THE LONDON SCHOOL OF ECONOMICS AND POLITICAL SCIENCE

# Week 3: Tabular Data

MY472: Data for Data Scientists
https://lse-my472.github.io/

Autumn Term 2025

**Ryan Hübert**
Associate Professor of Methodology

Slides last updated: 14 October 2025

# Setting the scene

Goal of data science: move from data to information

Last week, we focused on technical aspects of how data is represented in a digital format

This week, we move further down the chain from data to information

- What (conceptual) types of data do data scientists work with?
- A common “shape” of data: tabular data
- Working with tabular data

# Outline

# Types of data

Almost all data science involves **numerical** data

- ▷ This is data that is represented as numbers

This does *not* mean that everything is quantitative

- ▷ **Quantitative**: Capturing a *quantity* as a continuous or discrete variable
- ▷ **Qualitative**: Capturing a *quality* as a categorical variable

We will convert almost all data to numerical data

# Types of data

**Continuous (numerical) data** takes values within a range

- **Interval**: meaningful differences, arbitrary zero (e.g., temperature)
- **Ratio**: meaningful differences and meaningful (“absolute”) zero (e.g., weight)

**Discrete data** only takes specific values, often whole numbers

- **Count**: non-negative integers representing number of occurrences
- **Ordinal**: numbers (usually integers) with meaningful order, but no meaningful difference between values (e.g., rankings)
- **Nominal**: categories with no inherent order (e.g., colours)
- **Binary**: special case of nominal data with only two categories

# A motivating example: peaches

Source: The Today Show, Peach Benefits

# A motivating example: peaches

There are many varieties of peaches

- ▷ **Peach variety** is qualitative data, e.g. Donut, Nectarine, White

Their quality differs in many ways

- ▷ **Colour, taste, fuzziness** are all qualitative data

But they also differ in quantitatively measurable ways

- ▷ **Price per peach** is quantitative data, e.g. £1, £0.55, £1.15

Some ways they differ can be qualitative or quantitative

- ▷ **Size** can be qualitative (e.g., small or large), or quantitative (e.g., average weight in kg)

# Shapes of data

When a data scientist works with data, it comes in a “shape”

peach.json

```json
{
    "Nectarine" : {
        "colour" : "red",
        "taste" : "sweet",
        "price_gbp" : 0.75,
        "average_weight_g" : 140
    },
    "Donut" : {
        "colour" : "yellow",
        "taste" : "tangy",
        "price_gbp" : 1,
        "average_weight_g" : 100
    }
}
```

# Shapes of data

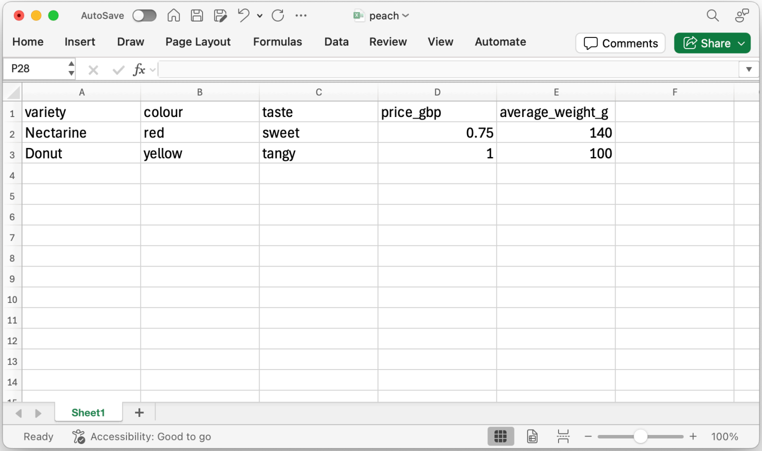
There are many shapes of data, e.g.:

- Raw text: unstructured sequences of characters (week 10)
- Key-value or array structures: semi-structured data such as JSON (week 5)
- Hierarchical or tree-structured data: formats expressing nested relationships such as HTML and XML (week 7)
- Geometric or spatial data: coordinates, shapes (week 4)

The most common shape of data is **tabular**

- Tabular data is arranged in *tables* with rows and columns
- Often called “datasets,” “data frames,” etc.
- Many data shapes are converted to tabular data for analysis

