

Wrangling with Databases

Data Wrangling, Session 7c

Kieran Healy

Code Horizons

January 2026

Load the packages, as always

```
library(here)      # manage file paths
library(socviz)    # data and some useful functions
library(tidyverse) # your friend and mine
library(gapminder) # inescapable

library(DBI) # DBMS interface layer
library(duckdb) # Local database server
```

“Big” Data

What we're talking about

Mostly in this case, datasets that are nominally larger than your laptop's memory.

There are other more specific uses, and truly huge data is beyond the scope of the course. But we can look at methods for working with data that's "big" for all practical purposes.

Databases

When we're working with data in the social sciences the basic case is a single table that we're going to do something with, like run a regression or make a plot.

```
gapminder
```

```
# A tibble: 1,704 × 6
  country   continent year lifeExp      pop gdpPercap
  <fct>     <fct>    <int>   <dbl>    <int>     <dbl>
1 Afghanistan Asia      1952    28.8  8425333    779.
2 Afghanistan Asia      1957    30.3  9240934    821.
3 Afghanistan Asia      1962    32.0  10267083   853.
4 Afghanistan Asia      1967    34.0  11537966   836.
5 Afghanistan Asia      1972    36.1  13079460   740.
6 Afghanistan Asia      1977    38.4  14880372   786.
7 Afghanistan Asia      1982    39.9  12881816   978.
8 Afghanistan Asia      1987    40.8  13867957   852.
9 Afghanistan Asia      1992    41.7  16317921   649.
10 Afghanistan Asia     1997    41.8  22227415   635.
# i 1,694 more rows
```

But the bigger a dataset gets, the more we have to think about whether we really want (or even can have) all of it in memory all the time.

Databases

In addition, much of what we want to do with a specific dataset will involve actually acting on some relatively small subset of it.

```
gapminder >
  select(gdpPercap, lifeExp)

# A tibble: 1,704 × 2
  gdpPercap lifeExp
  <dbl>     <dbl>
1 779.     28.8
2 821.     30.3
3 853.     32.0
4 836.     34.0
5 740.     36.1
6 786.     38.4
7 978.     39.9
8 852.     40.8
9 649.     41.7
10 635.    41.8
# i 1,694 more rows
```

Databases

In addition, much of what we want to do with a specific dataset will involve actually acting on some relatively small subset of it.

```
gapminder >
  filter(continent == "Europe",
         year == 1977)

# A tibble: 30 × 6
  country      continent  year lifeExp     pop gdpPercap
  <fct>        <fct>    <int>   <dbl>   <int>     <dbl>
1 Albania     Europe    1977    68.9  2509048    3533.
2 Austria     Europe    1977    72.2  7568430    19749.
3 Belgium     Europe    1977    72.8  9821800    19118.
4 Bosnia and Herzegovina Europe  1977    69.9  4086000    3528.
5 Bulgaria    Europe    1977    70.8  8797022    7612.
6 Croatia     Europe    1977    70.6  4318673    11305.
7 Czech Republic Europe    1977    70.7  10161915   14800.
8 Denmark     Europe    1977    74.7  5088419    20423.
9 Finland     Europe    1977    72.5  4738902    15605.
10 France     Europe    1977    73.8  53165019   18293.
# i 20 more rows
```

Databases

In addition, much of what we want to do with a specific dataset will involve actually acting on some relatively small subset of it.

```
gapminder >
  group_by(continent) >
  summarize(lifeExp = mean(lifeExp),
            pop = mean(pop),
            gdpPercap = mean(gdpPercap))
```

```
# A tibble: 5 × 4
  continent lifeExp      pop gdpPercap
  <fct>     <dbl>    <dbl>     <dbl>
1 Africa      48.9  9916003.    2194.
2 Americas    64.7  24504795.   7136.
3 Asia        60.1  77038722.   7902.
4 Europe      71.9  17169765.  14469.
5 Oceania     74.3  8874672.   18622.
```

Databases

Efficiently storing and querying really large quantities of data is the realm of the database and of Structured Query Languages.

As with everything in information technology there is a long and interesting story about various efforts to come up with a good theory of data storage and retrieval, and efficient algorithms for it. If you are interested, watch e.g. [this lecture from a DBMS course](#) from about twelve minutes in.

Where's the database?

Local or remote?

On disk or in memory?

The important thing from the database admin's point of view is that the data is stored *efficiently*, that we have a means of *querying* it, and those queries rely on some search-and-retrieval method that's *really fast*.

There's no free lunch. We want storage methods to be efficient and queries to be fast because the datasets are gonna be gigantic, and accessing them will take time.

