Applied Data Science with R: Working with Relational Data

Matthias Haber

06 March 2019

Last week's Homework

Load packages

```r
library(tidyverse)
library(nycflights13)
```

Carrier with longest average dep delay accounting for distance

```
flights %>%
  group_by(carrier) %>%
  mutate(delay = dep_delay / distance) %>%
  summarise(delay = mean(delay, na.rm =T)) %>%
  arrange(desc(delay)) %>%
  head(n=1)
```

```
## # A tibble: 1 x 2
##   carrier  delay
##   <chr>    <dbl>
## 1 YV      0.0618
```

Flights delayed by >=2h, made up >1h in flight

```
flights %>%
  filter(dep_delay >= 120, dep_delay-arr_delay > 60) %>%
  n_distinct()
```

```
## [1] 14
```

Destination with second largest spread of distance

```
flights %>%
  group_by(dest) %>%
  summarise(spread = sd(distance)) %>%
  top_n(2, spread)
```

```
## # A tibble: 2 x 2
##   dest  spread
##   <chr>  <dbl>
## 1 EGE     10.5
## 2 SAN     10.3
```

Carrier with larges arrival delay in summer

```r
flights %>%
  filter(month %in% c(6,7,8)) %>%
  group_by(carrier) %>%
  summarise(delay = mean(arr_delay, na.rm =TRUE)) %>%
  filter(delay >=20) %>%
  n_distinct()
```

```
## [1] 4
```

Average departure delay of AA

```
flights %>%
  filter(carrier =="AA") %>%
  summarise(delay = round(mean(dep_delay, na.rm =TRUE)))
```

```
## # A tibble: 1 x 1
##   delay
##   <dbl>
## 1     9
```

Plane with greatest distance travelled in March

```
flights %>%
  filter(!is.na(tailnum), month == 3) %>%
  group_by(tailnum) %>%
  summarise(dist = sum(distance)) %>%
  arrange(desc(dist))
```

```
## # A tibble: 3,186 x 2
##    tailnum  dist
##    <chr>   <dbl>
##  1 N324AA  87402
##  2 N554UA  86148
##  3 N722TW  84775
##  4 N502UA  83673
##  5 N323AA  83430
##  6 N557UA  83229
```

Prerequisites

Packages

```r
library(tidyverse)
library(dbplyr) # install.packages("dbplyr")
library(DBI)
library(RSQLite) # install.packages("RSQLite")
library(nycflights13)
library(readr)
```

Data

```r
url <- paste0("https://raw.githubusercontent.com/mhaber/",
              "AppliedDataScience/",
              "master/slides/week5/data/")
films <- read_csv(paste0(url, "films.csv"))
people <- read_csv(paste0(url, "people.csv"),
                   col_types = "iccc")
reviews <- read_csv(paste0(url, "reviews.csv"))
roles <- read_csv(paste0(url, "roles.csv"))
#flights
```

Relational data

Relational data

Data analysis rarely involves only a single table, but many tables of data, and you must combine them to answer the questions that you're interested in. Collectively, multiple tables of data are called relational data because it is the relations that are important.

The most common place to find relational data is in a relational database management system (or RDBMS).

Databases

What is a database:

> *A collection of information organized to afford efficient retrievel.*

> *"When people use the word database, fundamentally what they are saying is that the data should be self-describing and it should have a schema." (Jim Gray)*

Databases

Advantages of databases:

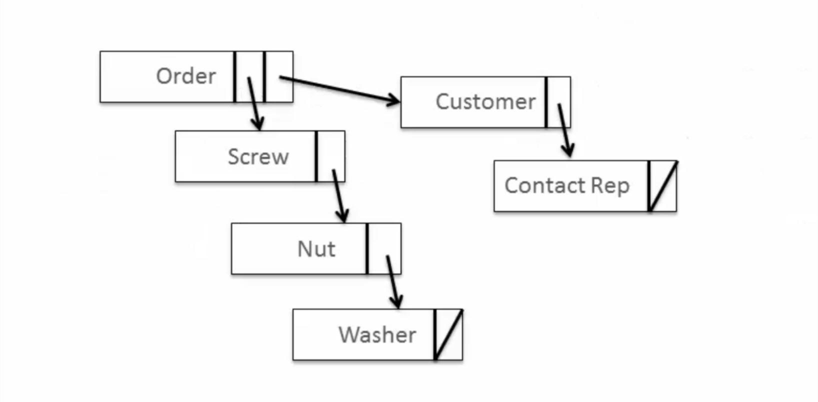
1. Sharing
    - Support concurrent access by multiple readers and writers
2. Data Model Enforcement
    - Make sure all applications see clean, organized data
3. Scale
    - Work with datasets too large to fit into memory
4. Flexibility
    - Use the data in new, unanticipated ways

