

Tidy R programming with the OMOP Common Data Model

Edward Burn Adam Black Berta Raventós Yuchen Guo
Mike Du Kim López-Güell Núria Mercadé-Besora
Martí Català

2025-10-23

Table of contents

Preface	6
Is this book for me?	6
How is the book organised?	6
Citation	6
License	7
Code	7
R Packages	7
 I Getting started with databases from R	 9
 1 A first analysis using data in a database	 11
1.1 Getting set up	11
1.2 Taking a peek at the data	11
1.3 Inserting data into a database	12
1.4 Translation from R to SQL	14
1.5 Example analysis	15
1.5.1 Postgres	17
1.5.2 SQL Server	17
1.5.3 Redshift	18
1.5.4 Snowflake	18
1.5.5 Spark	19
1.6 Disconnecting from the database	23
1.7 Further reading	23
 2 Core verbs for analytic pipelines utilising a database	 24
2.1 Tidyverse functions	27
2.2 Getting to an analytic dataset	30
2.3 Disconnecting from the database	32
2.4 Further reading	33
 3 Supported expressions for database queries	 34
3.1 Data types	34
3.1.1 DuckDB	35
3.1.2 Postgres	36

3.1.3	SQL Server	37
3.1.4	Redshift	38
3.1.5	Snowflake	39
3.1.6	Spark	40
3.2	Comparison and logical operators	41
3.2.1	DuckDB	41
3.2.2	Postgres	42
3.2.3	SQL Server	43
3.2.4	Redshift	44
3.2.5	Snowflake	45
3.2.6	Spark	46
3.3	Conditional statements	47
3.3.1	DuckDB	47
3.3.2	Postgres	48
3.3.3	SQL Server	48
3.3.4	Redshift	49
3.3.5	Snowflake	50
3.3.6	Spark	51
3.4	Working with strings	52
3.4.1	DuckDB	52
3.4.2	Postgres	54
3.4.3	SQL Server	57
3.4.4	Redshift	59
3.4.5	Snowflake	61
3.4.6	Spark	64
3.5	Working with dates	66
3.5.1	DuckDB	66
3.5.2	Postgres	67
3.5.3	SQL Server	68
3.5.4	Redshift	69
3.5.5	Snowflake	70
3.5.6	Spark	71
3.6	Data aggregation	72
3.6.1	DuckDB	72
3.6.2	Postgres	73
3.6.3	SQL Server	73
3.6.4	Redshift	74
3.6.5	Snowflake	75
3.6.6	Spark	75
3.7	Window functions	76
3.7.1	DuckDB	77
3.7.2	Postgres	78
3.7.3	SQL Server	79

3.7.4	Redshift	80
3.7.5	Snowflake	81
3.7.6	Spark	82
3.8	Calculating quantiles, including the median	83
3.8.1	DuckDB	84
3.8.2	Postgres	84
3.8.3	SQL Server	85
3.8.4	Redshift	85
3.8.5	Snowflake	86
3.8.6	Spark	86
4	Building analytic pipelines for a data model	88
4.1	Defining a data model	89
4.2	Creating functions for the data model	92
4.3	Building efficient analytic pipelines	100
4.3.1	The risk of “clean” R code	100
4.3.2	Piping and SQL	103
4.3.3	Computing intermediate queries	104
4.4	Disconnecting from the database	108
II	Working with the OMOP CDM from R	110
5	Creating a CDM reference	112
5.1	The OMOP CDM layout	112
5.2	Creating a reference to the OMOP CDM	113
5.3	CDM attributes	116
5.3.1	CDM name	116
5.3.2	CDM version	118
5.4	Including cohort tables in the cdm reference	118
5.5	Including achilles tables in the cdm reference	120
5.6	Adding other tables to the cdm reference	122
5.7	Mutability of the cdm reference	125
5.8	Working with temporary and permanent tables	128
5.9	Disconnecting	130
5.10	Further reading	130
6	Exploring the OMOP CDM	131
6.1	Counting people	132
6.2	Summarising observation periods	135
6.3	Summarising clinical records	136
6.4	Disconnecting	139

7	Identifying patient characteristics	140
7.1	Adding specific demographics	140
7.2	Adding multiple demographics simultaneously	144
7.3	Creating categories	147
7.4	Adding custom variables	151
7.5	Disconnecting	154
7.6	Further reading	154
8	Adding cohorts to the CDM	155
8.1	What is a cohort?	155
8.2	Set up	155
8.3	General concept based cohort	156
8.4	Applying inclusion criteria	161
8.4.1	Only include first cohort entry per person	161
8.4.2	Restrict to study period	161
8.4.3	Applying demographic inclusion criteria	162
8.4.4	Applying cohort-based inclusion criteria	162
8.5	Cohort attributes	164
8.6	Disconnecting	168
8.7	Further reading	168
9	Working with cohorts	169
9.1	Cohort intersections	169
9.2	Intersection between two cohorts	169
9.2.1	Flag	170
9.2.2	Count	173
9.2.3	Date and times	175
9.3	Intersection between a cohort and tables with patient data	176
9.4	Disconnecting	177
9.5	Further reading	177
	Final remarks	178
	Tidy R in OMOP collection	178
	Support us	178
	About the authors	179

Preface

Is this book for me?

We've written this book for anyone interested in working with Observational Medical Outcomes Partnership (OMOP) Common Data Model (CDM) instances using a tidyverse style approach. That is, human centered, consistent, composable, and inclusive (see [Tidy design principles](#) for more details on these principles).

New to R? We recommend you take a look at [R for data science](#) before reading this book. We assume that you have [R](#) installed together with an adequate Integrated Development Environment (IDE) such as [RStudio](#) or [Positron](#). See this [tutorial](#) if you need guidance on how to get started. The book uses multiple packages that you will need to install. See the list in the [R packages](#) section.

New to databases? We recommend you take a look at some web tutorials on SQL, such as [SQLBolt](#) or [SQLZoo](#) to have a basic understanding of how databases work.

New to the OMOP CDM? We'd recommend you pair this book with [The Book of OHDSI](#).

How is the book organised?

The book is divided into two parts. The first half of the book is focused on the general principles for working with databases from R. In these chapters you will see how you can use familiar tidyverse-style code to build up analytic pipelines that start with data held in a database and end with your analytic results. The second half of the book is focused on working with data in the OMOP CDM format, a widely used data format for health care data. In these chapters you will see how to work with this data format using the general principles from the first half of the book along with a set of R packages that have been built for the OMOP CDM.

Citation

If you found this book useful, please help us by citing it:

Burn E, Black A, Raventós B, Guo Y, Du M, López-Güell K, Mercadé-Besora N, Català M. Tidy R programming with the OMOP Common Data Model. GitHub; 2025. <https://github.com/oxford-pharmacoepi/Tidy-R-programming-with-OMOP>

License

Code

The source code for the book can be found at this [GitHub repository](#), please star it if you found it useful.

R Packages

This book is rendered automatically through [GitHub Actions](#) using the following version of packages:

Finding R package dependencies ... Done!

Note: we only included the packages called explicitly in the book.

Package	Version	Link
CDMConnector	2.2.0	
CodelistGenerator	3.5.0	
CohortCharacteristics	1.0.2	
CohortConstructor	0.5.0	
DBI	1.2.3	
Lahman	13.0-0	
PatientProfiles	1.4.3	
bit64	4.6.0-1	
cli	3.6.5	
clock	0.7.3	
dbplyr	2.5.1	
dm	1.0.12	
dplyr	1.1.4	
duckdb	1.4.1	
ggplot2	4.0.0	
nycflights13	1.0.2	
omock	0.5.0	
omopgenerics	1.3.2	
palmerpenguins	0.1.1	
purrr	1.1.0	
sloop	1.0.1	
stringr	1.5.2	
tidyr	1.3.1	

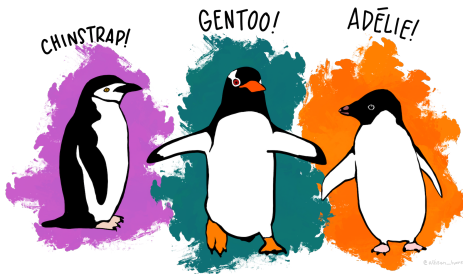
Part I

Getting started with databases from R

In this first half of the book, we will explore how to work with databases from R. In the following chapters, you'll see that when working with data held in a relational database, we can leverage various open-source R packages to perform tidyverse-style data analyses.

- In Chapter 1 we will perform a simple data analysis from start to finish using a table in a database.
- In Chapter 2 we will see in more detail how familiar dplyr functions can be used to combine data spread across different tables in a database into an analytic dataset for further analysis in R.
- In Chapter 3 we will see how we can perform more complex data manipulation via translation of R code into SQL specific to the database management system being used.
- In Chapter 4 we will see how we can build data pipelines by creating a data model in R to represent the relational database we're working with and creating functions and methods to work with it.

1 A first analysis using data in a database



Artwork by [@allison_horst](#)

Before we start working with healthcare data spread across a database using the OMOP Common Data Model, let's first do a simpler analysis. In this case, we will do a quick data analysis with R using a simple dataset held in a database to understand the general approach. For this we'll use data from [palmerpenguins](#) package, which contains data on penguins collected from the [Palmer Station](#) in Antarctica.

1.1 Getting set up

```
library(dplyr)
library(dbplyr)
library(ggplot2)
library(DBI)
library(duckdb)
library(palmerpenguins)
```

1.2 Taking a peek at the data

The package [palmerpenguins](#) contains two datasets, one of them called [penguins](#), which we will use in this chapter. We can get an overview of the data using the [glimpse\(\)](#) command.

```
glimpse(penguins)
```

```
Rows: 344
Columns: 8
$ species      <fct> Adelie, Adelie, Adelie, Adelie, Adelie, Adelie, Adel~
$ island       <fct> Torgersen, Torgersen, Torgersen, Torgersen, Torgerse~
$ bill_length_mm <dbl> 39.1, 39.5, 40.3, NA, 36.7, 39.3, 38.9, 39.2, 34.1, ~
$ bill_depth_mm <dbl> 18.7, 17.4, 18.0, NA, 19.3, 20.6, 17.8, 19.6, 18.1, ~
$ flipper_length_mm <int> 181, 186, 195, NA, 193, 190, 181, 195, 193, 190, 186~
$ body_mass_g   <int> 3750, 3800, 3250, NA, 3450, 3650, 3625, 4675, 3475, ~
$ sex          <fct> male, female, female, NA, female, male, female, male~
$ year         <int> 2007, 2007, 2007, 2007, 2007, 2007, 2007, 2007, 2007~
```

Or we could take a look at the first rows of the data using `head()`:

```
head(penguins, 5)
```

```
# A tibble: 5 x 8
  species island bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
  <fct>   <fct>         <dbl>         <dbl>         <int>         <int>
1 Adelie Torgersen      39.1           18.7           181           3750
2 Adelie Torgersen      39.5           17.4           186           3800
3 Adelie Torgersen      40.3            18           195           3250
4 Adelie Torgersen      NA             NA             NA             NA
5 Adelie Torgersen      36.7           19.3           193           3450
# i 2 more variables: sex <fct>, year <int>
```

1.3 Inserting data into a database

By default, the data provided by the package is local (stored in memory on your computer), so let's first put it into a [DuckDB](#) database. We need to first create the database.

```
con <- dbConnect(drv = duckdb())
```

See that now we have created an empty DuckDB database. We can easily add the penguins data to it.

```
dbWriteTable(conn = con, name = "penguins", value = penguins)
```

With the function `dbListTables()` we can list the tables of a database. In our case, we can see it now has one table:

```
dbListTables(conn = con)
```

```
[1] "penguins"
```

And now that the data is in a database we could use SQL to get the first rows that we saw before.

```
dbGetQuery(conn = con, statement = "SELECT * FROM penguins LIMIT 5")
```

	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g
1	Adelie	Torgersen	39.1	18.7	181	3750
2	Adelie	Torgersen	39.5	17.4	186	3800
3	Adelie	Torgersen	40.3	18.0	195	3250
4	Adelie	Torgersen	NA	NA	NA	NA
5	Adelie	Torgersen	36.7	19.3	193	3450

	sex	year
1	male	2007
2	female	2007
3	female	2007
4	<NA>	2007
5	female	2007

As you can see we have the same data that we had locally but now inside the database.

Connecting to databases from R

Database connections from R can be made using the [DBI package](#). The back-end for DBI is facilitated by database-specific driver packages. In the code snippets above, we created a new, empty, in-process DuckDB database to which we then added our dataset. But we could have instead connected to an existing duckdb database. This could, for example, look like:

```
con <- dbConnect(drv = duckdb(dbdir = "my_duckdb_database.duckdb"))
```

Note that if you point to a non-existing DuckDB file, this will be created with an empty database.

In this book for simplicity we will mostly be working with in-process DuckDB databases with synthetic data. However, when analysing real patient data we will be more often working with client-server databases, where we are connecting from our computer to a central server with the database or working with data held in the cloud. The approaches shown throughout this book will work in the same way for these other types of database management systems, but the way to connect to the database will be different (although still using DBI). In general, creating connections is supported by associated back-end packages. For example a connection to a Postgres database would use the RPostgres R package and look something like:

```
con <- dbConnect(drv = Postgres(),
                 dbname = "my_database",
                 host = "my_server",
                 user = "user",
                 password = "password")
```

For more examples on how to connect to databases using the DBI package please see [Connecting with DBI](#).

1.4 Translation from R to SQL

Instead of using SQL to query our database, we might instead want to use the same R code as before. However, instead of working with the local dataset, now we will need it to query the data held in the database. To do this, first we can create a reference to the table in the database as such:

```
penguins_db <- tbl(src = con, "penguins")
penguins_db
```

```
# Source:   table<penguins> [?? x 8]
# Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-azure:R 4.4.1/:memory:]
  species island  bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
  <fct>   <fct>         <dbl>         <dbl>         <int>         <int>
1 Adelie  Torgersen        39.1          18.7          181          3750
2 Adelie  Torgersen        39.5          17.4          186          3800
3 Adelie  Torgersen        40.3           18          195          3250
4 Adelie  Torgersen        NA            NA            NA            NA
5 Adelie  Torgersen        36.7          19.3          193          3450
6 Adelie  Torgersen        39.3          20.6          190          3650
7 Adelie  Torgersen        38.9          17.8          181          3625
```

```

 8 Adelie  Torgersen      39.2      19.6      195      4675
 9 Adelie  Torgersen      34.1      18.1      193      3475
10 Adelie  Torgersen      42       20.2      190      4250
# i more rows
# i 2 more variables: sex <fct>, year <int>

```

Once we have this reference, we can then use it with familiar looking R code.

```
head(penguins_db, 5)
```

```

# Source:   SQL [?? x 8]
# Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-azure:R 4.4.1/:memory:]
  species island  bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
  <fct>   <fct>         <dbl>         <dbl>         <int>         <int>
1 Adelie  Torgersen      39.1          18.7          181          3750
2 Adelie  Torgersen      39.5          17.4          186          3800
3 Adelie  Torgersen      40.3           18           195          3250
4 Adelie  Torgersen      NA             NA             NA             NA
5 Adelie  Torgersen      36.7          19.3          193          3450
# i 2 more variables: sex <fct>, year <int>

```

The magic here is provided by the [dbplyr](#) package, which takes the R code and converts it into SQL. In this case the query looks like the SQL we wrote directly before.

```
head(penguins_db, 5) |>
  show_query()
```

```

<SQL>
SELECT penguins.*
FROM penguins
LIMIT 5

```

1.5 Example analysis

More complicated SQL can also be generated by using familiar [dplyr](#) code. For example, we could get a summary of bill length by species like so:

```
penguins_db |>
  group_by(species) |>
  summarise(
    n = n(),
    min_bill_length_mm = min(bill_length_mm, na.rm = TRUE),
    mean_bill_length_mm = mean(bill_length_mm, na.rm = TRUE),
    max_bill_length_mm = max(bill_length_mm, na.rm = TRUE)
  ) |>
  mutate(min_max_bill_length_mm = paste0(
    min_bill_length_mm, " to ", max_bill_length_mm
  )) |>
  select("species", "mean_bill_length_mm", "min_max_bill_length_mm")
```

```
# Source:   SQL [?? x 3]
# Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-azure:R 4.4.1/:memory:]
  species   mean_bill_length_mm min_max_bill_length_mm
  <fct>                <dbl> <chr>
1 Adelie           38.8 32.1 to 46.0
2 Chinstrap        48.8 40.9 to 58.0
3 Gentoo           47.5 40.9 to 59.6
```

The benefit of using `dbplyr` now becomes quite clear if we take a look at the corresponding SQL that is generated for us:

```
penguins_db |>
  group_by(species) |>
  summarise(
    n = n(),
    min_bill_length_mm = min(bill_length_mm, na.rm = TRUE),
    mean_bill_length_mm = mean(bill_length_mm, na.rm = TRUE),
    max_bill_length_mm = max(bill_length_mm, na.rm = TRUE)
  ) |>
  mutate(min_max_bill_length_mm = paste0(
    min_bill_length_mm, " to ", max_bill_length_mm
  )) |>
  select("species", "mean_bill_length_mm", "min_max_bill_length_mm") |>
  show_query()
```

```
<SQL>
SELECT
  species,
```



```

mean_bill_length_mm,
CONCAT_WS(' ', min_bill_length_mm, ' to ', max_bill_length_mm) AS min_max_bill_length_mm
FROM (
  SELECT
    species,
    COUNT(*) AS n,
    MIN(bill_length_mm) AS min_bill_length_mm,
    AVG(bill_length_mm) AS mean_bill_length_mm,
    MAX(bill_length_mm) AS max_bill_length_mm
  FROM penguins
  GROUP BY species
) q01

```

Instead of having to write this somewhat complex SQL specific to DuckDB, we can use the friendlier `dplyr` syntax that will be more familiar if you're coming from an R programming background.

