the size of the result set with the `LIMIT` clause

```
SELECT title, rating, length FROM movies
  ORDER BY rating DESC, length
  LIMIT 10;
```

Would return the same as above, but only the first ten results in the result set would be returned, rather than the full table.

# 3
# Sending statements that don't return a result set

Whilst extracting data that exists in a database is certainly useful it is by no means the only thing we typically want to use SQL for. Typically we want to be able to update and create new tables within the database to store new records or modify existing information.

## Creating a new table

Let's now imagine that we want to create a new table which will record shoppers purchases at a chain of stores. When we tell the database that we want a new table to be prepared for storing data we need to

- Give the table a name
- Give each variable in that table a name
- Describe the type of data that each variable contains

The `CREATE TABlE` clause is used to achieve this with the following syntax
`CREATE TABLE table_name (     column_1 data_type,      column_2 data_type,      column_3 data_type   );`
Declaring variable types isn't something that R programmers are typically used to, but the type names are fairly self explanatory. Below is a table of the more common data type declarations

| Type | Description |
|---|---|
| integer | 4 byte signed integer, -2,147,483,648 to 2,147,483,647 |
| double precision | same as R's numeric variables |
| bool | logical values, true or false |
| date | A date of the form YYYY-MM-DD |
| time | A value of the form HH:MM:SS |
| timestamp | A combination of date and time |
| text | variable length string |
| char(size) | A string of fixed size |

This is by no means an exhaustive list of data types but is definitely the most common. For our table let's say we want to store the date of a purchase, the purchase amount and a customer name. We could create our SQL statement as follows

```
statement = "
CREATE TABLE purchases (
  name text,
  date date,
  invoice double precision
);
"
```

To send arbitrary SQL commands from R we can use the `dbExecute()` function.[1] The function will execute the given query on the PostgreSQL database and return to R a numeric value describing how many rows have been affected.

[1] `dbGetQuery()` is for when we want to return a result set as a data frame.

```
dbExecute(con, statement)
#> [1] 0
```

We can see that the new table is now part of the database

```
dbListTables(con)
#>  [1] "test_table" "movies"     "diamonds"
#>  [4] "test"       "airlines"   "airports"
#>  [7] "flights"    "planes"     "weather"
#> [10] "purchases"
dbListFields(con, "purchases")
#> [1] "name"    "date"    "invoice"
```

## *Modifying a table*

### *Inserting new data*

New data can be inserted into an existing table with the `INSERT INTO` and `VALUES` clauses. We will populate our table with it's first piece of data

```
statement = "
INSERT INTO purchases (name, date, invoice)
  VALUES ('John','2017-01-01', 23.99);
"
dbExecute(con, statement)
#> [1] 1
dbGetQuery(con, "SELECT * FROM purchases;")
#>   name       date invoice
#> 1 John 2017-01-01      24
```

Multiple rows cal also be inserted

```
statement = "
INSERT INTO purchases (name, date, invoice)
  VALUES ('Fred','2017-01-02', 6.75),
         ('Carol','2016-12-25', 99.99);
"
dbExecute(con, statement)
#> [1] 2
```

```
dbGetQuery(con, "SELECT * FROM purchases;")
#>    name       date invoice
#> 1  John 2017-01-01   23.99
#> 2  Fred 2017-01-02    6.75
#> 3 Carol 2016-12-25   99.99
```

### *Modifying existing data*

We now realise that some data entry in our table is wrong, Carol didn't really go shopping on Christmas day, so we update that row of our data. The `UPDATE` and `SET` clauses allow us to do this

```
statement = "
UPDATE purchases
  SET date = '2016-12-26'
  WHERE name = 'Carol';
"
dbExecute(con, statement)
#> [1] 1
dbGetQuery(con, "SELECT * FROM purchases;")
#>    name       date invoice
#> 1  John 2017-01-01   23.99
#> 2  Fred 2017-01-02    6.75
#> 3 Carol 2016-12-26   99.99
```

We may also decide that an extra variable should have been recorded. Perhaps the number of items on the invoice. `ALTER TABLE` allows us to make modification to the purchases table in the database. Together with `ADD COLUMN` we can get the desired effect.

```
statement = "
ALTER TABLE purchases
  ADD COLUMN items integer;
"
dbExecute(con, statement)
#> [1] 0
dbGetQuery(con, "SELECT * FROM purchases;")
#>    name       date invoice items
#> 1  John 2017-01-01   23.99    NA
#> 2  Fred 2017-01-02    6.75    NA
#> 3 Carol 2016-12-26   99.99    NA
```

We could then populate our new field with some values using `INSERT INTO`.