Database layouts

A real database is usually not a single giant table. Instead it is more like a list of tables that are partially connected through keys shared between tables. Those keys are indexed and the tables are stored in a tree-like way that makes searching much faster than just going down each row and looking for matches.

From a social science perspective, putting things in different tables might be thought of a matter of logically organizing entities at different *units of observation*. Querying tables is a matter of assembling tables ad hoc at various *units of analysis*.

Database layouts

```
gapminder_xtra ← read_csv(here("data", "gapminder_xtra.csv"))
gapminder_xtra

# A tibble: 1,704 × 13
  country continent year lifeExp      pop gdpPercap area_pct pop_pct
  <chr>    <chr>   <dbl>   <dbl>     <dbl>    <dbl>    <dbl>    <dbl>
1 Afghanistan Asia     1952     28.8  8425333    779.    29.8    59.4
2 Afghanistan Asia     1957     30.3  9240934    821.    29.8    59.4
3 Afghanistan Asia     1962     32.0  10267083   853.    29.8    59.4
4 Afghanistan Asia     1967     34.0  11537966   836.    29.8    59.4
5 Afghanistan Asia     1972     36.1  13079460   740.    29.8    59.4
6 Afghanistan Asia     1977     38.4  14880372   786.    29.8    59.4
7 Afghanistan Asia     1982     39.9  12881816   978.    29.8    59.4
8 Afghanistan Asia     1987     40.8  13867957   852.    29.8    59.4
9 Afghanistan Asia     1992     41.7  16317921   649.    29.8    59.4
10 Afghanistan Asia    1997     41.8  22227415   635.    29.8    59.4
# i 1,694 more rows
# i 5 more variables: gm_countries <dbl>, country_fr <chr>, iso2 <chr>,
#   iso3 <chr>, number <dbl>
```

Again, in social science terms, the redundancies are annoying in part because they apply to different levels or units of observation. From a Database point of view they are also bad because they allow the possibility of a variety of errors or anomalies when updating the table, and they make things really inefficient for search and querying.

Database normalization

A hierarchical set of rules and criteria for ensuring the integrity of data stored across multiple tables and for reducing redundancy in data storage.

Tries to eliminate various sources of error — so-called Insertion, Update, and Deletion anomalies — particularly ones that will pollute, damage, or corrupt things beyond the specific change.

Redundancy and error are minimized by breaking the database up into a series of linked or related tables. Hence the term “relational database”

Normal Forms

0NF: No duplicate rows!

1NF: Using row order to convey information is not allowed; Mixing data types in the same column is not allowed; No table without a primary key is not allowed. Primary keys can be defined by more than one column though. No “repeating groups”.

2NF: Each non-key attribute must depend on the entire primary key

3NF: Every non-key attribute should depend wholly and only on the key.

Think of these rules in connection with ideas about “tidy data” that we’ve already covered.

Database normalization

```
gapminder_xtra
```

```
# A tibble: 1,704 × 13
  country continent year lifeExp      pop gdpPercap area_pct pop_pct
  <chr>     <chr>   <dbl>    <dbl>     <dbl>    <dbl>    <dbl>    <dbl>
1 Afghanistan Asia     1952     28.8  8425333    779.    29.8    59.4
2 Afghanistan Asia     1957     30.3  9240934    821.    29.8    59.4
3 Afghanistan Asia     1962     32.0 10267083    853.    29.8    59.4
4 Afghanistan Asia     1967     34.0 11537966    836.    29.8    59.4
5 Afghanistan Asia     1972     36.1 13079460    740.    29.8    59.4
6 Afghanistan Asia     1977     38.4 14880372    786.    29.8    59.4
7 Afghanistan Asia     1982     39.9 12881816    978.    29.8    59.4
8 Afghanistan Asia     1987     40.8 13867957    852.    29.8    59.4
9 Afghanistan Asia     1992     41.7 16317921    649.    29.8    59.4
10 Afghanistan Asia    1997     41.8 22227415    635.    29.8    59.4
# i 1,694 more rows
# i 5 more variables: gm_countries <dbl>, country_fr <chr>, iso2 <chr>,
#   iso3 <chr>, number <dbl>
```

