Introduction to Data Science

Session 5: Relational databases and SQL

Simon Munzert
Hertie School | GRAD-C11/E1339

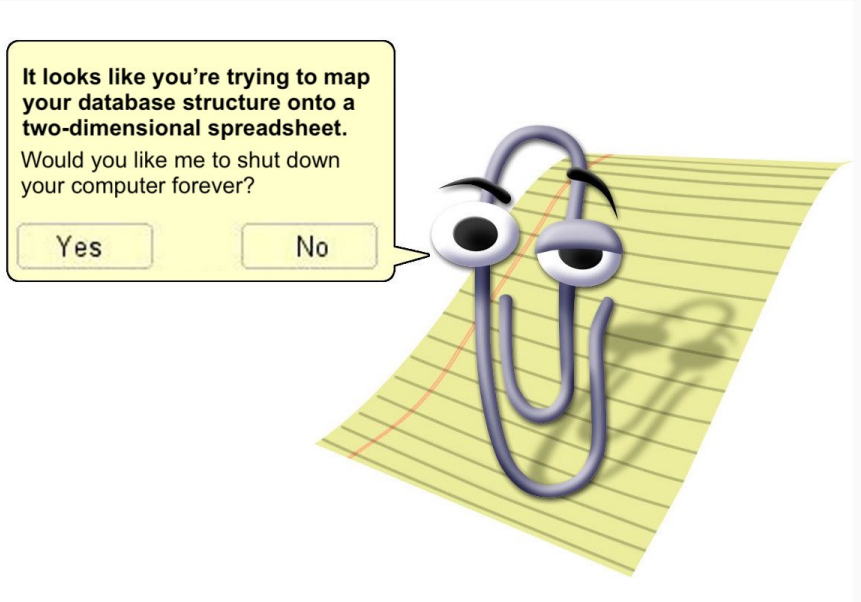
Table of contents

Why databases?

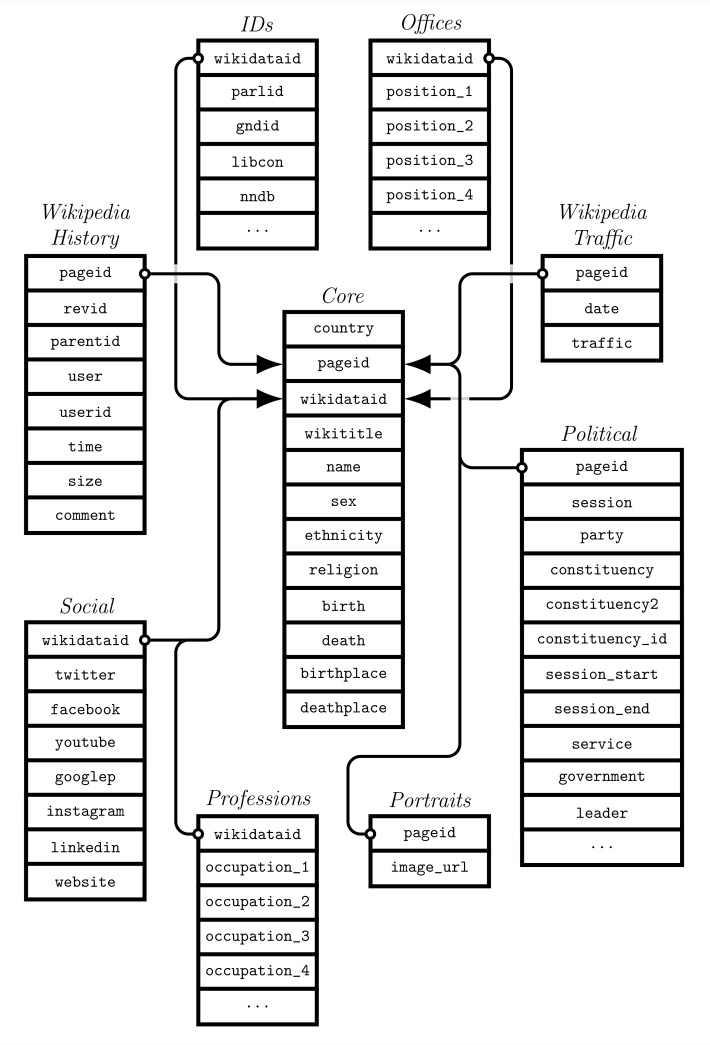
The ubiqity of multi-dimensional data structures

From data frames...

- When you have a background in social sciences, your top-of-the-head mental image of data might be a rectangular spreadsheet.
- In fact, much of classical "statistical" software (SPSS, Stata, MS Excel) operates with rectangular data frames by default.
- At the same time, your perception might be file-based. Data is stored in files, and these files are read (and produced) by our data management software.
- In many cases, the **two-dimensional structure** makes sense. For instance, we observe
  - persons x attitudes
  - countries x characteristics
  - social media posts x text features



The ubiqity of multi-dimensional data structures



`Credit` saschagobel/legislatoR

... to complex data structures

- However, the longer you think about it, the more problematic it becomes to store your data in two-dimensional structures.
- **Examples**:
  - countries x persons x characteristics x time
  - countries x states x communities x time x variables
  - social media posts x retweets x users x user characteristics x network features x meta data
- Mapping three- onto two-dimensional structures is easy (think: `pivot_longer`, `pivot_wider`).
- With **multiple heterogeneous data sources**, things get messy.
- Managing complex data structures is just one perk of using databases.

When databases become useful

Size and structure

- You have **loads of data that exceed the working memory** on your computer. Databases are only limited by available disk size (or can be distributed across multiple disks/machines).
- Your **data structure is complex**. Databases allow/encourage you to store, retrieve and subset data with complex data structures.
- Your data is big and you have to **access/subset/operate frequently**. Querying databases is fast.
- You care about **data quality** and have clear expectations how data should look like. Using databases you can define specific rules for extending and updating your database.

When databases become useful

Size and structure

- You have **loads of data that exceed the working memory** on your computer. Databases are only limited by available disk size (or can be distributed across multiple disks/machines).
- Your **data structure is complex**. Databases allow/encourage you to store, retrieve and subset data with complex data structures.
- Your data is big and you have to **access/subset/operate frequently**. Querying databases is fast.
- You care about **data quality** and have clear expectations how data should look like. Using databases you can define specific rules for extending and updating your database.

Accessibility and concurrency

- You **collaborate with others** on a data collection project. With a database, you have a common, simultaneously accessible, and reliable infrastructure at hand that multiple users can access at the same time.
- When several parties are involved, who is allowed to do what with the database might differ (e.g., read-only, access to parts of the data, limited admin rights, etc.). Most databases allow **defining different usage rights for different users**.

Talking about databases

What we should distinguish

- The **types of databases**, e.g.: relational, navigational, NoSQL, NewSQL
- The **database management system**, e.g.: PostgreSQL, Oracle, SQL Server, SQLite
- The **data structure**, e.g.: tables, columns, keys, normal forms
- The **data manipulations**, e.g.: selects, joins, grouping
- The **query language**, e.g., SQL, SPARQL

Also, there are so many more ways to classify databases. But that's enough for now.

Today, **we focus on relational databases**. They are by no means the only type of databases (see above), but they're ubiquitous and won't go away any time soon.

Talking about databases

What we should distinguish

- The **types of databases**, e.g.: relational, navigational, NoSQL, NewSQL
- The **database management system**, e.g.: PostgreSQL, Oracle, SQL Server, SQLite
- The **data structure**, e.g.: tables, columns, keys, normal forms
- The **data manipulations**, e.g.: selects, joins, grouping
- The **query language**, e.g., SQL, SPARQL

Also, there are so many more ways to classify databases. But that's enough for now.

Today, **we focus on relational databases**. They are by no means the only type of databases (see above), but they're ubiquitous and won't go away any time soon.

Databases versus data frames

When reading/talking about features of databases, you will encounter a particular jargon. Here's how database concepts map onto R data frame jargon:

R jargon	Database jargon
column	attribute/field
row	tuple/record
element/cell	attribute value
data frame	relation/table
column types	table schema
bunch of related data frames	database

Relational database fundamentals

Codd's relational model for databases

- The concept of relational databases builds on the relational model (RM) for database management, as proposed by Edgar F. "Ted" Codd in 1969/1970.
- Todd described the RM formally, but also introduced it using concepts that are still in use today (normalization, keys, joins, redundancy, etc.).
- The key assumption of the relational model is that all data can be represented as relations (tables).
- Information is then represented by data values in relations.
- When you think this is trivial, check out the history of databases and live through the pain of the early era of navigational DBMS in the 1960s and the NoSQL era that we've (not yet) overcome.

A Relational Model of Data for Large Shared Data Banks

E. F. CODD
IBM Research Laboratory, San Jose, California

Future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation). A prompting service which supplies such information is not a satisfactory solution. 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. Changes in data representation will often be needed as a result of changes in query, update, and report traffic and natural growth in the types of stored information.

Existing noninferential, formatted data systems provide users with tree-structured files or slightly more general network models of the data. In Section 1, inadequacies of these models are discussed. A model based on *n*-ary relations, a normal form for data base relations, and the concept of a universal data sublanguage are introduced. In Section 2, certain operations on relations (other than logical inference) are discussed and applied to the problems of redundancy and consistency in the user's model.

