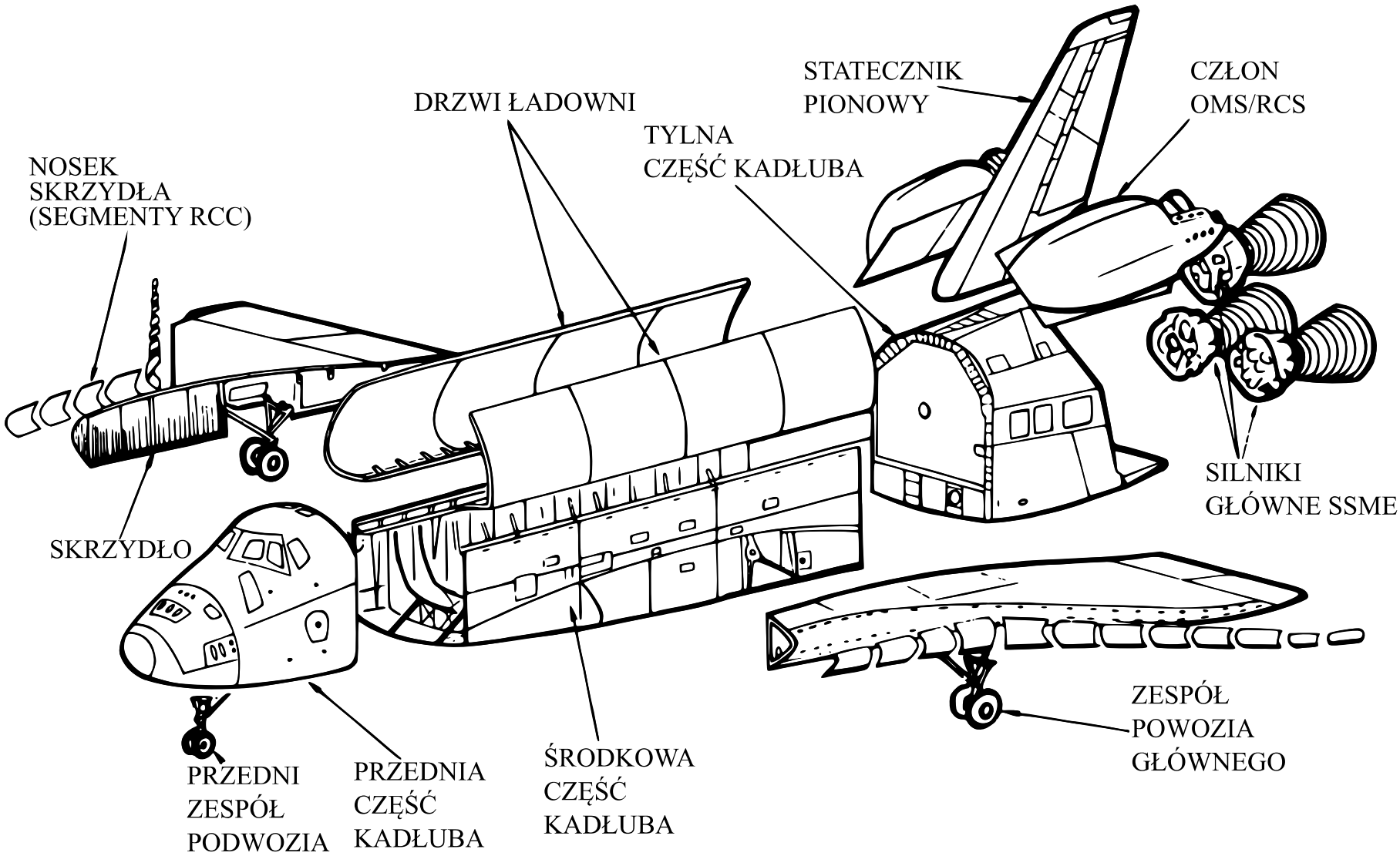
Unleashing SQL

with R

Daniel Fryer

NZSSN

2022/02/18


NOSEK
SKRZYDŁA
(SEGMENTY RCC)
DRZWI ŁADOWNI
STATECZNIK
PIONOWY
TYLNA
CZĘŚĆ KADŁUBA
CZŁON
OMS/RCS
SILNIKI
GŁÓWNE SSME
SKRZYDŁO
ZESPÓŁ
POWOZIA
GŁÓWNEGO
PRZEDNI
ZESPÓŁ
PODWOZIA
PRZEDNIA
CZĘŚĆ
KADŁUBA
ŚRODKOWA
CZĘŚĆ
KADŁUBA

The fundamentals

Why learn R?