# Peaches as tabular data



| variety | colour | taste | price_gbp | average_weight_g |
|---|---|---|---|---|
| Nectarine | red | sweet | 0.75 | 140 |
| Donut | yellow | tangy | 1 | 100 |

# Units and observations

**Units** are the entities or subjects being studied

- ▷ E.g., individuals, countries, companies

An **observation** is a single “peek” at a unit under specific conditions, such as in a time period

In a **cross-sectional dataset**, units = observations

- ▷ Take a slice (cross-section) at a single point in time

In a **hierarchical dataset**, units $\neq$ observations

- ▷ Each unit can have multiple observations, e.g. **time series** and **longitudinal (panel) data**

In tabular data, *rows are observations*

# Features (variables)

**Features** are attributes of a unit specified for each observation

These are also called **variables** since their values *vary* depending on the observation

In tabular data, *all columns are features*

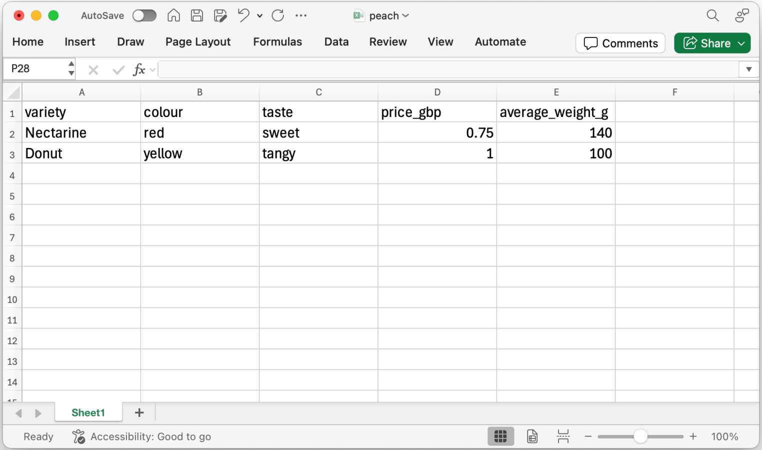
- They could be quantitative or qualitative, or merely identifiers

All tabular datasets should have a **primary key**: a variable which uniquely identifies each observation

- This could be implicit (a combination of two or more variables) or explicit, like a unique ID number

# Features (variables)

What is the primary key for this tabular data?




| variety | colour | taste | price_gbp | average_weight_g |
|---|---|---|---|---|
| Nectarine | red | sweet | 0.75 | 140 |
| Donut | yellow | tangy | 1 | 100 |

# Outline

1. Types and shapes of data
2. Tabular data in R
3. Transforming, summarising and manipulating data
4. Tidy data
5. Databases

# From concepts to practice: tabular data in R

R has special object types for tabular data

- A **matrix** object, such as `matrix(1:6, nrow=2)`
- A **array** object, such as `array(1:8, dim=c(2,2,2))`
- A **data frame** object, such as `data.frame(a=1:3, b=4:6)`

Matrices and arrays:

- must contain homogenous object types
- differ in dimensions: matrices are 2D, arrays can be an arbitrary number of dimensions

Data frames can contain heterogenous object types

- Data frames “look like” matrices (2-dimensional), but they are technically named `list()` objects in R

# From concepts to practice: tabular data in R

We'll develop ideas around a hypothetical peach seller in the UK

- Suppose she has some sales data written in her notebook
- Can create tabular data on the fly in R using `data.frame()`

```
sales = data.frame(
  variety = c("Donut", "Nectarine", "White"),
  colour = c("Yellow", "Red", "White"),
  kg_sold = c(150, 100, 120),
  gbp_per_kg = c(4.4, 7.75, 5)
)
print(sales)
```

```
    variety colour kg_sold gbp_per_kg
1     Donut Yellow     150       4.40
2 Nectarine    Red     100       7.75
3     White  White     120       5.00
```

# From concepts to practice: tabular data in R

Note the heterogeneous column types by using `str()`

```
str(sales)
```

```
'data.frame':	3 obs. of  4 variables:
 $ variety   : chr  "Donut" "Nectarine" "White"
 $ colour    : chr  "Yellow" "Red" "White"
 $ kg_sold   : num  150 100 120
 $ gbp_per_kg: num  4.4 7.75 5
```

Each column is a variable that

- ▷ Has a name
- ▷ Is a vector of same length (4)
- ▷ Has homogenous object types

Tabular data in R can have “standard” object types (`num`, `chr`, etc.), or more specialised types (`factor`, `date`, etc.)