Translation to different SQL dialects

Note this same R code will also work for other SQL dialects such as Postgres, SQL server, Snowflake and Spark. Here you can see the different generated translations:

1.5.1 Postgres

```

<SQL>
SELECT
  `species`,
  `mean_bill_length_mm`,
  CONCAT_WS(' ', `min_bill_length_mm`, ' to ', `max_bill_length_mm`) AS `min_max_bill_length_mm`
FROM (
  SELECT
    `species`,
    COUNT(*) AS `n`,
    MIN(`bill_length_mm`) AS `min_bill_length_mm`,
    AVG(`bill_length_mm`) AS `mean_bill_length_mm`,
    MAX(`bill_length_mm`) AS `max_bill_length_mm`
  FROM `df`
  GROUP BY `species`
) AS `q01`

```

1.5.2 SQL Server

```

<SQL>

```

```

SELECT
  `species`,
  `mean_bill_length_mm`,
  `min_bill_length_mm` + ' to ' + `max_bill_length_mm` AS `min_max_bill_length_mm`
FROM (
  SELECT
    `species`,
    COUNT_BIG(*) AS `n`,
    MIN(`bill_length_mm`) AS `min_bill_length_mm`,
    AVG(`bill_length_mm`) AS `mean_bill_length_mm`,
    MAX(`bill_length_mm`) AS `max_bill_length_mm`
  FROM `df`
  GROUP BY `species`
) AS `q01`

```

1.5.3 Redshift

```

<SQL>
SELECT
  `species`,
  `mean_bill_length_mm`,
  `min_bill_length_mm` || ' to ' || `max_bill_length_mm` AS `min_max_bill_length_mm`
FROM (
  SELECT
    `species`,
    COUNT(*) AS `n`,
    MIN(`bill_length_mm`) AS `min_bill_length_mm`,
    AVG(`bill_length_mm`) AS `mean_bill_length_mm`,
    MAX(`bill_length_mm`) AS `max_bill_length_mm`
  FROM `df`
  GROUP BY `species`
) AS `q01`

```

1.5.4 Snowflake

```

<SQL>
SELECT
  `species`,
  `mean_bill_length_mm`,
  ARRAY_TO_STRING(ARRAY_CONSTRUCT_COMPACT(`min_bill_length_mm`, ' to ', `max_bill_length_mm`)) AS `min_max_bill_length_mm`
FROM (
  SELECT

```

```

    `species`,
    COUNT(*) AS `n`,
    MIN(`bill_length_mm`) AS `min_bill_length_mm`,
    AVG(`bill_length_mm`) AS `mean_bill_length_mm`,
    MAX(`bill_length_mm`) AS `max_bill_length_mm`
FROM `df`
GROUP BY `species`
) AS `q01`

```

1.5.5 Spark

```

<SQL>
SELECT
  `species`,
  `mean_bill_length_mm`,
  CONCAT_WS(' ', `min_bill_length_mm`, ' to ', `max_bill_length_mm`) AS `min_max_bill_length`
FROM (
  SELECT
    `species`,
    COUNT(*) AS `n`,
    MIN(`bill_length_mm`) AS `min_bill_length_mm`,
    AVG(`bill_length_mm`) AS `mean_bill_length_mm`,
    MAX(`bill_length_mm`) AS `max_bill_length_mm`
  FROM `df`
  GROUP BY `species`
) AS `q01`

```

Note that even though the different SQL statements look similar, each SQL dialect has its own particularities. Using the [dbplyr](#) approach allows us to support multiple different SQL dialects and back-ends by just writing R code.

Not having to worry about the SQL translation behind our queries allows us to query the database in a simple way even for more complex questions. For instance, suppose now that we are particularly interested in the body mass variable. We can first notice that there are a couple of missing records for this.

```

penguins_db |>
  mutate(missing_body_mass_g = if_else(is.na(body_mass_g), 1, 0)) |>
  group_by(species, missing_body_mass_g) |>
  tally()

```

Source: SQL [?? x 3]

```
# Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-azure:R 4.4.1/:memory:]
  species  missing_body_mass_g      n
  <fct>           <dbl> <dbl>
1 Adelie                1      1
2 Gentoo                1      1
3 Chinstrap            0     68
4 Adelie                0    151
5 Gentoo                0    123
```

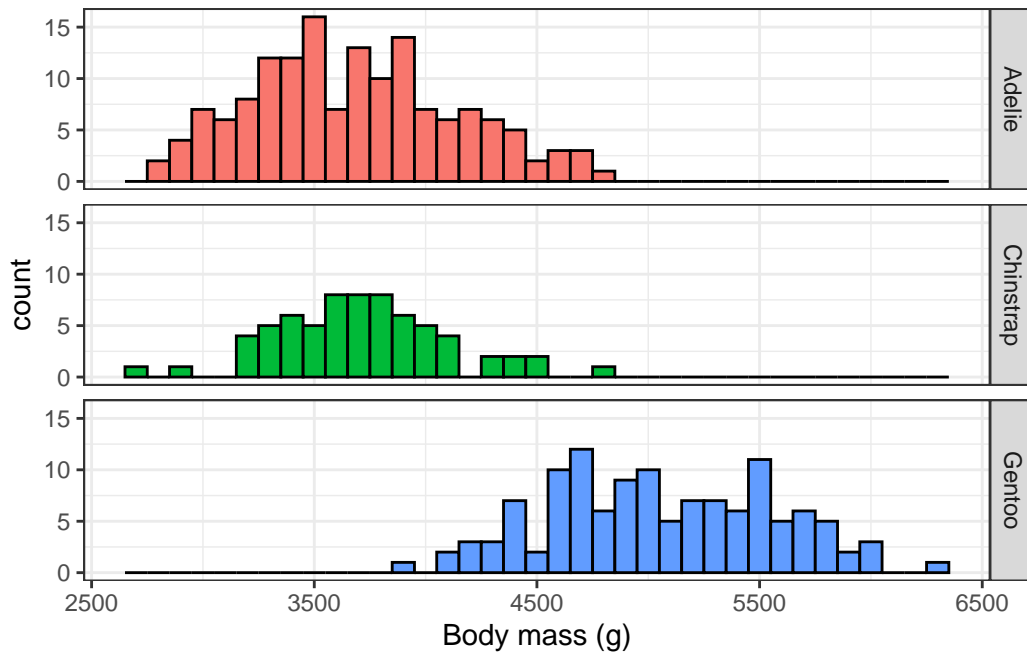
We can get the mean for each of the species (dropping those two missing records).

```
penguins_db |>
  group_by(species) |>
  summarise(mean_body_mass_g = round(mean(body_mass_g, na.rm = TRUE)))
```

```
# Source:   SQL [?? x 2]
# Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-azure:R 4.4.1/:memory:]
  species  mean_body_mass_g
  <fct>           <dbl>
1 Adelie        3701
2 Chinstrap     3733
3 Gentoo        5076
```

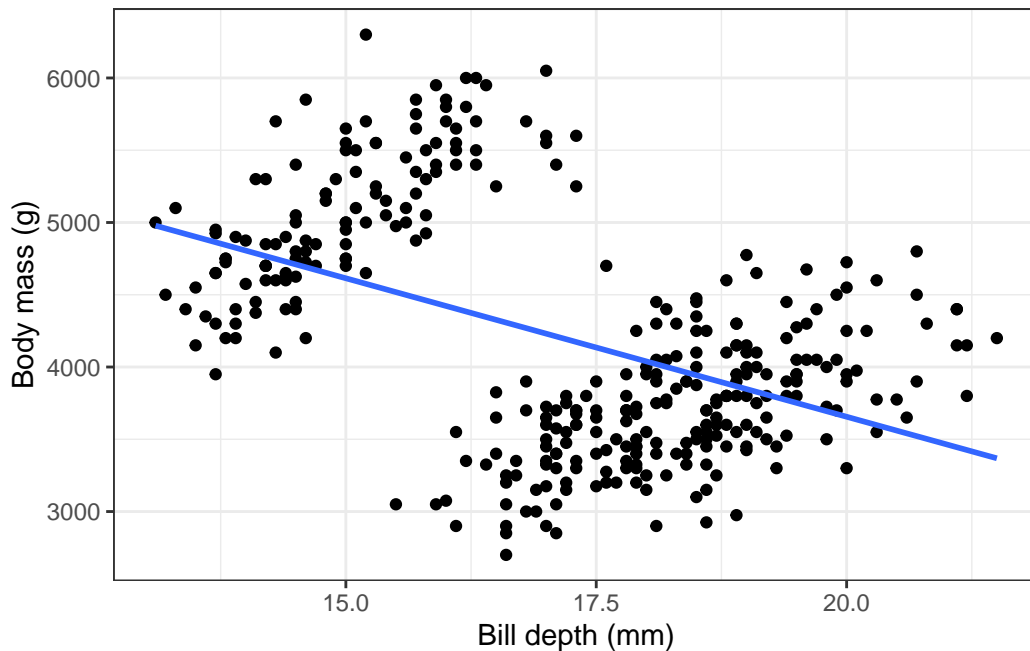
We could also make a histogram of values for each of the species using the `ggplot2` package. Here we would bring our data back into R before creating our plot with the `collect()` function.

```
penguins_db |>
  select("species", "body_mass_g") |>
  collect() |>
  ggplot(aes(group = species, fill = species)) +
  facet_grid(species ~ .) +
  geom_histogram(aes(body_mass_g), colour = "black", binwidth = 100) +
  xlab("Body mass (g)") +
  theme_bw() +
  theme(legend.position = "none")
```



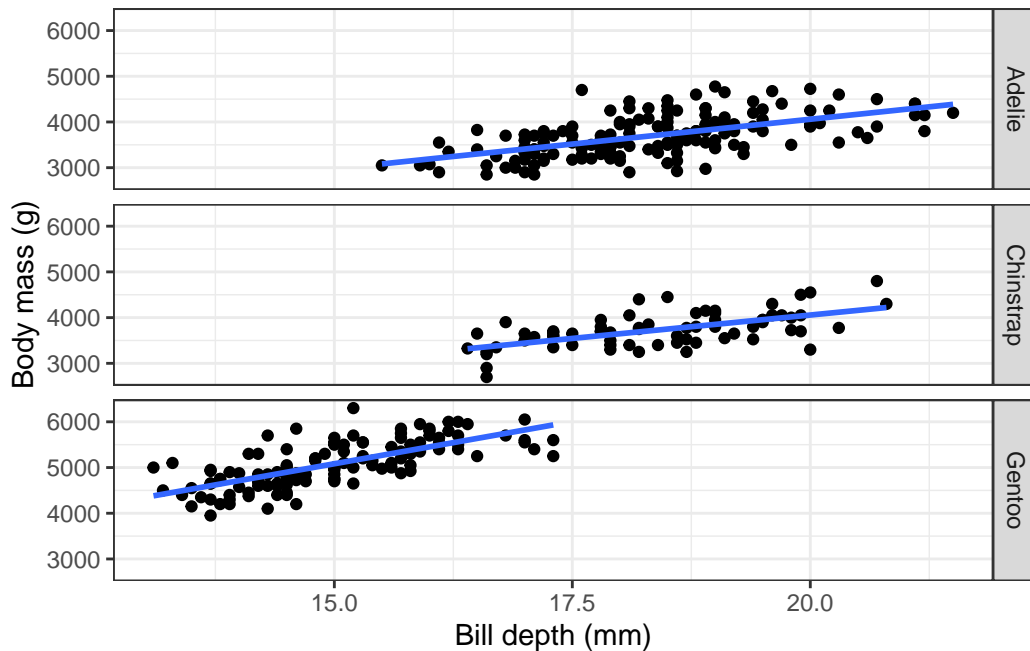
Now let's look at the relationship between body mass and bill depth.

```
penguins_db |>
  select("species", "body_mass_g", "bill_depth_mm") |>
  collect() |>
  ggplot(aes(x = bill_depth_mm, y = body_mass_g)) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE) +
  xlab("Bill depth (mm)") +
  ylab("Body mass (g)") +
  theme_bw() +
  theme(legend.position = "none")
```



Here we see a negative correlation between body mass and bill depth which seems rather unexpected. But what about if we stratify this query by species?

```
penguins_db |>
  select("species", "body_mass_g", "bill_depth_mm") |>
  collect() |>
  ggplot(aes(x = bill_depth_mm, y = body_mass_g)) +
  facet_grid(species ~ .) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE) +
  xlab("Bill depth (mm)") +
  ylab("Body mass (g)") +
  theme_bw() +
  theme(legend.position = "none")
```



As well as having an example of working with data in database from R, you also have an example of [Simpson's paradox](#)!

1.6 Disconnecting from the database

Now that we've reached the end of this example, we can close our connection to the database.

```
dbDisconnect(conn = con)
```

1.7 Further reading

- [R for Data Science \(Chapter 13: Relational data\)](#)
- [Writing SQL with dbplyr](#)
- [Data Carpentry: SQL databases and R](#)

2 Core verbs for analytic pipelines utilising a database

We saw in the previous chapter that we can use familiar `dplyr` verbs with data held in a database. In the last chapter, we were working with just a single table which we loaded into the database. When working with databases, we will typically be working with multiple tables (as we'll see later when working with data in the OMOP CDM format). For this chapter, we will see more tidyverse functionality that can be used with data in a database, this time using the `nycflights13` data. As we can see, we now have a set of related tables with data on flights departing from New York City airports in 2013.

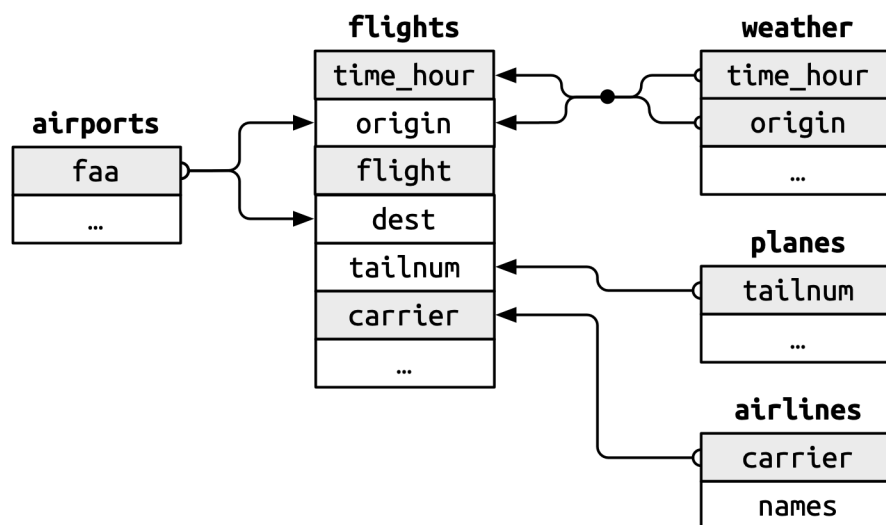


Figure 2.1: `nycflights13` relational diagram from <https://collinn.github.io/teaching/2023/labs/joins.html>.

