Statistical Programming and Open Science Methods

Relational databases and the concept of normalized data

Joachim Gassen
Humboldt-Universität zu Berlin

September 02, 2022

SFB/Transregio 266
ACCOUNTING FOR TRANSPARENCY

Time table October 11

When?	What?
09:00	Writing readable and reusable code
10:30	Coffee
11:00	Debugging tools
12:30	Lunch and coffee
13:30	Relational databases and the concept of normalized data
14:30	Data wrangling and visualization fundamentals
15:30	Assignments and wrap up
16:00	End of event

Disclaimer

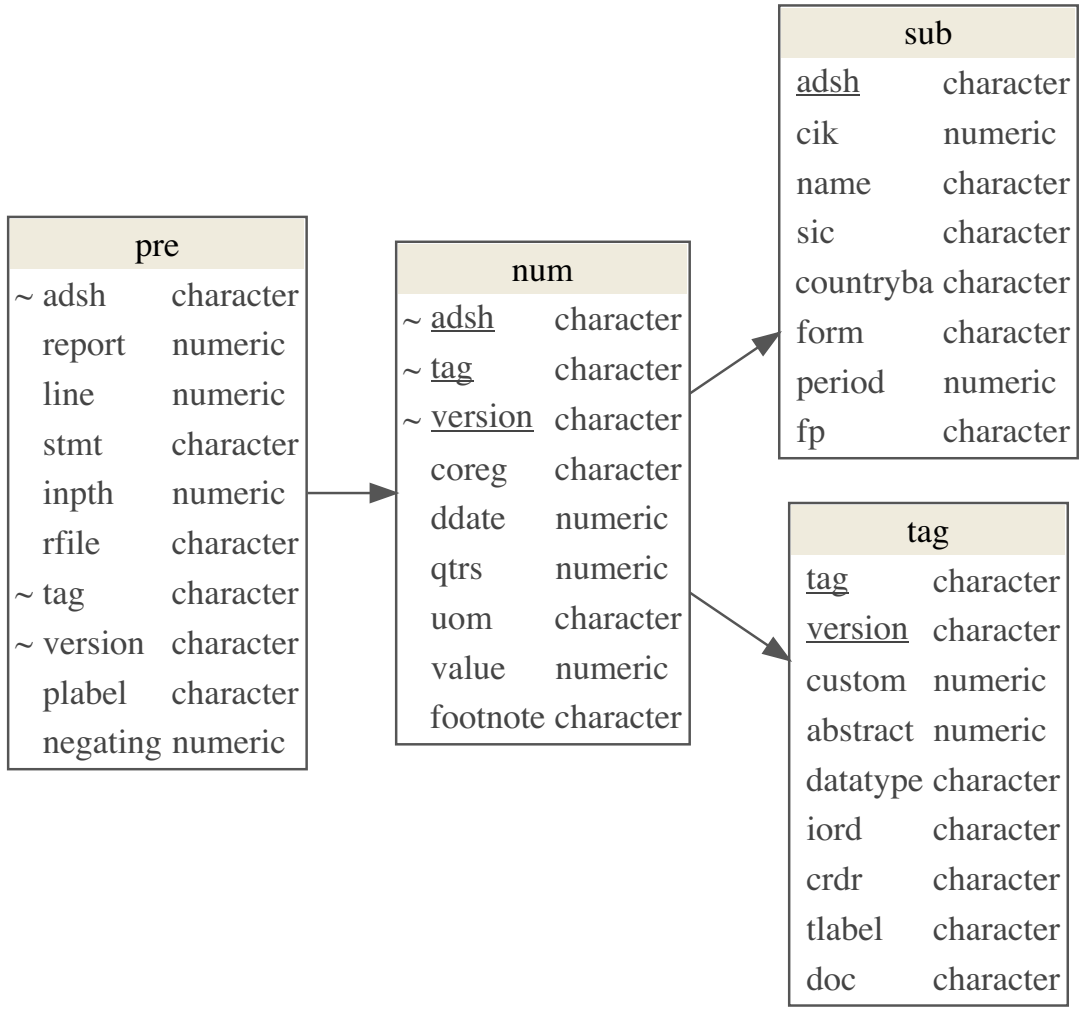
Some of the following — in particular the nice-looking figures — are borrowed from Garrett Grolemund and Hadley Wickham (2017): R for Data Science, https://r4ds.had.co.nz