# Data frames in the tidyverse

We will mostly use the **tidyverse** collection of packages to work with tabular data

- These packages contain a bunch of useful tools; it's worth familiarising yourself at `https://www.tidyverse.org/`
- You can use base R for your own work, but your assignments must replicate the tidyverse way of doing things
- Can load all the packages with `library("tidyverse")`

In the tidyverse, tabular data is stored in a `tibble` object

- Differences between base R `data.frame` and `tibble` are somewhat in-the-weeds
- `tibble` is mostly "backward compatible" with `data.frame`

# Data frames in the tidyverse

Data for the hypothetical peach seller, now in a tibble

```r
library("tidyverse")
sales = tibble(
  variety = c("Donut", "Nectarine", "White"),
  colour = c("Yellow", "Red", "White"),
  kg_sold = c(150, 100, 120),
  gbp_per_kg = c(4.4, 7.75, 5)
)
print(sales)
```

```
# A tibble: 3 x 4
  variety   colour kg_sold gbp_per_kg
  <chr>     <chr>    <dbl>      <dbl>
1 Donut     Yellow     150       4.4
2 Nectarine Red        100       7.75
3 White     White      120       5
```

# Storing tabular data

Common file formats for storing tabular data:

- Comma-separated values (`.csv`) – ubiquitous and simple
  - Each *line* is an observation
  - Each variable value is separated by a comma
- Application specific (proprietary) formats (.dta, .sav, .xlsx etc.)
  - Can allow for richer representations including meta-data
  - More complex, and not necessarily human-readable
  - Can cause problems (for example lost Covid-19 data)

Often choice is dictated by the source (and size) of the data

Packages like `{haven}` allow for reading in non-csv formats

# Reading and writing tabular data in R

We can use the `{readr}` package to:

1. Write tabular data to our computer's storage device

```
library("readr")
write_csv(sales, "data/peach_sales.csv")
```

2. Read tabular data to our computer's storage device

```
sales <- read_csv("data/peach_sales.csv")
```

With base R, you can use `write.csv()` and `read.csv()`

# Adding data

You can add data—either columns or rows—by “binding” them

- In the tidyverse: `bind_cols()` and `bind_rows()`
- In base R: `rbind()` and `cbind()`

Suppose the peach sales data was from a specific date (1st July 2025), and she wants to indicate this

```
sales <- bind_cols(sales, date = "2025-07-01")
print(sales)
```

```
# A tibble: 3 x 5
  variety   colour kg_sold gbp_per_kg date
  <chr>     <chr>    <dbl>      <dbl> <chr>
1 Donut     Yellow     150       4.4  2025-07-01
2 Nectarine Red        100       7.75 2025-07-01
3 White     White      120       5    2025-07-01
```

# Adding data

Suppose she wants to add data in her notebook from another date (1st August 2025)

```
sales2 = tibble(
  variety = c("Donut", "Nectarine", "White"),
  colour = c("Yellow", "Red", "White"),
  kg_sold = c(140, 200, 60),
  gbp_per_kg = c(4, 7.5, 5.1),
  date = "2025-08-01"
)
sales <- bind_rows(sales, sales2)
sales
```

```
# A tibble: 6 x 5
  variety   colour kg_sold gbp_per_kg date
  <chr>     <chr>    <dbl>      <dbl> <chr>
1 Donut     Yellow     150       4.4  2025-07-01
2 Nectarine Red        100       7.75 2025-07-01
3 White     White      120       5    2025-07-01
4 Donut     Yellow     140       4    2025-08-01
5 Nectarine Red        200       7.5  2025-08-01
6 White     White       60       5.1  2025-08-01
```

# Dealing with dates

Dates are challenging — my advice:

- Always try to use ISO 8601 format for dates: YYYY-MM-DD
- Sometimes, even safer: YYYYMMDD format
- Read and write as `chr`; convert to `date` format for analysis only
- Avoid editing `.csv` files in Excel (or other GUIs)

# Dealing with dates

Suppose the seller saves her data to `.csv`

```
write_csv(sales, "data/peach_sales.csv")
```