Pre-Relational Databases

Evaluate databases:

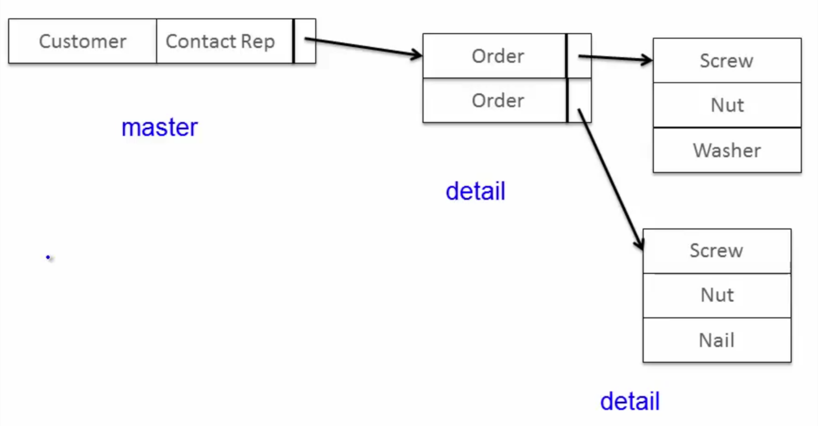
- How are the data physically organized on disk?
- What kinds of queries are efficiently supported, and which are not?
- How hard is it to update the data, or add new data
- What happes when I encounter new queries that I didn't anticipate. Do I reorganize the data?

Historical example: network databases



- All programs must be rewritten to accommodate reorganizations of the data
- difficultto look up all orders that involve a specific part

Historical example: hierachical databases



- hierachical order of data in terms of segments
    - works great if you want to find all orders for a particular customer
    - but what if you want to find all customers who ordered a Nail?

Relational databases

Key idea of relational databases

> *"Relational Database Management Systems were invented to let you use one set of data in multiple ways. Including ways that were unforeseen at the time the database is built, and then at the time the first applications are written." (Curt Monash)*

Relational database philisophy

God made the integers; all else is the work of man. (Leopold Kronecker, 19th century mathematician)

Cood made relations; all else is the work of man. (Raghu Ramakrishnan, DB text book author)

Relational databases concept

Relational databases (Edgar Codd 1970)

- Everything is a table
- Every row in a table has the same (types of) columns
- Relationships are implicit: no pointers

Relational databases concept

course	Student_ID
GRAD-E123	223
GRAD-E123	224
GRAD-E456	255
GRAD-E456	244

Student_ID	Student_Name
223	Jane
224	Joe
255	Susan

Relational databases concept

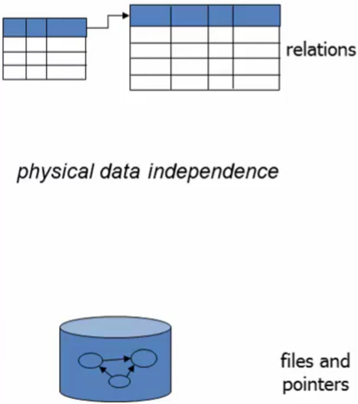
- Pre relational: if your data changed, your application broke
- Early RDBMS were buggy and slow, but required only 5% of the application code

Notion of data independence:

> *"Activities of users at terminals and most application programs should remain unaffected when the internal representation of data is changed and even when some aspects of the external representation are changed." (Ted Cod)*

Physical data independence

Access data through a high-level language (SQL), not physical pointers.