# 4
# *Data manipulation using `dplyr`*

## *What is `dplyr`*

`dplyr` is a fantastic package for manipulating data frame structures[1]. It aims to make more involved manipulation easier by giving a collection of functions with consistent syntax where each function is aimed at doing one small task very well. The package focuses on easy to read and easy to use functions, the motivation being that a user spends more time worrying about data than they do about writing code. It is not part of the base R installation, so must be installed separately.

[1] To install the package, just run `install.packages("dplyr")`

```r
# library to load functions
library("dplyr")
```

## *Data partitioning (`filter()`)*

The `dplyr` package has a function `filter()` which we can use to replicate the data partitioning from the previous chapter.

To compare with the previous chapter we will use the same examples, previously we wrote

```r
ratings = movies$rating # pick out relevant column
good_films = ratings > 9 # perform the relation operation
sub_movies = movies[good_films,] # take the subset
```

using `filter()` from `dplyr` we could instead write

```r
sub_movies = filter(movies, rating > 9)
```

or

```r
## movies which < 120 minutes and released in 1994
sub_movies = movies[movies$length < 120 & movies$year == 1994,]
## becomes
sub_movies = filter(movies, length < 120 & year == 1994)
```

Usefully we now no longer need to write `movies$` every time we wish to refer to a particular column of the `movies` data.

### *summarise()*

`dplyr` contains lots of useful similar looking functions. For example we can use the `summarise()` function to apply a numerical summary to a variable (column) in our data frame.

```
# average length of films
summarise(movies, mean(length))
#> # A tibble: 1 x 1
#>   `mean(length)`
#>            <dbl>
#> 1           82.3
# median rating of films
summarise(movies, median(rating))
#> # A tibble: 1 x 1
#>   `median(rating)`
#>              <dbl>
#> 1              6.1
## multiple summaries & rename
summarise(movies,
          avg_length = mean(length),
          med_rating = median(rating))
#> # A tibble: 1 x 2
#>   avg_length med_rating
#>        <dbl>      <dbl>
#> 1       82.3        6.1
```

### *The piping operator*

All `dplyr` functions follow a consistent syntax

```
function_name(datasetname, variable conditions)
```

and always return another data frame[2]

[2] Strictly it returns a tibble which is a particular type of data frame with a more useful print method and are slightly more efficient.

Apart from making the structure consistent this has another useful consequence. That is we can make use of an operator called the piping operator `%>%` which feeds forward a result as the first argument of the function. Again the motivation here is legibility when chaining together multiple commands, we could first `filter()` then `summarise()`

```
movies %>%
  filter(ratings > 9) %>%
  summarise(mean(length))
#> # A tibble: 1 x 1
#>   `mean(length)`
#>            <dbl>
#> 1           64.2
```

Because `%>%` passes forward the result as the first argument of the next function we don't need to store intermediate results, or refer to the data frame after the beginning. The idea is that this is "easy to read" as "take the movies data, filter it this way then summarise it that way".

If I wanted the mean and standard deviation of movie ratings for all PG rated films I could

```
movies %>%
  filter(mpaa == "PG") %>%
  summarise(mean = mean(rating), sd = sd(rating))
#> # A tibble: 1 x 2
#>    mean    sd
#>   <dbl> <dbl>
#> 1  5.61  1.47
```

## *group_by()*

The final `dplyr` function that we will examine here is `group_by()`. Imagine I also now want means and standard deviations of all the other film certifications. I could run similar code for "PG", "PG-13" .... But the author of `dplyr` knows this is the sort of thing we often want to do, so provides the `group_by()` function to take care of this for us.

```
movies %>%
  group_by(mpaa)
```

Notice that `group_by()` by itself doesn't change the appearance of our data, all that it has done is create a categorical structure behind the scenes. We can see the fruits of our labour when applying the `summarise()` function to the grouped data.

```
movies %>%
  group_by(mpaa) %>%
  summarise(mean(length))
#> # A tibble: 5 x 2
#>    mpaa `mean(length)`
#>   <chr>          <dbl>
#> 1                 80.6
#> 2 NC-17          110.2
#> 3    PG           97.4
#> 4 PG-13          105.0
#> # ... with 1 more rows
```

Everything we do after the `group_by` statement (using `dplyr` functions) will happen to each unique group according to that variable.