Think about the following scenarios

- You have a project that collects data via a web form/survey
- You have a time-intensive process that you want to multi-thread across several platforms
- You have big data that you need to query but the amount of data that you actually need for your analysis is relatively small
- You are reusing data for multiple projects

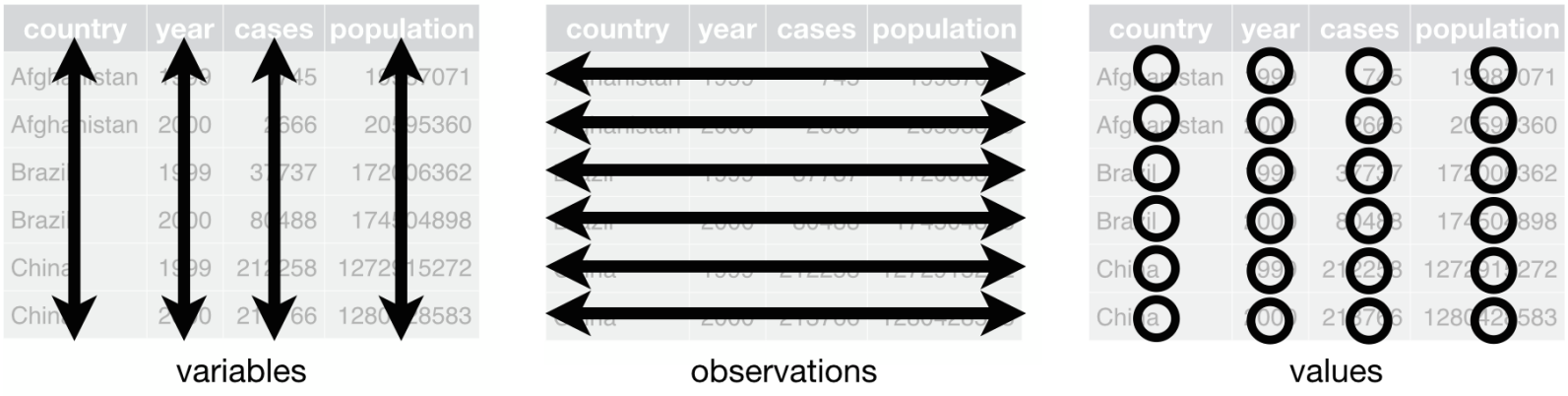
In all these cases you might need a relational database management system (RDBMS)

Our SEC data has a relational structure . . .



Understanding the concept of relations: Fundamentals

- Observations are stored in rows, variables are stored in columns
- Only one observation per row, only one variable per column
- Each observation has a unique primary key (might be a set of keys)

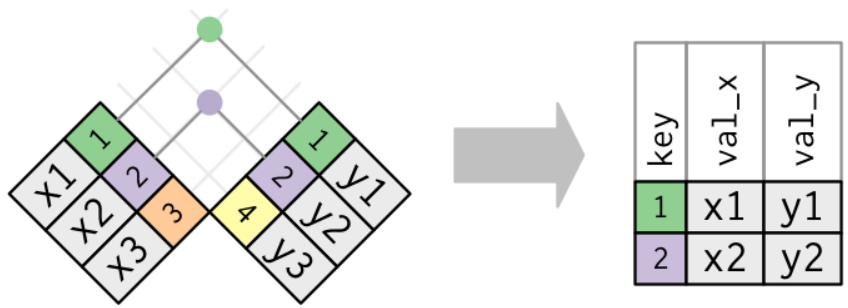


Primary and foreign keys

- A primary key (or set of keys) identifies an observation in a given dataset/table
- A foreign key (or a set of keys) identifies an observation in *another* dataset/table
- This implies that a foreign key is not necessary unique (while the primary key is)
- In is a very useful habit to organize joins so that you use the foreign key of the first (left) and the primary key of the second (right) dataset