KEY WORDS AND PHRASES: data bank, data base, data structure, data organization, hierarchies of data, networks of data, relations, derivability, redundancy, consistency, composition, join, retrieval language, predicate calculus, security, data integrity
CR CATEGORIES: 3.70, 3.73, 3.75, 4.20, 4.22, 4.29

1. Relational Model and Normal Form

1.1. Introduction

This paper is concerned with the application of elementary relation theory to systems which provide shared access to large banks of formatted data. Except for a paper by Childs [1], the principal application of relations to data systems has been to deductive question-answering systems. Levein and Maron [2] provide numerous references to work in this area.

The relational view (or model) of data described in Section 1 appears to be superior in several respects to the graph or network model [3, 4] presently in vogue for noninferential systems. It provides a means of describing data with its natural structure only—that is, without superimposing any additional structure for machine representation purposes. Accordingly, it provides a basis for a high level data language which will yield maximal independence between programs on the one hand and machine representation and organization of data on the other.

A further advantage of the relational view is that it forms a sound basis for treating derivability, redundancy, and consistency of relations—these are discussed in Section 2. The network model, on the other hand, has spawned a number of confusions, not the least of which is mistaking the derivation of connections for the derivation of relations (see remarks in Section 2 on the "connection trap").

Finally, the relational view permits a clearer evaluation of the scope and logical limitations of present formatted data systems, and also the relative merits (from a logical standpoint) of competing representations of data within a single system. Examples of this clearer perspective are cited in various parts of this paper. Implementations of systems to support the relational model are not discussed.

1.2. Data Dependencies in Present Systems

The provision of data description tables in recently developed information systems represents a major advance toward the goal of data independence [5, 6, 7]. Such tables facilitate changing certain characteristics of the data representation stored in a data bank. However, the variety of data representation characteristics which can be changed *without logically impairing some application programs* is still quite limited. Further, the model of data with which users interact is still cluttered with representational properties, particularly in regard to the representation of collections of data (as opposed to individual items). Three of the principal kinds of data dependencies which still need to be removed are: ordering dependence, indexing dependence, and access path dependence. In some systems these dependencies are not clearly separable from one another.

1.2.1. *Ordering Dependence.* Elements of data in a data bank may be stored in a variety of ways, some involving no concern for ordering, some permitting each element to participate in one ordering only, others permitting each element to participate in several orderings. Let us consider those existing systems which either require or permit data elements to be stored in at least one total ordering which is

`Credit` Communications of the ACM 13(6), 1970

Codd's relational model for databases (cont.)

Storing data in tables

- Again, the key concept of relational databases is that all information can be represented in a table.
- A single table already introduces relations: All data in one row belongs to the same record.
- If we want to represent more complex relations (i.e., measuring a person's weight twice or measuring the weight of their children as well), we can relate data from one table to another.

Example

- We have collected data on Peter, Paul, and Mary.
- We have information on birthdays, telephone numbers, and favorite foods.
- How can we represent this information in tables?

Codd's relational model for databases (cont.)

Storing data in tables

- Again, the key concept of relational databases is that all information can be represented in a table.
- A single table already introduces relations: All data in one row belongs to the same record.
- If we want to represent more complex relations (i.e., measuring a person's weight twice or measuring the weight of their children as well), we can relate data from one table to another.

Example

- We have collected data on Peter, Paul, and Mary.
- We have information on birthdays, telephone numbers, and favorite foods.
- How can we represent this information in tables?

Table 7.1 Our friends' birthdays and favorite foods

nameid	name	birthday	favoritefood1	favoritefood2	favoritefood3
1	Peter Pascal	01/02/1991	spaghetti	hamburger	
2	Paul Panini	02/03/1992	fruit salad		
3	Mary Meyer	03/04/1993	chocolate	fish fingers	hamburger

Table 7.2 Our friends' telephone numbers

telephoneid	nameid	telephonenumber
1	1	001665443
2	2	00255555
3	1	001878345

- We start representing the data in two tables.
- They are linked via the key `nameid`, so we don't have to add the full names to the phone numbers table.
- Note that we have a 1:m (one-to-many) relation here because Peter has two phone numbers.

Codd's relational model for databases (cont.)

Storing data in tables

- Again, the key concept of relational databases is that all information can be represented in a table.
- A single table already introduces relations: All data in one row belongs to the same record.
- If we want to represent more complex relations (i.e., measuring a person's weight twice or measuring the weight of their children as well), we can relate data from one table to another.

Example

- We have collected data on Peter, Paul, and Mary.
- We have information on birthdays, telephone numbers, and favorite foods.
- How can we represent this information in tables?

Table 7.1 Our friends' birthdays and favorite foods

nameid	name	birthday	favoritefood1	favoritefood2	favoritefood3
1	Peter Pascal	01/02/1991	spaghetti	hamburger	
2	Paul Panini	02/03/1992	fruit salad		
3	Mary Meyer	03/04/1993	chocolate	fish fingers	hamburger

Table 7.2 Our friends' telephone numbers

telephoneid	nameid	telephonenumber
1	1	001665443
2	2	00255555
3	1	001878345

- However, the way we store the data is not ideal. In the first table, we have three columns measuring effectively the same thing. And what if there's more favorite food? Adding information in such a fashion creates a lot of redundant information.

Codd's relational model for databases (cont.)

Storing data in tables

- Again, the key concept of relational databases is that all information can be represented in a table.
- A single table already introduces relations: All data in one row belongs to the same record.
- If we want to represent more complex relations (i.e., measuring a person's weight twice or measuring the weight of their children as well), we can relate data from one table to another.

Table 7.3 Our friends' birthdays—revised

name id	first name	last name	birthday
1	Peter	Pascal	01/02/1991
2	Paul	Panini	02/03/1992
3	Mary	Meyer	03/04/1993

Table 7.4 Our friends' food preferences

nameid	foodname	rank
1	spaghetti	1
1	hamburger	2
2	fruit salad	1
3	chocolate	1
3	fish fingers	2
3	hamburger	3

Example

- We have collected data on Peter, Paul, and Mary.
- We have information on birthdays, telephone numbers, and favorite foods.
- How can we represent this information in tables?

- Splitting up the information by creating another table for food preferences is better.
- There's still some redundancy left. Is it really necessary to have `hamburger` in the table twice?

Codd's relational model for databases (cont.)

Storing data in tables

- Again, the key concept of relational databases is that all information can be represented in a table.
- A single table already introduces relations: All data in one row belongs to the same record.
- If we want to represent more complex relations (i.e., measuring a person's weight twice or measuring the weight of their children as well), we can relate data from one table to another.

Example

- We have collected data on Peter, Paul, and Mary.
- We have information on birthdays, telephone numbers, and favorite foods.
- How can we represent this information in tables?

Table 7.5 Our friend's food preferences—revised

rankid	nameid	foodid	rank
1	1	1	1
2	1	2	2
3	2	3	1
4	3	4	1
5	3	5	2
6	3	2	3

Table 7.6 Food types

foodid	food name	healthy	kcalp100g
1	spaghetti	no	0.158
2	hamburger	no	0.295
3	fruit salad	yes	0.043
4	chocolate	no	0.546
5	fish fingers	no	0.290

- Now that's better.
- In restructuring the information in our database, we **avoided redundancy (duplication)**.
- This is the process of **database normalization**.

Database normalization

What is database normalization?

From the Wikipedia: "Database normalization is the process of **structuring a database**, usually a relational database, in accordance with a series of so-called **normal forms** in order to **reduce data redundancy and improve data integrity**. It was first proposed by Edgar F. Codd as part of his relational model."

- You'll probably not have to apply normalization yourself because you are a user not a designer of databases.
- However, it helps to have an idea of what the first normal forms are.
- Higher-order normal forms imply lower-order normal forms (e.g., in order to satisfy the 3rd normal form, the 1st and 2nd normal forms have to be satisfied, too).

Database normalization

What is database normalization?

From the Wikipedia: "Database normalization is the process of **structuring a database**, usually a relational database, in accordance with a series of so-called **normal forms** in order to **reduce data redundancy and improve data integrity**. It was first proposed by Edgar F. Codd as part of his relational model."

- You'll probably not have to apply normalization yourself because you are a user not a designer of databases.
- However, it helps to have an idea of what the first normal forms are.
- Higher-order normal forms imply lower-order normal forms (e.g., in order to satisfy the 3rd normal form, the 1st and 2nd normal forms have to be satisfied, too).

Normalization and tidy data

There is also a straightforward link to Hadley Wickham's "tidy data":

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a table.

This is Codd's 3rd normal form using "statistical" jargon and applied to a single dataset.

Database normalization (cont.)

The **normal forms** (from least normalized to most normalized):

- UNF: Unnormalized form
- 1NF: First normal form
- 2NF: Second normal form
- 3NF: Third normal form
- EKNF: Elementary key normal form
- BCNF: Boyce–Codd normal form
- 4NF: Fourth normal form
- ETNF: Essential tuple normal form
- 5NF: Fifth normal form
- DKNF: Domain-key normal form
- 6NF: Sixth normal form

	UNF (1970)	1NF (1970)	2NF (1971)	3NF (1971)	EKNF (1982)	BCNF (1974)	4NF (1977)	ETNF (2012)	5NF (1979)	DKNF (1981)	6NF (2003)
Primary key (no duplicate tuples)[4]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Atomic columns (cells cannot have tables as values)[5]	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Every non-trivial functional dependency either does not begin with a proper subset of a candidate key or ends with a prime attribute (no partial functional dependencies of non-prime attributes on candidate keys)[5]	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓
Every non-trivial functional dependency either begins with a superkey or ends with a prime attribute (no transitive functional dependencies of non-prime attributes on candidate keys)[5]	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓	✓
Every non-trivial functional dependency either begins with a superkey or ends with an elementary prime attribute	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓	N/A
Every non-trivial functional dependency begins with a superkey	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	N/A
Every non-trivial multivalued dependency begins with a superkey	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	N/A
Every join dependency has a superkey component[8]	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	N/A
Every join dependency has only superkey components	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	N/A
Every constraint is a consequence of domain constraints and key constraints	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗
Every join dependency is trivial	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓

`Credit` English Wikipedia, "Database normalization"

Database normalization (cont.)