This chaining together of commands can make it much easier to do fairly advanced data subsetting and summarising. For example let's say that I want to know about films that are a reasonable length, say less than 2.5 hours, for which we know information on the budgets, and I want to know how the average ratings, lengths and budgets all compare for Action and non Action films among the four film certification ratings.

We can take the steps and put them into some order:

| Function | purpose |
|---|---|
| `filter()` | Extracting subsets of data |
| `arrange()` | Re–order your data by sorting on (a) given column(s) |
| `select()` | Choose specific columns from the data |
| `rename()` | Rename specific columns from the data |
| `distinct()` | Keep only unique rows of the data |
| `mutate()` | Add a new column to the data, can be formed from calculation |
| `sample_n()` | Take a random sample from your data |
| `count()` | Tally the number of observations in each group of data |

Table 4.1: Useful dplyr functions

1. keep only films for where we know about the budget and it fits length criteria (`filter()`),
2. group our data into the relevant subsets (`group_by()`),
3. calculate summary statistics (`summarise()`).

```
movies %>%
  filter(length < 150 & !is.na(budget)) %>%
  group_by(mpaa, Action) %>%
  summarise(avglength = mean(length),
            avgrating = mean(rating),
            avgbudget = mean(budget))
#> # A tibble: 9 x 5
#> # Groups:   mpaa [?]
#>    mpaa Action avglength avgrating avgbudget
#>   <chr>  <int>     <dbl>     <dbl>     <dbl>
#> 1            0      85.1      6.27   4335842
#> 2            1      96.7      5.88  12204388
#> 3 NC-17      0      99.7      5.95   7897833
#> 4    PG      0      98.2      5.80  31858683
#> # ... with 5 more rows
```

This is something that would be very laborious to do using only the base R subsetting and functions that you have already seen.

Whilst it can be a bit tricky getting to grips with thinking about data manipulation using `dplyr` I fully believe that it is easier to get what you want than using base R's functions and definitely worth the effort.

There are other `dplyr` functions which we have not had time to explore here but they tend to all look and work in the same sort of way. See for example table4.1 for a short (certainly not exhaustive) list. Another bonus of getting used to `dplyr` and tibbles is that it allows you to interact with other types of data. For example you can connect directly to SQL data bases and retain all of the nice R syntax and data manipulation, but act on your relational data.

## *dplyr and databases*

One of `dplyr`'s lesser known benefits is the ability to connect to a database back end for it's data source. This is great because it allows us to manipulate data in a way that is familiar to us, without worrying too much about how the data is stored.

To connect dplyr to a data base we need to make sure that `dbplyr` is installed. This package takes care of the interface between R and the database back-end. It can be installed in the usual way.

```
install.packages("dbplyr")
```

To connect `dplyr` to the data base

```
library(dplyr)
user = "user.name"
pass = "user.password"
dbname = "user.db"
dcon = src_postgres(dbname = dbname, user = user, password = pass)
```

then we can register an object for a particular table.

```
db_diamonds = tbl(dcon, "diamonds")
```

Now that we have done this, we have created a connection to the data in the database and an object in R. On the surface it looks like `db_diamonds` is just another data frame object. However, the data does not live in memory as an R object. This makes this a suitable approach to dealing with big data sets in R. We have an object that lets us manipulate (via dplyr) a dataframe looking object, that is actually just a reference to data on an SQL server (which may well not be local).