Community!

- Nerds
- Statisticians
- Data analysts
- Social scientists
- Psychologists
- Bioinformaticians
- Medical researchers
- Programmers
- Artists
- Machine learners
- Industry
- ...

State of the art statistical analysis **packages**.

Free, open source, transparent.

Tons of help and inspiration:

- Twitter
- Blogs
- Educators
- Books
- Galleries
- Journals
- Conferences
- CRAN

Powerful and capable:

- Reproducible research!
- Make slides (xaringan)
- Automatic reports (RMarkdown)
- Dashboards (Shiny)
- Interactive plots (plotly)
- Used in industry

It's from New Zealand!

Why learn R with SQL?

Keep your data organised.

Understand tidy data better.

Understand the tidyverse better.

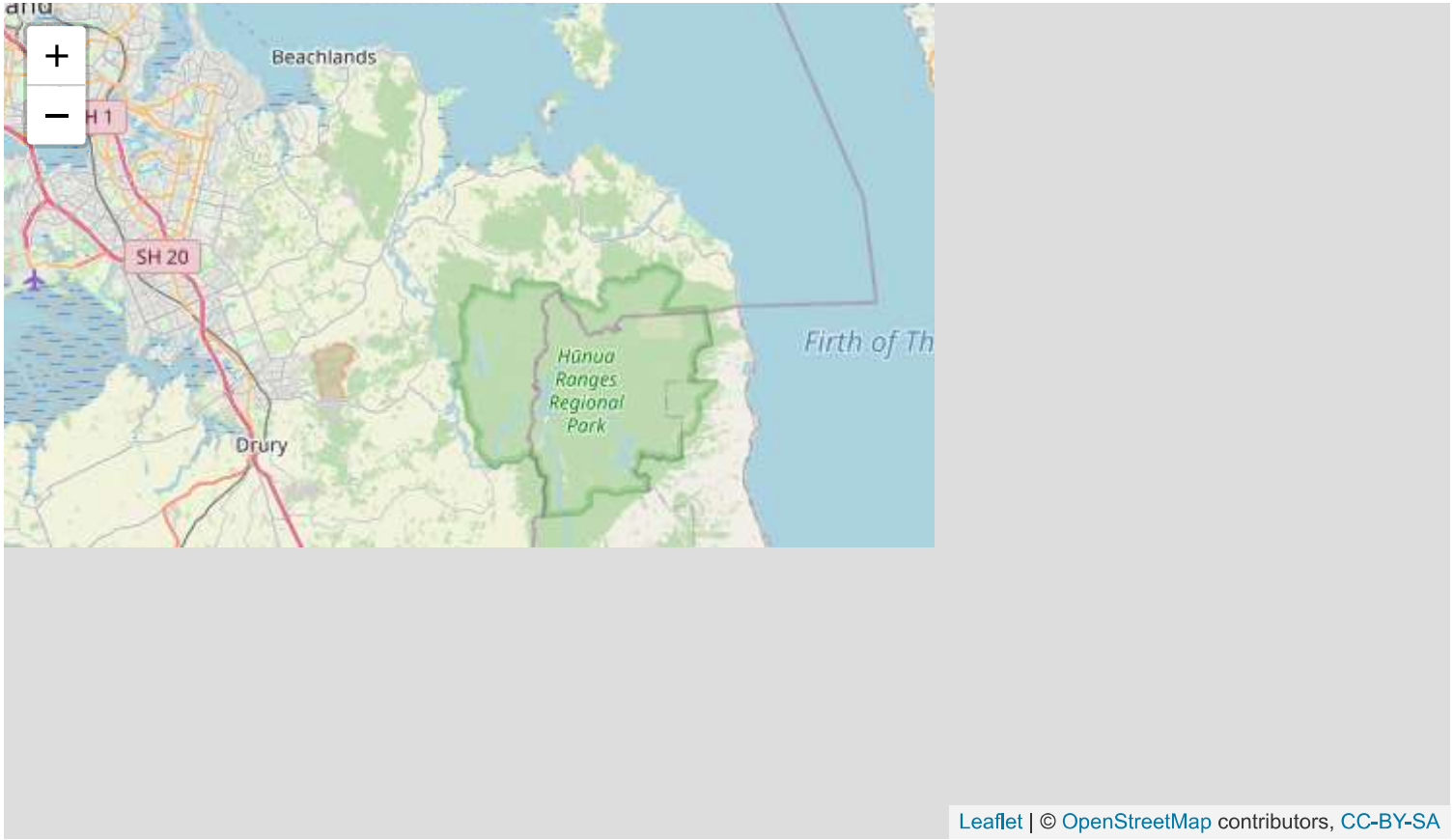
Work with remote servers in large collaborations.