Table 7.7 First normal form error—1

zip code and city
789222 Big Blossom
43211 Little Hamstaedt
123456 Bloomington
…

This table fails the first rule of the first normal form because two different types of information are saved in one column—a city's zip code and a city's name.

Table 7.8 First normal form error—2

telephone
0897729344, 0666556322
123123454
675345334
…

This table fails the first rule of the first normal form because two telephone numbers are saved within the first row.

Table 7.9 First normal form error—3

telephone1	telephone2
0897729344	0666556322
123123454	
675345334	
…	

This table fails the first rule of the first normal form because it uses two columns to store the same kind of information.

Table 7.10 A second normal form error

nameid	firstname	birthday	favoritefood	foodid	rank
1	Peter	01/02/1991	spaghetti	1	1
1	Peter	01/02/1991	hamburger	2	2
2	Paul	02/03/1992	fruit salad	3	1
3	Mary	03/04/1993	chocolate	4	1
3	Mary	03/04/1993	fish fingers	5	2
3	Mary	03/04/1993	hamburger	1	3

This table does not comply to second normal form because all columns except `rank` either relate to one part of the combined primary key (`nameid` and `foodid`) or to other part but not to both.

Table 7.11 A third normal form error

nameid	firstname	birthday	favoritefood	healthy	kcalp100g
1	Peter	01/02/1991	spaghetti	no	0.158
2	Paul	02/03/1992	fruit salad	yes	0.043
3	Mary	03/04/1993	chocolate	no	0.546

This table does not comply to third normal form because `favoritefood`, `healthy` and `kcalp100g` do not relate directly to persons which the primary key (`nameid`) is based on.

Database schema

What schemas are

- The database schema describes the structure of a database. It represents the map or blueprint of how the database is constructed.
- The schema specifies all core ingredients of the database, including tables, fields, keys relationships, views, etc.
- The visualization helps database users understand the relationships between the tables.

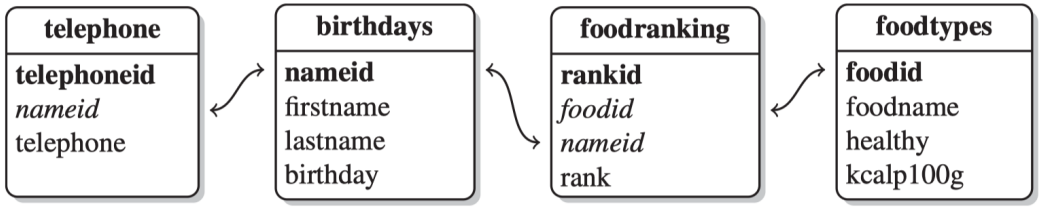
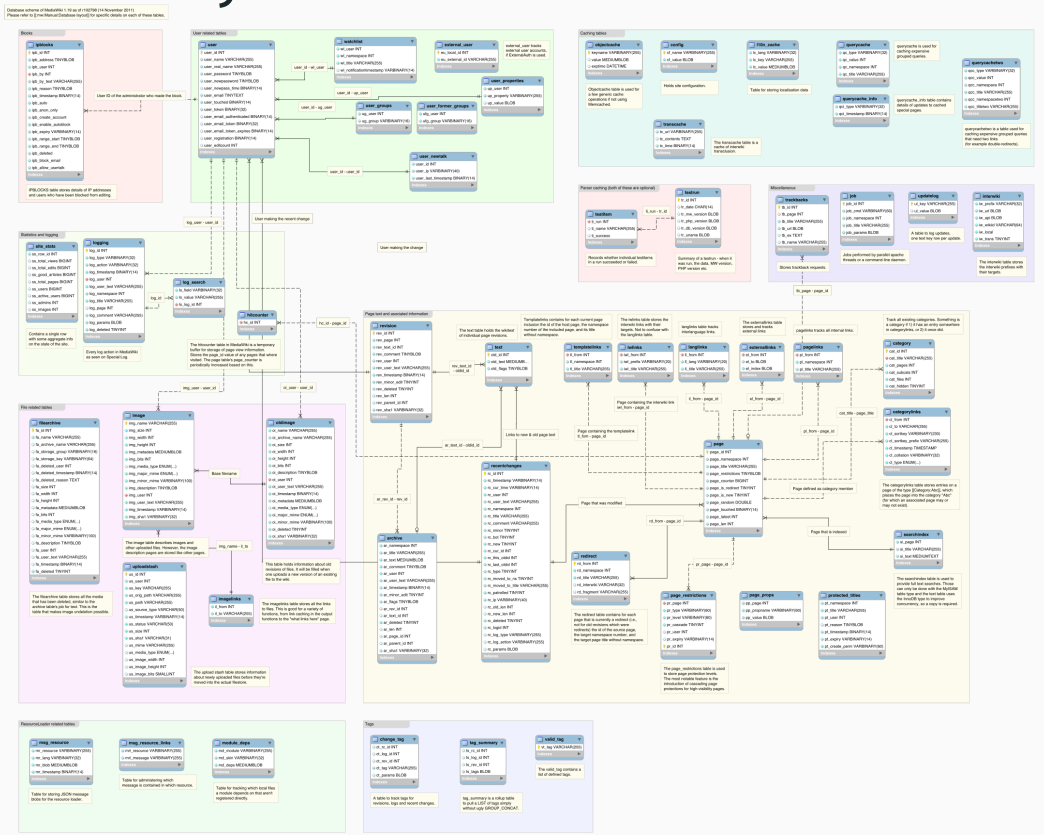


Figure 7.2 Database scheme

How they can look like



`Credit` Timo Tijhof/Wikimedia Commons

Back to dplyr: joins

Relational data in R

For the simple examples that I'm going to show here, we'll need some data sets that come bundled with the **nycflights13** package.

Let's load it now and then inspect these data frames in your own console.

```
R> library(nycflights13)
```

Relational data in R (cont.)

The package contains the tables `flights`, `airlines`, `airports`, `planes`, and `weather`.

Relational data in R (cont.)

The package contains the tables `flights`, `airlines`, `airports`, `planes`, and `weather`.

The `airlines` data frame lets you look up the full carrier name from its abbreviated code:

```
R> head(airlines, 10)
```

```
## # A tibble: 10 × 2
##    carrier name
##    <chr>   <chr>
##  1 9E      Endeavor Air Inc.
##  2 AA      American Airlines Inc.
##  3 AS      Alaska Airlines Inc.
##  4 B6      JetBlue Airways
##  5 DL      Delta Air Lines Inc.
##  6 EV      ExpressJet Airlines Inc.
##  7 F9      Frontier Airlines Inc.
##  8 FL      AirTran Airways Corporation
##  9 HA      Hawaiian Airlines Inc.
## 10 MQ      Envoy Air
```

Relational data in R (cont.)

The package contains the tables `flights`, `airlines`, `airports`, `planes`, and `weather`.

`airports` gives information about each airport, identified by the `faa` airport code:

```
R> head(airports, 10)
```

```
## # A tibble: 10 × 8
##    faa   name                           lat    lon   alt    tz dst   tzone
##    <chr> <chr>                        <dbl>  <dbl> <dbl> <dbl> <chr> <chr>
##  1 04G   Lansdowne Airport             41.1  -80.6  1044    -5 A     America/…
##  2 06A   Moton Field Municipal Airport  32.5  -85.7   264    -6 A     America/…
##  3 06C   Schaumburg Regional            42.0  -88.1   801    -6 A     America/…
##  4 06N   Randall Airport                41.4  -74.4   523    -5 A     America/…
##  5 09J   Jekyll Island Airport          31.1  -81.4    11    -5 A     America/…
##  6 0A9   Elizabethton Municipal Airport  36.4  -82.2  1593    -5 A     America/…
##  7 0G6   Williams County Airport        41.5  -84.5   730    -5 A     America/…
##  8 0G7   Finger Lakes Regional Airport  42.9  -76.8   492    -5 A     America/…
##  9 0P2   Shoestring Aviation Airfield   39.8  -76.6  1000    -5 U     America/…
## 10 0S9   Jefferson County Intl          48.1 -123.    108    -8 A     America/…
```

Relational data in R (cont.)

The package contains the tables `flights`, `airlines`, `airports`, `planes`, and `weather`.