Let's load the required libraries, add our data to a DuckDB database, and then create references to each of these tables.

```
library(nycflights13)
library(dplyr)
library(dbplyr)
```



```

library(tidyr)
library(duckdb)
library(DBI)

# create duckdb connection
con <- dbConnect(drv = duckdb())

# copy tables in a loop
for (nm in c("airlines", "airports", "flights", "planes", "weather")) {
  dbWriteTable(conn = con, name = nm, value = get(nm))
}

airports_db <- tbl(src = con, "airports")
glimpse(airports_db)

```

```

Rows: ??
Columns: 8
Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-azure:R 4.4.1/:memory:]
$ faa    <chr> "04G", "06A", "06C", "06N", "09J", "0A9", "0G6", "0G7", "0P2", "~
$ name   <chr> "Lansdowne Airport", "Moton Field Municipal Airport", "Schaumbur~
$ lat    <dbl> 41.13047, 32.46057, 41.98934, 41.43191, 31.07447, 36.37122, 41.4~
$ lon    <dbl> -80.61958, -85.68003, -88.10124, -74.39156, -81.42778, -82.17342~
$ alt    <dbl> 1044, 264, 801, 523, 11, 1593, 730, 492, 1000, 108, 409, 875, 10~
$ tz     <dbl> -5, -6, -6, -5, -5, -5, -5, -5, -5, -8, -5, -6, -5, -5, -5, -
5, ~
$ dst    <chr> "A", "A", "A", "A", "A", "A", "A", "A", "U", "A", "A", "U", "A",~
$ tzone  <chr> "America/New_York", "America/Chicago", "America/Chicago", "Ameri~

```

```

flights_db <- tbl(src = con, "flights")
glimpse(flights_db)

```

```

Rows: ??
Columns: 19
Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-azure:R 4.4.1/:memory:]
$ year      <int> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2~
$ month     <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1~
$ day       <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1~
$ dep_time  <int> 517, 533, 542, 544, 554, 554, 555, 557, 557, 558, 558, ~
$ sched_dep_time <int> 515, 529, 540, 545, 600, 558, 600, 600, 600, 600, 600, ~
$ dep_delay <dbl> 2, 4, 2, -1, -6, -4, -5, -3, -3, -2, -2, -2, -2, -2, -
1~

```

```

$ arr_time      <int> 830, 850, 923, 1004, 812, 740, 913, 709, 838, 753, 849, ~
$ sched_arr_time <int> 819, 830, 850, 1022, 837, 728, 854, 723, 846, 745, 851, ~
$ arr_delay     <dbl> 11, 20, 33, -18, -25, 12, 19, -14, -8, 8, -2, -3, 7, -
1~
$ carrier       <chr> "UA", "UA", "AA", "B6", "DL", "UA", "B6", "EV", "B6", "~
$ flight        <int> 1545, 1714, 1141, 725, 461, 1696, 507, 5708, 79, 301, 4~
$ tailnum       <chr> "N14228", "N24211", "N619AA", "N804JB", "N668DN", "N394~
$ origin        <chr> "EWR", "LGA", "JFK", "JFK", "LGA", "EWR", "EWR", "LGA", ~
$ dest         <chr> "IAH", "IAH", "MIA", "BQN", "ATL", "ORD", "FLL", "IAD", ~
$ air_time      <dbl> 227, 227, 160, 183, 116, 150, 158, 53, 140, 138, 149, 1~
$ distance      <dbl> 1400, 1416, 1089, 1576, 762, 719, 1065, 229, 944, 733, ~
$ hour          <dbl> 5, 5, 5, 5, 6, 5, 6, 6, 6, 6, 6, 6, 6, 6, 5, 6, 6, 6~
$ minute        <dbl> 15, 29, 40, 45, 0, 58, 0, 0, 0, 0, 0, 0, 0, 0, 59, 0~
$ time_hour     <dtm> 2013-01-01 10:00:00, 2013-01-01 10:00:00, 2013-01-
01 1~

```

```

weather_db <- tbl(src = con, "weather")
glimpse(weather_db)

```

```

Rows: ??
Columns: 15
Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-azure:R 4.4.1/:memory:]
$ origin      <chr> "EWR", "EWR", "EWR", "EWR", "EWR", "EWR", "EWR", "EWR", "EW~
$ year        <int> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, ~
$ month       <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ~
$ day         <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ~
$ hour        <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 14, 15, 16, 17, 18, ~
$ temp        <dbl> 39.02, 39.02, 39.02, 39.92, 39.02, 37.94, 39.02, 39.92, 39.~
$ dewp        <dbl> 26.06, 26.96, 28.04, 28.04, 28.04, 28.04, 28.04, 28.04, 28.~
$ humid       <dbl> 59.37, 61.63, 64.43, 62.21, 64.43, 67.21, 64.43, 62.21, 62.~
$ wind_dir    <dbl> 270, 250, 240, 250, 260, 240, 240, 250, 260, 260, 260, 330, ~
$ wind_speed  <dbl> 10.35702, 8.05546, 11.50780, 12.65858, 12.65858, 11.50780, ~
$ wind_gust   <dbl> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, 20.~
$ precip      <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ~
$ pressure    <dbl> 1012.0, 1012.3, 1012.5, 1012.2, 1011.9, 1012.4, 1012.2, 101~
$ visib       <dbl> 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, ~
$ time_hour   <dtm> 2013-01-01 06:00:00, 2013-01-01 07:00:00, 2013-01-01 08:00~

```

```

planes_db <- tbl(src = con, "planes")
glimpse(planes_db)

```

```

Rows: ??

```

```
Columns: 9
Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-azure:R 4.4.1/:memory:]
$ tailnum      <chr> "N10156", "N102UW", "N103US", "N104UW", "N10575", "N105UW~
$ year         <int> 2004, 1998, 1999, 1999, 2002, 1999, 1999, 1999, 1999, 199~
$ type         <chr> "Fixed wing multi engine", "Fixed wing multi engine", "Fi~
$ manufacturer <chr> "EMBRAER", "AIRBUS INDUSTRIE", "AIRBUS INDUSTRIE", "AIRBU~
$ model        <chr> "EMB-145XR", "A320-214", "A320-214", "A320-214", "EMB-~
145~
$ engines      <int> 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, ~
$ seats       <int> 55, 182, 182, 182, 55, 182, 182, 182, 182, 182, 182, 55, 55, 5~
$ speed       <int> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N~
$ engine      <chr> "Turbo-fan", "Turbo-fan", "Turbo-fan", "Turbo-fan", "Turb~
```

```
airlines_db <- tbl(src = con, "airlines")
glimpse(airlines_db)
```

```
Rows: ??
Columns: 2
Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-azure:R 4.4.1/:memory:]
$ carrier <chr> "9E", "AA", "AS", "B6", "DL", "EV", "F9", "FL", "HA", "MQ", "O~
$ name    <chr> "Endeavor Air Inc.", "American Airlines Inc.", "Alaska Airline~
```

2.1 Tidyverse functions

For almost all analyses, we want to go from having our starting data spread out across multiple tables in the database to a single tidy table containing all the data we need for the specific analysis. We can often get to our tidy analytic dataset using the tidyverse functions below (most of which come from [dplyr](#), but a couple also from the [tidyr](#) package). These functions all work with data in a database by generating SQL that will have the same purpose as if these functions were being run against data in R.

! Important

Until we use [compute\(\)](#) or [collect\(\)](#) (or printing the first few rows of the result), all we're doing is translating R code into SQL. This means no code is being executed on the database side.

- [compute\(\)](#) will execute the query and store it in a new table in the database.
- [collect\(\)](#) will execute the query and bring the result back to R.
- printing (e.g. [glimpse\(\)](#) or [print\(\)](#)) will execute the query, limiting the result to

the first set of rows, which leads to shorter computation time on the database side.

Purpose	Functions	Description
Selecting rows	<code>filter</code> , <code>distinct</code>	To select rows in a table.
Ordering rows	<code>arrange</code>	To order rows in a table.
Column Transformation	<code>mutate</code> , <code>select</code> , <code>relocate</code> , <code>rename</code>	To create new columns or change existing ones.
Grouping and ungrouping	<code>group_by</code> , <code>rowwise</code> , <code>ungroup</code>	To group data by one or more variables and to remove grouping.
Aggregation	<code>count</code> , <code>tally</code> , <code>summarise</code>	These functions are used for summarising data.
Data merging and joining	<code>inner_join</code> , <code>left_join</code> , <code>right_join</code> , <code>full_join</code> , <code>anti_join</code> , <code>semi_join</code> , <code>cross_join</code>	These functions are used to combine data from different tables based on common columns.
Data reshaping	<code>pivot_wider</code> , <code>pivot_longer</code>	These functions are used to reshape data between wide and long formats.
Data union	<code>union_all</code> , <code>union</code>	This function combines two tables.
Randomly selects rows	<code>slice_sample</code>	We can use this to take a random subset a table.

💡 Behind the scenes

By using the above functions we can use the same code regardless of whether the data is held in the database or locally in R. This is because the functions used above are generic functions which behave differently depending on the type of input they are given. Let's take `inner_join()` for example. We can see that this function is a S3 generic function (with S3 being the most common object-oriented system used in R).

```
library(sloop)
ftype(inner_join)
```

```
[1] "S3"      "generic"
```

Among others, the references we create to tables in a database have `tbl_lazy` as a class attribute. Meanwhile, we can see that when collected into R, the object changes to have different attributes, one of which is `data.frame`:

```
class(flights_db)
```

```
[1] "tbl_duckdb_connection" "tbl_dbi"      "tbl_sql"
[4] "tbl_lazy"             "tbl"
```

```
class(flights_db |> head(1) |> collect())
```

```
[1] "tbl_df"      "tbl"        "data.frame"
```

We can see that `inner_join()` has different methods for `tbl_lazy` and `data.frame`.

```
s3_methods_generic("inner_join")
```

```
# A tibble: 2 x 4
  generic      class      visible source
  <chr>        <chr>      <lgl>   <chr>
1 inner_join data.frame FALSE   registered S3method
2 inner_join tbl_lazy   FALSE   registered S3method
```

When working with references to tables in the database the `tbl_lazy` method will be used.

```
s3_dispatch(flights_db |>
  inner_join(planes_db))
```

```
inner_join.tbl_duckdb_connection
inner_join.tbl_dbi
inner_join.tbl_sql
=> inner_join.tbl_lazy
inner_join.tbl
inner_join.default
```

But once we bring data into R, the `data.frame` method will be used.

```
s3_dispatch(flights_db |> head(1) |> collect() |>
  inner_join(planes_db |> head(1) |> collect()))

inner_join.tbl_df
inner_join.tbl
=> inner_join.data.frame
inner_join.default
```

2.2 Getting to an analytic dataset

To see a little more on how we can use the above functions, let's say we want to do an analysis of late flights from JFK airport. We want to see whether there is some relationship between plane characteristics and the risk of delay.

For this, we'll first use the `filter()` and `select()` `dplyr` verbs to get the data from the flights table. Note, we'll rename `arr_delay` to just `delay`.

```
delayed_flights_db <- flights_db |>
  filter(!is.na(arr_delay) & origin == "JFK") |>
  select("dest", "distance", "carrier", "tailnum", "delay" = "arr_delay")
```

i Show query

See the resultant DuckDB query:

```
<SQL>
SELECT dest, distance, carrier, tailnum, arr_delay AS delay
FROM flights
WHERE (NOT((arr_delay IS NULL)) AND origin = 'JFK')
```

When executed, our results will look like the following:

```
delayed_flights_db
```

```
# Source:   SQL [?? x 5]
# Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-azure:R 4.4.1/:memory:]
  dest distance carrier tailnum delay
<chr>    <dbl> <chr>    <chr>    <dbl>
```

1	MIA	1089	AA	N619AA	33
2	BQN	1576	B6	N804JB	-18
3	MCO	944	B6	N593JB	-8
4	PBI	1028	B6	N793JB	-2
5	TPA	1005	B6	N657JB	-3
6	LAX	2475	UA	N29129	7
7	BOS	187	B6	N708JB	-4
8	ATL	760	DL	N3739P	-8
9	SFO	2586	UA	N532UA	14
10	RSW	1074	B6	N635JB	4

i more rows

Now we'll add plane characteristics from the planes table. We will use an inner join so that only records for which we have the plane characteristics are kept.

```
delayed_flights_db <- delayed_flights_db |>
  inner_join(
    planes_db |>
      select("tailnum", "seats"),
    by = "tailnum"
  )
```

Note that our first query was not executed, as we didn't use either `compute()` or `collect()`, so we'll now have added our join to the original query.

Show query

See that now the SQL code combines both queries:

```
<SQL>
SELECT LHS.*, seats
FROM (
  SELECT dest, distance, carrier, tailnum, arr_delay AS delay
  FROM flights
  WHERE (NOT((arr_delay IS NULL)) AND origin = 'JFK')
) LHS
INNER JOIN planes
  ON (LHS.tailnum = planes.tailnum)
```

And when executed, our results will look like the following:

```
delayed_flights_db
```

```
# Source:   SQL [?? x 6]
# Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-azure:R 4.4.1/:memory:]
  dest  distance carrier tailnum delay seats
  <chr>   <dbl> <chr>   <chr>   <dbl> <int>
1 MIA      1089 AA      N619AA     33    178
2 BQN      1576 B6      N804JB    -18    200
3 MCO       944 B6      N593JB     -8    200
4 PBI      1028 B6      N793JB     -2    200
5 TPA      1005 B6      N657JB     -3    200
6 LAX      2475 UA      N29129     7     178
7 BOS       187 B6      N708JB     -4    200
8 ATL       760 DL      N3739P     -8    189
9 RSW      1074 B6      N635JB     4     200
10 SJU     1598 B6      N794JB    -21    200
# i more rows
```

This tidy dataset has been created in the database via R code translated to SQL. With this, we can now collect our analytic dataset into R and proceed from there (for example, to perform statistical analyses locally that might not be possible to run in a database, such as plots, density distributions, regression, or anything beyond data manipulation).

```
delayed_flights <- delayed_flights_db |>
  collect()

glimpse(delayed_flights)
```

```
Rows: 93,298
Columns: 6
$ dest      <chr> "LAX", "CLT", "MCO", "SFO", "ATL", "FLL", "BUF", "RSW", "LAS"~
$ distance  <dbl> 2475, 541, 944, 2586, 760, 1069, 301, 1074, 2248, 1182, 2153,~
$ carrier   <chr> "UA", "US", "B6", "UA", "DL", "B6", "B6", "B6", "B6", "B6", "~
$ tailnum   <chr> "N34137", "N117UW", "N632JB", "N502UA", "N681DA", "N568JB", "~
$ delay     <dbl> -10, -34, -2, 7, -12, -3, 2, 2, 0, -19, -35, -8, 7, -12, 11, ~
$ seats     <int> 178, 182, 200, 178, 178, 200, 20, 200, 20, 200, 20, 200,~
```

2.3 Disconnecting from the database

Now that we've reached the end of this example, we can close our connection to the database.


```
dbDisconnect(conn = con)
```

2.4 Further reading

- Wickham H, François R, Henry L, Müller K, Vaughan D (2025). *dplyr: A Grammar of Data Manipulation*. R package version 1.1.4, <https://dplyr.tidyverse.org>
- Wickham H, Vaughan D, Girlich M (2025). *tidyr: Tidy Messy Data*. R package version 1.3.1, <https://tidyr.tidyverse.org>.

3 Supported expressions for database queries

In the previous chapter, Chapter 2, we saw that there are a core set of tidyverse functions that can be used with databases to extract data for analysis. The SQL code used in the previous chapter would be the same for all database management systems, with only joins and variable selection being used.

For more complex data pipelines, we will, however, often need to incorporate additional expressions within these functions. Because of differences across database management systems, the SQL these pipelines get translated to can vary. Moreover, some expressions may only be supported for some subset of databases. When writing code that we want to work across different database management systems, we therefore need to keep in mind what is supported where. To help with this, the sections below show the available translations for common expressions we might want to use.