She opens the file in Excel to look at something and it autosaves
Next time she imports it, she gets January dates:

```
sales <- read_csv("data/peach_sales.csv")
sales
```

```
# A tibble: 6 x 5
  variety   colour kg_sold gbp_per_kg date
  <chr>     <chr>    <dbl>      <dbl> <chr>
1 Donut     Yellow     150       4.4  7/1/25
2 Nectarine Red        100       7.75 7/1/25
3 White     White      120       5    7/1/25
4 Donut     Yellow     140       4    8/1/25
5 Nectarine Red        200       7.5  8/1/25
6 White     White       60       5.1  8/1/25
```

# Dealing with dates

The tidyverse includes a great package called `{lubridate}`

- If dates are your thing, check out the docs

Assuming I know the correct order of day and month:

```
sales$date <- lubridate::mdy(sales$date)
sales$date
```

```
[1] "2025-07-01" "2025-07-01" "2025-07-01" "2025-08-01"
[5] "2025-08-01" "2025-08-01"
```

Can re-save, and also use `date` format directly for analysis:

```
sales$date + 14
```

```
[1] "2025-07-15" "2025-07-15" "2025-07-15" "2025-08-15"
[5] "2025-08-15" "2025-08-15"
```

# Dealing with qualitative data

Tabular datasets usually contain qualitative data

- Here: `variety` and `colour` are both qualitative

Often you want to leave these as is

- Here: `variety` is more like an unique identifier

But if you want to do statistical analysis, you will need to convert to numeric data

- One approach: convert to `factor` variable
- A much better approach: create **dummy variables**

R automatically converts `factor` variables to dummies when, e.g., running regressions—get in habit of doing it yourself!

# Dealing with qualitative data

```r
sales$colour_Yellow <- ifelse(sales$colour == "Yellow", 1, 0)
sales$colour_Red <- ifelse(sales$colour == "Red", 1, 0)
sales$colour_White <- ifelse(sales$colour == "White", 1, 0)
sales$colour <- NULL # remove colour column, as no longer needed
sales[,c("variety", "colour_Yellow", "colour_Red", "colour_White")]
```

```
# A tibble: 6 x 4
  variety   colour_Yellow colour_Red colour_White
  <chr>             <dbl>      <dbl>        <dbl>
1 Donut                 1          0            0
2 Nectarine             0          1            0
3 White                 0          0            1
4 Donut                 1          0            0
5 Nectarine             0          1            0
6 White                 0          0            1
```

# Outline

# Wrangling data in base R

To work with data in base R, we will typically have to manipulate objects directly:

```r
# add a new variable
sales$revenue <- sales$kg_sold * sales$gbp_per_kg
# keep only these columns
sales <- sales[, c("variety", "revenue")]
# sort by revenue in descending order
sales <- sales[order(-sales$revenue), ]

head(sales)
```

```
    variety revenue
5 Nectarine    1500
2 Nectarine     775
1     Donut     660
3     White     600
4     Donut     560
6     White     306
```

# Wrangling data in tidyverse

Tidy R gives us an alternative approach

`{dplyr}` gives us useful and literal tools for wrangling data in R:

- `mutate()`: Add or modify variables in a data frame
- `select()`: Choose specific columns from a data frame
- `filter()`: Subset rows based on conditions
- `arrange()`: Sort rows by one or more variables
- and many more (also see other tidyverse packages)

# Wrangling data in tidyverse

Using the pipe `|>` (or `%>%`) allows us to chain operations:

```
sales <- read_csv("data/peach_sales.csv")
sales |>
  mutate(revenue = kg_sold * gbp_per_kg) |> # add variable
  select(variety, revenue) |>               # select columns
  arrange(desc(revenue))                    # sort
```

```
# A tibble: 6 x 2
  variety   revenue
  <chr>       <dbl>
1 Nectarine    1500
2 Nectarine     775
3 Donut         660
4 White         600
5 Donut         560
6 White         306
```

# Grouping and hierarchies

Sometimes data has a nested structure, such as:

1. Repeated observations of the same units:
   - ▷ each observation is *nested* under a single unit
2. Hierarchical data:
   - ▷ each unit is *nested* under a higher-level unit (cluster)
3. Binned data:
   - ▷ each observation is *nested* under a bin based on a variable