Database normalization

```
gapminder
```

```
# A tibble: 1,704 × 6
  country   continent year lifeExp     pop gdpPercap
  <fct>     <fct>    <int>   <dbl>   <int>     <dbl>
1 Afghanistan Asia      1952    28.8  8425333    779.
2 Afghanistan Asia      1957    30.3  9240934    821.
3 Afghanistan Asia      1962    32.0 10267083    853.
4 Afghanistan Asia      1967    34.0 11537966    836.
5 Afghanistan Asia      1972    36.1 13079460    740.
6 Afghanistan Asia      1977    38.4 14880372    786.
7 Afghanistan Asia      1982    39.9 12881816    978.
8 Afghanistan Asia      1987    40.8 13867957    852.
9 Afghanistan Asia      1992    41.7 16317921    649.
10 Afghanistan Asia     1997    41.8 22227415    635.
# i 1,694 more rows
```

Database normalization

```
continent_tbl ← read_tsv(here("data", "continent_tab.tsv"))
country_tbl ← read_tsv(here("data", "country_tab.tsv"))
year_tbl ← read_tsv(here("data", "year_tab.tsv"))
```

```
continent_tbl
```

```
# A tibble: 5 × 5
  continent_id continent area_pct pop_pct gm_countries
    <dbl> <chr>     <dbl>   <dbl>      <dbl>
1           1 Africa     20.3    17.6       52
2           2 Americas   28.1     13        25
3           3 Asia       29.8    59.4       33
4           4 Europe     6.7      9.4       30
5           5 Oceania    5.7      0.6        2
```

```
gapminder
```

```
# A tibble: 1,704 × 6
  country    continent  year lifeExp      pop gdpPercap
  <fct>      <fct>    <int>   <dbl>    <int>      <dbl>
1 Afghanistan Asia     1952    28.8  8425333    779.
2 Afghanistan Asia     1957    30.3  9240934    821.
3 Afghanistan Asia     1962    32.0  10267083   853.
4 Afghanistan Asia     1967    34.0  11537966   836.
5 Afghanistan Asia     1972    36.1  13079460   740.
6 Afghanistan Asia     1977    38.4  14880372   786.
7 Afghanistan Asia     1982    39.9  12881816   978.
8 Afghanistan Asia     1987    40.8  13867957   852.
9 Afghanistan Asia     1992    41.7  16317921   649.
10 Afghanistan Asia    1997    41.8  22227415   635.
# i 1,694 more rows
```

Database normalization

```
continent_tbl
```

```
# A tibble: 5 × 5
  continent_id continent area_pct pop_pct gm_countries
    <dbl> <chr>      <dbl>   <dbl>        <dbl>
1         1 Africa     20.3    17.6       52
2         2 Americas   28.1     13        25
3         3 Asia       29.8    59.4       33
4         4 Europe     6.7      9.4       30
5         5 Oceania    5.7      0.6        2
```

```
country_tbl
```

```
# A tibble: 249 × 8
  country_id continent_id country    iso_country country_fr iso2  iso3 number
    <dbl> <dbl> <chr>      <chr>      <chr>   <chr> <chr> <dbl>
1         1         3 Afghanistan Afghanistan Afghanist... AF   AFG     4
2         2         4 Albania     Albania     Albanie (... AL   ALB     8
3         3         1 Algeria    Algeria    Algérie (... DZ   DZA    12
4         4        NA <NA>      American S... Samoa amé... AS   ASM    16
5         5        NA <NA>      Andorra    Andorre (... AD   AND    20
6         6         1 Angola     Angola     Angola (l... AO   AGO    24
7         7        NA Anguilla  Anguilla   Anguilla   AI   AIA   660
8         8        NA Antarctica Antarctica Antarctiq... AQ   ATA    10
9         9        NA Antigua an... Antigua an... Antigua-e... AG   ATG    28
10        10        2 Argentina  Argentina Argentine... AR   ARG    32
# i 239 more rows
```

Database normalization

```
country_tbl
```

```
# A tibble: 249 × 8
  country_id continent_id country      iso_country country_fr iso2  iso3 number
        <dbl>       <dbl> <chr>        <chr>      <chr>   <chr> <chr> <dbl>
1           1          3 Afghanistan Afghanistan Afghanist... AF   AFG     4
2           2          4 Albania      Albania      Albanie (... AL   ALB     8
3           3          1 Algeria     Algeria     Algérie (... DZ   DZA    12
4           4         NA <NA>      American S... Samoa amé... AS   ASM    16
5           5         NA <NA>      Andorra     Andorre (... AD   AND    20
6           6          1 Angola     Angola     Angola (l... AO   AGO    24
7           7         NA Anguilla  Anguilla    Anguilla   AI   AIA  660
8           8         NA Antarctica Antarctica Antarctiq... AQ   ATA    10
9           9         NA Antigua an... Antigua an... Antigua-e... AG   ATG    28
10          10         2 Argentina  Argentina  Argentine... AR   ARG    32
# i 239 more rows
```

```
year_tbl
```

```
# A tibble: 1,704 × 5
  year country_id lifeExp      pop gdpPercap
    <dbl>       <dbl>    <dbl>    <dbl>    <dbl>
1  1952          1    28.8  8425333    779.
2  1957          1    30.3  9240934    821.
3  1962          1    32.0 10267083    853.
4  1967          1    34.0 11537966    836.
5  1972          1    36.1 13079460    740.
6  1977          1    38.4 14880372    786.
7  1982          1    39.9 12881816    978.
8  1987          1    40.8 13867957   852.
9  1992          1    41.7 16317921   649.
10 1997          1    41.8 22227415   635.
# i 1,694 more rows
```