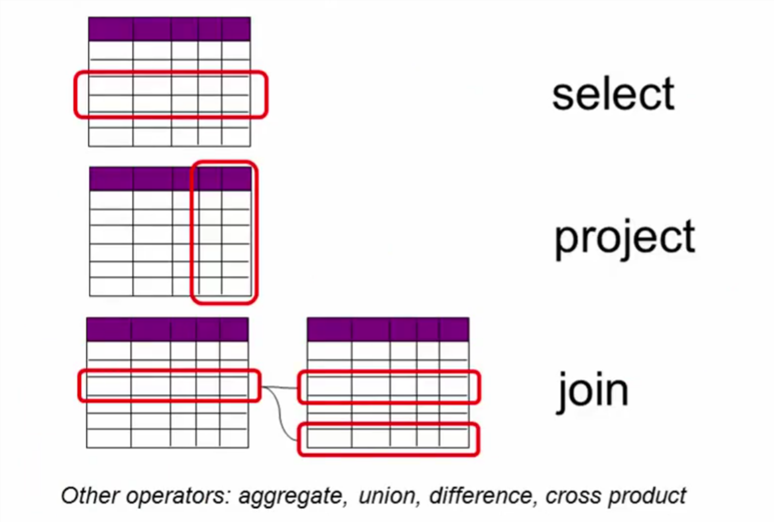



```
SELECT seq
  FROM ncbi_sequences
 WHERE seq = 'GATTACGATATTA';
```

```
f = fopen('table_file');
fseek(10030440);
while (True) {
  fread(&buf, 1, 8192, f);
  if (buf == GATTACGATATTA) {
     . . .
```

Algebra of tables

Use operations from relational algebra and set theory.



Algebraic optimization

$$N = ((z * 2) + ((z * 3) + 0))/1$$

Algebraic optimization

$N = ((z * 2) + ((z * 3) + 0))/1$

Algebraic laws:

1. $x + 0 = x$
2. $x/1 = x$
3. $(n * x + n * y) = n * (x + y)$
4. $x * y = y * x$

Apply rules 1,3,4,2: $N = (2 + 3) * z$

All relational databases do algebraic optimization when you write a query.

Relational algebra operators

- Union ∪, intersection ∩, difference -
- Selection s
- Projection Π
- Join ⋈

RA

- Duplicate elimination d
- Grouping and aggregation g
- Sorting t

Extended RA

Sets vs. bags

Sets: $\{a,b,c\}$, $\{a,d,e,f\}$, $\{\}$,... Bags: $\{a,a,b,c\}$, $\{b,b,b,b,b\}$,...

Relational algebra has two semantics:

- Set semantics = standard relational algebra
- Bag semantics = extented relational algebra

Rule of thumb:

- Every commercial database will assume bag semantics

Union

$$R1 \cup R2$$

```
SELECT * FROM R1
UNION
SELECT * FROM R2
```

R1

A	B
a1	b1
a2	b1

$\cup$

R2

A	B
a1	b1
a3	b4

=

$R1 \cup R2$

A	B
a1	b1
a2	b1
a3	b4

Difference

R1 – R2

```
SELECT * FROM R1
EXCEPT
SELECT * FROM R2
```

R1

A	B
a1	b1
a2	b1

–

R2

A	B
a1	b1
a3	b4

=

R1 – R2

A	B
a2	b1

Intersection

- Derived operator using minus:

$$R1 \cap R2 = R1 - (R1 - R2)$$

- Derived using joins:

$$R1 \cap R2 = R1 \bowtie R2$$

Selection

- Returns all tuples which satisfy a condition

$\sigma_c(R)$

- The condition c can be $=, <, \leq, >, \geq, <>$
- And you can also use boolean expression (and, or ,not)

Selection

Employee

SSN	Name	Salary
1234545	John	20000
5423341	Smith	60000
4352342	Fred	50000

$\sigma_{\text{Salary} > 40000}$ (Employee)

SSN	Name	Salary
5423341	Smith	60000
4352342	Fred	50000

Projection

- Eliminates columns

$\prod_{A1,\ldots,An}(R)$

Projection

Employee

SSN	Name	Salary
1234545	John	20000
5423341	John	60000
4352342	John	20000

$\Pi_{\text{Name,Salary}}$ (Employee)

Name	Salary
John	20000
John	60000
John	20000

Name	Salary
John	20000
John	60000

Set semantics

Cross Product

- Each tuple in $R1$ with each tuple in $R2$: $R1 \times R2$
- Rare in practice, but can come up in analytics
    - e.g. find all pairs of similar tweets

Cross Product

Employee

Name	SSN
John	999999999
Tony	777777777

Dependent

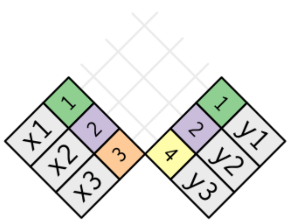
EmpSSN	DepName
999999999	Emily
777777777	Joe

Employee □ Dependent

Name	SSN	EmpSSN	DepName
John	999999999	999999999	Emily
John	999999999	777777777	Joe
Tony	777777777	999999999	Emily
Tony	777777777	777777777	Joe

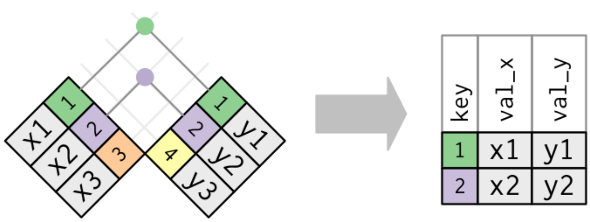
Join

A join is a way of connecting each row in `x` to zero, one, or more rows in `y`. The following diagram shows each potential match as an intersection of a pair of lines.