Let's first load the packages which these expressions come from. In addition to base R types, `bit64` adds support for integer64. The `stringr` package provides functions for working with strings, while `clock` has various functions for working with dates. Many other useful expressions will come from `dplyr` itself.

```
library(duckdb)
library(bit64)
library(dplyr)
library(dbplyr)
library(stringr)
library(clock)
```

3.1 Data types

Commonly used data types are consistently supported across database backends. We can use the base `as.numeric()`, `as.integer()`, `as.character()`, `as.Date()`, and `as.POSIXct()`. We can also use `as.integer64()` from the `bit64` package to coerce to integer64, and the `as_date()` and `as_datetime()` from the `clock` package instead of `as.Date()` and `as.POSIXct()`, respectively.

💡 Show SQL

3.1.1 DuckDB

```
con <- simulate_duckdb()
translate_sql(as.numeric(var), con = con)
```

```
<SQL> CAST(`var` AS NUMERIC)
```

```
translate_sql(as.integer(var), con = con)
```

```
<SQL> CAST(`var` AS INTEGER)
```

```
translate_sql(as.integer64(var), con = con)
```

```
<SQL> CAST(`var` AS BIGINT)
```

```
translate_sql(as.character(var), con = con)
```

```
<SQL> CAST(`var` AS TEXT)
```

```
translate_sql(as.Date(var), con = con)
```

```
<SQL> CAST(`var` AS DATE)
```

```
translate_sql(as_date(var), con = con)
```

```
<SQL> CAST(`var` AS DATE)
```

```
translate_sql(as.POSIXct(var), con = con)
```

```
<SQL> CAST(`var` AS TIMESTAMP)
```

```
translate_sql(as_datetime(var), con = con)
```

```
<SQL> CAST(`var` AS TIMESTAMP)
```

```
translate_sql(as.logical(var), con = con)
```

```
<SQL> CAST(`var` AS BOOLEAN)
```

3.1.2 Postgres

```
con <- simulate_postgres()
translate_sql(as.numeric(var), con = con)
```

```
<SQL> CAST(`var` AS NUMERIC)
```

```
translate_sql(as.integer(var), con = con)
```

```
<SQL> CAST(`var` AS INTEGER)
```

```
translate_sql(as.integer64(var), con = con)
```

```
<SQL> CAST(`var` AS BIGINT)
```

```
translate_sql(as.character(var), con = con)
```

```
<SQL> CAST(`var` AS TEXT)
```

```
translate_sql(as.Date(var), con = con)
```

```
<SQL> CAST(`var` AS DATE)
```

```
translate_sql(as_date(var), con = con)
```

```
<SQL> CAST(`var` AS DATE)
```

```
translate_sql(as.POSIXct(var), con = con)
```

```
<SQL> CAST(`var` AS TIMESTAMP)
```

```
translate_sql(as_datetime(var), con = con)
```

```
<SQL> CAST(`var` AS TIMESTAMP)
```

```
translate_sql(as.logical(var), con = con)
```

```
<SQL> CAST(`var` AS BOOLEAN)
```

3.1.3 SQL Server

```
con <- simulate_mssql()
translate_sql(as.numeric(var), con = con)
```

```
<SQL> TRY_CAST(`var` AS FLOAT)
```

```
translate_sql(as.integer(var), con = con)
```

```
<SQL> TRY_CAST(TRY_CAST(`var` AS NUMERIC) AS INT)
```

```
translate_sql(as.integer64(var), con = con)
```

```
<SQL> TRY_CAST(TRY_CAST(`var` AS NUMERIC(38, 0)) AS BIGINT)
```

```
translate_sql(as.character(var), con = con)
```

```
<SQL> TRY_CAST(`var` AS VARCHAR(MAX))
```

```
translate_sql(as.Date(var), con = con)
```

```
<SQL> TRY_CAST(`var` AS DATE)
```

```
translate_sql(as_date(var), con = con)
```

```
<SQL> TRY_CAST(`var` AS DATE)
```

```
translate_sql(as.POSIXct(var), con = con)
```

```
<SQL> TRY_CAST(`var` AS DATETIME2)
```

```
translate_sql(as_datetime(var), con = con)
```

```
<SQL> TRY_CAST(`var` AS DATETIME2)
```

```
translate_sql(as.logical(var), con = con)
```

```
<SQL> TRY_CAST(`var` AS BIT)
```

3.1.4 Redshift

```
con <- simulate_redshift()  
translate_sql(as.numeric(var), con = con)
```

```
<SQL> CAST(`var` AS FLOAT)
```

```
translate_sql(as.integer(var), con = con)
```

```
<SQL> CAST(`var` AS INTEGER)
```

```
translate_sql(as.integer64(var), con = con)
```

```
<SQL> CAST(`var` AS BIGINT)
```

```
translate_sql(as.character(var), con = con)
```

```
<SQL> CAST(`var` AS TEXT)
```

```
translate_sql(as.Date(var), con = con)
```

```
<SQL> CAST(`var` AS DATE)
```

```
translate_sql(as_date(var), con = con)
```

```
<SQL> CAST(`var` AS DATE)
```

```
translate_sql(as.POSIXct(var), con = con)
```

```
<SQL> CAST(`var` AS TIMESTAMP)
```

```
translate_sql(as_datetime(var), con = con)
```

```
<SQL> CAST(`var` AS TIMESTAMP)
```

```
translate_sql(as.logical(var), con = con)
```

```
<SQL> CAST(`var` AS BOOLEAN)
```

3.1.5 Snowflake

```
con <- simulate_snowflake()  
translate_sql(as.numeric(var), con = con)
```

```
<SQL> CAST(`var` AS DOUBLE)
```

```
translate_sql(as.integer(var), con = con)
```

```
<SQL> CAST(`var` AS INT)
```

```
translate_sql(as.integer64(var), con = con)
```

```
<SQL> CAST(`var` AS BIGINT)
```

```
translate_sql(as.character(var), con = con)
```

```
<SQL> CAST(`var` AS STRING)
```

```
translate_sql(as.Date(var), con = con)
```

```
<SQL> CAST(`var` AS DATE)
```

```
translate_sql(as_date(var), con = con)
```

```
<SQL> CAST(`var` AS DATE)
```

```
translate_sql(as.POSIXct(var), con = con)
```

```
<SQL> CAST(`var` AS TIMESTAMP)
```

```
translate_sql(as_datetime(var), con = con)
```

```
<SQL> CAST(`var` AS TIMESTAMP)
```

```
translate_sql(as.logical(var), con = con)
```

```
<SQL> CAST(`var` AS BOOLEAN)
```

3.1.6 Spark

```
con <- simulate_spark_sql()
translate_sql(as.numeric(var), con = con)
```

```
<SQL> CAST(`var` AS DOUBLE)
```

```
translate_sql(as.integer(var), con = con)
```

```
<SQL> CAST(`var` AS INT)
```

```
translate_sql(as.integer64(var), con = con)
```

```
<SQL> CAST(`var` AS BIGINT)
```

```
translate_sql(as.character(var), con = con)
```

```
<SQL> CAST(`var` AS STRING)
```

```
translate_sql(as.Date(var), con = con)
```

```
<SQL> CAST(`var` AS DATE)
```

```
translate_sql(as_date(var), con = con)
```

```
<SQL> CAST(`var` AS DATE)
```

```
translate_sql(as.POSIXct(var), con = con)
```

```
<SQL> CAST(`var` AS TIMESTAMP)
```

```
translate_sql(as_datetime(var), con = con)
```

```
<SQL> CAST(`var` AS TIMESTAMP)
```

```
translate_sql(as.logical(var), con = con)
```

```
<SQL> CAST(`var` AS BOOLEAN)
```


3.2 Comparison and logical operators

Base R comparison operators, such as `<`, `<=`, `==`, `>=`, `>`, are also well supported in all database backends. Logical operators, such as `&` and `|`, can also be used as if the data were in R.

💡 Show SQL

3.2.1 DuckDB

```
con <- simulate_duckdb()
translate_sql(var_1 == var_2, con = con)
```

```
<SQL> `var_1` = `var_2`
```

```
translate_sql(var_1 >= var_2, con = con)
```

```
<SQL> `var_1` >= `var_2`
```

```
translate_sql(var_1 < 100, con = con)
```

```
<SQL> `var_1` < 100.0
```

```
translate_sql(var_1 %in% c("a", "b", "c"), con = con)
```

```
<SQL> `var_1` IN ('a', 'b', 'c')
```

```
translate_sql(!var_1 %in% c("a", "b", "c"), con = con)
```

```
<SQL> NOT(`var_1` IN ('a', 'b', 'c'))
```

```
translate_sql(is.na(var_1), con = con)
```

```
<SQL> (`var_1` IS NULL)
```

```
translate_sql(!is.na(var_1), con = con)
```

```
<SQL> NOT((`var_1` IS NULL))
```

```
translate_sql(var_1 >= 100 & var_1 < 200, con = con)
```

```
<SQL> `var_1` >= 100.0 AND `var_1` < 200.0
```

```
translate_sql(var_1 >= 100 | var_1 < 200, con = con)
```

```
<SQL> `var_1` >= 100.0 OR `var_1` < 200.0
```

3.2.2 Postgres

```
con <- simulate_postgres()
translate_sql(var_1 == var_2, con = con)
```

```
<SQL> `var_1` = `var_2`
```

```
translate_sql(var_1 >= var_2, con = con)
```

```
<SQL> `var_1` >= `var_2`
```

```
translate_sql(var_1 < 100, con = con)
```

```
<SQL> `var_1` < 100.0
```

```
translate_sql(var_1 %in% c("a", "b", "c"), con = con)
```

```
<SQL> `var_1` IN ('a', 'b', 'c')
```

```
translate_sql(!var_1 %in% c("a", "b", "c"), con = con)
```

```
<SQL> NOT(`var_1` IN ('a', 'b', 'c'))
```

```
translate_sql(is.na(var_1), con = con)
```

```
<SQL> (`var_1` IS NULL)
```

```
translate_sql(!is.na(var_1), con = con)
```

```
<SQL> NOT((`var_1` IS NULL))
```

```
translate_sql(var_1 >= 100 & var_1 < 200, con = con)
```

```
<SQL> `var_1` >= 100.0 AND `var_1` < 200.0
```

```
translate_sql(var_1 >= 100 | var_1 < 200, con = con)
```

```
<SQL> `var_1` >= 100.0 OR `var_1` < 200.0
```

3.2.3 SQL Server

```
con <- simulate_mssql()
translate_sql(var_1 == var_2, con = con)
```

```
<SQL> `var_1` = `var_2`
```

```
translate_sql(var_1 >= var_2, con = con)
```

```
<SQL> `var_1` >= `var_2`
```

```
translate_sql(var_1 < 100, con = con)
```

```
<SQL> `var_1` < 100.0
```

```
translate_sql(var_1 %in% c("a", "b", "c"), con = con)
```

```
<SQL> `var_1` IN ('a', 'b', 'c')
```

```
translate_sql(!var_1 %in% c("a", "b", "c"), con = con)
```

```
<SQL> NOT(`var_1` IN ('a', 'b', 'c'))
```

```
translate_sql(is.na(var_1), con = con)
```

```
<SQL> (`var_1` IS NULL)
```

```
translate_sql(!is.na(var_1), con = con)
```

```
<SQL> NOT((`var_1` IS NULL))
```

```
translate_sql(var_1 >= 100 & var_1 < 200, con = con)
```

```
<SQL> `var_1` >= 100.0 AND `var_1` < 200.0
```

```
translate_sql(var_1 >= 100 | var_1 < 200, con = con)
```

```
<SQL> `var_1` >= 100.0 OR `var_1` < 200.0
```

3.2.4 Redshift

```
con <- simulate_redshift()
translate_sql(var_1 == var_2, con = con)
```

```
<SQL> `var_1` = `var_2`
```

```
translate_sql(var_1 >= var_2, con = con)
```

```
<SQL> `var_1` >= `var_2`
```

```
translate_sql(var_1 < 100, con = con)
```

```
<SQL> `var_1` < 100.0
```

```
translate_sql(var_1 %in% c("a", "b", "c"), con = con)
```

```
<SQL> `var_1` IN ('a', 'b', 'c')
```

```
translate_sql(!var_1 %in% c("a", "b", "c"), con = con)
```

```
<SQL> NOT(`var_1` IN ('a', 'b', 'c'))
```

```
translate_sql(is.na(var_1), con = con)
```

```
<SQL> (`var_1` IS NULL)
```

```
translate_sql(!is.na(var_1), con = con)
```

```
<SQL> NOT((`var_1` IS NULL))
```

```
translate_sql(var_1 >= 100 & var_1 < 200, con = con)
```

```
<SQL> `var_1` >= 100.0 AND `var_1` < 200.0
```

```
translate_sql(var_1 >= 100 | var_1 < 200, con = con)
```

```
<SQL> `var_1` >= 100.0 OR `var_1` < 200.0
```

3.2.5 Snowflake

```
con <- simulate_snowflake()
translate_sql(var_1 == var_2, con = con)
```

```
<SQL> `var_1` = `var_2`
```

```
translate_sql(var_1 >= var_2, con = con)
```

```
<SQL> `var_1` >= `var_2`
```

```
translate_sql(var_1 < 100, con = con)
```

```
<SQL> `var_1` < 100.0
```

```
translate_sql(var_1 %in% c("a", "b", "c"), con = con)
```

```
<SQL> `var_1` IN ('a', 'b', 'c')
```

```
translate_sql(!var_1 %in% c("a", "b", "c"), con = con)
```

```
<SQL> NOT(`var_1` IN ('a', 'b', 'c'))
```

```
translate_sql(is.na(var_1), con = con)
```

```
<SQL> (`var_1` IS NULL)
```

```
translate_sql(!is.na(var_1), con = con)
```

```
<SQL> NOT((`var_1` IS NULL))
```

```
translate_sql(var_1 >= 100 & var_1 < 200, con = con)
```

```
<SQL> `var_1` >= 100.0 AND `var_1` < 200.0
```

```
translate_sql(var_1 >= 100 | var_1 < 200, con = con)
```

```
<SQL> `var_1` >= 100.0 OR `var_1` < 200.0
```

3.2.6 Spark

```
con <- simulate_spark_sql()
translate_sql(var_1 == var_2, con = con)
```

```
<SQL> `var_1` = `var_2`
```

```
translate_sql(var_1 >= var_2, con = con)
```

```
<SQL> `var_1` >= `var_2`
```

```
translate_sql(var_1 < 100, con = con)
```

```
<SQL> `var_1` < 100.0
```

```
translate_sql(var_1 %in% c("a", "b", "c"), con = con)
```

```
<SQL> `var_1` IN ('a', 'b', 'c')
```

```
translate_sql(!var_1 %in% c("a", "b", "c"), con = con)
```

```
<SQL> NOT(`var_1` IN ('a', 'b', 'c'))
```

```
translate_sql(is.na(var_1), con = con)
```

```
<SQL> (`var_1` IS NULL)
```

```
translate_sql(!is.na(var_1), con = con)
```

```
<SQL> NOT((`var_1` IS NULL))
```

```
translate_sql(var_1 >= 100 & var_1 < 200, con = con)
```

```
<SQL> `var_1` >= 100.0 AND `var_1` < 200.0
```

```
translate_sql(var_1 >= 100 | var_1 < 200, con = con)
```

```
<SQL> `var_1` >= 100.0 OR `var_1` < 200.0
```

3.3 Conditional statements

The base `ifelse` function, along with `if_else` and `case_when` from `dplyr` are translated for each database backend. As can be seen in the translations, `case_when` maps to the SQL CASE WHEN statement.