We can see that this is a bit different purely by printing out the object

```
db_diamonds
#> # Source:   table<diamonds> [?? x 11]
#> # Database: postgres 9.5.9
#> #   [jamie@localhost:5432/test]
#>   row.names carat     cut color clarity
#>       <chr> <dbl>   <chr> <chr>   <chr>
#> 1         1  0.23   Ideal     E     SI2
#> 2         2  0.21 Premium     E     SI1
#> 3         3  0.23    Good     E     VS1
#> 4         4  0.29 Premium     I     VS2
#> # ... with more rows, and 6 more variables:
#> #   depth <dbl>, table <dbl>, price <int>,
#> #   x <dbl>, y <dbl>, z <dbl>
```

Note that whilst we can see a few rows here, the data is not being stored in memory. The full size of the data is not known to R at this point. This remains true when we perform operations on our data set too. We get an answer shown to us in the form of a few rows, but data and operations are on the database. To pull the data into R we need to use the `collect()` function.

```
db_diamonds %>% collect
#> # A tibble: 53,940 x 11
#>   row.names carat     cut color clarity
#> *     <chr> <dbl>   <chr> <chr>   <chr>
#> 1         1  0.23   Ideal     E     SI2
#> 2         2  0.21 Premium     E     SI1
#> 3         3  0.23    Good     E     VS1
#> 4         4  0.29 Premium     I     VS2
#> # ... with 5.394e+04 more rows, and 6 more
#> #   variables: depth <dbl>, table <dbl>,
#> #   price <int>, x <dbl>, y <dbl>, z <dbl>
```

# 5 Multiple tables

Very often data is distributed across more than one table. Consider some data on flights in the US. One table might contain information on all the flights for a given day, including the operator that ran the flight. A second table might contain information on the flight carriers.

We will add some data to our database to recreate this situation. The package `nycflights13` has a number of data sets that concern flight times, operators etc. We can grab these data sets from the R package to our database with the following code.

```r
library(nycflights13)

## remember to make the connection to your database first (see chapter 1)
dbWriteTable(con, "airlines",airlines,overwrite = TRUE, row.names = FALSE)
#> [1] TRUE
dbWriteTable(con,"airports",airports, overwrite = TRUE, row.names = FALSE)
#> [1] TRUE
dbWriteTable(con, "flights", flights, overwrite = TRUE, row.names = FALSE)
#> [1] TRUE
dbWriteTable(con, "planes", planes, overwrite = TRUE, row.names = FALSE)
#> [1] TRUE
dbWriteTable(con, "weather", weather, overwrite = TRUE, row.names = FALSE)
#> [1] TRUE
```

To give some context, `flights` contains information on what flights have been sent during the year with a variable which has an abbreviation of the carrier. The `airlines` table has a collection of carrier abbreviations and the corresponding full names. It is common in analyses that we have multiple tables of data that relate to something similar, for which we want flexible tools to combine them.

We can create our dplyr connection to each table as well:

```r
airlines = tbl(con, "airlines")
airports = tbl(con, "airports")
flights = tbl(con,"flights")
planes = tbl(con, "planes")
weather = tbl(con, "weather")
```

## Outer joins

Joins are operations which take place between two or more tables to combine data from each together. Outer joins are the most common type of join of which there are three types. All 3 of these are supported by SQL standard directly. For each of the joins below we will see both the `dplyr` and the `SQL` code to achieve the result. When a row does not have a corresponding match in an outer join, a missing value is registered.

### Left join

A left join is the most common join type. It joins a left table to a right table where all rows of the left table are kept, but only rows in the right table which have a match on the joining variables will be kept. It is the most common because it guarantees that no records from your primary table (left) are excluded from the result set.

`dplyr` makes this sort of join trivial with the `left_join()` function. The first two arguments are the names of the left and right tables to be joined. By default the join will take place on all variables that have a common name in both the left and right. This can be changed however, see `?left_join` for extra details.

```
flights %>% left_join(airlines)
#> Joining, by = "carrier"
#> # Source:   lazy query [?? x 20]
#> # Database: postgres 9.5.9
#> #   [jamie@localhost:5432/test]
#>    year month   day dep_time sched_dep_time
#>   <int> <int> <int>    <int>          <int>
#> 1  2013     1     1      517            515
#> 2  2013     1     1      533            529
#> 3  2013     1     1      542            540
#> 4  2013     1     1      544            545
#> # ... with more rows, and 15 more variables:
#> #   dep_delay <dbl>, arr_time <int>,
#> #   sched_arr_time <int>, arr_delay <dbl>,
#> #   carrier <chr>, flight <int>,
#> #   tailnum <chr>, origin <chr>, dest <chr>,
#> #   air_time <dbl>, distance <dbl>,
#> #   hour <dbl>, minute <dbl>,
#> #   time_hour <dttm>, name <chr>
```

In this case since both tables have a “carrier” column, the join is performed on this column.