Combine all your CSV files into a single file (SQLite).

Search and query your combined CSV files with ease (DB Browser).

Showing off

```
library(leaflet)
leaflet() %>% addTiles() %>% setView(174.76898, -36.85231, zoom = 10)
```



Project management 

Scripts are essentially text files that you save your code to.

Every **project** should have its own folder (usually with subfolders).

A single project typically involves:

- Preparation, analysis, presentation, data.

The **working directory** should always be the **project root**.

Your **workspace** is the RStudio environment, including all the variables you have created, and your 'history'. *Do not rely on this*.

"*The source code is real. The objects are realizations of the source code.*"

Where do you put your data?

In the data folder!

- Raw data
- Processed data

Always think about a new person opening your project directory for the first time. Will it work seamlessly? Will they know how to use it?

Restart your session frequently.

```
umbrella
│   README.md
│   start-here.R
│
└──data
│   │
│   └──raw
│   └──processed
│
└──prepare
│   │   cleaning-functions.R
│
└──analyse
│   │   analysis-functions.R
│
└──present
    │
    └──figures
    └──unleashing-SQL
```

Umbrella: a simple example project

Click here to open

R is a calculator

```
# a boring calculation
5 + 7
```

```
# [1] 12
```

A calculator that can save variables

```
x <- 5 + 7
x*2
```

```
## [1] 24
```

In all sorts of ways

```
x <- c(1,2,3,4,5)
x*2
```

```
## [1]  2  4  6  8 10
```

R has lists

These are very powerful

```
Friends <- list(
  id = c(1,2,3),
  FirstName = c('X','Y','Z'),
  LastName = c('A','B','C'),
  FavColour = c('red', 'blue', NA)
)
Friends
```

```
## $id
## [1] 1 2 3
##
## $FirstName
## [1] "X" "Y" "Z"
##
## $LastName
## [1] "A" "B" "C"
##
## $FavColour
## [1] "red"  "blue" NA
```

But the output looks kind of weird.

Let's make the lists look better

```
Friends <- data.frame(Friends)
Friends
```

```
##   id FirstName LastName FavColour
## 1  1         X        A       red
## 2  2         Y        B      blue
## 3  3         Z        C      <NA>
```

🤔 *Ummmm... can we do better?*

```
library(tibble)
Friends <- tibble(Friends)
Friends
```

```
## # A tibble: 3 x 4
##      id FirstName LastName FavColour
##   <dbl> <chr>     <chr>    <chr>
## 1     1 X         A        red
## 2     2 Y         B        blue
## 3     3 Z         C        <NA>
```

🧠 *Wow, that's informative!*

But *can we do better?*

```
library(DT)
datatable(Friends)
```

Show 10 entries Search:

	id	FirstName	LastName	FavColour
1	1	X	A	red
2	2	Y	B	blue
3	3	Z	C	

Showing 1 to 3 of 3 entries Previous 1 Next

A better dataset

Show 8 entries Search:

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.1	3.5	1.4	0.2	setosa
6	5.4	3.9	1.7	0.4	setosa
7	4.6	3.4	1.4	0.3	setosa
9	4.4	2.9	1.4	0.2	setosa
12	4.8	3.4	1.6	0.2	setosa
15	5.8	4	1.2	0.2	setosa
17	5.4	3.9	1.3	0.4	setosa
18	5.1	3.5	1.4	0.3	setosa