💡 Show SQL

3.3.1 DuckDB

```
con <- simulate_duckdb()
translate_sql(ifelse(var == "a", 1L, 2L), con = con)
```

```
<SQL> CASE WHEN (`var` = 'a') THEN 1 WHEN NOT (`var` = 'a') THEN 2 END
```

```
translate_sql(if_else(var == "a", 1L, 2L), con = con)
```

```
<SQL> CASE WHEN (`var` = 'a') THEN 1 WHEN NOT (`var` = 'a') THEN 2 END
```

```
translate_sql(case_when(var == "a" ~ 1L, .default = 2L), con = con)
```

```
<SQL> CASE WHEN (`var` = 'a') THEN 1 ELSE 2 END
```

```
translate_sql(case_when(var == "a" ~ 1L, var == "b" ~ 2L, var == "c" ~ 3L, .default = NULL),
              con = con)
```

```
<SQL> CASE
WHEN (`var` = 'a') THEN 1
WHEN (`var` = 'b') THEN 2
WHEN (`var` = 'c') THEN 3
END
```

```
translate_sql(case_when(var == "a" ~ 1L, var == "b" ~ 2L, var == "c" ~ 3L, .default = "something else"),
              con = con)
```

```
<SQL> CASE
WHEN (`var` = 'a') THEN 1
WHEN (`var` = 'b') THEN 2
WHEN (`var` = 'c') THEN 3
ELSE 'something else'
END
```

3.3.2 Postgres

```
con <- simulate_postgres()
translate_sql(ifelse(var == "a", 1L, 2L), con = con)
```

```
<SQL> CASE WHEN (`var` = 'a') THEN 1 WHEN NOT (`var` = 'a') THEN 2 END
```

```
translate_sql(if_else(var == "a", 1L, 2L), con = con)
```

```
<SQL> CASE WHEN (`var` = 'a') THEN 1 WHEN NOT (`var` = 'a') THEN 2 END
```

```
translate_sql(case_when(var == "a" ~ 1L, .default = 2L), con = con)
```

```
<SQL> CASE WHEN (`var` = 'a') THEN 1 ELSE 2 END
```

```
translate_sql(case_when(var == "a" ~ 1L, var == "b" ~ 2L, var == "c" ~ 3L, .default = NULL),
              con = con)
```

```
<SQL> CASE
WHEN (`var` = 'a') THEN 1
WHEN (`var` = 'b') THEN 2
WHEN (`var` = 'c') THEN 3
END
```

```
translate_sql(case_when(var == "a" ~ 1L, var == "b" ~ 2L, var == "c" ~ 3L, .default = "something else"),
              con = con)
```

```
<SQL> CASE
WHEN (`var` = 'a') THEN 1
WHEN (`var` = 'b') THEN 2
WHEN (`var` = 'c') THEN 3
ELSE 'something else'
END
```

3.3.3 SQL Server

```
con <- simulate_mssql()
translate_sql(ifelse(var == "a", 1L, 2L), con = con)
```

```
<SQL> IIF(`var` = 'a', 1, 2)
```



```
translate_sql(if_else(var == "a", 1L, 2L), con = con)
```

```
<SQL> IIF(`var` = 'a', 1, 2)
```

```
translate_sql(case_when(var == "a" ~ 1L, .default = 2L), con = con)
```

```
<SQL> CASE WHEN (`var` = 'a') THEN 1 ELSE 2 END
```

```
translate_sql(case_when(var == "a" ~ 1L, var == "b" ~ 2L, var == "c" ~ 3L, .default = NULL),  
              con = con)
```

```
<SQL> CASE  
WHEN (`var` = 'a') THEN 1  
WHEN (`var` = 'b') THEN 2  
WHEN (`var` = 'c') THEN 3  
END
```

```
translate_sql(case_when(var == "a" ~ 1L, var == "b" ~ 2L, var == "c" ~ 3L, .default = "some"),  
              con = con)
```

```
<SQL> CASE  
WHEN (`var` = 'a') THEN 1  
WHEN (`var` = 'b') THEN 2  
WHEN (`var` = 'c') THEN 3  
ELSE 'something else'  
END
```

3.3.4 Redshift

```
con <- simulate_redshift()  
translate_sql(ifelse(var == "a", 1L, 2L), con = con)
```

```
<SQL> CASE WHEN (`var` = 'a') THEN 1 WHEN NOT (`var` = 'a') THEN 2 END
```

```
translate_sql(if_else(var == "a", 1L, 2L), con = con)
```

```
<SQL> CASE WHEN (`var` = 'a') THEN 1 WHEN NOT (`var` = 'a') THEN 2 END
```

```
translate_sql(case_when(var == "a" ~ 1L, .default = 2L), con = con)
```

```
<SQL> CASE WHEN (`var` = 'a') THEN 1 ELSE 2 END
```

```
translate_sql(case_when(var == "a" ~ 1L, var == "b" ~ 2L, var == "c" ~ 3L, .default = NULL),  
              con = con)
```

```
<SQL> CASE  
WHEN (`var` = 'a') THEN 1  
WHEN (`var` = 'b') THEN 2  
WHEN (`var` = 'c') THEN 3  
END
```

```
translate_sql(case_when(var == "a" ~ 1L, var == "b" ~ 2L, var == "c" ~ 3L, .default = "some"),  
              con = con)
```

```
<SQL> CASE  
WHEN (`var` = 'a') THEN 1  
WHEN (`var` = 'b') THEN 2  
WHEN (`var` = 'c') THEN 3  
ELSE 'something else'  
END
```

3.3.5 Snowflake

```
con <- simulate_snowflake()  
translate_sql(ifelse(var == "a", 1L, 2L), con = con)
```

```
<SQL> CASE WHEN (`var` = 'a') THEN 1 WHEN NOT (`var` = 'a') THEN 2 END
```

```
translate_sql(if_else(var == "a", 1L, 2L), con = con)
```

```
<SQL> CASE WHEN (`var` = 'a') THEN 1 WHEN NOT (`var` = 'a') THEN 2 END
```

```
translate_sql(case_when(var == "a" ~ 1L, .default = 2L), con = con)
```

```
<SQL> CASE WHEN (`var` = 'a') THEN 1 ELSE 2 END
```

```
translate_sql(case_when(var == "a" ~ 1L, var == "b" ~ 2L, var == "c" ~ 3L, .default = NULL),  
              con = con)
```

```
<SQL> CASE
WHEN (`var` = 'a') THEN 1
WHEN (`var` = 'b') THEN 2
WHEN (`var` = 'c') THEN 3
END
```

```
translate_sql(case_when(var == "a" ~ 1L, var == "b" ~ 2L, var == "c" ~ 3L, .default = "something else" ~ 4L), con = con)
```

```
<SQL> CASE
WHEN (`var` = 'a') THEN 1
WHEN (`var` = 'b') THEN 2
WHEN (`var` = 'c') THEN 3
ELSE 'something else'
END
```

3.3.6 Spark

```
con <- simulate_spark_sql()
translate_sql(ifelse(var == "a", 1L, 2L), con = con)
```

```
<SQL> CASE WHEN (`var` = 'a') THEN 1 WHEN NOT (`var` = 'a') THEN 2 END
```

```
translate_sql(if_else(var == "a", 1L, 2L), con = con)
```

```
<SQL> CASE WHEN (`var` = 'a') THEN 1 WHEN NOT (`var` = 'a') THEN 2 END
```

```
translate_sql(case_when(var == "a" ~ 1L, .default = 2L), con = con)
```

```
<SQL> CASE WHEN (`var` = 'a') THEN 1 ELSE 2 END
```

```
translate_sql(case_when(var == "a" ~ 1L, var == "b" ~ 2L, var == "c" ~ 3L, .default = NULL), con = con)
```


```
<SQL> CASE
WHEN (`var` = 'a') THEN 1
WHEN (`var` = 'b') THEN 2
WHEN (`var` = 'c') THEN 3
END
```

```
translate_sql(case_when(var == "a" ~ 1L, var == "b" ~ 2L, var == "c" ~ 3L, .default = "something else",
                        con = con)
```

```
<SQL> CASE
WHEN (`var` = 'a') THEN 1
WHEN (`var` = 'b') THEN 2
WHEN (`var` = 'c') THEN 3
ELSE 'something else'
END
```

3.4 Working with strings

Compared to the previous sections, there is much more variation in support of functions to work with strings across database management systems. In particular, although various useful `stringr` functions do have translations ubiquitously, it can be seen below that more translations are available for some databases compared to others.

 Show SQL

3.4.1 DuckDB

```
con <- simulate_duckdb()
translate_sql(nchar(var), con = con)
```

```
<SQL> LENGTH(`var`)
```

```
translate_sql(nzchar(var), con = con)
```

```
<SQL> ((`var` IS NULL) OR `var` != '')
```

```
translate_sql(substr(var, 1, 2), con = con)
```

```
<SQL> SUBSTR(`var`, 1, 2)
```

```
translate_sql(trimws(var), con = con)
```

```
<SQL> LTRIM(RTRIM(`var`))
```

```
translate_sql(tolower(var), con = con)
```

<SQL> LOWER(`var`)

```
translate_sql(str_to_lower(var), con = con)
```

<SQL> LOWER(`var`)

```
translate_sql(toupper(var), con = con)
```

<SQL> UPPER(`var`)

```
translate_sql(str_to_upper(var), con = con)
```

<SQL> UPPER(`var`)

```
translate_sql(str_to_title(var), con = con)
```

<SQL> INITCAP(`var`)

```
translate_sql(str_trim(var), con = con)
```

<SQL> LTRIM(RTRIM(`var`))

```
translate_sql(str_squish(var), con = con)
```

<SQL> TRIM(REGEXP_REPLACE(`var`, '\s+', ' ', 'g'))

```
translate_sql(str_detect(var, "b"), con = con)
```

<SQL> REGEXP_MATCHES(`var`, 'b')

```
translate_sql(str_detect(var, "b", negate = TRUE), con = con)
```

<SQL> (NOT(REGEXP_MATCHES(`var`, 'b')))

```
translate_sql(str_detect(var, "[aeiou]"), con = con)
```

<SQL> REGEXP_MATCHES(`var`, '[aeiou]')

```
translate_sql(str_replace(var, "a", "b"), con = con)
```

```
<SQL> REGEXP_REPLACE(`var`, 'a', 'b')
```

```
translate_sql(str_replace_all(var, "a", "b"), con = con)
```

```
<SQL> REGEXP_REPLACE(`var`, 'a', 'b', 'g')
```

```
translate_sql(str_remove(var, "a"), con = con)
```

```
<SQL> REGEXP_REPLACE(`var`, 'a', '')
```

```
translate_sql(str_remove_all(var, "a"), con = con)
```

```
<SQL> REGEXP_REPLACE(`var`, 'a', '', 'g')
```

```
translate_sql(str_like(var, "a"), con = con)
```

```
<SQL> `var` LIKE 'a'
```

```
translate_sql(str_starts(var, "a"), con = con)
```

```
<SQL> REGEXP_MATCHES(`var`, '^(?:' || 'a' || ')')
```

```
translate_sql(str_ends(var, "a"), con = con)
```

```
<SQL> REGEXP_MATCHES(`var`, '(?:' || 'a' || ')$')
```

3.4.2 Postgres

```
con <- simulate_postgres()  
translate_sql(nchar(var), con = con)
```

```
<SQL> LENGTH(`var`)
```

```
translate_sql(nzchar(var), con = con)
```

```
<SQL> ((`var` IS NULL) OR `var` != '')
```

```
translate_sql(substr(var, 1, 2), con = con)
```

```
<SQL> SUBSTR(`var`, 1, 2)
```

```
translate_sql(trimws(var), con = con)
```

```
<SQL> LTRIM(RTRIM(`var`))
```

```
translate_sql(tolower(var), con = con)
```

```
<SQL> LOWER(`var`)
```

```
translate_sql(str_to_lower(var), con = con)
```

```
<SQL> LOWER(`var`)
```

```
translate_sql(toupper(var), con = con)
```

```
<SQL> UPPER(`var`)
```

```
translate_sql(str_to_upper(var), con = con)
```

```
<SQL> UPPER(`var`)
```

```
translate_sql(str_to_title(var), con = con)
```

```
<SQL> INITCAP(`var`)
```

```
translate_sql(str_trim(var), con = con)
```

```
<SQL> LTRIM(RTRIM(`var`))
```

```
translate_sql(str_squish(var), con = con)
```

```
<SQL> LTRIM(RTRIM(REGEXP_REPLACE(`var`, '\s+', ' ', 'g')))
```

```
translate_sql(str_detect(var, "b"), con = con)
```

<SQL> `var` ~ 'b'

```
translate_sql(str_detect(var, "b", negate = TRUE), con = con)
```

<SQL> !(`var` ~ 'b')

```
translate_sql(str_detect(var, "[aeiou]"), con = con)
```

<SQL> `var` ~ '[aeiou]'

```
translate_sql(str_replace(var, "a", "b"), con = con)
```

<SQL> REGEXP_REPLACE(`var`, 'a', 'b')

```
translate_sql(str_replace_all(var, "a", "b"), con = con)
```

<SQL> REGEXP_REPLACE(`var`, 'a', 'b', 'g')

```
translate_sql(str_remove(var, "a"), con = con)
```

<SQL> REGEXP_REPLACE(`var`, 'a', '')

```
translate_sql(str_remove_all(var, "a"), con = con)
```

<SQL> REGEXP_REPLACE(`var`, 'a', '', 'g')

```
translate_sql(str_like(var, "a"), con = con)
```

<SQL> `var` ILIKE 'a'

```
translate_sql(str_starts(var, "a"), con = con)
```

Error in `str_starts()`:

! Only fixed patterns are supported on database backends.

```
translate_sql(str_ends(var, "a"), con = con)
```

Error in `str_ends()`:

! Only fixed patterns are supported on database backends.

3.4.3 SQL Server

```
con <- simulate_mssql()
translate_sql(nchar(var), con = con)
```

```
<SQL> LEN(`var`)
```

```
translate_sql(nzchar(var), con = con)
```

```
<SQL> ((`var` IS NULL) OR `var` != '')
```

```
translate_sql(substr(var, 1, 2), con = con)
```

```
<SQL> SUBSTRING(`var`, 1, 2)
```

```
translate_sql(trimws(var), con = con)
```

```
<SQL> LTRIM(RTRIM(`var`))
```

```
translate_sql(tolower(var), con = con)
```

```
<SQL> LOWER(`var`)
```

```
translate_sql(str_to_lower(var), con = con)
```

```
<SQL> LOWER(`var`)
```

```
translate_sql(toupper(var), con = con)
```

```
<SQL> UPPER(`var`)
```

```
translate_sql(str_to_upper(var), con = con)
```

```
<SQL> UPPER(`var`)
```

```
translate_sql(str_to_title(var), con = con)
```

```
Error in `str_to_title()`:  
! `str_to_title()` is not available in this SQL variant.
```

```
translate_sql(str_trim(var), con = con)
```

<SQL> LTRIM(RTRIM(`var`))

```
translate_sql(str_squish(var), con = con)
```

Error in `str_squish()`:
! `str_squish()` is not available in this SQL variant.

```
translate_sql(str_detect(var, "b"), con = con)
```

Error in `str_detect()`:
! Only fixed patterns are supported on database backends.

```
translate_sql(str_detect(var, "b", negate = TRUE), con = con)
```

Error in `str_detect()`:
! Only fixed patterns are supported on database backends.

```
translate_sql(str_detect(var, "[aeiou]"), con = con)
```

Error in `str_detect()`:
! Only fixed patterns are supported on database backends.

```
translate_sql(str_replace(var, "a", "b"), con = con)
```

Error in `str_replace()`:
! `str_replace()` is not available in this SQL variant.

```
translate_sql(str_replace_all(var, "a", "b"), con = con)
```

Error in `str_replace_all()`:
! `str_replace_all()` is not available in this SQL variant.

```
translate_sql(str_remove(var, "a"), con = con)
```

Error in `str_remove()`:
! `str_remove()` is not available in this SQL variant.

```
translate_sql(str_remove_all(var, "a"), con = con)
```

Error in `str_remove_all()`:
! `str_remove_all()` is not available in this SQL variant.

```
translate_sql(str_like(var, "a"), con = con)
```

```
<SQL> `var` LIKE 'a'
```

```
translate_sql(str_starts(var, "a"), con = con)
```

Error in `str_starts()`:
! Only fixed patterns are supported on database backends.

```
translate_sql(str_ends(var, "a"), con = con)
```

Error in `str_ends()`:
! Only fixed patterns are supported on database backends.