`planes` gives information about each plane, identified by its `tailnum` (aircraft registration a.k.a. tail number):

```
R> head(planes, 10)
```

```
## # A tibble: 10 × 9
##    tailnum  year type              manufacturer model engines seats speed engine
##    <chr>   <int> <chr>             <chr>        <chr>   <int> <int> <int> <chr>
##  1 N10156   2004 Fixed wing multi… EMBRAER      EMB-…       2    55    NA Turbo…
##  2 N102UW   1998 Fixed wing multi… AIRBUS INDU… A320…       2   182    NA Turbo…
##  3 N103US   1999 Fixed wing multi… AIRBUS INDU… A320…       2   182    NA Turbo…
##  4 N104UW   1999 Fixed wing multi… AIRBUS INDU… A320…       2   182    NA Turbo…
##  5 N10575   2002 Fixed wing multi… EMBRAER      EMB-…       2    55    NA Turbo…
##  6 N105UW   1999 Fixed wing multi… AIRBUS INDU… A320…       2   182    NA Turbo…
##  7 N107US   1999 Fixed wing multi… AIRBUS INDU… A320…       2   182    NA Turbo…
##  8 N108UW   1999 Fixed wing multi… AIRBUS INDU… A320…       2   182    NA Turbo…
##  9 N109UW   1999 Fixed wing multi… AIRBUS INDU… A320…       2   182    NA Turbo…
## 10 N110UW   1999 Fixed wing multi… AIRBUS INDU… A320…       2   182    NA Turbo…
```

Relational data in R (cont.)

The package contains the tables `flights`, `airlines`, `airports`, `planes`, and `weather`.

`weather` gives the weather at each NYC airport for each hour:

```
R> head(weather, 10)
```

```
## # A tibble: 10 × 15
##    origin  year month   day  hour  temp  dewp humid wind_dir wind_speed
##    <chr>  <int> <int> <int> <int> <dbl> <dbl> <dbl>    <dbl>      <dbl>
##  1 EWR     2013     1     1     1  39.0  26.1  59.4      270      10.4
##  2 EWR     2013     1     1     2  39.0  27.0  61.6      250       8.06
##  3 EWR     2013     1     1     3  39.0  28.0  64.4      240      11.5
##  4 EWR     2013     1     1     4  39.9  28.0  62.2      250      12.7
##  5 EWR     2013     1     1     5  39.0  28.0  64.4      260      12.7
##  6 EWR     2013     1     1     6  37.9  28.0  67.2      240      11.5
##  7 EWR     2013     1     1     7  39.0  28.0  64.4      240      15.0
##  8 EWR     2013     1     1     8  39.9  28.0  62.2      250      10.4
##  9 EWR     2013     1     1     9  39.9  28.0  62.2      260      15.0
## 10 EWR     2013     1     1    10  41    28.0  59.6      260      13.8
## # ℹ 5 more variables: wind_gust <dbl>, precip <dbl>, pressure <dbl>,
## #   visib <dbl>, time_hour <dttm>
```

Relational data in R (cont.)

The package contains the tables `flights`, `airlines`, `airports`, `planes`, and `weather`.

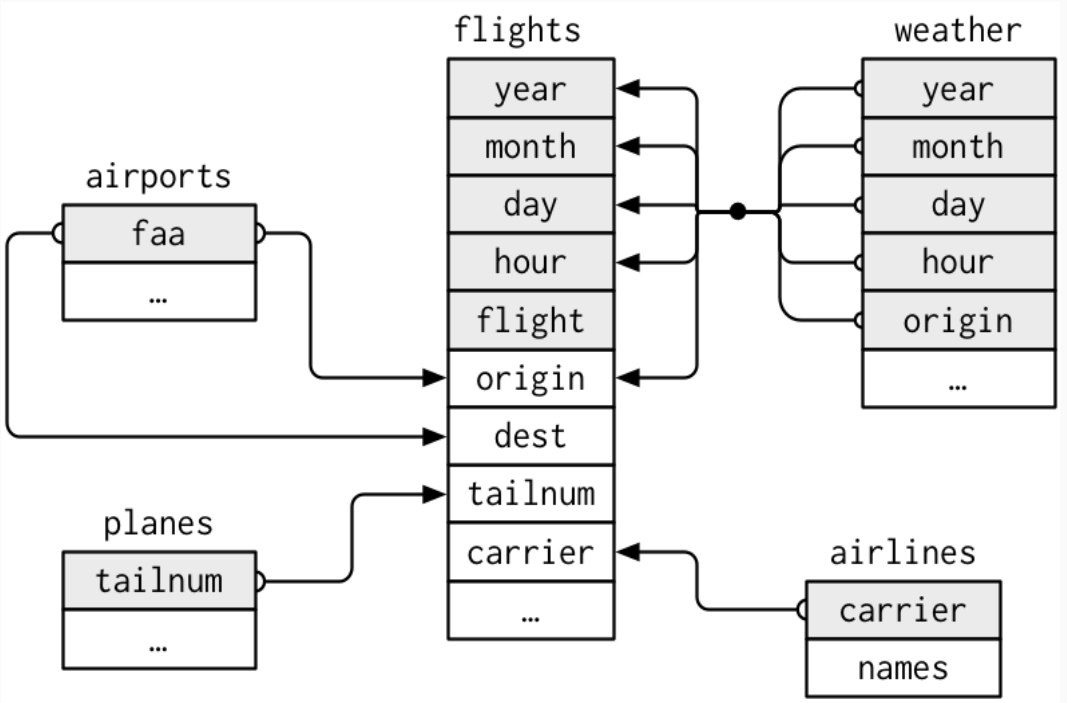
Finally, `flights` gives data on each of the 336776 flights in the dataset:

```
R> head(flights, 10)
```

```
## # A tibble: 10 × 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     1      517            515         2      830            819
##  2  2013     1     1      533            529         4      850            830
##  3  2013     1     1      542            540         2      923            850
##  4  2013     1     1      544            545        -1     1004           1022
##  5  2013     1     1      554            600        -6      812            837
##  6  2013     1     1      554            558        -4      740            728
##  7  2013     1     1      555            600        -5      913            854
##  8  2013     1     1      557            600        -3      709            723
##  9  2013     1     1      557            600        -3      838            846
## 10  2013     1     1      558            600        -2      753            745
## # ℹ 11 more variables: 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>
```

Relational data in R (cont.)

- We can illustrate the relationships between the different tables with a schematic drawing.[1]
- One table seems central (`flights`), but that's not a necessary feature of relational databases.
- Key to understanding the diagram is that each relation always concerns a pair of tables:
  - `flights` connects to `planes` via `tailnum`.
  - `flights` connects to `airlines` via `carrier`.
  - `flights` connects to `airports` via `origin` and `dest`.
  - `flights` connects to `weather` via `origin` (location) and `year`, `month`, `day`, and `hour` (time).



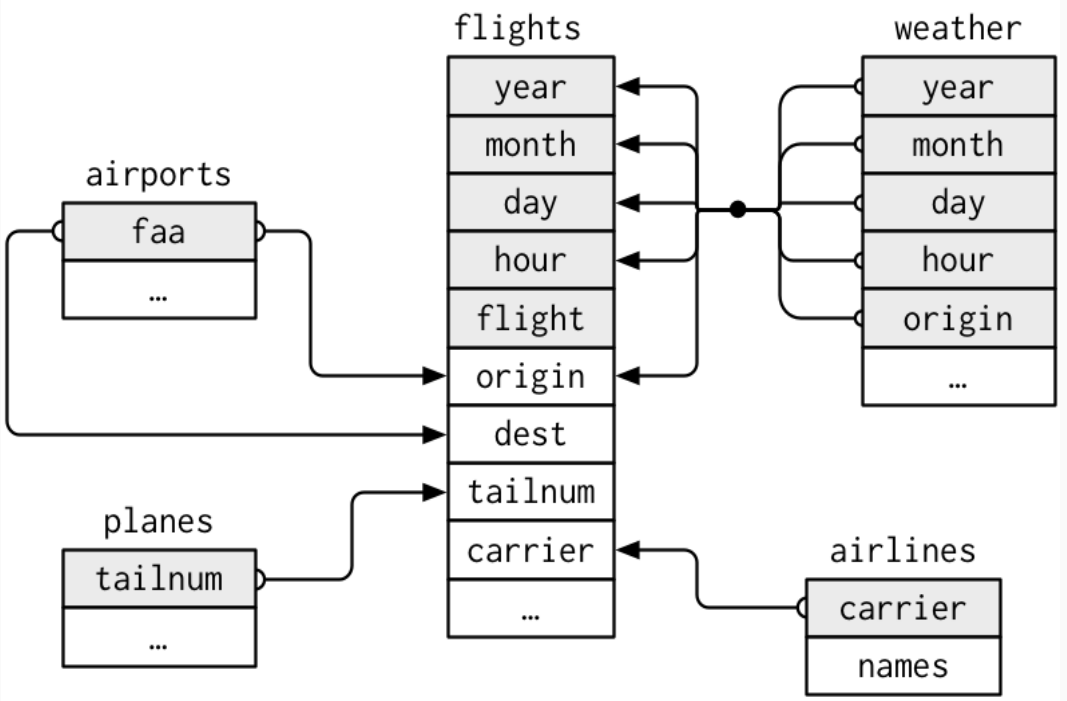
[1] Of course there are R packages that help you create such data models visually, e.g., the dm package.

Relational data in R (cont.)

Question time

1. Imagine you wanted to draw (approximately) the route each plane flies from its origin to its destination. What variables would you need? What tables would you need to combine?

2. We know that some days of the year are “special”, and fewer people than usual fly on them. How might you represent that data as a data frame? What would be the primary keys of that table? How would it connect to the existing tables?