Inner join

The simplest type of join is the **inner join**. An inner join matches pairs of observations whenever their keys are equal. Unmatched rows are not included in the result



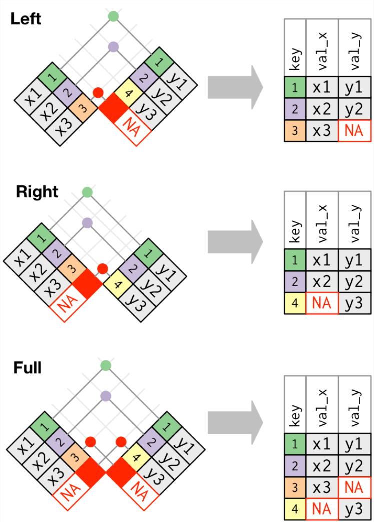
Outer joins

An inner join keeps observations that appear in both tables. An **outer join** keeps observations that appear in at least one of the tables. There are three types of outer joins:

- A **left join** keeps all observations in `x`.
- A **right join** keeps all observations in `y`.
- A **full join** keeps all observations in `x` and `y`.

The most commonly used join is the left join: you use this whenever you look up additional data from another table, because it preserves the original observations even when there isn't a match.

Outer joins


Left
key
val_x
val_y
1 x1 y1
2 x2 y2
3 x3 NA
Right
key
val_x
val_y
1 x1 y1
2 x2 y2
4 NA y3
Full
key
val_x
val_y
1 x1 y1
2 x2 y2
3 x3 NA
4 NA y3

SQL

SQL

Structured Query Language is a language for interacting with databases. SQL is over 40 years old, and is used by pretty much every database in existence.

- A query is a request for data from a database table (or combination of tables)
- SQL can be used to query but also to create and modify databases.

SELECT

In SQL, you can select data from a table using a `SELECT` statement. For example, the following query selects the name column `FROM` the people table:

```
SELECT name
FROM people
```

`SELECT` and `FROM` are keywords. They are not case-sensitive but it's good practice to make SQL keywords uppercase to distinguish them from other parts of your query, like column and table names.

SELECT

To select multiple columns from a table, simply separate the column names with commas!

```
SELECT name, birthdate
FROM people
```

To select all columns

```
SELECT *
FROM people
```

You can also `LIMIT` the number of rows returned:

```
SELECT *
FROM people
LIMIT 10
```

SELECT DISTINCT

You can use the DISTINCT keyword to select all the unique values from a column.

```
SELECT DISTINCT language
FROM films
```

COUNT

The COUNT statement returns the number of rows in one or more columns.

```
SELECT COUNT(*)
FROM people
```

It's common to combine COUNT with DISTINCT to count the number of distinct values in a column.

```
SELECT COUNT(DISTINCT birthdate)
FROM people
```

The `WHERE` keyword allows you to filter based on both text and numeric values in a table. The `WHERE` clause always comes after the `FROM` statement.

```
SELECT title
FROM films
WHERE title = 'Metropolis'
```

You can build up your WHERE queries by combining multiple conditions with the AND keyword.

```
SELECT title
FROM films
WHERE release_year > 1994
AND release_year < 2000
```

The `OR` operator allows you to select rows based on multiple conditions where some but not all of the conditions need to be met?

```
SELECT title
FROM films
WHERE release_year = 1994
OR release_year = 2000
```

BETWEEN

The BETWEEN keyword provides a useful shorthand for filtering values within a specified range. BETWEEN is always inclusive can be used with multiple AND and OR operators

```
SELECT title
FROM films
WHERE release_year
BETWEEN 1994 AND 2000
```

WHERE IN

The IN operator allows you to specify multiple values in a WHERE clause, making it easier and quicker to specify multiple OR conditions.

```
SELECT name
FROM kids
WHERE age IN (2, 4, 6, 8, 10)
```

NULL and IS NULL

NULL represents a missing or unknown value. You can check for NULL values using the expression IS NULL.

```
SELECT COUNT(*)
FROM people
WHERE birthdate IS NULL
```

You can use the IS NOT NULL operator to filter out missing values.