3.4.4 Redshift

```
con <- simulate_redshift()  
translate_sql(nchar(var), con = con)
```

```
<SQL> LENGTH(`var`)
```

```
translate_sql(nzchar(var), con = con)
```

```
<SQL> ((`var` IS NULL) OR `var` != '')
```

```
translate_sql(substr(var, 1, 2), con = con)
```

```
<SQL> SUBSTRING(`var`, 1, 2)
```

```
translate_sql(trimws(var), con = con)
```

```
<SQL> LTRIM(RTRIM(`var`))
```

```
translate_sql(tolower(var), con = con)
```

<SQL> LOWER(`var`)

```
translate_sql(str_to_lower(var), con = con)
```

<SQL> LOWER(`var`)

```
translate_sql(toupper(var), con = con)
```

<SQL> UPPER(`var`)

```
translate_sql(str_to_upper(var), con = con)
```

<SQL> UPPER(`var`)

```
translate_sql(str_to_title(var), con = con)
```

<SQL> INITCAP(`var`)

```
translate_sql(str_trim(var), con = con)
```

<SQL> LTRIM(RTRIM(`var`))

```
translate_sql(str_squish(var), con = con)
```

<SQL> LTRIM(RTRIM(REGEXP_REPLACE(`var`, '\s+', ' ', 'g')))

```
translate_sql(str_detect(var, "b"), con = con)
```

<SQL> `var` ~ 'b'

```
translate_sql(str_detect(var, "b", negate = TRUE), con = con)
```

<SQL> !(`var` ~ 'b')

```
translate_sql(str_detect(var, "[aeiou]"), con = con)
```

<SQL> `var` ~ '[aeiou]'

```
translate_sql(str_replace(var, "a", "b"), con = con)
```

Error in `str_replace()`:
! `str_replace()` is not available in this SQL variant.

```
translate_sql(str_replace_all(var, "a", "b"), con = con)
```

```
<SQL> REGEXP_REPLACE(`var`, 'a', 'b')
```

```
translate_sql(str_remove(var, "a"), con = con)
```

```
<SQL> REGEXP_REPLACE(`var`, 'a', '')
```

```
translate_sql(str_remove_all(var, "a"), con = con)
```

```
<SQL> REGEXP_REPLACE(`var`, 'a', '', 'g')
```

```
translate_sql(str_like(var, "a"), con = con)
```

```
<SQL> `var` ILIKE 'a'
```

```
translate_sql(str_starts(var, "a"), con = con)
```

Error in `str_starts()`:
! Only fixed patterns are supported on database backends.

```
translate_sql(str_ends(var, "a"), con = con)
```

Error in `str_ends()`:
! Only fixed patterns are supported on database backends.

3.4.5 Snowflake

```
con <- simulate_snowflake()  
translate_sql(nchar(var), con = con)
```

```
<SQL> LENGTH(`var`)
```

```
translate_sql(nzchar(var), con = con)
```

<SQL> ((`var` IS NULL) OR `var` != '')

```
translate_sql(substr(var, 1, 2), con = con)
```

<SQL> SUBSTR(`var`, 1, 2)

```
translate_sql(trimws(var), con = con)
```

<SQL> LTRIM(RTRIM(`var`))

```
translate_sql(tolower(var), con = con)
```

<SQL> LOWER(`var`)

```
translate_sql(str_to_lower(var), con = con)
```

<SQL> LOWER(`var`)

```
translate_sql(toupper(var), con = con)
```

<SQL> UPPER(`var`)

```
translate_sql(str_to_upper(var), con = con)
```

<SQL> UPPER(`var`)

```
translate_sql(str_to_title(var), con = con)
```

<SQL> INITCAP(`var`)

```
translate_sql(str_trim(var), con = con)
```

<SQL> TRIM(`var`)

```
translate_sql(str_squish(var), con = con)
```

<SQL> REGEXP_REPLACE(TRIM(`var`), '\\s+', ' ')

```
translate_sql(str_detect(var, "b"), con = con)
```

```
<SQL> REGEXP_INSTR(`var`, 'b') != 0
```

```
translate_sql(str_detect(var, "b", negate = TRUE), con = con)
```

```
<SQL> REGEXP_INSTR(`var`, 'b') = 0
```

```
translate_sql(str_detect(var, "[aeiou]"), con = con)
```

```
<SQL> REGEXP_INSTR(`var`, '[aeiou]') != 0
```

```
translate_sql(str_replace(var, "a", "b"), con = con)
```

```
<SQL> REGEXP_REPLACE(`var`, 'a', 'b', 1.0, 1.0)
```

```
translate_sql(str_replace_all(var, "a", "b"), con = con)
```

```
<SQL> REGEXP_REPLACE(`var`, 'a', 'b')
```

```
translate_sql(str_remove(var, "a"), con = con)
```

```
<SQL> REGEXP_REPLACE(`var`, 'a', '', 1.0, 1.0)
```

```
translate_sql(str_remove_all(var, "a"), con = con)
```

```
<SQL> REGEXP_REPLACE(`var`, 'a')
```

```
translate_sql(str_like(var, "a"), con = con)
```

```
<SQL> `var` LIKE 'a'
```

```
translate_sql(str_starts(var, "a"), con = con)
```

```
<SQL> REGEXP_INSTR(`var`, 'a') = 1
```

```
translate_sql(str_ends(var, "a"), con = con)
```

```
<SQL> REGEXP_INSTR(`var`, 'a', 1, 1, 1) = (LENGTH(`var`) + 1)
```

3.4.6 Spark

```
con <- simulate_spark_sql()
translate_sql(nchar(var), con = con)
```

```
<SQL> LENGTH(`var`)
```

```
translate_sql(nzchar(var), con = con)
```

```
<SQL> ((`var` IS NULL) OR `var` != '')
```

```
translate_sql(substr(var, 1, 2), con = con)
```

```
<SQL> SUBSTR(`var`, 1, 2)
```

```
translate_sql(trimws(var), con = con)
```

```
<SQL> LTRIM(RTRIM(`var`))
```

```
translate_sql(tolower(var), con = con)
```

```
<SQL> LOWER(`var`)
```

```
translate_sql(str_to_lower(var), con = con)
```

```
<SQL> LOWER(`var`)
```

```
translate_sql(toupper(var), con = con)
```

```
<SQL> UPPER(`var`)
```

```
translate_sql(str_to_upper(var), con = con)
```

```
<SQL> UPPER(`var`)
```

```
translate_sql(str_to_title(var), con = con)
```

```
<SQL> INITCAP(`var`)
```



```
translate_sql(str_trim(var), con = con)
```

```
<SQL> LTRIM(RTRIM(`var`))
```

```
translate_sql(str_squish(var), con = con)
```

```
Error in `str_squish()`:  
! `str_squish()` is not available in this SQL variant.
```

```
translate_sql(str_detect(var, "b"), con = con)
```

```
Error in `str_detect()`:  
! Only fixed patterns are supported on database backends.
```

```
translate_sql(str_detect(var, "b", negate = TRUE), con = con)
```

```
Error in `str_detect()`:  
! Only fixed patterns are supported on database backends.
```

```
translate_sql(str_detect(var, "[aeiou]"), con = con)
```

```
Error in `str_detect()`:  
! Only fixed patterns are supported on database backends.
```

```
translate_sql(str_replace(var, "a", "b"), con = con)
```

```
Error in `str_replace()`:  
! `str_replace()` is not available in this SQL variant.
```

```
translate_sql(str_replace_all(var, "a", "b"), con = con)
```

```
Error in `str_replace_all()`:  
! `str_replace_all()` is not available in this SQL variant.
```

```
translate_sql(str_remove(var, "a"), con = con)
```

```
Error in `str_remove()`:  
! `str_remove()` is not available in this SQL variant.
```

```
translate_sql(str_remove_all(var, "a"), con = con)
```

Error in `str_remove_all()`:
! `str_remove_all()` is not available in this SQL variant.

```
translate_sql(str_like(var, "a"), con = con)
```

```
<SQL> `var` LIKE 'a'
```

```
translate_sql(str_starts(var, "a"), con = con)
```

Error in `str_starts()`:
! Only fixed patterns are supported on database backends.

```
translate_sql(str_ends(var, "a"), con = con)
```

Error in `str_ends()`:
! Only fixed patterns are supported on database backends.

3.5 Working with dates

Like with strings, support for working with dates is somewhat mixed. In general, we would use functions from the `clock` package such as `get_day()`, `get_month()`, `get_year()` to extract parts from a date, `add_days()` to add or subtract days to a date, and `date_count_between()` to get the number of days between two date variables.

 Show SQL

3.5.1 DuckDB

```
con <- simulate_duckdb()
translate_sql(get_day(date_1), con = con)
```

```
<SQL> DATE_PART('day', `date_1`)
```

```
translate_sql(get_month(date_1), con = con)
```

```
<SQL> DATE_PART('month', `date_1`)
```

```
translate_sql(get_year(date_1), con = con)
```

```
<SQL> DATE_PART('year', `date_1`)
```

```
translate_sql(add_days(date_1, 1), con = con)
```

```
<SQL> DATE_ADD(`date_1`, INTERVAL (1.0) day)
```

```
translate_sql(add_years(date_1, 1), con = con)
```

```
<SQL> DATE_ADD(`date_1`, INTERVAL (1.0) year)
```

```
translate_sql(difftime(date_1, date_2), con = con)
```

```
<SQL> difftime(`date_1`, `date_2`)
```

```
translate_sql(date_count_between(date_1, date_2, "day"), con = con)
```

```
<SQL> DATEDIFF('day', `date_1`, `date_2`)
```

```
translate_sql(date_count_between(date_1, date_2, "year"), con = con)
```

Error in date_count_between(date_1, date_2, "year"): The only supported value for `precision` is "day"

3.5.2 Postgres

```
con <- simulate_postgres()
translate_sql(get_day(date_1), con = con)
```

```
<SQL> DATE_PART('day', `date_1`)
```

```
translate_sql(get_month(date_1), con = con)
```

```
<SQL> DATE_PART('month', `date_1`)
```

```
translate_sql(get_year(date_1), con = con)
```

```
<SQL> DATE_PART('year', `date_1`)
```

```
translate_sql(add_days(date_1, 1), con = con)
```

```
<SQL> (`date_1` + 1.0*INTERVAL'1 day')
```

```
translate_sql(add_years(date_1, 1), con = con)
```

```
<SQL> (`date_1` + 1.0*INTERVAL'1 year')
```

```
translate_sql(difftime(date_1, date_2), con = con)
```

```
<SQL> (CAST(`date_1` AS DATE) - CAST(`date_2` AS DATE))
```

```
translate_sql(date_count_between(date_1, date_2, "day"), con = con)
```

```
<SQL> `date_2` - `date_1`
```

```
translate_sql(date_count_between(date_1, date_2, "year"), con = con)
```

```
Error in `date_count_between()`:  
! `precision = "year"` isn't supported on database backends.  
i It must be "day" instead.
```

3.5.3 SQL Server

```
con <- simulate_mssql()  
translate_sql(get_day(date_1), con = con)
```

```
<SQL> DATEPART(DAY, `date_1`)
```

```
translate_sql(get_month(date_1), con = con)
```

```
<SQL> DATEPART(MONTH, `date_1`)
```

```
translate_sql(get_year(date_1), con = con)
```

```
<SQL> DATEPART(YEAR, `date_1`)
```

```
translate_sql(add_days(date_1, 1), con = con)
```

```
<SQL> DATEADD(DAY, 1.0, `date_1`)
```

```
translate_sql(add_years(date_1, 1), con = con)
```

```
<SQL> DATEADD(YEAR, 1.0, `date_1`)
```

```
translate_sql(difftime(date_1, date_2), con = con)
```

```
<SQL> DATEDIFF(DAY, `date_2`, `date_1`)
```

```
translate_sql(date_count_between(date_1, date_2, "day"), con = con)
```

```
<SQL> DATEDIFF(DAY, `date_1`, `date_2`)
```

```
translate_sql(date_count_between(date_1, date_2, "year"), con = con)
```

```
Error in `date_count_between()`:  
! `precision = "year"` isn't supported on database backends.  
i It must be "day" instead.
```

3.5.4 Redshift

```
con <- simulate_redshift()  
translate_sql(get_day(date_1), con = con)
```

```
<SQL> DATE_PART('day', `date_1`)
```

```
translate_sql(get_month(date_1), con = con)
```

```
<SQL> DATE_PART('month', `date_1`)
```

```
translate_sql(get_year(date_1), con = con)
```

```
<SQL> DATE_PART('year', `date_1`)
```

```
translate_sql(add_days(date_1, 1), con = con)
```

```
<SQL> DATEADD(DAY, 1.0, `date_1`)
```

```
translate_sql(add_years(date_1, 1), con = con)
```

```
<SQL> DATEADD(YEAR, 1.0, `date_1`)
```

```
translate_sql(difftime(date_1, date_2), con = con)
```

```
<SQL> DATEDIFF(DAY, `date_2`, `date_1`)
```

```
translate_sql(date_count_between(date_1, date_2, "day"), con = con)
```

```
<SQL> DATEDIFF(DAY, `date_1`, `date_2`)
```

```
translate_sql(date_count_between(date_1, date_2, "year"), con = con)
```

```
Error in `date_count_between()`:  
! `precision = "year"` isn't supported on database backends.  
i It must be "day" instead.
```

3.5.5 Snowflake

```
con <- simulate_snowflake()  
translate_sql(get_day(date_1), con = con)
```

```
<SQL> DATE_PART(DAY, `date_1`)
```

```
translate_sql(get_month(date_1), con = con)
```

```
<SQL> DATE_PART(MONTH, `date_1`)
```

```
translate_sql(get_year(date_1), con = con)
```

```
<SQL> DATE_PART(YEAR, `date_1`)
```

```
translate_sql(add_days(date_1, 1), con = con)
```

```
<SQL> DATEADD(DAY, 1.0, `date_1`)
```

```
translate_sql(add_years(date_1, 1), con = con)
```

```
<SQL> DATEADD(YEAR, 1.0, `date_1`)
```

```
translate_sql(difftime(date_1, date_2), con = con)
```

```
<SQL> DATEDIFF(DAY, `date_2`, `date_1`)
```

```
translate_sql(date_count_between(date_1, date_2, "day"), con = con)
```

```
<SQL> DATEDIFF(DAY, `date_1`, `date_2`)
```

```
translate_sql(date_count_between(date_1, date_2, "year"), con = con)
```

```
Error in `date_count_between()`:  
! `precision = "year"` isn't supported on database backends.  
i It must be "day" instead.
```

3.5.6 Spark

```
con <- simulate_spark_sql()  
translate_sql(get_day(date_1), con = con)
```

```
<SQL> DATE_PART('DAY', `date_1`)
```

```
translate_sql(get_month(date_1), con = con)
```

```
<SQL> DATE_PART('MONTH', `date_1`)
```

```
translate_sql(get_year(date_1), con = con)
```

```
<SQL> DATE_PART('YEAR', `date_1`)
```

```
translate_sql(add_days(date_1, 1), con = con)
```

```
<SQL> DATE_ADD(`date_1`, 1.0)
```

```
translate_sql(add_years(date_1, 1), con = con)
```

```
<SQL> ADD_MONTHS(`date_1`, 1.0 * 12.0)
```

```
translate_sql(difftime(date_1, date_2), con = con)
```

```
<SQL> DATEDIFF(`date_2`, `date_1`)
```

```
translate_sql(date_count_between(date_1, date_2, "day"), con = con)
```

```
<SQL> DATEDIFF(`date_2`, `date_1`)
```

```
translate_sql(date_count_between(date_1, date_2, "year"), con = con)
```

```
Error in `date_count_between()`:  
! `precision = "year"` isn't supported on database backends.  
i It must be "day" instead.
```

3.6 Data aggregation

Within the context of using `summarise()`, we can get aggregated results across entire columns using functions such as `n()`, `n_distinct()`, `sum()`, `min()`, `max()`, `mean()`, and `sd()`. As can be seen below, the SQL for these calculations is similar across different database management systems.