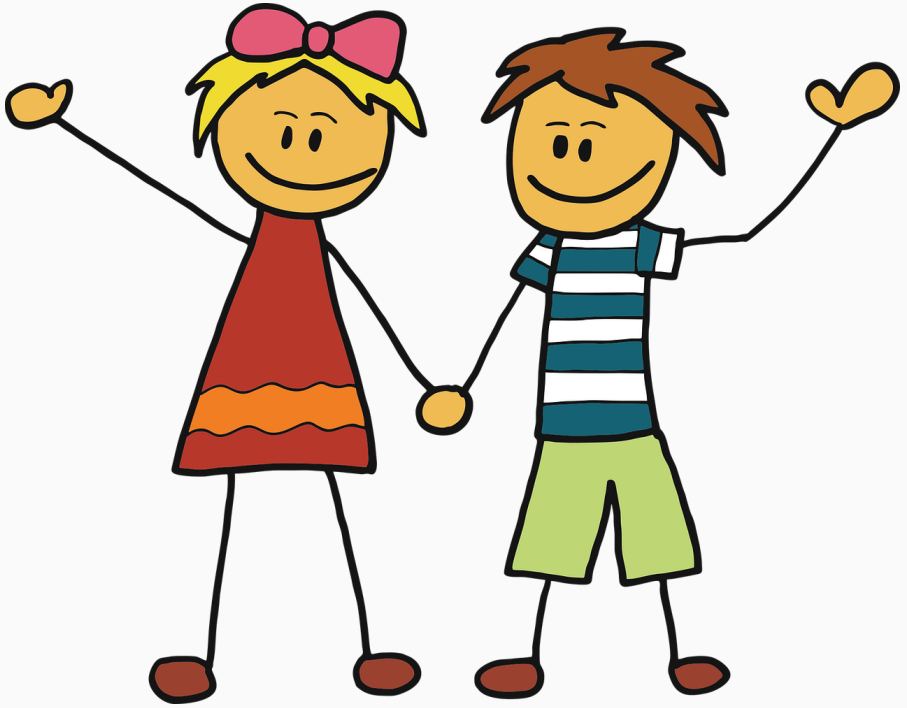
Joins

One of the mainstays of the `dplyr` package is merging data with the family of join operations.

Luckily, the functions are both intuitive to apply and consistent with SQL join vocabulary.

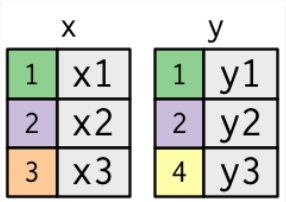
The main functions are:

- `inner_join(df1, df2)`
- `left_join(df1, df2)`
- `right_join(df1, df2)`
- `full_join(df1, df2)`
- `semi_join(df1, df2)`
- `anti_join(df1, df2)`



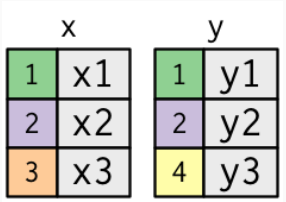
The logic of joins

We start with a simple setup of two tables `x` and `y`:



The logic of joins

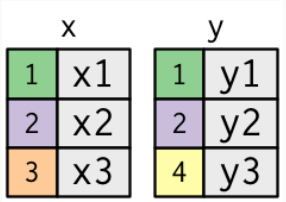
We start with a simple setup of two tables `x` and `y`:



The **colored columns** represent the key variables, the **gray columns** the value variables. The basic ideas of joining will generalize to multiple keys and values.

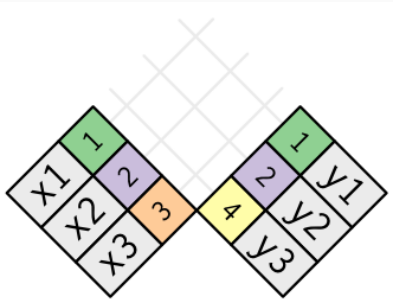
The logic of joins

We start with a simple setup of two tables `x` and `y`:



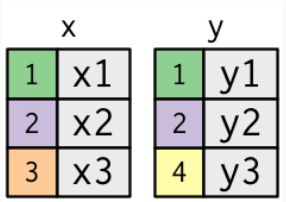
The **colored columns** represent the key variables, the **gray columns** the value variables. The basic ideas of joining will generalize to multiple keys and values.

Joining is about connecting each row in `x` to zero, one, or more rows in `y`:



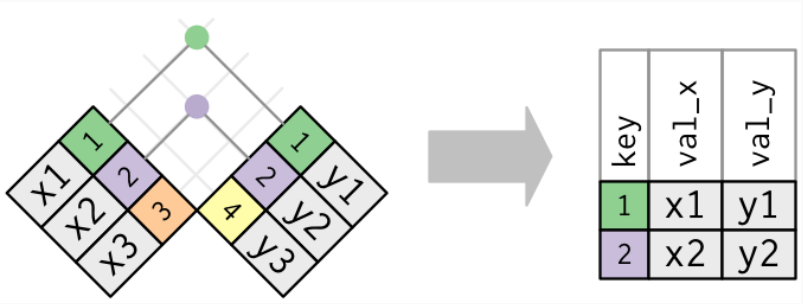
The logic of joins

We start with a simple setup of two tables `x` and `y`:



The **colored columns** represent the key variables, the **gray columns** the value variables. The basic ideas of joining will generalize to multiple keys and values.

Joining is about connecting each row in `x` to zero, one, or more rows in `y`:



In an actual join, matches will be indicated with dots. The number of dots = the number of matches = the number of rows in the output.

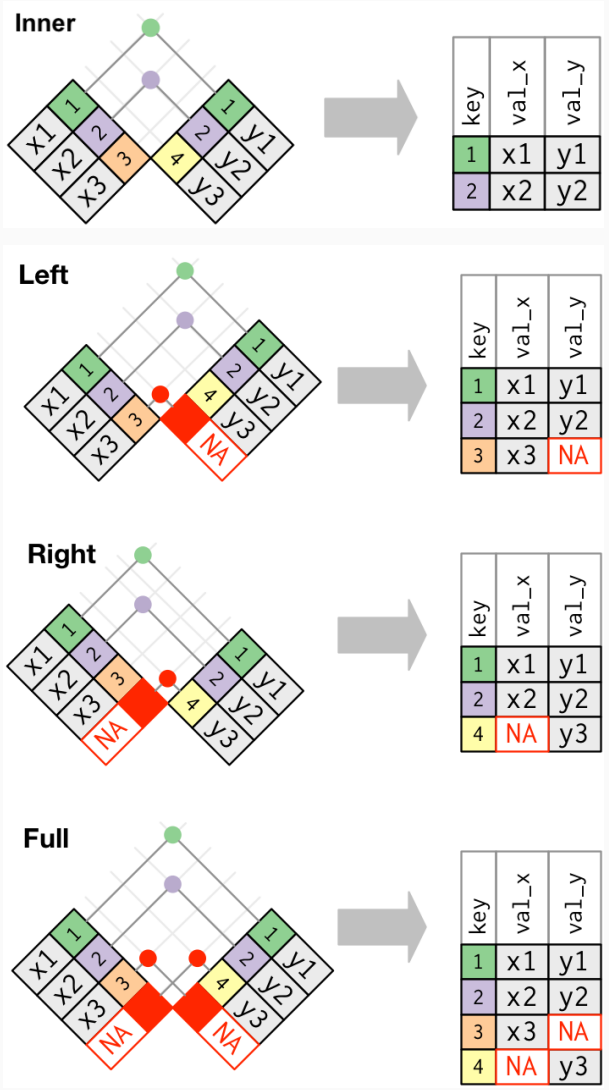
The logic of joins (cont.)

Inner join

- An inner join matches pairs of observations when their keys are equal.
- The output of an inner join is a new table that contains the key and values of both tables.
- Unmatched rows are not included in the result. To be used with caution, because it's easy to lose observations!

Outer joins

- Outer joins keep 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`.
- Use the left join unless you have a strong reason not to do so; it preserves original observations even when there isn't a match.



Keys

- Variables used to connect each pair of tables are called keys.
- A key is a variable (or set of variables) that uniquely identifies an observation.
- In simple cases, a single variable is sufficient to identify an observation (plane → `tailnum`).
- In other cases, multiple variables are needed (look again at the `weather` table).

Keys

- Variables used to connect each pair of tables are called keys.
- A key is a variable (or set of variables) that uniquely identifies an observation.
- In simple cases, a single variable is sufficient to identify an observation (plane → `tailnum`).
- In other cases, multiple variables are needed (look again at the `weather` table).

Primary keys

- A primary key uniquely identifies an observation in its own table.
- It is either a column containing a (sometimes autogenerated and otherwise meaningless) identifier that uniquely identifies each row, or
- Several substantively meaningful columns whose row values *taken together* uniquely identify each row.

Keys

- Variables used to connect each pair of tables are called keys.
- A key is a variable (or set of variables) that uniquely identifies an observation.
- In simple cases, a single variable is sufficient to identify an observation (plane → `tailnum`).
- In other cases, multiple variables are needed (look again at the `weather` table).

Primary keys

- A primary key uniquely identifies an observation in its own table.
- It is either a column containing a (sometimes autogenerated and otherwise meaningless) identifier that uniquely identifies each row, or
- Several substantively meaningful columns whose row values *taken together* uniquely identify each row.