Showing 1 to 8 of 50 entries

Previous 1 2 3 4 5 6 7 Next

🦜 Yet underneath, 'tis still but a list 🦜

```
str(Friends)
```

```
## tibble [3 x 4] (S3: tbl_df/tbl/data.frame)
##  $ id       : num [1:3] 1 2 3
##  $ FirstName: chr [1:3] "X" "Y" "Z"
##  $ LastName : chr [1:3] "A" "B" "C"
##  $ FavColour: chr [1:3] "red" "blue" NA
```

Lists, precious lists

- 🤖 SQL stores data in a way that is great for machines
- 👨‍👩‍👧 R stores data in a way that is great for programmers

Everything is...

Everything in R is either **data** or a **function**. We refer to 'things' as 'objects'.

The contents of lists can be accessed with `$`

```
Friends$FavColour
```

```
## [1] "red"  "blue" NA
```

So, what is `$`?

Well, if `$` is not data, then it's a function.

```
`$`(Friends, FavColour)
```

```
## [1] "red"  "blue" NA
```

Just a sneaky function.

Here's another function for accessing things! `[`

```
Friends[1,4]
```

```
## # A tibble: 1 x 1
##   FavColour
##   <chr>
## 1 red
```

Thinking in vectors

Here's a vector of `TRUE` / `FALSE`

```
x <- c(TRUE, FALSE, TRUE)
```

We can use it to get friends.

```
Friends[x,]
```

```
## # A tibble: 2 x 4
##      id FirstName LastName FavColour
##   <dbl> <chr>     <chr>    <chr>
## 1     1 X         A        red
## 2     3 Z         C        <NA>
```

```
Friends[!x,]
```

```
## # A tibble: 1 x 4
##      id FirstName LastName FavColour
##   <dbl> <chr>     <chr>    <chr>
## 1     2 Y         B        blue
```

Packages make functions and data

But we can (and should) make our own functions.

```
# A function that checks if x is 'blue'
is_x_blue <- function(x) {
  x == 'blue'
}
```

And we should use them often.

```
is_x_blue('green')
```

```
## [1] FALSE
```

```
is_x_blue('blue')
```

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

We can use the argument explicitly, if we like.

```
is_x_blue(x='red')
```

```
## [1] FALSE
```

Functions are building blocks.

The longer code gets, the harder it is to think about.

Do you prefer this?

```
spec <- iris$Species
iris_setosa <- iris[spec == "setosa", ]
m <- lm(Sepal.Length ~ Petal.Length, data=iris_setosa)
coef(summary(m))
```

Or this?

```
# Get the iris Setosa species
iris_setosa <- where_species_is_setosa(iris)

# Fit Model (1), and get results
results <- fit_linear_model(iris_setosa, number = 1)

# Print results
results
```

The pipe operator %>%

```
# Get the iris Setosa species
iris_setosa <- where_species_is_setosa(iris)

# Fit Model (1), and get results
results <- fit_linear_model(iris_setosa, number = 1)

# Print results
results
```

Many people prefer this instead:

```
# Get Setosa species and fit Model (1)
iris %>%
  where_species_is_setosa() %>%
  fit_linear_model(number = 1)
```

The pipe comes from  `magrittr`

```
library(magrittr)
```


tidyverse
www.rstudio.com

Tidy data

"*Happy families are all alike;*
every unhappy family is unhappy in its own way"
- Leo Tolstoy.

"*Tidy datasets are all alike,*
but every messy dataset is messy in its own way"
- Hadley Wickham.

"*The principles of tidy data are closely tied to those of relational databases... but are framed in a language familiar to statisticians*" [1]

[1] Wickham, H (2014), *Tidy Data*, The R Journal

Tidy data principles

1. Every variable is a column.
2. Every observation is a row.
3. Every cell holds a single (atomic) value.

Conditions 1 and 2 are often referred to as **long format**.

Advantages

- Datasets can be read and understood universally.
- Easier for R packages to work together.
- Enables SQL-like operations.

🕵️ Messy data

The R package 📦 `tidyr` has functions for tidying messy data.