Might want to restructure data given a hierarchy

# Grouping in tidyverse

Consider data where each *unit* belongs to some hierarchy

Suppose the peach seller has a dataset of each peach's weight

```
# Simulate a hypothetical dataset (each peach = unit)
peach_weights <- data.frame(
  variety = sample(c("Donut", "Nectarine", "White"),
                   100, replace = TRUE),
  weight = runif(100, 0.5, 1.5)
)
head(peach_weights) # show first six rows
```

```
    variety    weight
1     Donut 0.9634350
2 Nectarine 1.4178032
3     Donut 0.7276818
4     White 1.4602638
5 Nectarine 0.7056252
6     White 1.1796567
```

Unit = a peach; higher-level cluster = variety

# Grouping in tidyverse

We can group by a higher variable and summarise across that variable:

```
# Group by variety and summarise:
peach_weights |>
  group_by(variety) |>            # variable on which to group
  summarise(                      # summarise across grouping var
    count = n(),                  # counts observations
    total_weight = sum(weight),   # apply a function (sum(x))
    mean_weight = mean(weight)    # a different function (mean(x))
  )
```

```
# A tibble: 3 x 4
  variety   count total_weight mean_weight
  <chr>     <int>        <dbl>       <dbl>
1 Donut        34         31.3       0.921
2 Nectarine    30         30.9       1.03
3 White        36         36.9       1.03
```

# Reshaping in R

Now consider data with multiple *observations* per *unit*

Suppose the seller has some data on yield per variety over time

```r
# Simulate a hypothetical dataset
# (variety = unit, observed over 25 time periods)
peach_panel <- tibble(
  variety = rep(c("Donut", "Nectarine", "White"), 25),
  year = rep(2000:2024, each = 3),
  yield = runif(75, 50, 200)
)
head(peach_panel)
```

```
# A tibble: 6 x 3
  variety    year yield
  <chr>     <int> <dbl>
1 Donut      2000  89.1
2 Nectarine  2000  94.5
3 White      2000  79.1
4 Donut      2001 182.
5 Nectarine  2001 100.
6 White      2001  91.2
```

# Merges and joins

We often have multiple datasets with “related” data that we want to **join** (or **merge**) together

Tables are joined/merged on columns that appear in each table

- Columns appearing in all tables to be joined are called **keys**

All joins will create a new table with the columns from the tables being joined but they differ on what *rows* they keep, e.g.:

- **Inner join**: keep only rows with matching keys in both tables
- **Left (right) join**: keep all rows from the left (right) table, and any matching rows from the right (left) table
- **Full join**: keep all rows from both tables

**Always check your data after joins/merges!**

# Merges and joins

The seller has a chart with the various culinary qualities of different peach varieties, which she enters into R

```
peach_features = tibble(
  variety = c("Donut", "Redhaven", "White"),
  taste = c("Tangy", "Sweet", "Sweet"),
  fuzziness = c("Fuzzy", "Fuzzy", "Fuzzy")
)
peach_features
```

```
# A tibble: 3 x 3
  variety  taste fuzziness
  <chr>    <chr> <chr>
1 Donut    Tangy Fuzzy
2 Redhaven Sweet Fuzzy
3 White    Sweet Fuzzy
```

Suppose she eventually wants to analyse how her sales of peach varieties correlates with culinary features

# Merges and joins

So, she needs to merge the peach feature data into her sales data

- The **key** is `variety`, which is in both tables
- A left join (with `sales` on left) makes most sense here (why?)

```
sales |>
  select(variety, kg_sold, gbp_per_kg) |> # only needs sales data
  left_join(peach_features, by = "variety") # join on 'variety'
```

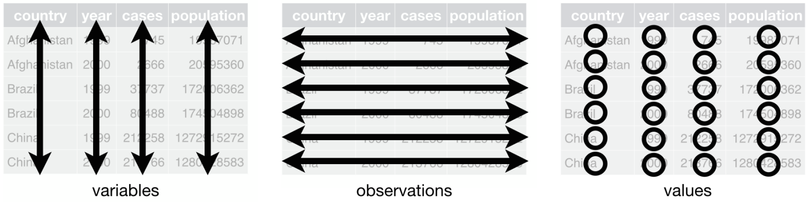
```
# A tibble: 6 x 5
  variety   kg_sold gbp_per_kg taste fuzziness
  <chr>       <dbl>      <dbl> <chr> <chr>
1 Donut         150       4.4  Tangy Fuzzy
2 Nectarine     100       7.75 <NA>  <NA>
3 White         120       5    Sweet Fuzzy
4 Donut         140       4    Tangy Fuzzy
5 Nectarine     200       7.5  <NA>  <NA>
6 White          60       5.1  Sweet Fuzzy
```