`LIKE` and `NOT LIKE`

The `LIKE` operator can be used in a `WHERE` clause to search for a pattern in a column. You use a wildcard as a placeholder for some other values. There are two wildcards you can use with `LIKE`:

- The % wildcard will match zero, one, or many characters in text:
- The _ wildcard will match a single character.

For example,

```
SELECT name
FROM companies
WHERE name LIKE 'Data%'
```

finds 'Data', 'Datab', 'Database'.

Aggregate functions

SQL provides a few functions to perform some calculation on the data in a database. Those aggregate functions can also be combined with the `WHERE` clause

For example,

```
SELECT AVG(budget)
FROM films
```

gives you the average value from the budget column of the films table. Similarly, the `MAX` function returns the highest budget:

```
SELECT MAX(budget)
FROM films
```

Aggregate functions

The `SUM` function returns the result of adding up the numeric values in a column:

```
SELECT SUM(budget)
FROM films
```

Arithmetics

You can perform basic arithmetic with symbols like +, -, *, and /.

For example, this gives a result of 12:

```sql
SELECT (4 * 3)
```

However, the following gives a result of 1,

```sql
SELECT (4 / 3)
```

SQL assumes that if you divide an integer by an integer, you want to get an integer back. If you want more precision when dividing, you can add decimal places to your numbers.

AS

You use the AS keyword to assign a temporary name to something (i.e. alias).

```
SELECT MAX(budget) AS max_budget,
       MAX(duration) AS max_duration
FROM films
```

The `ORDER BY` keyword is used to sort results according to the values of one or more columns. By default `ORDER BY` will sort in ascending order. If you want to sort the results in descending order, you can use the `DESC` keyword.

```
SELECT title
FROM films
ORDER BY release_year DESC
```

`ORDER BY` multiple columns

`ORDER BY` can also be used to sort on multiple columns. It will sort by the first column specified, then sort by the next, then the next, and so on.

```
SELECT birthdate, name
FROM people
ORDER BY birthdate, name
```

GROUP BY

GROUP BY allows you to group a result by one or more columns.
GROUP BY always goes after the FROM clause.

```
SELECT sex, count(*)
FROM employees
GROUP BY sex
```

Commonly, GROUP BY is used with aggregate functions like COUNT() or MAX().

HAVING

In SQL, aggregate functions can't be used in WHERE clauses. If you want to filter based on the result of an aggregate function, you need to use the HAVING clause.

```
SELECT release_year
FROM films
GROUP BY release_year
HAVING COUNT(title) > 10
```

Joins

dplyr
`inner_join(x, y, by = "z")`
`left_join(x, y, by = "z")`
`right_join(x, y, by = "z")`
`full_join(x, y, by = "z")`

SQL
`SELECT * FROM x INNER JOIN y USING (z)`
`SELECT * FROM x LEFT JOIN y USING (z)`
`SELECT * FROM x RIGHT JOIN y USING (z)`
`SELECT * FROM x FULL JOIN y USING (z)`

UNION, INTERSECT, EXCEPT

The UNION operator is used to combine the results of two or more SELECT statements without returning any duplicate rows.

```
SELECT column1
FROM table1
UNION
SELECT column1
FROM table1
```

The INTERSECT operator is used to combine two SELECT statements but returns rows only from the first SELECT statement that are identical to a row in the second SELECT statement.

The EXCEPT operator is used to combine two SELECT statements and returns rows from the first SELECT statement that are not returned by the second SELECT statement.

User-defined function support

Most SQL database also allow user-defined functions,routines that perform an action, such as a complex calculation, and return the result of that action as a value.

- PostgreSQL
  - SQL, PL/pSQL, Python, C/C++, R
- Microsoft SQL Server
  - SQL, T-SQL, C#
- Oracle
  - SQL, PL-SQL, Java, C/C++, Python, others
- SQLite
  - None

SQL in R

SQL vs R

SQL is not designed to do data analysis. For example, calculate the median arrival delay per carrier.

Using `dplyr`:

```
flights %>%
  dplyr::group_by(carrier) %>%
  dplyr::summarize(delay = median(arr_delay, na. rm =TRUE))
```

SQL vs R

PostgreSQL:

```
WITH ordered_flights AS (
 SELECT arr_delay,
        row_number() OVER (order by id) AS row_id,
        (SELECT COUNT(1) from flights) AS ct
 FROM flights
)

SELECT AVG(arr_delay) AS median
FROM ordered_flights
WHERE row_id BETWEEN ct/2.0 AND ct/2.0 + 1
```

SQL and R

Good idea to store data in SQL and use R to analyze it. `dbplyr`, for example, also works with remote on-disk data stored in databases.