And it also has some datasets for us to play with.

```
library(tidyr)
relig_income
```

Show 3 entries Search:

	religion	<$10k	$10-20k	$20-30k	$30-40k	$40-50k
1	Agnostic	27	34	60	81	76
2	Atheist	12	27	37	52	35
3	Buddhist	27	21	30	34	33

Showing 1 to 3 of 18 entries

Previous 1 2 3 4 5 6 Next

🧹 Cleaning `relig_income`

This kind of dataset is sometimes referred to as **wide format**.

📦 `tidyr` gives us `pivot_longer`

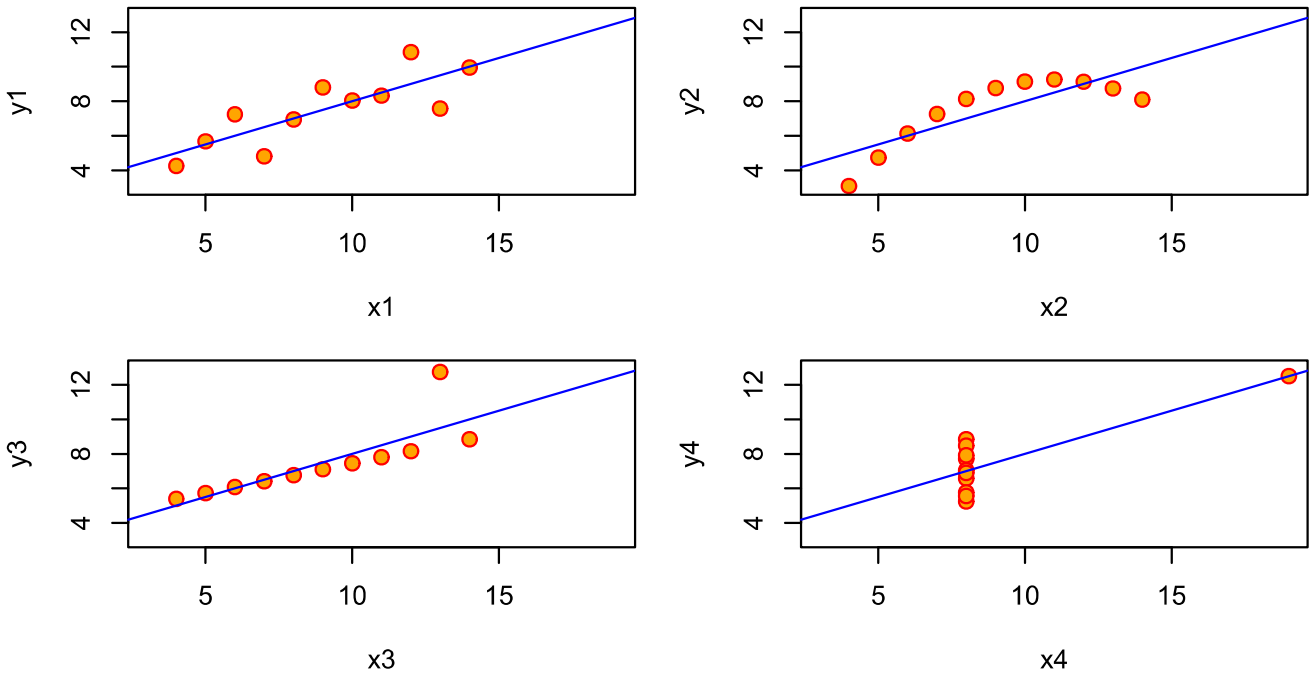
```
pivot_longer(
  data = relig_income,
  cols = !religion,
  names_to = "income",
  values_to = "count"
)
```

Show 3 entries Search:

	religion	income	count
1	Agnostic	<$10k	27
2	Agnostic	$10-20k	34
3	Agnostic	$20-30k	60

Showing 1 to 3 of 180 entries

Messy data no.2: Anscombe's Quartet



Anscombe's Quartet

Show 6 entries Search:

	x1	x2	x3	x4	y1	y2	y3	y4
1	10	10	10	8	8.04	9.14	7.46	6.58
2	8	8	8	8	6.95	8.14	6.77	5.76
3	13	13	13	8	7.58	8.74	12.74	7.71
4	9	9	9	8	8.81	8.77	7.11	8.84
5	11	11	11	8	8.33	9.26	7.81	8.47
6	14	14	14	8	9.96	8.1	8.84	7.04