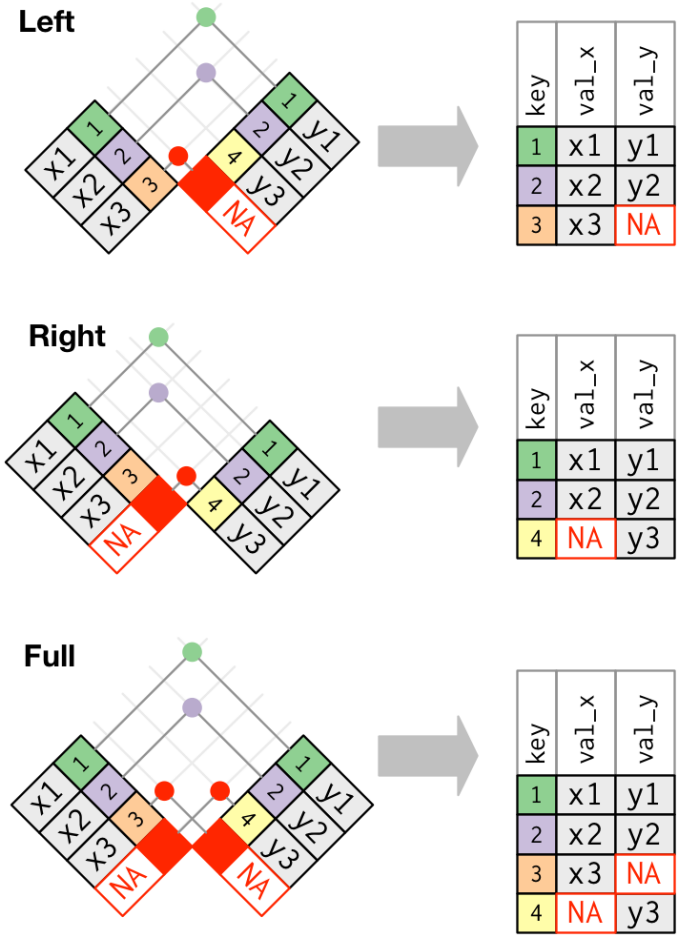
Joining datasets: 1:1 joins

- There are various type of joins: The inner join



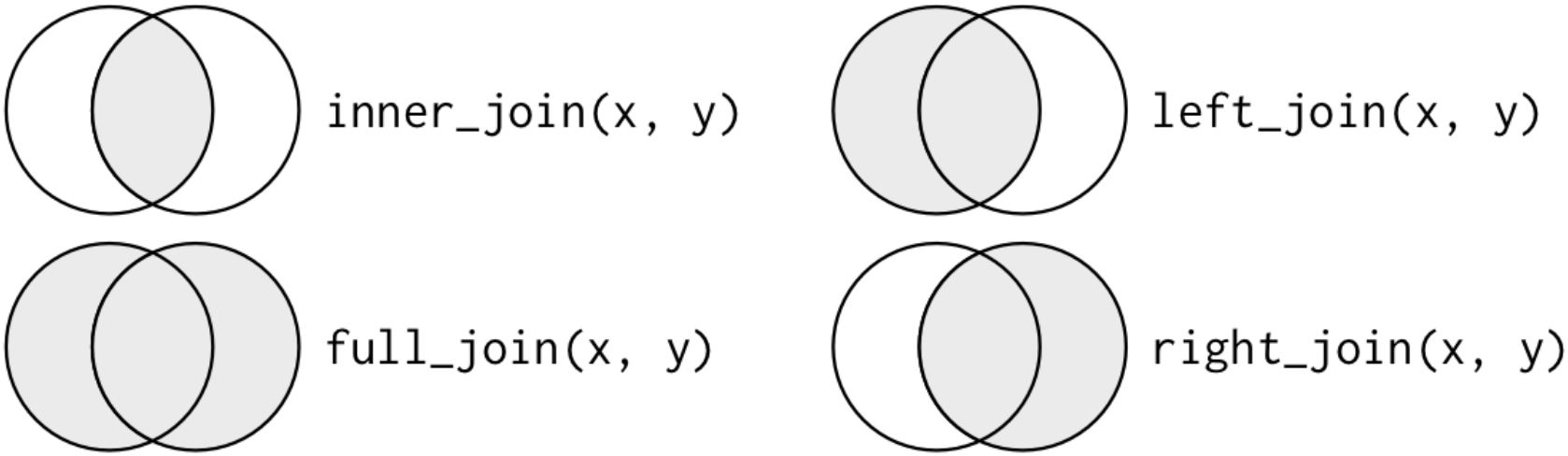
Joining datasets: 1:1 joins

- The outer joins



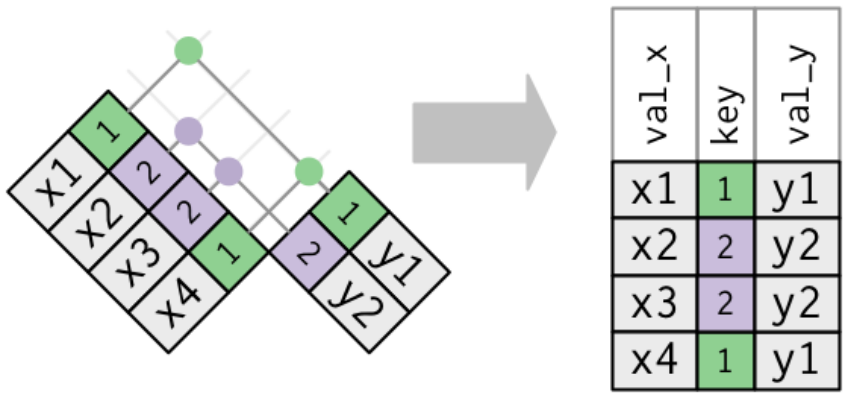
Joining datasets: 1:1 joins

- The 1:1 joins combined



Joining datasets: 1:n joins

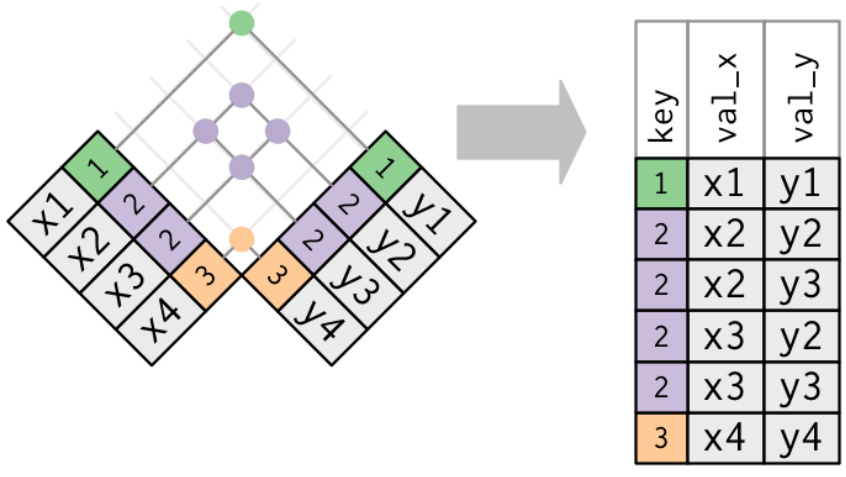
- Joins one observation from one dataset to potentially many observations from another dataset. Can and in most cases will enlarge the sample size



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

Joining datasets: n:m joins (the wrong way)

- Directly joining two datasets that form a n:m relation is not feasible as observations are no longer uniquely defined in *both* tables.



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

Joining done right: A “case study” about normalization

Assume that we run an event for the TRR 266. People register for the event. To keep track of registration, we set up the following data set.

```
participant = tibble(
  name = c("Andreas", "Tina", "Xhi"),
  email = c("andi@gmail.com", "tina@hu-berlin.de", "xhi@upb.de"),
  project = c("A02", "B02", "A05, B08")
)
```

What do you think about this approach?

Oh, OK. You want a key

All good now?

```
participant <- tibble(
  participant_id = 1:3,
  name = c("Andreas", "Tina", "Xhi"),
  email = c("andi@gmail.com", "tina@hu-berlin.de", "xhi@upb.de"),
  project = c("A02", "B02", "A05, B08")
)
```

`project` is not atomic

`participants$project[3]` (`"A05, B08"`) contains two observations. How to fix this? Maybe by:

```
participant <- tibble(
  participant_id = 1:3,
  name = c("Andreas", "Tina", "Xhi"),
  email = c("andi@gmail.com", "tina@hu-berlin.de", "xhi@upb.de"),
  project1 = c("A02", "B02", "A05"),
  project2 = c(NA, NA, "B08")
)
```

Now everything is atomic (One value per cell). Technically, we have reached the first normal form of a relational database. Is it a smart solution?