 Show SQL

3.6.1 DuckDB

```
lazy_frame(x = c(1, 2), con = simulate_duckdb()) |>  
  summarise(  
    n = n(),  
    n_unique = n_distinct(x),  
    sum = sum(x, na.rm = TRUE),  
    sum_is_1 = sum(x == 1, na.rm = TRUE),  
    min = min(x, na.rm = TRUE),  
    mean = mean(x, na.rm = TRUE),  
    max = max(x, na.rm = TRUE),  
    sd = sd(x, na.rm = TRUE)  
  ) |>  
  show_query()
```

```
<SQL>
```



```

SELECT
  COUNT(*) AS `n`,
  COUNT(DISTINCT row(`x`)) AS `n_unique`,
  SUM(`x`) AS `sum`,
  SUM(`x` = 1.0) AS `sum_is_1`,
  MIN(`x`) AS `min`,
  AVG(`x`) AS `mean`,
  MAX(`x`) AS `max`,
  STDDEV(`x`) AS `sd`
FROM `df`

```

3.6.2 Postgres

```

lazy_frame(x = c(1, 2), con = simulate_postgres()) |>
  summarise(
    n = n(),
    n_unique = n_distinct(x),
    sum = sum(x, na.rm = TRUE),
    sum_is_1 = sum(x == 1, na.rm = TRUE),
    min = min(x, na.rm = TRUE),
    mean = mean(x, na.rm = TRUE),
    max = max(x, na.rm = TRUE),
    sd = sd(x, na.rm = TRUE)
  ) |>
  show_query()

```

```

<SQL>
SELECT
  COUNT(*) AS `n`,
  COUNT(DISTINCT `x`) AS `n_unique`,
  SUM(`x`) AS `sum`,
  SUM(`x` = 1.0) AS `sum_is_1`,
  MIN(`x`) AS `min`,
  AVG(`x`) AS `mean`,
  MAX(`x`) AS `max`,
  STDDEV_SAMP(`x`) AS `sd`
FROM `df`

```

3.6.3 SQL Server

```

lazy_frame(x = c(1, 2), con = simulate_mssql()) |>
  summarise(
    n = n(),
    n_unique = n_distinct(x),
    sum = sum(x, na.rm = TRUE),
    sum_is_1 = sum(x == 1, na.rm = TRUE),
    min = min(x, na.rm = TRUE),
    mean = mean(x, na.rm = TRUE),
    max = max(x, na.rm = TRUE),
    sd = sd(x, na.rm = TRUE)
  ) |>
  show_query()

```

<SQL>

```

SELECT
  COUNT_BIG(*) AS `n`,
  COUNT(DISTINCT `x`) AS `n_unique`,
  SUM(`x`) AS `sum`,
  SUM(CAST(IIF(`x` = 1.0, 1, 0) AS BIT)) AS `sum_is_1`,
  MIN(`x`) AS `min`,
  AVG(`x`) AS `mean`,
  MAX(`x`) AS `max`,
  STDEV(`x`) AS `sd`
FROM `df`

```

3.6.4 Redshift

```

lazy_frame(x = c(1, 2), con = simulate_redshift()) |>
  summarise(
    n = n(),
    n_unique = n_distinct(x),
    sum = sum(x, na.rm = TRUE),
    sum_is_1 = sum(x == 1, na.rm = TRUE),
    min = min(x, na.rm = TRUE),
    mean = mean(x, na.rm = TRUE),
    max = max(x, na.rm = TRUE),
    sd = sd(x, na.rm = TRUE)
  ) |>
  show_query()

```

<SQL>

```

SELECT
  COUNT(*) AS `n`,
  COUNT(DISTINCT `x`) AS `n_unique`,
  SUM(`x`) AS `sum`,
  SUM(`x` = 1.0) AS `sum_is_1`,
  MIN(`x`) AS `min`,
  AVG(`x`) AS `mean`,
  MAX(`x`) AS `max`,
  STDDEV_SAMP(`x`) AS `sd`
FROM `df`

```

3.6.5 Snowflake

```

lazy_frame(x = c(1, 2), con = simulate_snowflake()) |>
  summarise(
    n = n(),
    n_unique = n_distinct(x),
    sum = sum(x, na.rm = TRUE),
    sum_is_1 = sum(x == 1, na.rm = TRUE),
    min = min(x, na.rm = TRUE),
    mean = mean(x, na.rm = TRUE),
    max = max(x, na.rm = TRUE),
    sd = sd(x, na.rm = TRUE)
  ) |>
  show_query()

```

<SQL>

```

SELECT
  COUNT(*) AS `n`,
  COUNT(DISTINCT `x`) AS `n_unique`,
  SUM(`x`) AS `sum`,
  SUM(`x` = 1.0) AS `sum_is_1`,
  MIN(`x`) AS `min`,
  AVG(`x`) AS `mean`,
  MAX(`x`) AS `max`,
  STDDEV(`x`) AS `sd`
FROM `df`

```

3.6.6 Spark

```

lazy_frame(x = c(1, 2), con = simulate_spark_sql()) |>
  summarise(
    n = n(),
    n_unique = n_distinct(x),
    sum = sum(x, na.rm = TRUE),
    sum_is_1 = sum(x == 1, na.rm = TRUE),
    min = min(x, na.rm = TRUE),
    mean = mean(x, na.rm = TRUE),
    max = max(x, na.rm = TRUE),
    sd = sd(x, na.rm = TRUE)
  ) |>
  show_query()

```

```

<SQL>
SELECT
  COUNT(*) AS `n`,
  COUNT(DISTINCT `x`) AS `n_unique`,
  SUM(`x`) AS `sum`,
  SUM(`x` = 1.0) AS `sum_is_1`,
  MIN(`x`) AS `min`,
  AVG(`x`) AS `mean`,
  MAX(`x`) AS `max`,
  STDDEV_SAMP(`x`) AS `sd`
FROM `df`

```

3.7 Window functions

entire columns. Window functions differ in that they perform calculations across rows that are in some way related to a current row. For these, we now use `mutate()` instead of `summarise()`. In the previous section, we saw how aggregate functions can be used to perform operations across

We can use window functions like `cumsum()` and `cummean()` to calculate running totals and averages, or `lag()` and `lead()` to help compare rows to their preceding or following rows.

Given that window functions compare rows to rows before or after them, we will often use `arrange()` or `window_order()` to specify the order of rows. This will translate into an `ORDER BY` clause in the SQL. In addition, we may well also want to apply window functions within some specific groupings in our data. Using `group_by()` would result in a `PARTITION BY` clause in the translated SQL so that the window function operates on each group independently.

💡 Show SQL

3.7.1 DuckDB

```
con <- simulate_duckdb()
lazy_frame(x = c(10, 20, 30), z = c(1, 2, 3), con = con) |>
  window_order(z) |>
  mutate(
    sum_x = cumsum(x),
    mean_x = cummean(x),
    lag_x = lag(x),
    lead_x = lead(x)
  ) |>
  show_query()
```

```
<SQL>
SELECT
  `df`.*,
  SUM(`x`) OVER (ORDER BY `z` ROWS UNBOUNDED PRECEDING) AS `sum_x`,
  AVG(`x`) OVER (ORDER BY `z` ROWS UNBOUNDED PRECEDING) AS `mean_x`,
  LAG(`x`, 1, NULL) OVER (ORDER BY `z`) AS `lag_x`,
  LEAD(`x`, 1, NULL) OVER (ORDER BY `z`) AS `lead_x`
FROM `df`
```

```
lazy_frame(x = c(10, 20), y = c("a", "b"), z = c(1, 2), con = con) |>
  window_order(z) |>
  group_by(y) |>
  mutate(
    sum_x = cumsum(x),
    mean_x = cummean(x),
    lag_x = lag(x),
    lead_x = lead(x)
  ) |>
  show_query()
```

```
<SQL>
SELECT
  `df`.*,
  SUM(`x`) OVER (PARTITION BY `y` ORDER BY `z` ROWS UNBOUNDED PRECEDING) AS `sum_x`,
  AVG(`x`) OVER (PARTITION BY `y` ORDER BY `z` ROWS UNBOUNDED PRECEDING) AS `mean_x`,
  LAG(`x`, 1, NULL) OVER (PARTITION BY `y` ORDER BY `z`) AS `lag_x`,
  LEAD(`x`, 1, NULL) OVER (PARTITION BY `y` ORDER BY `z`) AS `lead_x`
FROM `df`
```

3.7.2 Postgres

```
con <- simulate_postgres()
lazy_frame(x = c(10, 20, 30), z = c(1, 2, 3), con = con) |>
  window_order(z) |>
  mutate(
    sum_x = cumsum(x),
    mean_x = cummean(x),
    lag_x = lag(x),
    lead_x = lead(x)
  ) |>
  show_query()
```

```
<SQL>
SELECT
  `df`.*,
  SUM(`x`) OVER `win1` AS `sum_x`,
  AVG(`x`) OVER `win1` AS `mean_x`,
  LAG(`x`, 1, NULL) OVER `win2` AS `lag_x`,
  LEAD(`x`, 1, NULL) OVER `win2` AS `lead_x`
FROM `df`
WINDOW
  `win1` AS (ORDER BY `z` ROWS UNBOUNDED PRECEDING),
  `win2` AS (ORDER BY `z`)
```

```
lazy_frame(x = c(10, 20), y = c("a", "b"), z = c(1, 2), con = con) |>
  window_order(z) |>
  group_by(y) |>
  mutate(
    sum_x = cumsum(x),
    mean_x = cummean(x),
    lag_x = lag(x),
    lead_x = lead(x)
  ) |>
  show_query()
```

```
<SQL>
SELECT
  `df`.*,
  SUM(`x`) OVER `win1` AS `sum_x`,
  AVG(`x`) OVER `win1` AS `mean_x`,
```

```

LAG(`x`, 1, NULL) OVER `win2` AS `lag_x`,
LEAD(`x`, 1, NULL) OVER `win2` AS `lead_x`
FROM `df`
WINDOW
  `win1` AS (PARTITION BY `y` ORDER BY `z` ROWS UNBOUNDED PRECEDING),
  `win2` AS (PARTITION BY `y` ORDER BY `z`)

```

3.7.3 SQL Server

```

con <- simulate_mssql()
lazy_frame(x = c(10, 20, 30), z = c(1, 2, 3), con = con) |>
  window_order(z) |>
  mutate(
    sum_x = cumsum(x),
    mean_x = cummean(x),
    lag_x = lag(x),
    lead_x = lead(x)
  ) |>
  show_query()

```

```

<SQL>
SELECT
  `df`.*,
  SUM(`x`) OVER (ORDER BY `z` ROWS UNBOUNDED PRECEDING) AS `sum_x`,
  AVG(`x`) OVER (ORDER BY `z` ROWS UNBOUNDED PRECEDING) AS `mean_x`,
  LAG(`x`, 1, NULL) OVER (ORDER BY `z`) AS `lag_x`,
  LEAD(`x`, 1, NULL) OVER (ORDER BY `z`) AS `lead_x`
FROM `df`

```

```

lazy_frame(x = c(10, 20), y = c("a", "b"), z = c(1, 2), con = con) |>
  window_order(z) |>
  group_by(y) |>
  mutate(
    sum_x = cumsum(x),
    mean_x = cummean(x),
    lag_x = lag(x),
    lead_x = lead(x)
  ) |>
  show_query()

```

```

<SQL>

```

```

SELECT
  `df`.*,
  SUM(`x`) OVER (PARTITION BY `y` ORDER BY `z` ROWS UNBOUNDED PRECEDING) AS `sum_x`,
  AVG(`x`) OVER (PARTITION BY `y` ORDER BY `z` ROWS UNBOUNDED PRECEDING) AS `mean_x`,
  LAG(`x`, 1, NULL) OVER (PARTITION BY `y` ORDER BY `z`) AS `lag_x`,
  LEAD(`x`, 1, NULL) OVER (PARTITION BY `y` ORDER BY `z`) AS `lead_x`
FROM `df`

```

3.7.4 Redshift

```

con <- simulate_redshift()
lazy_frame(x = c(10, 20, 30), z = c(1, 2, 3), con = con) |>
  window_order(z) |>
  mutate(
    sum_x = cumsum(x),
    mean_x = cummean(x),
    lag_x = lag(x),
    lead_x = lead(x)
  ) |>
  show_query()

```

```

<SQL>
SELECT
  `df`.*,
  SUM(`x`) OVER (ORDER BY `z` ROWS UNBOUNDED PRECEDING) AS `sum_x`,
  AVG(`x`) OVER (ORDER BY `z` ROWS UNBOUNDED PRECEDING) AS `mean_x`,
  LAG(`x`, 1) OVER (ORDER BY `z`) AS `lag_x`,
  LEAD(`x`, 1) OVER (ORDER BY `z`) AS `lead_x`
FROM `df`

```

```

lazy_frame(x = c(10, 20), y = c("a", "b"), z = c(1, 2), con = con) |>
  window_order(z) |>
  group_by(y) |>
  mutate(
    sum_x = cumsum(x),
    mean_x = cummean(x),
    lag_x = lag(x),
    lead_x = lead(x)
  ) |>
  show_query()

```



```

<SQL>
SELECT
  `df`.*,
  SUM(`x`) OVER (PARTITION BY `y` ORDER BY `z` ROWS UNBOUNDED PRECEDING) AS `sum_x`,
  AVG(`x`) OVER (PARTITION BY `y` ORDER BY `z` ROWS UNBOUNDED PRECEDING) AS `mean_x`,
  LAG(`x`, 1) OVER (PARTITION BY `y` ORDER BY `z`) AS `lag_x`,
  LEAD(`x`, 1) OVER (PARTITION BY `y` ORDER BY `z`) AS `lead_x`
FROM `df`

```

3.7.5 Snowflake

```

con <- simulate_snowflake()
lazy_frame(x = c(10, 20, 30), z = c(1, 2, 3), con = con) |>
  window_order(z) |>
  mutate(
    sum_x = cumsum(x),
    mean_x = cummean(x),
    lag_x = lag(x),
    lead_x = lead(x)
  ) |>
  show_query()

```

```

<SQL>
SELECT
  `df`.*,
  SUM(`x`) OVER (ORDER BY `z` ROWS UNBOUNDED PRECEDING) AS `sum_x`,
  AVG(`x`) OVER (ORDER BY `z` ROWS UNBOUNDED PRECEDING) AS `mean_x`,
  LAG(`x`, 1, NULL) OVER (ORDER BY `z`) AS `lag_x`,
  LEAD(`x`, 1, NULL) OVER (ORDER BY `z`) AS `lead_x`
FROM `df`

```

```

lazy_frame(x = c(10, 20), y = c("a", "b"), z = c(1, 2), con = con) |>
  window_order(z) |>
  group_by(y) |>
  mutate(
    sum_x = cumsum(x),
    mean_x = cummean(x),
    lag_x = lag(x),
    lead_x = lead(x)
  ) |>
  show_query()

```

```

<SQL>
SELECT
  `df`.*,
  SUM(`x`) OVER (PARTITION BY `y` ORDER BY `z` ROWS UNBOUNDED PRECEDING) AS `sum_x`,
  AVG(`x`) OVER (PARTITION BY `y` ORDER BY `z` ROWS UNBOUNDED PRECEDING) AS `mean_x`,
  LAG(`x`, 1, NULL) OVER (PARTITION BY `y` ORDER BY `z`) AS `lag_x`,
  LEAD(`x`, 1, NULL) OVER (PARTITION BY `y` ORDER BY `z`) AS `lead_x`
FROM `df`

```

3.7.6 Spark

```

con <- simulate_spark_sql()
lazy_frame(x = c(10, 20, 30), z = c(1, 2, 3), con = con) |>
  window_order(z) |>
  mutate(
    sum_x = cumsum(x),
    mean_x = cummean(x),
    lag_x = lag(x),
    lead_x = lead(x)
  ) |>
  show_query()

```

```

<SQL>
SELECT
  `df`.*,
  SUM(`x`) OVER `win1` AS `sum_x`,
  AVG(`x`) OVER `win1` AS `mean_x`,
  LAG(`x`, 1, NULL) OVER `win2` AS `lag_x`,
  LEAD(`x`, 1, NULL) OVER `win2` AS `lead_x`
FROM `df`
WINDOW
  `win1` AS (ORDER BY `z` ROWS UNBOUNDED PRECEDING),
  `win2` AS (ORDER BY `z`)

```

```

lazy_frame(x = c(10, 20), y = c("a", "b"), z = c(1, 2), con = con) |>
  window_order(z) |>
  group_by(y) |>
  mutate(
    sum_x = cumsum(x),
    mean_x = cummean(x),
    lag_x = lag(x),
    lead_x = lead(x)
  ) |>
  show_query()

```

```

<SQL>
SELECT
  `df`.*,
  SUM(`x`) OVER `win1` AS `sum_x`,
  AVG(`x`) OVER `win1` AS `mean_x`,
  LAG(`x`, 1, NULL) OVER `win2` AS `lag_x`,
  LEAD(`x`, 1, NULL) OVER `win2` AS `lead_x`
FROM `df`
WINDOW
  `win1` AS (PARTITION BY `y` ORDER BY `z` ROWS UNBOUNDED PRECEDING),
  `win2` AS (PARTITION BY `y` ORDER BY `z`)

```

TODO add note arrange vs window_order

i arrange() vs window_order()

Although they are sometimes used interchangeably

3.8 Calculating quantiles, including the median

So far we've seen that we can perform various data manipulations and calculate summary statistics for different database management systems using the same R code. Although the translated SQL has been different, the databases all supported similar approaches to perform these queries.