Showing 1 to 6 of 11 entries Previous 1 2 Next

Cleaning Anscombe's Quartet

```
pivot_longer(
  data = anscombe,
  cols = everything(),
  names_to = c(".value", "set"),
  names_pattern = "(.)(.)"
)
```

Show 4 entries Search:

	set	x	y
1	1	10	8.04
2	2	10	9.14
3	3	10	7.46
4	4	8	6.58

Showing 1 to 4 of 44 entries

Previous 1 2 3 4 5 … 11 Next

Getting connected to SQL

For MySQL and T-SQL, see
this guide
covering local and remote connections from R

The wonderful SQLite

SQLite is:

- Small
- Fast
- Self contained
- High reliability
- Full-featured

The most used database engine in the world.

Around one trillion active databases in use.

Hundreds of SQLite databases on any smart phone.

One of the top 5 most widely deployed software modules of any kind.


SQLite

Connect or create, one line!

First get our directories organised (see umbrella).

```
library(here)
```

Now, create or connect

```
library(RSQLite)
con <- dbConnect(
  SQLite(),
  here("data", "raw", "Sandpit.sqlite")
)
```

If it can't be found, then `Sandpit.sqlite` *will be created (empty).*

Use the Sandpit database

The package  `dplyr` gives us a whole 'grammar of data manipulation'.

The package  `dbplyr` allows `dplyr` to talk to SQL.

```
library(dplyr)
library(dbplyr)
```

So many packages! From now on I'll write them like this:

```
dplyr::tbl()
```

The `dplyr::` means we are using a function from `dplyr`. The function we're using is called `tbl`.

Connect to a table

```
banana <- dplyr::tbl(con, "Ape_Banana")
banana
```

```
## # Source:   table<Ape_Banana> [?? x 7]
## # Database: sqlite 3.36.0
## #   [D:\CloudDrive\OneDrive\Cloud-drive\Teach\Courses\Workshops\SQL\repositories\umbrella
##    BananaID TasteRank DatePicked DateEaten  Ripe TreeID Comments
##       <int>     <int>      <int>     <int> <int>  <int> <chr>
##  1        1         2   20181003  20181004     0      1 <NA>
##  2        2         4   20181003  20181004     1      2 <NA>
##  3        3         4   20181003  20181004     1      2 <NA>
##  4        4         5   20181003  20181006     1      1 <NA>
##  5        5         5   20181003  20181006     1      2 best banana ever
##  6        6         3   20181003  20181004     1      2 <NA>
##  7        7         2   20181002  20181004     0      3 <NA>
##  8        8         5   20181002  20181005     1      3 smooth and delectable
##  9        9         3   20181002  20181003     1      4 <NA>
## 10       10         3   20181002  20181003     1      5 <NA>
## # ... with more rows
```

Grammar of data manipulation  SQL

The function `dplyr::filter` is like the SQL `WHERE` clause.

```
ripe_banana <- banana %>% dplyr::filter(Ripe == 1)
ripe_banana
```

```
## # Source:   lazy query [?? x 7]
## # Database: sqlite 3.36.0
## #   [D:\CloudDrive\OneDrive\Cloud-drive\Teach\Courses\Workshops\SQL\repositories\umbrella
##    BananaID TasteRank DatePicked DateEaten  Ripe TreeID Comments
##       <int>     <int>      <int>     <int> <int>  <int> <chr>
##  1        2         4   20181003  20181004     1      2 <NA>
##  2        3         4   20181003  20181004     1      2 <NA>
##  3        4         5   20181003  20181006     1      1 <NA>
##  4        5         5   20181003  20181006     1      2 best banana ever
##  5        6         3   20181003  20181004     1      2 <NA>
##  6        8         5   20181002  20181005     1      3 smooth and delectable
##  7        9         3   20181002  20181003     1      4 <NA>
##  8       10         3   20181002  20181003     1      5 <NA>
##  9       12         5   20181002  20181005     1      4 <NA>
## 10       16         5   20181001  20181004     1      5 a culinary delight
## # ... with more rows
```