This type of join is also supported by the SQL standard. `LEFT JOIN` is the clause that we require.

```
SELECT * FROM flights
  LEFT JOIN airlines
  USING (carrier);
```

The `USING (name)` syntax says use column "carrier" from each of the left and right tables to perform the join on. Where `dplyr` uses default behavior on omission of the variable names, this is not possible in SQL and will result in an error and no result set being returned.

## *Right join*

A right join is very similar to a left join, except that all rows in the right table are kept and only rows in the left table which match on the joining variables will be returned.

```
airlines %>% right_join(flights)
```

returns the same as

```
flights %>% left_join(airlines)
```

but with the columns in a different order.

In SQL we could achieve this as

```
SELECT * FROM airlines
  RIGHT JOIN flights
  USING (carrier);
```

## *Full join*

A full join will include all observations from both the left and the right table. This is a little easier to see with a smaller example.

```
df1 = data.frame(x = c(1,2), y = 2:1)
df2 = data.frame(x = c(1,3), a = 10, b = "a")
df1
#>   x y
#> 1 1 2
#> 2 2 1
df2
#>   x  a b
#> 1 1 10 a
#> 2 3 10 a
df1 %>% full_join(df2)
#> Joining, by = "x"
#>   x  y  a    b
#> 1 1  2 10    a
#> 2 2  1 NA <NA>
#> 3 3 NA 10    a
```

The corresponding SQL clause for this type of join is `FULL JOIN` and follows the same syntax as above.

For completion the left and right joins given the small data frames in the previous example.

```
df1 %>% left_join(df2)
#> Joining, by = "x"
#>   x y  a    b
#> 1 1 2 10    a
#> 2 2 1 NA <NA>
df1 %>% right_join(df2)
#> Joining, by = "x"
#>   x  y  a b
#> 1 1  2 10 a
#> 2 3 NA 10 a
```

When the names of the joining variables do no match, we can specify which variables from each table should be joined on. For example

```
colnames(df1) = c("jr","y")
df1 %>% left_join(df2, by = c("jr" = "x"))
#>   jr y  a    b
#> 1  1 2 10    a
#> 2  2 1 NA <NA>
```

`dplyr` will automatically choose one of the copies of the joined variables to show in the results. I.e not both "x" and "jr" are shown here. This is not the case for SQL. To achieve the same result we would want

```
SELECT jr,y,a,b FROM
  df1 LEFT JOIN df2 ON
  df1.jr = df2.x;
```

- Note that we are being specific in both which variables the result set should contain, and the variable names in the tables that should be joined on with syntax `tablename.columnname`. If our `SELECT` clause here was `SELECT *` we would have both column "jr" and column "x" in our results set.

## *Inner join*

An inner join on two tables will return a result set that has only rows that have an exact match in both the left table and the right table.

```
df1 %>% inner_join(df2, by = c("jr" = "x"))
#>   jr y  a b
#> 1  1 2 10 a
```

This type of join is often referred to as a simple join and can be executed in SQL as

```
SELECT jr, y, a, b FROM
  df1 JOIN df2 ON
  df1.jr = df2.x;
```

## *Other joins*

The four join types above are specified in the SQL standard and are therefore implemented in PostgreSQL. `dplyr` however additionally has some other join functions. These are not implemented directly as

PostgreSQL clauses. For example a `semi_join()` will keep rows of the left table that have a match in the right table. The consequence being that the result set is a subset of the left table.

```
df1 %>% semi_join(df2, by = c("jr" = "x"))
#>   jr y
#> 1  1 2
```

`dplyr` does come with a very useful function when it comes to dealing with databases. That is `show_query()`. Since there is no direct equivalent in PostgreSQL of a semi join, but dplyr allows us to do one, we could use the `show_query()` function to find what SQL code will perform the appropriate action.

```
airlines %>% semi_join(flights) %>% show_query()
#> Joining, by = "carrier"
#> <SQL>
#> SELECT * FROM "airlines" AS "TBL_LEFT"
#>
#> WHERE EXISTS (
#>  SELECT 1 FROM "flights" AS "TBL_RIGHT"
#>  WHERE ("TBL_LEFT"."carrier" = "TBL_RIGHT"."carrier")
#> )
```