Talking to databases

The main idea

Ultimately, we query databases with SQL. There are several varieties, because there are a variety of database systems and each has their own wrinkles and quirks.

We try to *abstract away* from some of those quirk by using a DBI (DataBase Interface) layer, which is a generic set of commands for talking to some database. It's analogous to an API.

We also need to use a package for the DBMS we're talking to. It translates DBI instructions into the specific dialect the DBMS speaks.

Talking to databases

Some databases are small, and some are far away.

Client-server databases are like websites, serving up responses to queries. The database lives on a machine somewhere in the building, or on campus or whatever.

Cloud DBMSs are like this, too, except the database lives on a machine in someone else's building.

In-process DBMSs live and run on your laptop. We'll use one of these, **duckdb** for examples here.

Talking to databases

We need to open a *connection* to a database before talking to it. Conventionally this is called `con`.

Once connected, we ask it questions. Either we use functions or packages designed to translate our R / dplyr syntax into SQL, or we use functions to pass SQL queries on directly.

We try to minimize the amount of time we are actually making the database do a lot of work.

The key thing is that when working with databases our queries are *lazy* – they don't actually do anything on the whole database unless its strictly necessary or they're explicitly told to.

Example: flights

The nice example

Where everything is lovely and clean. Thanks to Grant McDermott for the following example.

duckdb and DBI

```
# library(DBI)  
  
con ← dbConnect(duckdb :: duckdb(), path = ":memory:")
```

Here we open a connection to an in-memory duckdb database. It's empty. We're going to populate it with data from [nycflights](#).

duckdb and DBI

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

Remember, keys and indexes are what make databases *fast*.

Make a lazy tibble from it

This says “go to `con` and get the ‘flights’ table in it, and pretend it’s a tibble called `flights_db`.

```
flights_db ← tbl(con, "flights")

flights_db

# Source:   table<flights> [?? x 19]
# Database: DuckDB 1.4.2 [root@Darwin 25.2.0:R 4.5.2/:memory:]
#   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 more rows
# 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>
```

Run some dplyr-like queries

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

```
# Source: SQL [?? x 5]
# Database: DuckDB 1.4.2 [root@Darwin 25.2.0:R 4.5.2/:memory:]
  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
4 2013     1     1      -1     -18
5 2013     1     1      -6     -25
6 2013     1     1      -4      12
7 2013     1     1      -5      19
8 2013     1     1      -3     -14
9 2013     1     1      -3      -8
10 2013    1     1      -2       8
# i more rows
```

Run some dplyr-like queries

```
flights_db %>% filter(dep_delay > 240)

# Source:   SQL [?? x 19]
# Database: DuckDB 1.4.2 [root@Darwin 25.2.0:R 4.5.2/:memory:]
#   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      848           1835      853    1001       1950
# 2 2013     1     1     1815           1325      290    2120       1542
# 3 2013     1     1     1842           1422      260    1958       1535
# 4 2013     1     1     2115           1700      255    2330       1920
# 5 2013     1     1     2205           1720      285      46       2040
# 6 2013     1     1     2343           1724      379     314       1938
# 7 2013     1     2     1332           904      268    1616       1128
# 8 2013     1     2     1412           838      334    1710       1147
# 9 2013     1     2     1607           1030      337    2003       1355
# 10 2013    1     2     2131           1512      379    2340       1741
# i more rows
# 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>
```

Run some dplyr-like queries

```
flights_db %>  
  group_by(dest) %>  
  summarise(mean_dep_delay = mean(dep_delay))  
  
# Source:   SQL [?? x 2]  
# Database: DuckDB 1.4.2 [root@Darwin 25.2.0:R 4.5.2/:memory:]  
  dest  mean_dep_delay  
  <chr>  <dbl>  
1 ATL      12.5  
2 MCO      11.3  
3 BNA      16.0  
4 ALB      23.6  
5 PVD      21.8  
6 BQN      12.4  
7 CLT      9.22  
8 MDW      18.6  
9 SDF      16.4  
10 LAS     9.42  
# i more rows
```

Lazy, lazy, lazy

```
tailnum_delay_db ←  
  flights_db ▷  
  group_by(tailnum) ▷  
  summarise(  
    mean_dep_delay = mean(dep_delay),  
    mean_arr_delay = mean(arr_delay),  
    n = n()) ▷  
  filter(n > 100) ▷  
  arrange(desc(mean_arr_delay))
```

This doesn't touch the database.