Grammar of data manipulation  SQL

But seriously, `dplyr::filter` is *actually* the `WHERE` clause.

```
ripe_banana %>% dplyr::show_query()
```

```
## <SQL>
## SELECT *
## FROM `Ape_Banana`
## WHERE (`Ripe` = 1.0)
```

To execute the query:

```
ripe_banana %>% dplyr::collect()
```

```
## # A tibble: 34 x 7
##    BananaID TasteRank DatePicked DateEaten  Ripe TreeID Comments
##       <int>     <int>      <int>     <int> <int>  <int> <chr>
##  1        2         4   20181003  20181004     1      2 <NA>
##  2        3         4   20181003  20181004     1      2 <NA>
##  3        4         5   20181003  20181006     1      1 <NA>
##  4        5         5   20181003  20181006     1      2 best banana ever
##  5        6         3   20181003  20181004     1      2 <NA>
##  6        8         5   20181002  20181005     1      3 smooth and delectable
##  7        9         3   20181002  20181003     1      4 <NA>
##  8       10         3   20181002  20181003     1      5 <NA>
##  9       12         5   20181002  20181005     1      4 <NA>
```

We can write our own SQL

```
DBI::dbGetQuery(con,
"
SELECT TreeID, COUNT(*) AS NumRipe, AVG(TasteRank) AS AvgTaste
FROM Ape_Banana
WHERE DatePicked > '20180101'
GROUP BY TreeID;
"
)
```

Show 4 entries Search:

	TreeID	NumRipe	AvgTaste
1	1	2	3.5
2	2	7	3.71428571428571
3	3	3	3.33333333333333
4	4	3	3

Showing 1 to 4 of 18 entries

Previous 1 2 3 4 5 Next

It's up to you

```
tasty_bananas <- banana %>%
  dplyr::filter(DatePicked > '20180101') %>%
  dplyr::group_by(TreeID) %>%
  dplyr::summarise(NumRipe = n(), AvgTaste = mean(TasteRank, na.rm=T))
```

We can always inspect the SQL code created by `dplyr`.

```
tasty_bananas %>% show_query()
```

```
## <SQL>
## SELECT `TreeID`, COUNT(*) AS `NumRipe`, AVG(`TasteRank`) AS `AvgTaste`
## FROM `Ape_Banana`
## WHERE (`DatePicked` > '20180101')
## GROUP BY `TreeID`
```

Advanced: programmatically edit queries

```
for (this_taste in c(3,4,5)) {
  res <- DBI::dbGetQuery(con, stringr::str_interp("
    SELECT *
    FROM Ape_Banana
    WHERE TasteRank = ${this_taste}
  "))
  cat("\nResults for taste = ", this_taste, "\n")
  print(nrow(res))
}
```

```
##
## Results for taste =  3
## [1] 7
##
## Results for taste =  4
## [1] 10
##
## Results for taste =  5
## [1] 18
```

Advanced no.2: batch process results

Send the query without executing it.

```
rs <- DBI::dbSendQuery(con, "
  SELECT *
  FROM Ape_Banana
")
```

Process results, 20 at a time

```
while (!DBI::dbHasCompleted(rs)) {
  twenty_bananas <- DBI::dbFetch(rs, n = 20)

  # << insert processing on twenty_bananas here >>

  print(nrow(twenty_bananas))
}
```

```
## [1] 20
## [1] 20
## [1] 10
```

Saving results

Collect the results

```
tasty_bananas <- dplyr::collect(tasty_bananas)
```

Save as a CSV ( `readr` is better at it).

```
library(readr)

# Choose the processed data directory
location <- here("data", "processed", "tasty_bananas.csv")

# Write CSV
readr::write_csv(tasty_bananas, location)
```