# Outline

# Tabular data should be tidy

**Tidy data** is data that follows three rules:

1. Each observation is a row
2. Each variable is a column
3. Each cell is a value




Source: Hadley Wickham, Data Tidying

# What does “tidy” data look like in R?

The seller’s original sales data is tidy

```r
sales
```

```
# A tibble: 6 x 5
  variety   colour kg_sold gbp_per_kg date
  <chr>     <chr>    <dbl>      <dbl> <date>
1 Donut     Yellow     150       4.4  2025-07-01
2 Nectarine Red        100       7.75 2025-07-01
3 White     White      120       5    2025-07-01
4 Donut     Yellow     140       4    2025-08-01
5 Nectarine Red        200       7.5  2025-08-01
6 White     White       60       5.1  2025-08-01
```

# What can go wrong?

**Untidy example 1:** columns represent values of a variable

```r
untidy1
```

```
# A tibble: 3 x 3
  variety   `2025-07-01` `2025-08-01`
  <chr>            <dbl>        <dbl>
1 Donut              150          140
2 Nectarine          100          200
3 White              120           60
```

Note:

- Bad (in part) because we don't know what the data is
- Variable names are also bad (see backticks??)

# How to fix it?

To “tidy” `untidy1`: **pivot** columns using `{tidyr}` function

```
pivot_longer(data = untidy1,
             cols = c(`2025-07-01`, `2025-08-01`),
             names_to = "date",
             values_to = "kg_sold")
```

```
# A tibble: 6 x 3
  variety   date       kg_sold
  <chr>     <chr>        <dbl>
1 Donut     2025-07-01     150
2 Donut     2025-08-01     140
3 Nectarine 2025-07-01     100
4 Nectarine 2025-08-01     200
5 White     2025-07-01     120
6 White     2025-08-01      60
```

Specifically: pivoted from **wide** to **long** format

# What else can go wrong?

**Untidy example 2:** observations scattered across multiple rows

```r
untidy2
```

```
# A tibble: 12 x 5
   variety   colour date       var        value
   <chr>     <chr>  <date>     <chr>      <dbl>
 1 Donut     Yellow 2025-07-01 kg_sold   150   
 2 Donut     Yellow 2025-07-01 gbp_per_kg   4.4 
 3 Nectarine Red    2025-07-01 kg_sold   100   
 4 Nectarine Red    2025-07-01 gbp_per_kg   7.75
 5 White     White  2025-07-01 kg_sold   120   
 6 White     White  2025-07-01 gbp_per_kg   5   
 7 Donut     Yellow 2025-08-01 kg_sold   140   
 8 Donut     Yellow 2025-08-01 gbp_per_kg   4   
 9 Nectarine Red    2025-08-01 kg_sold   200   
10 Nectarine Red    2025-08-01 gbp_per_kg   7.5 
11 White     White  2025-08-01 kg_sold    60   
12 White     White  2025-08-01 gbp_per_kg   5.1 
```

# How to fix it?

To “tidy” `untidy2`: **pivot** those rows into a new pair of columns

```
pivot_wider(data = untidy2,
            names_from = "var", values_from = "value")
```

```
# A tibble: 6 x 5
  variety   colour date       kg_sold gbp_per_kg
  <chr>     <chr>  <date>       <dbl>      <dbl>
1 Donut     Yellow 2025-07-01     150       4.4
2 Nectarine Red    2025-07-01     100       7.75
3 White     White  2025-07-01     120       5
4 Donut     Yellow 2025-08-01     140       4
5 Nectarine Red    2025-08-01     200       7.5
6 White     White  2025-08-01      60       5.1
```

Specifically: pivoted from **long** to **wide** format

# Why care?

Data exists in service of producing useful information

- Untidy data obscures informational content of data

But sometimes untidy data is appropriate

- Dummy variables are not tidy, but it's okay!
- Long formats (e.g. `untidy2`) can be useful for storage/memory management (very wide data frames are computationally taxing)

Best practice:

- Store data as simple plain-text in tidy format when possible
- Convert to untidy on the fly, only when needed

# Outline

# Databases

**Database system**: an organized collection of data that is stored and accessed via a computer

- The way a database is organized is a **schema**
- Since a database is used for data *storage*, a user typically “reads” and “writes” to a database
- Access data via **queries**
- Queries are often constructed/written in **domain-specific languages** like SQL, but not always
- A user can typically read and write via `R` (or `python`)