The goal of `dbplyr` is to automatically generate SQL for you so that you're not forced to use it. Most of the time you don't need to know anything about SQL, and you can continue to use the `dplyr` verbs that you're already familiar with

Connecting to a database

You need to install a specific backend for the database that you want to connect to:

- `RMySQL` connects to MySQL and MariaDB
- `RPostgreSQL` connects to Postgres and Redshift
- `RSQLite` embeds a SQLite database (comes with `dbplyr`)
- `odbc` connects to many commercial databases via the open database connectivity protocol
- `bigrquery` connects to Google's BigQuery

Connecting to a database file

```r
con<-DBI::dbConnect(RSQLite::SQLite(), dbname = ":memory:")
```

Connecting to an online database

```
con <- DBI::dbConnect(RMySQL::MySQL(),
  host = "database.rstudio.com",
  user = "hadley",
  password = rstudioapi::askForPassword(
    "Database password")
)
```

Creating a database

```
con<-DBI::dbConnect(RSQLite::SQLite(), dbname = ":memory:")
dbWriteTable(con, "flights", nycflights13::flights)
dbListTables(con)
```

```
## [1] "flights"
```

```
flights_db <- tbl(con, "flights")
```

Generating queries

You can pass SQL queries with `dbGetQuery()` or `dbExecute()`.

```
results <- DBI::dbGetQuery(con,
                           "SELECT arr_time AS arrivalTime
                           FROM flights
                           WHERE carrier = 'UA'")
```

Generating queries

Most of the time you don't need to know anything about SQL, and you can continue to use the dplyr verbs that you're already familiar with:

```
tailnum_delay_db <- flights_db %>%
  dplyr::group_by(tailnum) %>%
  dplyr::summarize(
    delay = mean(arr_delay),
    n = n()
  ) %>%
  dplyr::arrange(desc(delay)) %>%
  dplyr::filter(n > 100)
```

Your R code is translated into SQL and executed in the database, not in R. dplyr tries to be as lazy as possible:

Generating queries

You can see the SQL dplyr generating with show_query():

```
tailnum_delay_db %>% show_query()
```

```
## Warning: Missing values are always removed in SQL.
## Use `AVG(x, na.rm = TRUE)` to silence this warning

## <SQL>
## SELECT *
## FROM (SELECT *
## FROM (SELECT `tailnum`, AVG(`arr_delay`) AS `delay`, COU
## FROM `flights`
## GROUP BY `tailnum`)
## ORDER BY `delay` DESC)
## WHERE (`n` > 100.0)
```

Pull data from a database

Once you've figured it out what data you need from the database, use `collect()` to pull all the data down into a local tibble.

```
tailnum_delay <- tailnum_delay_db %>% collect()
```

```
## Warning: Missing values are always removed in SQL.
## Use `AVG(x, na.rm = TRUE)` to silence this warning
```

`collect()` requires that database does some work, so it may take a long time to complete.

Homework Exercises

Homework Exercises

In today's exercise you'll be working with a database containing information on almost 5000 films. Your goal is to

1. set up our own (SQLite) database using `DBI::dbconnect()`
2. write the four objects `films`, `people`, `review`, `role` as tables into your database and use `dbListTables()` to make sure you did it correctly

Homework Exercises

3. complete the following tasks using either SQL or `dplyr`'s language:

   - Get the title, release year and country for every film
   - Get all the different type of film roles
   - Count the number of unique languages
   - Get the number of films released before 2000
   - Get the name and birth date of the person born on November 11th, 1974. Remember to use ISO date format ('1974-11-11')
   - Get all information for Spanish language films released after 2000, but before 2010
   - Get the names of people who are still alive
   - Get the amount grossed by the best performing film between 2000 and 2012

Homework Exercises

3. complete the following tasks using either SQL or `dplyr`'s language:
   - Get the average duration in hours for all films, aliased as `avg_duration_hours`
   - Get the IMDB score and count of film reviews grouped by IMDB score
   - Get the country, average budget, and average gross take of countries that have made more than 10 films. Order the result by country name, and limit the number of results displayed to 5. You should alias the averages as avg_budget and avg_gross respectively
   - Disconnect from your database

That's it for today. Questions?