Foreign keys

- A foreign key is a column containing primary key(s) from another table.
- It is the piece of information necessary to join both tables.
- Note that a variable can be both a primary *and* a foreign key. In our example, `origin` is part of the `weather` primary key, and is also a foreign key for the `airports` table.

Joins in R

Let's perform a left join on the flights and planes datasets.

- *Note*: I'm going subset columns after the join, but only to keep text on the slide.

Joins in R

Let's perform a left join on the flights and planes datasets.

- *Note*: I'm going subset columns after the join, but only to keep text on the slide.

```
R> left_join(flights, planes) %>%
+   select(year, month, day, dep_time, arr_time, carrier, flight, tailnum, type, model)
```

```
## Joining with `by = join_by(year, tailnum)`
```

```
## # A tibble: 336,776 × 10
##     year month   day dep_time arr_time carrier flight tailnum type  model
##    <int> <int> <int>    <int>    <int> <chr>    <int> <chr>   <chr> <chr>
##  1  2013     1     1      517      830 UA        1545 N14228  <NA>  <NA>
##  2  2013     1     1      533      850 UA        1714 N24211  <NA>  <NA>
##  3  2013     1     1      542      923 AA        1141 N619AA  <NA>  <NA>
##  4  2013     1     1      544     1004 B6         725 N804JB  <NA>  <NA>
##  5  2013     1     1      554      812 DL         461 N668DN  <NA>  <NA>
##  6  2013     1     1      554      740 UA        1696 N39463  <NA>  <NA>
##  7  2013     1     1      555      913 B6         507 N516JB  <NA>  <NA>
##  8  2013     1     1      557      709 EV        5708 N829AS  <NA>  <NA>
##  9  2013     1     1      557      838 B6          79 N593JB  <NA>  <NA>
## 10  2013     1     1      558      753 AA         301 N3ALAA  <NA>  <NA>
## # ℹ 336,766 more rows
```

Joins in R (cont.)

Note that `dplyr` made a reasonable guess about which columns to join on (i.e. columns that share the same name). It also told us its choices:

```
### Joining, by = c("year", "tailnum")
```

However, there's an obvious problem here: the variable "year" does not have a consistent meaning across our joining datasets!

- In one it refers to the *year of flight*, in the other it refers to *year of construction*.

Joins in R (cont.)

Note that `dplyr` made a reasonable guess about which columns to join on (i.e. columns that share the same name). It also told us its choices:

```
### Joining, by = c("year", "tailnum")
```

However, there's an obvious problem here: the variable "year" does not have a consistent meaning across our joining datasets!

- In one it refers to the *year of flight*, in the other it refers to *year of construction*.

Luckily, there's an easy way to avoid this problem.

- See if you can figure it out before turning to the next slide.
- Try `?dplyr::join`.

Joins in R (cont.)

You just need to be more explicit in your join call by using the `by =` argument.

- You can also rename any ambiguous columns to avoid confusion.

```
R> left_join(
+   flights,
+   planes %>% rename(year_built = year), ### Not necessary w/ below line, but helpful
+   by = "tailnum" ### Be specific about the joining column
+   ) %>%
+   select(year, month, day, dep_time, arr_time, carrier, flight, tailnum, year_built, type, model) %>%
+   head(3) ### Just to save vertical space on the slide
```

```
## # A tibble: 3 × 11
##    year month   day dep_time arr_time carrier flight tailnum year_built type
##   <int> <int> <int>    <int>    <int> <chr>    <int> <chr>        <int> <chr>
## 1  2013     1     1      517      830 UA        1545 N14228        1999 Fixed w…
## 2  2013     1     1      533      850 UA        1714 N24211        1998 Fixed w…
## 3  2013     1     1      542      923 AA        1141 N619AA        1990 Fixed w…
## # ℹ 1 more variable: model <chr>
```

Joins in R (cont.)

Last thing to mention on joins for now; note what happens if we again specify the join column... but don't rename the ambiguous "year" column in at least one of the given data frames:

```
R> left_join(
+   flights,
+   planes, ### Not renaming "year" to "year_built" this time
+   by = "tailnum"
+   ) %>%
+   select(contains("year"), month, day, dep_time, arr_time, carrier, flight, tailnum, type, model) %>%
+   head(3)

## # A tibble: 3 × 11
##   year.x year.y month   day dep_time arr_time carrier flight tailnum type  model
##    <int>  <int> <int> <int>    <int>    <int> <chr>    <int> <chr>   <chr> <chr>
## 1   2013   1999     1     1      517      830 UA        1545 N14228  Fixe… 737-…
## 2   2013   1998     1     1      533      850 UA        1714 N24211  Fixe… 737-…
## 3   2013   1990     1     1      542      923 AA        1141 N619AA  Fixe… 757-…
```

Joins in R (cont.)

Last thing to mention on joins for now; note what happens if we again specify the join column... but don't rename the ambiguous "year" column in at least one of the given data frames:

```
R> left_join(
+   flights,
+   planes, ### Not renaming "year" to "year_built" this time
+   by = "tailnum"
+   ) %>%
+   select(contains("year"), month, day, dep_time, arr_time, carrier, flight, tailnum, type, model) %>%
+   head(3)
```

```
## # A tibble: 3 × 11
##   year.x year.y month   day dep_time arr_time carrier flight tailnum type  model
##    <int>  <int> <int> <int>    <int>    <int> <chr>    <int> <chr>   <chr> <chr>
## 1   2013   1999     1     1      517      830 UA        1545 N14228  Fixe… 737-…
## 2   2013   1998     1     1      533      850 UA        1714 N24211  Fixe… 737-…
## 3   2013   1990     1     1      542      923 AA        1141 N619AA  Fixe… 757-…
```

Make sure you know what "year.x" and "year.y" are. Again, it pays to be specific.

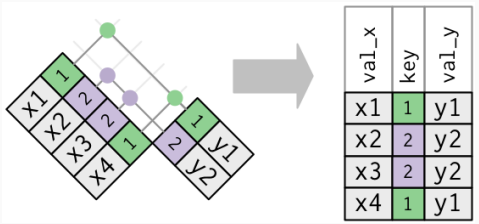
Duplicate keys

If you're lucky, keys are unique. But that's not always the case. There are two common scenarios:

Duplicate keys

If you're lucky, keys are unique. But that's not always the case. There are two common scenarios:

1. One table has duplicate keys, the other hasn't. This gives us a one-to-many relationship.

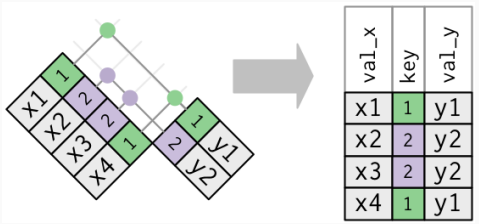


val_x	key	val_y
x1	1	y1
x2	2	y2
x3	2	y2
x4	1	y1

Duplicate keys

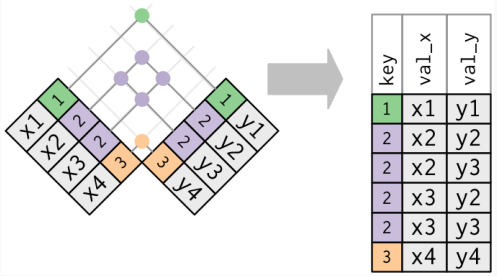
If you're lucky, keys are unique. But that's not always the case. There are two common scenarios:

1. One table has duplicate keys, the other hasn't. This gives us a one-to-many relationship.



val_x	key	val_y
x1	1	y1
x2	2	y2
x3	2	y2
x4	1	y1

2. Both tables have duplicate keys. This is usually an error because in neither table do the keys uniquely identify an observation. When joining duplicated keys, we get all possible combinations (the Cartesian product, a many-to-many relationship):



key	val_x	val_y
1	x1	y1
2	x2	y2
2	x2	y3
2	x3	y2
2	x3	y3
3	x4	y4

SQL

What is SQL?

Background and history

- SQL (pronounced [ɛsˌkjuːˈɛl] as in S-Q-L, or [siːkwəl] as in sequel) stands for **Structured Query Language**. Initially it was called SEQUEL (Structured English Query Language), but this was dropped due to trademark issues.
- It's a domain-specific language designed to query data contained in relational databases.
- Initially developed at IBM by Donald D. Chamberlin and Raymond F. Boyce in 1974.

What is SQL?

Background and history

- SQL (pronounced [ɛsˌkjuːˈɛl] as in S-Q-L, or [siːkwəl] as in sequel) stands for **Structured Query Language**. Initially it was called SEQUEL (Structured English Query Language), but this was dropped due to trademark issues.
- It's a domain-specific language designed to query data contained in relational databases.
- Initially developed at IBM by Donald D. Chamberlin and Raymond F. Boyce in 1974.

Why SQL?

- While database types differ, most of the relational databases you'll encounter speak SQL.
- The key skill to work with databases (outside R) is to learn how to speak SQL. Once you've mastered this, you should be able to work with any of them.
- SQL is featured as a required skill in many (most?) data science job ads out there.