# Types of databases

**Relational databases**

- data is stored in multiple tables to avoid redundancy
- tables are linked based on common **keys**
- SQL is dominant DSL used to access data

**Non-relational databases**

- data stored in a way that is not based on tabular relations
- Data is accessed using a wide variety of (sometimes customised) languages

# SQL: Structured Query Language

SQL is a **domain specific language (DSL)** designed to define, control access to, manipulate, and query *relational* databases

- Pronounced both "S-Q-L" and "SEQUEL"

Unlike R, it is a **nonprocedural/declarative language**: user defines what to do, inputs, and outputs, but not the control flow

- How the statement is executed is left to the **optimizer**, which is opaque to the user
- How long SQL queries depends on optimization
- Performance will vary, but generally faster than standard data frame manipulation in R (and much more scalable)

# SQL and tidyverse

You just learned how to work with, and manipulate, tabular data using `{tidyverse}`, which is *conceptually* identical to SQL

Many SQL queries "resemble" `{tidyverse}` functions, e.g.:

| SQL | `{tidyverse}` |
|---|---|
| `SELECT column1` | `select(column1)` |
| `FROM table` | `table \|> ...` |
| `WHERE condition` | `filter(condition)` |
| `GROUP BY column` | `group_by(column)` |
| `ORDER BY column` | `arrange(column)` |
| `LIMIT n` | `slice_head(n = n)` |
| `SUM()`, `COUNT()`, `AVG()` | `summarize()` with `sum()`, `n()`, `mean()` |
| `LEFT JOIN`, `INNER JOIN`, etc | `left_join()`, `inner_join()`, etc |

- ▷ Every SQL query needs at least `SELECT` and `FROM`
- ▷ Result of both SQL queries and `{dplyr}` pipelines is a table

# SQL query examples

Table 1 named `client`

```
# A tibble: 3 x 5
     id name    gender billed account_id
  <dbl> <chr>   <chr>   <dbl>      <dbl>
1     1 Alice   F         500        101
2     2 Bob     M         750        102
3     3 Charlie F         200        103
```

Table 2 named `account`

```
# A tibble: 3 x 2
     id balance
  <dbl>   <dbl>
1   101    5000
2   102    3000
3   103    7000
```

# SQL query examples

Returns a table with `name`, `account_id` columns of `client`:

```
SELECT name, account_id FROM client;
```

The `{tidyverse}` equivalent:

```
client |>
  select(name, account_id)
```

```
# A tibble: 3 x 2
  name    account_id
  <chr>        <dbl>
1 Alice          101
2 Bob            102
3 Charlie        103
```

# SQL query examples

Returns a table with all columns of `client` but only rows where the `gender` variable is “F”:

```
SELECT * FROM client WHERE gender = 'F';
```

The `{tidyverse}` equivalent:

```
client |>
  filter(gender == "F")
```

```
# A tibble: 2 x 5
     id name    gender billed account_id
  <dbl> <chr>   <chr>   <dbl>      <dbl>
1     1 Alice   F         500        101
2     3 Charlie F         200        103
```

# SQL query examples

This returns a table with two columns, `total_billed` and `avg_billed` and one row giving the total billed and average billed amounts for female clients in `client` table:

```
SELECT SUM(billed) AS total_billed,
      AVG(billed) AS avg_billed
FROM client
WHERE gender = 'F';
```

The `{tidyverse}` equivalent:

```
client |>
  filter(gender == "F") |>
  summarise(total_billed = sum(billed),
            avg_billed = mean(billed))
```

```
# A tibble: 1 x 2
  total_billed avg_billed
         <dbl>      <dbl>
1          700        350
```

# SQL join examples

This returns a table with two columns `name` and `balance` created by inner joining tables `client` and `account` by their shared keys, `account_id` and `id`:

```
SELECT client.name, account.balance
FROM client JOIN account
ON client.account_id = account.id;
```

The `{tidyverse}` equivalent:

```
client |>
  inner_join(account, by = c("account_id" = "id")) |>
  select(name, balance)
```

```
# A tibble: 3 x 2
  name    balance
  <chr>     <dbl>
1 Alice      5000
2 Bob        3000
3 Charlie    7000
```