Saving results

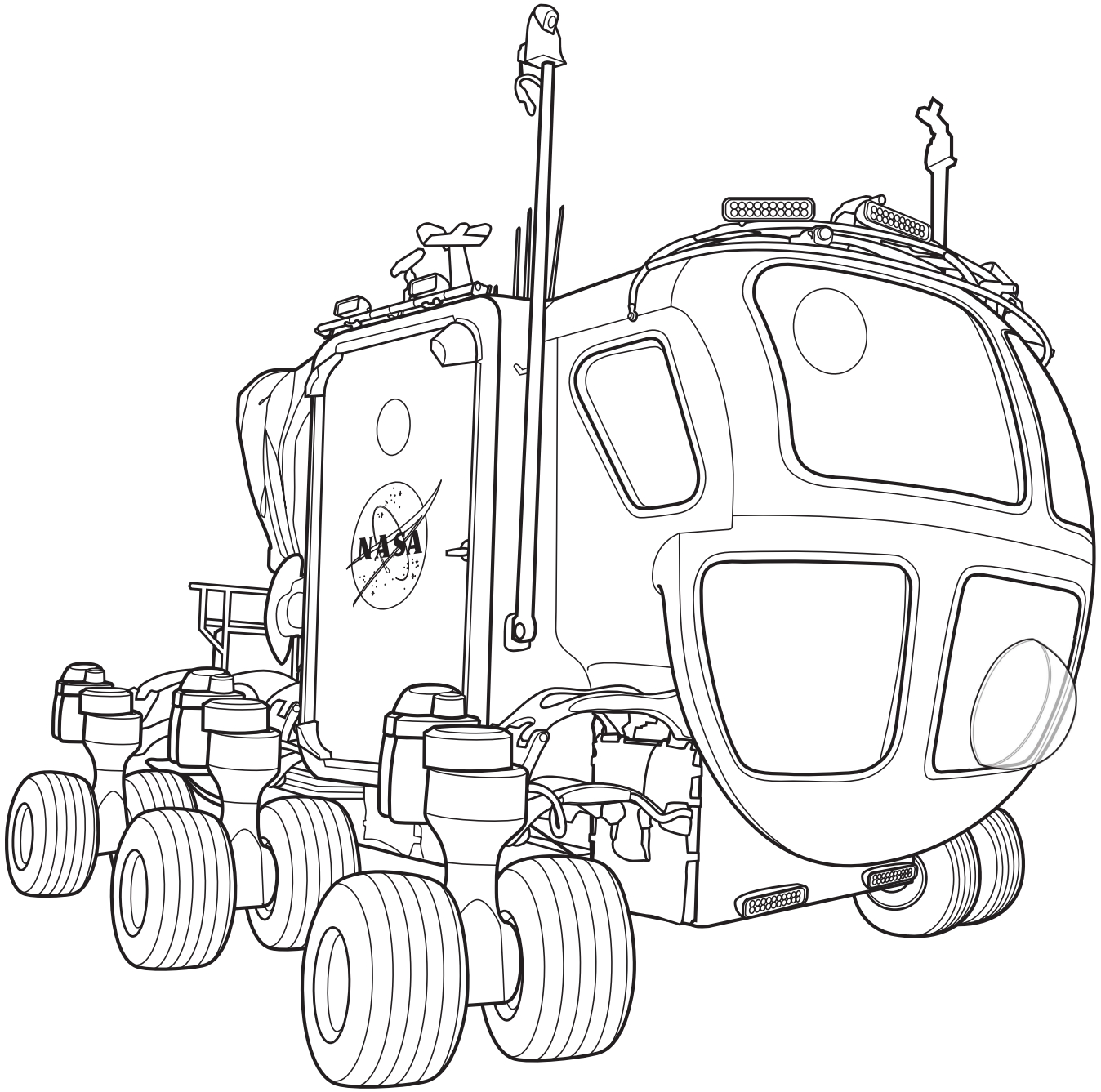
Or save to SQLite.

```
# Choose the processed data directory
location <- here("data", "processed", "Sandpit_results.sqlite")

# Connect or create database
res_con <- DBI::dbConnect(RSQLite::SQLite(), location)

# Save the table
DBI::dbWriteTable(res_con, "tasty_bananas", tasty_bananas)
```

Don't forget to disconnect

When you're done.

```
DBI::dbDisconnect(con)
DBI::dbDisconnect(res_con)
```


NASA

Live guide and demonstrations

Live guide and demonstrations

- Using GitHub
- Downloading umbrella repository
- Using DB Browser with SQLite
- Live demo of connecting and exploring

Also, see the R code folder on the course repo.