General SQL syntax

Classes of SQL syntax

- **Data query language** (DQL) to perform queries on the data [`SELECT`, `FROM`, `WHERE`]
- **Data definition language** (DDL) to describe data structure and its relations (create tables, columns, define data types, keys, constraints) [`CREATE`, `ALTER`, `DROP`]
- **Data manipulation language** (DML) to fill database or retrieve information from it [`SELECT`, `FROM`, `WHERE`, `INSERT`, `UPDATE`, `DELETE`]
- **Data control language** (DCL) to define usage/admin rights [`GRANT`, `REVOKE`]

General SQL syntax

Classes of SQL syntax

- **Data query language** (DQL) to perform queries on the data [`SELECT`, `FROM`, `WHERE`]
- **Data definition language** (DDL) to describe data structure and its relations (create tables, columns, define data types, keys, constraints) [`CREATE`, `ALTER`, `DROP`]
- **Data manipulation language** (DML) to fill database or retrieve information from it [`SELECT`, `FROM`, `WHERE`, `INSERT`, `UPDATE`, `DELETE`]
- **Data control language** (DCL) to define usage/admin rights [`GRANT`, `REVOKE`]

A generic query

The main SQL tool is `SELECT`, which allows you to perform queries on a table in a database. It has the generic form:

```
R> SELECT columns or computations
+   FROM table
+   WHERE condition
+   GROUP BY columns
+   HAVING condition
+   ORDER BY column [ASC | DESC]
+   LIMIT offset,count;
```

General SQL syntax (cont.)

Basic rules

- SQL statements start with a command describing the desired action (`SELECT`), followed by the unit on which it should be executed (`SELECT column1`), and one or more clauses (`WHERE column 2 = 1`).
- Although it's customary to write all SQL statements in capital letters, SQL is actually case insensitive towards its key words.
- Each SQL statement ends with a semicolon, so SQL statements can span across multiple lines.
- Comments either start with `--` or have to be put in between `/*` and `*/`.

```
R> CREATE DATABASE database1 ;
+ SELECT column1 FROM table1 WHERE column2 = 1 ;
+ UPDATE table1 SET column1 = 1 WHERE column2 > 3 ;
+ INSERT INTO table1 (column1, column2)
+ VALUES ('rc11', 'rc12'), ('rc21', 'rc22') ;

R> -- One line comment.
+ /*
+ Comment spanning
+ several lines
+ */
```

General SQL syntax (cont.)

Notes from a dplyr user

- SQL syntax is intuitive until it isn't.
- One problem: SQL imposes a **lexical order** of operations ("order of execution") which does not necessarily match the **logical order** of operations you'd have in mind.
- Conceptually, every step (like "`WHERE`") of the query transforms its input. However, the query's steps don't happen in the order they're written (e.g., `SELECT` is written first but the operation comes much later in the process).
- Compare this to our `dplyr` logic: Take this, do this, then do this, etc.

General SQL syntax (cont.)

Notes from a dplyr user

- SQL syntax is intuitive until it isn't.
- One problem: SQL imposes a **lexical order** of operations ("order of execution") which does not necessarily match the **logical order** of operations you'd have in mind.
- Conceptually, every step (like "`WHERE`") of the query transforms its input. However, the query's steps don't happen in the order they're written (e.g., `SELECT` is written first but the operation comes much later in the process).
- Compare this to our `dplyr` logic: Take this, do this, then do this, etc.

The good news

- With `dplyr` we have a package that can effectively speak SQL for us.
- Those queries (formulated with `dplyr` commands!) can then easily be submitted as SQL queries to the database (also thanks to the `DBI` package).
- So, do you still have to learn SQL? Probably not, but it won't hurt!

More resources to get started with SQL

Probably the best way to learn SQL hands-on are interactive tutorials. Check out the one at SQLBolt, SQL Teaching, or Lost SQL.

Another hands-on beginner SQL tutorial is at DataQuest.

Yet another good starting point to learn about the basics of SQL lives at Codeacademy.

Once you have made yourself more familiar with the language, work on your style. This SQL style guide is a good starting point.

If you want to know more about how indexes work in SQLite, check out this resource.

Finally, another good read is 10 easy steps to a complete understanding of SQL by Lukas Eder.

Honorary mentions go to SQL for the rest of us and SQL Zoo.

Talking to databases with R

Databases and Database Management Systems

What are databases?

- Databases are an organized collection of data.
- They are organized to afford efficient retrieval of (selections) of data.
- They entail data + metadata about structure and organization.
- They are generally accessed through a database management system.

Where are databases?

- Databases can exist locally or remotely, in-memory or on-disk.
- When they are stored locally, they are stored as binary file (not text file).
- Commonly, we think of a **client-server model**:
  - Databases live on a **server**, which manages them.
  - Users interact with the server through a **client** program.
  - Multiple users can **access** the same database **simultaneously**.

Databases and Database Management Systems

What are DBMS?

- Database Management Systems (DBMS) provide **efficient**, **reliable**, **convenient**, **safe**, **multi-user** storage of and access to **massive** data.
  - **Efficient**: Just fast.
  - **Reliable**: High uptime.
  - **Convenient**: High-level query languages.
  - **Safe**: Robust to power outages, node failures, etc.
  - **Multi-user**: Concurrency control. Not one user, but multiple.
  - **Massive**: Think Terabytes, not Gigabytes. Handle data that resides outside memory.
- There are so many DBMS for relational database structures alone.
- RDBMS differ in terms of capabilities, implemented features, operating system support, and much more. See The Database Database for a nice overview!

Databases and Database Management Systems (cont.)

Three basic forms of DBMS

- **Client-server DBMS**: Runs on a powerful central server, which you connect from your computer (the client). Great for sharing data with multiple people in an organization. Popular client-server DBMS's include PostgreSQL, MariaDB, SQL Server, and Oracle.
- **Cloud DBMS**: Similar to client server DBMS's, but they run in the cloud. This means that they can easily handle extremely large datasets and can automatically provide more compute resources as needed. , Popular examples: Snowflake, Amazon's RedShift, and Google's BigQuery.
- **Embedded (in-process) DBMS**: Run entirely on your computer or within an application. They're great for working with large datasets where you're the primary user. Examples: SQLite or DuckDB.

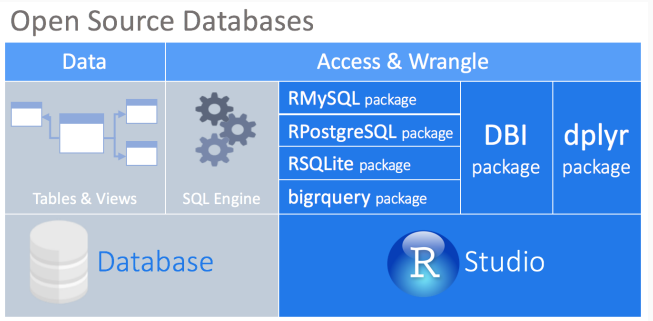
What's the right DBMS for you?

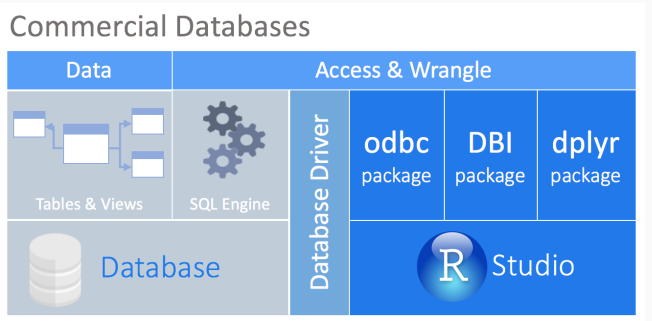
- You'll probably not be in the position to decide which database to use. If you're still interested in the differences, this or this overview might be a good starting point.

Connecting to databases with R

A database interface

- DBMS implement SQL but all have somewhat different conventions.
- R can connect to all major existing DBMS types.
- The R package `DBI` (**D**ata**B**ase **I**nterface) is a unified interface to them (car analogy: wheel, pedals).
- In addition, we need a separate "driver" to connect to the database engine (car analogy: key).





`Credit` db.rstudio.com

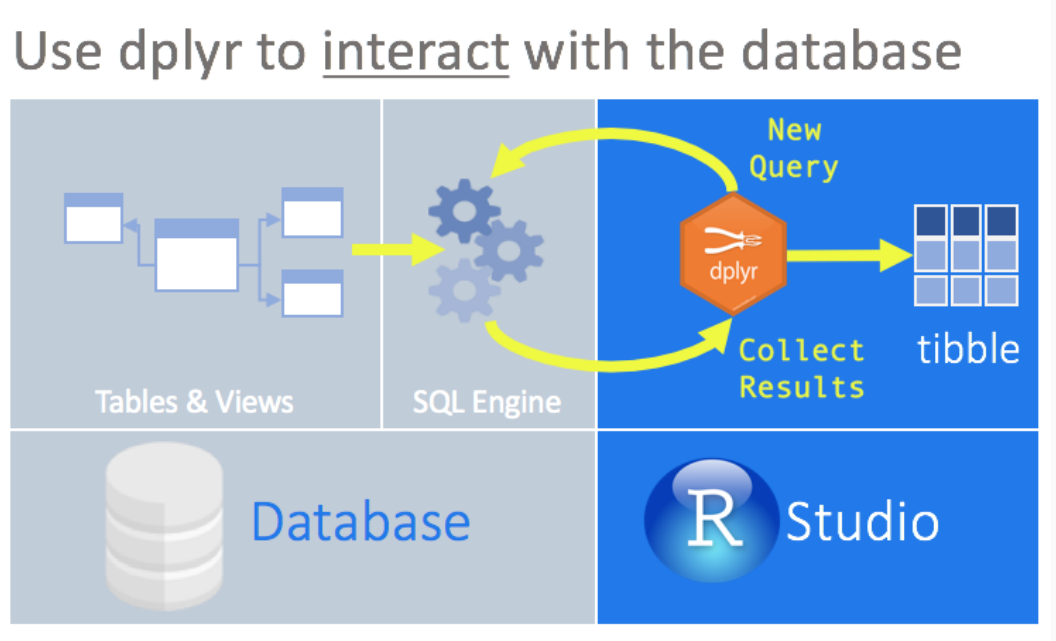
Drivers for open-source/commercial DBs

- There are various R packages that allow you to connect to particular **open-source** DBMS types including SQLite (via `RSQLite`), MySQL (`RMySQL`), PostgreSQL (`RPostgres`), Google BigQuery (`bigrquery`), MariaDB (`RMariaDB`), DuckDB (`duckdb`), and more.
- For **commercial databases** such as Microsoft SQL Server or Oracle, the `odbc` package provides a DBI-compliant interface to Open Database Connectivity (ODBC) drivers.