Addressing insertion anomalies

No. We created an *insertion anomaly*. If somebody happens to be on three projects, we would be unable to enter this information in our data model. Let's address this.

```
participant <- tibble(
  participant_id = 1:3,
  name = c("Andi", "Tina", "Xhi"),
  email = c("andi@gmail.com", "tina@hu-berlin.de", "xhi@upb.de")
)

people_project <- tibble(
  participant_id = c(1, 2, 3, 3),
  project_id = c("A02", "B02", "A05", "B08")
)
```

Adding data

We are happy about our new data model but we do not know what these fancy project IDs stand for. Why not add project names?

```
people_project$project_title <- c(
  "Transparency Effects of Organizational Innovations",
  "Private Firm Transparency",
  "Accounting for Tax Complexity",
  "Tax Burden Transparency"
)
```

We are happy. Right?

Meet the second normal form (2NF)

We just introduced data into your model that only depends on `project_id`, not on `participant_id`. As both together define the key of the dataset 'people_in_projects, we violated the requirements of second form normality: All non key data in a table have to depend on *all* keys, not on a subset.

This is important as a violation creates an *update anomaly*. In our case: When we change a project title, we potentially need to change in it multiple instances in your data.

Let's fix this.

```
project <- tibble(
  project_id = people_project$project_id,
  project_title = people_project$project_title
)

people_project <- people_project %>%
  select(- project_title)
```

Deleting data . . .

We are very proud of ourselves. Now Tina calls. She can't make it to our event. No problem. We simply delete her entry and send her an email to acknowledge that she is no longer registered.

```
participant <- participant %>%
  filter(participant_id != 2)
```

But, hey, where did her email go?

The third normal form

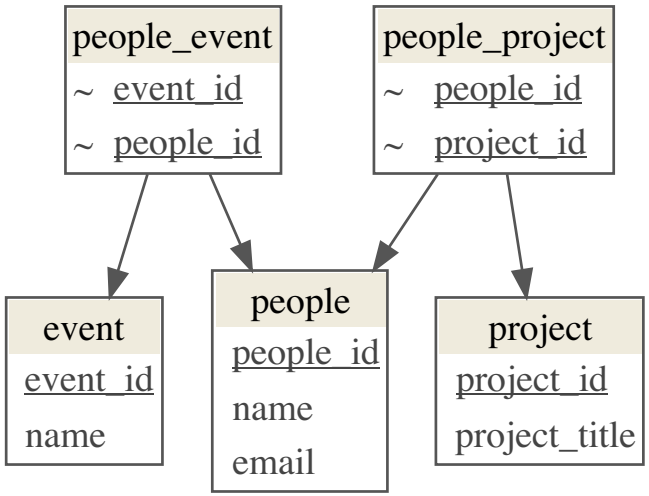
Our `participant` table is subject to a *deletion anomaly* as it is not in the third normal form (3NF). When we delete data, we also delete data that is still informative (the email in our case).

The third normal form requires on top of the second normal form that there are no *transitional dependencies* in our data, meaning that all the non-keys are directly depending on the keys and not on some other variable that in turn depends on the keys.

Time for a last change

```
people <- tibble(
  people_id = 1:3,
  name = c("Andi", "Tina", "Xhi"),
  email = c("andi@gmail.com", "tina@hu-berlin.de", "xhi@upb.de")
)

people_event <- tibble(
  event_id = c(1, 1),
  people_id = c(1, 3)
)

event <- tibble(
  event_id = 1,
  name = "Our super nice event"
)

people_project <- people_project %>%
  rename(people_id = participant_id)

rm(participant)
```

Our data model and voila: two n:m relations



Using external RDBMS for storage

- Using database solutions external to your own programming code has many advantages
- Your can access your data from various platforms
- RDBMS take care of potential race conditions when concurrent access is feasible
- Data integrity is maintained across applications
- Query speed (at least in many cases)