Lazy, lazy, lazy

Even when we ask to look at it, it just does the absolute minimum required.

```
tailnum_delay_db
```

```
# Source:      SQL [?? x 4]
# Database:   DuckDB 1.4.2 [root@Darwin 25.2.0:R 4.5.2/:memory:]
# Ordered by: desc(mean_arr_delay)
  tailnum mean_dep_delay mean_arr_delay     n
  <chr>        <dbl>        <dbl> <dbl>
1 N11119       32.6        30.3  148
2 N16919       32.4        29.9  251
3 N14998       29.4        27.9  230
4 N15910       29.3        27.6  280
5 N13123       29.6        26.0  121
6 N11192       27.5        25.9  154
7 N14950       26.2        25.3  219
8 N21130       27.0        25.0  126
9 N24128        24.8        24.9  129
10 N22971      26.5        24.7  230
# i more rows
```

When ready, use `collect()`

```
tailnum_delay ←  
tailnum_delay_db ⚡  
collect()  
  
tailnum_delay  
  
# A tibble: 1,201 × 4  
  tailnum mean_dep_delay mean_arr_delay     n  
  <chr>      <dbl>          <dbl> <dbl>  
1 N11119      32.6          30.3   148  
2 N16919      32.4          29.9   251  
3 N14998      29.4          27.9   230  
4 N15910      29.3          27.6   280  
5 N13123      29.6          26.0   121  
6 N11192      27.5          25.9   154  
7 N14950      26.2          25.3   219  
8 N21130      27.0          25.0   126  
9 N24128      24.8          24.9   129  
10 N22971      26.5          24.7  230  
# i 1,191 more rows
```

Now it exists for realsies.

Joins

Database systems will have more than one table. We query and join them. The idea is that getting the DBMS to do this will be way faster and more memory-efficient than trying to get `dplyr` to do it.

Joins

```
## Copy over the "planes" dataset to the same "con" DuckDB connection.  
copy_to(  
  dest = con,  
  df = nycflights13::planes,  
  name = "planes",  
  temporary = FALSE,  
  indexes = "tailnum"  
)  
  
## List tables in our "con" database connection (i.e. now "flights" and "planes")  
dbListTables(con)
```

```
[1] "flights" "planes"
```

```
## Reference from dplyr  
planes_db ← tbl(con, 'planes')
```

See what we did there? It's like `con` the database connection has a list of tables in it.

Joins

```
# Still not done for realsies!
left_join(
  flights_db,
  planes_db %>% rename(year_built = year),
  by = "tailnum" ## Important: Be specific about the joining column
) %>
  select(year, month, day, dep_time, arr_time, carrier, flight, tailnum,
         year_built, type, model)
```

```
# Source:  SQL [?? x 11]
# Database: DuckDB 1.4.2 [root@Darwin 25.2.0:R 4.5.2/:memory:]
  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 ...
2 2013     1     1      533      850  UA       1714 N24211    1998 Fixed ...
3 2013     1     1      542      923  AA       1141 N619AA    1990 Fixed ...
4 2013     1     1      544     1004  B6       725 N804JB    2012 Fixed ...
5 2013     1     1      554      812  DL       461 N668DN    1991 Fixed ...
6 2013     1     1      554      740  UA       1696 N39463    2012 Fixed ...
7 2013     1     1      555      913  B6       507 N516JB    2000 Fixed ...
8 2013     1     1      557      709  EV       5708 N829AS   1998 Fixed ...
9 2013     1     1      557      838  B6        79 N593JB    2004 Fixed ...
10 2013    1     1      558      849  B6       49 N793JB    2011 Fixed ...
# i more rows
# i 1 more variable: model <chr>
```

Finishing up

Close your connection!

```
dbDisconnect(con)
```

Example: ARCOOS Opioids data

This one is messier

I'm not going to do it on the slides. We'll try to process a pretty big data file on a machine of modest proportions.