Talking to databases

dplyr as a database interface

- Good news: `dplyr` is able to interact with databases directly by translating the `dplyr` verbs into SQL queries. This convenient feature allows you to "speak" directly with the database from R.
- Using `dplyr` as an interface allows you to:
  1. Run data exploration routines over all of the data, instead of importing parts of the data into R.
  2. Use the SQL engine to run the data transformations. In effect, computation is being pushed to the database.
  3. Collect into R only a targeted dataset.
  4. Keep all your code in R. There is no need to alternate between languages or tools to perform the data exploration.



`Credit` db.rstudio.com

A hands-on database session with R

Before we start, we load the `DBI` package to connect to a DBMS and the `RSQLite` package to communicate with an SQLite database. Tidyverse and `dplyr` / `dbplyr` will be used, too. Also, we will rely on `nycflights13` for some toy data sets.

```
R> library(DBI)
R> library(RSQLite)
R> library(tidyverse)
R> library(nycflights13)
```

Connecting to the database

Now, let's set up a connection with an SQLite database. In fact, we will not interact with an existing one but build up our own, which will live in the memory:

```
R> con ← dbConnect(RSQLite::SQLite(), dbname = ":memory:")
```

`con` represents our database connection via which we'll interact with the database.

Connecting to the database

Now, let's set up a connection with an SQLite database. In fact, we will not interact with an existing one but build up our own, which will live in the memory:

```
R> con ← dbConnect(RSQLite::SQLite(), dbname = ":memory:")
```

`con` represents our database connection via which we'll interact with the database.

The arguments to `DBI::dbConnect()` vary from database to database, but the first argument is always the database backend. For instance, it's `RSQLite::SQLite()` for RSQLite, `RMariaDB::MariaDB()` for RMariaDB, `RPostgres::Postgres()`, and `bigrquery::bigquery()` for BigQuery.

Connecting to the database

Now, let's set up a connection with an SQLite database. In fact, we will not interact with an existing one but build up our own, which will live in the memory:

```
R> con ← dbConnect(RSQLite::SQLite(), dbname = ":memory:")
```

`con` represents our database connection via which we'll interact with the database.

The arguments to `DBI::dbConnect()` vary from database to database, but the first argument is always the database backend. For instance, it's `RSQLite::SQLite()` for RSQLite, `RMariaDB::MariaDB()` for RMariaDB, `RPostgres::Postgres()`, and `bigrquery::bigquery()` for BigQuery.

Also, in real life, chances are that the database lives on a server and you have to authenticate to connect to it. This could look as follows:

```
R> con ← DBI::dbConnect(RSQLite::SQLite(),
+                       host = "mydatabase.host.com",
+                       user = "simon",
+                       password = "mypassword"
+ )
```

Filling the database

Next, we upload a local data frame into the remote data source; here: our database. Note that this is specific to our (toy) example. You'll probably not have to build up your own database.

```
R> dplyr::copy_to(
+   dest = con,
+   df = nycflights13::flights,
+   name = "flights")
```

Filling the database

Next, we upload a local data frame into the remote data source; here: our database. Note that this is specific to our (toy) example. You'll probably not have to build up your own database.

```
R> dplyr::copy_to(
+   dest = con,
+   df = nycflights13::flights,
+   name = "flights",
+   temporary = FALSE,
+   indexes = list(
+     c("year", "month", "day"),
+     "carrier",
+     "tailnum",
+     "dest"
+   )
+ )
```

We can also explicitly set up indexes that will allow us to quickly process the data by day, carrier, plane, and destination. While creating the right indices is key to good database performance, in common applications this will be taken care of by the database maintainer.

Querying the database

Now it's time to start querying our database. First, we generate a reference table from the database using `dplyr`'s `tbl()`:

```
R> flights_db ← tbl(con, "flights")
```

Note that `flights_db` is a remote source; the table is not stored in our local environment. We can use it as a "pointer" to the actual database. Next, we perform various queries, such as:

```
R> flights_db %>% select(year:day, dep_delay, arr_delay) %>% head(3)
```

```
## # Source:   SQL [3 x 5]
## # Database: sqlite 3.41.2 [/Users/simonmunzert/Munzert Dropbox/Simon Munzert/github/intro-to-data-science-23/lectu
##    year month   day dep_delay arr_delay
##   <int> <int> <int>     <dbl>     <dbl>
## 1  2013     1     1         2        11
## 2  2013     1     1         4        20
## 3  2013     1     1         2        33
```

Yes, we can use `dplyr` syntax to do database queries!

Querying the database

The most important difference between ordinary data frames and remote database queries is that your R code is translated into SQL and executed in the database on the remote server, not in R on your local machine. When working with databases, `dplyr` **tries to be as lazy as possible**:

- It never pulls data into R unless you explicitly ask for it.
- It delays doing any work until the last possible moment: it collects together everything you want to do and then sends it to the database in one step.

This even applies when you assign the output of a database query to an object:

```
R> tailnum_delay_db ← flights_db %>%
+   group_by(tailnum) %>%
+   summarise(
+     delay = mean(arr_delay),
+     n = n()
+   ) %>%
+   arrange(desc(delay)) %>%
+   filter(n > 100)
```

Querying the database

Laziness also has some downsides. Because there's generally no way to determine how many rows a query will return unless you actually run it, nrow() is always NA:

```
R> nrow(tailnum_delay_db)
```

```
## [1] NA
```

Because you can't find the last few rows without executing the whole query, you can't use tail():

```
R> tail(tailnum_delay_db)
```

```
## Error in `tail()`:
## ! `tail()` is not supported on database backends.
```

If you then want to pull the data into a local data frame, use `dplyr::collect()`:

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

```
## # A tibble: 1,201 × 3
##    tailnum delay     n
##    <chr>   <dbl> <int>
```

Using SQL directly in R

Again, because it is so cool: Yes, we can use `dplyr` syntax to do database queries! Behind the scenes, `dplyr` is translating your R code into SQL. The `dbplyr` package is doing the work for us.

You can use the `show_query()` function to display the SQL code that was used to generate a queried table:

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

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

If you still want to formulate SQL queries and pass them on to the DBMS, use `DBI::dbGetQuery()`:

```
R> sql_query ← "SELECT * FROM flights WHERE dep_delay > 240.0 LIMIT 5"
R> dbGetQuery(con, sql_query)
```

```
##   year month day dep_time sched_dep_time dep_delay arr_time sched_arr_time
## 1 2013     1   1      848           1835       853     1001           1950
## 2 2013     1   1     1815           1325       290     2120           1542
```

More useful resources to get started with databases

Relational databases FTW

Get started with The Beginner's Guide to Databases.

A comprehensive overview of database interaction using R is available at db.rstudio.com. It also features a set of best practices that go beyond what we covered today.

Not to mention Databases 101 by Grant McDermott, which also features example code to connect to DuckDB and Google BigQuery.

Also, there is the introduction to `dbplyr` which comes as a package vignette.

Finally, Awesome MySQL, a goldmine when you work with MySQL.

And what about NoSQL databases?

We haven't talked about non-relational databases so far, but you might encounter them in the future.

NoSQL became popular in the 2000s as a consequence of several developments: Cost of data storage decreased, which made the need of efficient, non-redundant but complex data storage less urgent. At the same time, heterogeneity in data formats increased and coming up with well-defined schemas was difficult. NoSQL offered more flexibility.

There are several types of NoSQL databases, including document DBs where documents contain pairs of fields and values (think: JSON), key-value DBs, or graph DBs.

Depending on your use case, this might be exactly what you need. If you want to learn more, start here or here.

Summary

Summing it up

Most data scientists **never design a database**.

- But they almost all end up interacting with them.
- Also, with database administrators (DBAs).

Lots of academic data work consists of working with *bad re-inventions* of the relational database.

- Looking at you, Excel users.
- Looking at you, instructor.

Thinking about data relationally **will help you with statistics**.

- Multilevel models
- Time-series
- Network analysis
- NLP

Database concept	Statistical concept
Table	Sample
Column	Variable
Row	Unit
Value	Observation
Foreign key relationships	Nested variables
Many-to-many relationships	Crossed variables (possibly unbalanced)

Databases: Maybe not the most exciting technology...

... but awesomely useful and not going away.

Coming up

Assignment

None! But you'll get the chance to practice databasing with R in the lab.

Next lecture

Web data and technologies - relational data structures FTW!