A simple example: SQLite

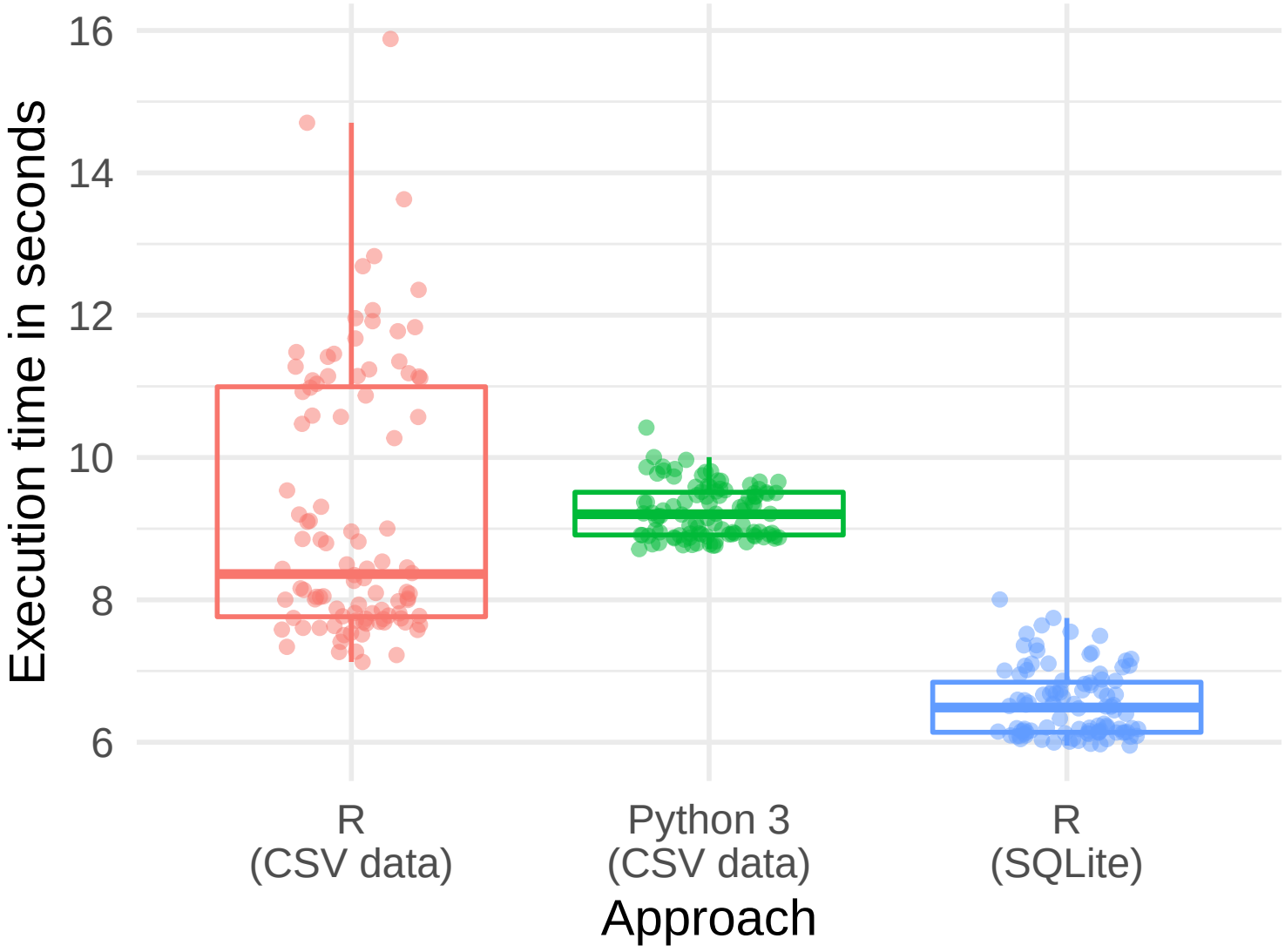
- `SQLite` is a light-weight single-user file-based RDBMS
- It qualifies as ACID: Transactions are *atomic*, *consistent*, *isolated*, and *durable*.
- Using such a RDBMS does not avoid all race conditions (concurrent data changes leaving data in inconsistent state) but significantly reduces the situations where race conditions can incur

Larger scale RDBMS that can be outsourced to different servers include *PostgreSQL*, *MySQL*, and *MariaDB*

Moving our SEC data to SQLite

```
library(DBI)
if (file.exists("../data/sec.sqlite3"))
  unlink("../data/sec.sqlite3")
con <- dbConnect(RSQLite::SQLite(), "../data/sec.sqlite3")
for(df_name in c("sub", "tag", "num", "pre")) {
  dbWriteTable(
    con, df_name,
    read_csv(paste0("../data/", df_name, ".csv"))
  )
}
dbListTables(con)
## [1] "num" "pre" "sub" "tag"
dbDisconnect(con)
```

It pays performance-wise (100 runs each)



Note: Runs for Stata (CSV data) executed from plot (mean = 148)