A case where this is not true is when we are interested in summarizing distributions of the data and estimating quantiles. For example, let's take estimating the median as an example. Some databases only support calculating the median as an aggregation function similar to how min, mean, and max were calculated above. However, some others only support it as a

window function like lead and lag above. Unfortunately, this means that for some databases, quantiles can only be calculated using the summarise aggregation approach, while in others only the mutate window approach can be used.

💡 Show SQL

3.8.1 DuckDB

```
con <- simulate_duckdb()
lazy_frame(x = c(1,2), con = con) |>
  summarise(median = median(x, na.rm = TRUE)) |>
  show_query()
```

```
<SQL>
SELECT MEDIAN(`x`) AS `median`
FROM `df`
```

```
lazy_frame(x = c(1,2), con = con) |>
  mutate(median = median(x, na.rm = TRUE)) |>
  show_query()
```

```
<SQL>
SELECT `df`.*, MEDIAN(`x`) OVER () AS `median`
FROM `df`
```

3.8.2 Postgres

```
con <- simulate_postgres()
lazy_frame(x = c(1,2), con = con) |>
  summarise(median = median(x, na.rm = TRUE)) |>
  show_query()
```

```
<SQL>
SELECT PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY `x`) AS `median`
FROM `df`
```

```
lazy_frame(x = c(1,2), con = con) |>
  mutate(median = median(x, na.rm = TRUE)) |>
  show_query()
```

```
Error in `median()` :
! Translation of `median()` in `mutate()` is not supported for
  PostgreSQL.
i Use a combination of `summarise()` and `left_join()` instead:
  `df %>% left_join(summarise(<col> = median(x, na.rm = TRUE)))`.
```

3.8.3 SQL Server

```
con <- simulate_mssql()
lazy_frame(x = c(1,2), con = con) |>
  summarise(median = median(x, na.rm = TRUE)) |>
  show_query()
```

```
Error in `median()` :
! Translation of `median()` in `summarise()` is not supported for SQL
  Server.
i Use a combination of `distinct()` and `mutate()` for the same result:
  `mutate(<col> = median(x, na.rm = TRUE)) %>% distinct(<col>)`
```

```
lazy_frame(x = c(1,2), con = con) |>
  mutate(median = median(x, na.rm = TRUE)) |>
  show_query()
```

```
<SQL>
SELECT
  `df`.*,
  PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY `x`) OVER () AS `median`
FROM `df`
```

3.8.4 Redshift

```
con <- simulate_redshift()
lazy_frame(x = c(1,2), con = con) |>
  summarise(median = median(x, na.rm = TRUE)) |>
  show_query()
```

```
<SQL>
SELECT PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY `x`) AS `median`
FROM `df`
```

```
lazy_frame(x = c(1,2), con = con) |>
  mutate(median = median(x, na.rm = TRUE)) |>
  show_query()
```

Error in `median()`:

! Translation of `median()` in `mutate()` is not supported for PostgreSQL.

i Use a combination of `summarise()` and `left_join()` instead:
 `df %>% left_join(summarise(<col> = median(x, na.rm = TRUE)))`.

3.8.5 Snowflake

```
con <- simulate_snowflake()
lazy_frame(x = c(1,2), con = con) |>
  summarise(median = median(x, na.rm = TRUE)) |>
  show_query()
```

<SQL>

```
SELECT PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY `x`) AS `median`
FROM `df`
```

```
lazy_frame(x = c(1,2), con = con) |>
  mutate(median = median(x, na.rm = TRUE)) |>
  show_query()
```

<SQL>

```
SELECT
  `df`.*,
  PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY `x`) OVER () AS `median`
FROM `df`
```

3.8.6 Spark

```
con <- simulate_spark_sql()
lazy_frame(x = c(1,2), con = con) |>
  summarise(median = median(x, na.rm = TRUE)) |>
  show_query()
```

<SQL>

```
SELECT MEDIAN(`x`) AS `median`
FROM `df`
```

```
lazy_frame(x = c(1,2), con = con) |>  
  mutate(median = median(x, na.rm = TRUE)) |>  
  show_query()
```

```
<SQL>  
SELECT `df`.*, MEDIAN(`x`) OVER () AS `median`  
FROM `df`
```

4 Building analytic pipelines for a data model

In the previous chapters, we've seen that after connecting to a database, we can create references to the various tables we're interested in and write custom analytic code to query them. However, if we are working with the same database over and over again, we are likely to want to build some tooling for tasks we often perform.

To see how we can develop a data model with associated methods and functions, we'll use the Lahman baseball data. The data is stored across various related tables.

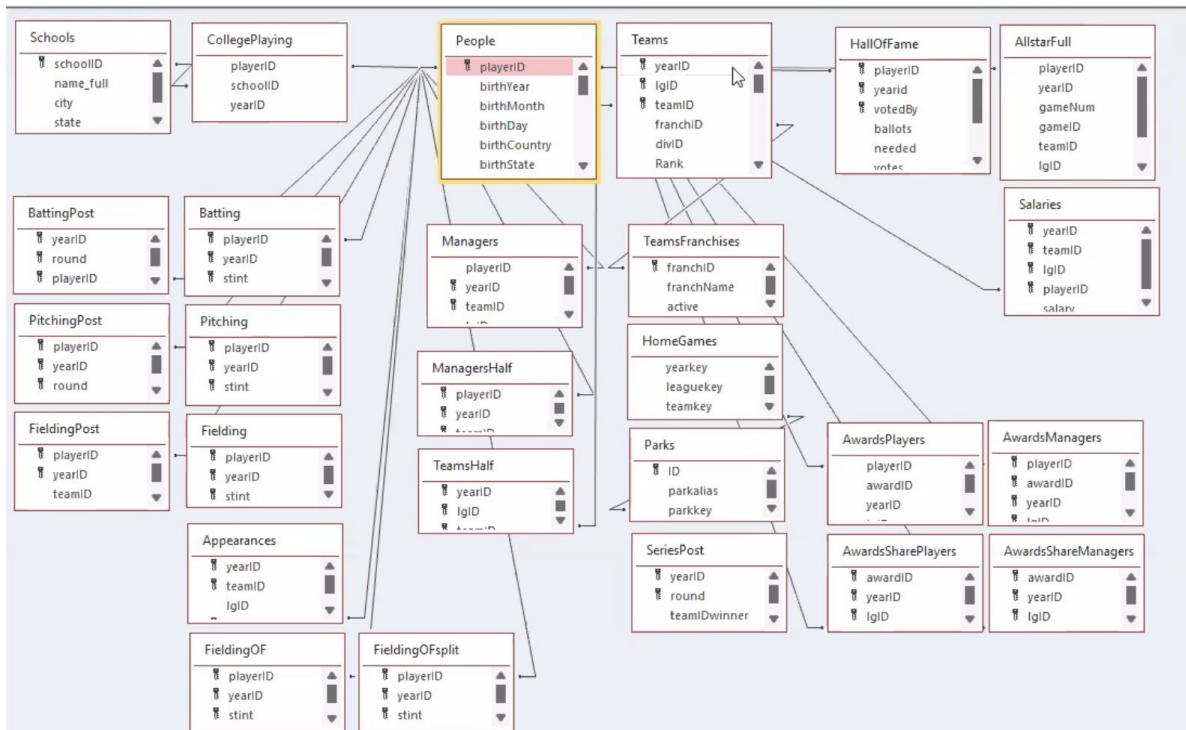


Figure 4.1: Lahman's Baseball Database schema from <https://cdalzell.github.io/Lahman/>.

4.1 Defining a data model

```
library(duckdb)
library(dplyr)
library(tidyr)
library(purrr)
library(cli)
library(dbplyr)
library(Lahman)

con <- dbConnect(drv = duckdb())
copy_lahman(con = con)
```

copy_lahman

The `copy_lahman()` function inserts all the different tables in the connection, it works similarly as we have done before with the for loop and the `dbWriteTable()` function. See that there are 28 new tables inserted in our DuckDB database:

```
dbListTables(conn = con)
```

[1] "AllstarFull"	"Appearances"	"AwardsManagers"
[4] "AwardsPlayers"	"AwardsShareManagers"	"AwardsSharePlayers"
[7] "Batting"	"BattingPost"	"CollegePlaying"
[10] "Fielding"	"FieldingOF"	"FieldingOFsplit"
[13] "FieldingPost"	"HallOfFame"	"HomeGames"
[16] "LahmanData"	"Managers"	"ManagersHalf"
[19] "Parks"	"People"	"Pitching"
[22] "PitchingPost"	"Salaries"	"Schools"
[25] "SeriesPost"	"Teams"	"TeamsFranchises"
[28] "TeamsHalf"		

Instead of manually creating references to tables of interest as we go, we will write a function to create a single reference to the Lahman data.

```
lahmanFromCon <- function(con) {
  lahmanRef <- c(
    "AllstarFull", "Appearances", "AwardsManagers", "AwardsPlayers", "AwardsManagers",
    "AwardsShareManagers", "Batting", "BattingPost", "CollegePlaying", "Fielding",
    "FieldingOF", "FieldingOFsplit", "FieldingPost", "HallOfFame", "HomeGames",
    "LahmanData", "Managers", "ManagersHalf", "Parks", "People", "Pitching",
```

```

    "PitchingPost", "Salaries", "Schools", "SeriesPost", "Teams", "TeamsFranchises",
    "TeamsHalf"
  ) |>
  set_names() |>
  map(\(x) tbl(src = con, from = x))
class(lahmanRef) <- c("lahman_ref", class(lahmanRef))
lahmanRef
}

```

With this function we can now easily get references to all our lahman tables in one go using our `lahmanFromCon()` function.

```

lahman <- lahmanFromCon(con = con)

lahman$People |>
  glimpse()

```

```

Rows: ??
Columns: 26
Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-azure:R 4.4.1/:memory:]
$ playerId    <chr> "aardsda01", "aaronha01", "aaronro01", "aasedo01", "abada~
$ birthYear   <int> 1981, 1934, 1939, 1954, 1972, 1985, 1850, 1877, 1869, 186~
$ birthMonth  <int> 12, 2, 8, 9, 8, 12, 11, 4, 11, 10, 6, 9, 3, 10, 2, 8, 9, ~
$ birthDay    <int> 27, 5, 5, 8, 25, 17, 4, 15, 11, 14, 1, 20, 16, 22, 16, 17~
$ birthCity   <chr> "Denver", "Mobile", "Mobile", "Orange", "Palm Beach", "La~
$ birthCountry <chr> "USA", "USA", "USA", "USA", "USA", "D.R.", "USA", "USA", ~
$ birthState  <chr> "CO", "AL", "AL", "CA", "FL", "La Romana", "PA", "PA", "V~
$ deathYear   <int> NA, 2021, 1984, NA, NA, NA, 1905, 1957, 1962, 1926, NA, N~
$ deathMonth  <int> NA, 1, 8, NA, NA, NA, 5, 1, 6, 4, NA, NA, 2, 6, NA, NA, N~
$ deathDay    <int> NA, 22, 16, NA, NA, NA, 17, 6, 11, 27, NA, NA, 13, 11, NA~
$ deathCountry <chr> NA, "USA", "USA", NA, NA, NA, "USA", "USA", "USA", "USA",~
$ deathState  <chr> NA, "GA", "GA", NA, NA, NA, "NJ", "FL", "VT", "CA", NA, N~
$ deathCity   <chr> NA, "Atlanta", "Atlanta", NA, NA, NA, "Pemberton", "Fort ~
$ nameFirst   <chr> "David", "Hank", "Tommie", "Don", "Andy", "Fernando", "Jo~
$ nameLast    <chr> "Aardsma", "Aaron", "Aaron", "Aase", "Abad", "Abad", "Aba~
$ nameGiven   <chr> "David Allan", "Henry Louis", "Tommie Lee", "Donald Willi~
$ weight      <int> 215, 180, 190, 190, 184, 235, 192, 170, 175, 169, 192, 22~
$ height      <int> 75, 72, 75, 75, 73, 74, 72, 71, 71, 68, 72, 74, 71, 70, 7~
$ bats        <fct> R, R, R, R, L, L, R, R, R, L, L, R, R, R, R, R, L, R, L, ~
$ throws      <fct> R, R, R, R, L, L, R, R, R, L, L, R, R, R, R, L, L, R, L, ~
$ debut       <chr> "2004-04-06", "1954-04-13", "1962-04-10", "1977-07-
26", "~

```

```
$ bbrefID      <chr> "aardsda01", "aaronha01", "aaronto01", "aasedo01", "abada~
$ finalGame    <chr> "2015-08-23", "1976-10-03", "1971-09-26", "1990-10-
03", "~
$ retroID      <chr> "aardd001", "aarah101", "aarot101", "aased001", "abada001~
$ deathDate    <date> NA, 2021-01-22, 1984-08-16, NA, NA, NA, 1905-05-17, 1957~
$ birthDate    <date> 1981-12-27, 1934-02-05, 1939-08-05, 1954-09-08, 1972-
08--
```

i The dm package

In this chapter we will be creating a bespoke data model for our database. This approach can be further extended using the [dm](#) package, which also provides various helpful functions for creating a data model and working with it.

Similar to above, we can use [dm\(\)](#) to create a single object to access our database tables.

```
library(dm)
lahman_dm <- dm(batting = tbl(con, "Batting"), people = tbl(con, "People"))
lahman_dm
```

```
-- Table source -----
src: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-azure:R 4.4.1/:memory:]
-- Metadata -----
Tables: `batting`, `people`
Columns: 48
Primary keys: 0
Foreign keys: 0
```

Using this approach, we can make use of various utility functions. For example here we specify [primary and foreign keys](#) and then check that the key constraints are satisfied.

```
lahman_dm <- lahman_dm |>
  dm_add_pk(table = "people", columns = "playerID") |>
  dm_add_fk(table = "batting", columns = "playerID", ref_table = "people")
lahman_dm
```

```
-- Table source -----
src: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-azure:R 4.4.1/:memory:]
-- Metadata -----
Tables: `batting`, `people`
Columns: 48
Primary keys: 1
Foreign keys: 1
```

```
dm_examine_constraints(.dm = lahman_dm)
```

```
i All constraints satisfied.
```

For more information on the dm package see <https://dm.cynkra.com/index.html>

4.2 Creating functions for the data model

We can also now make various functions specific to our Lahman data model to facilitate data analyses. Given we know the structure of the data, we can build a set of functions that abstract away some of the complexities of working with data in a database.

Let's start by making a small function to get the teams players have played for. We can see that the code we use follows on from the last couple of chapters.

```
getTeams <- function(lahman, name = "Barry Bonds") {  
  lahman$Batting |>  
    inner_join(  
      lahman$People |>  
        mutate(full_name = paste0(nameFirst, " ", nameLast)) |>  
        filter(full_name %in% name) |>  
        select("playerID"),  
      by = "playerID"  
    ) |>  
    distinct(teamID, yearID) |>  
    left_join(lahman$Teams, by = c("teamID", "yearID")) |>  
    distinct(name)  
}
```

Now we can easily get the different teams a player represented. We can see how changing the player name changes the SQL that is getting run behind the scenes.

```
getTeams(lahman = lahman, name = "Babe Ruth")
```

```
# Source:   SQL [?? x 1]  
# Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-azure:R 4.4.1/:memory:]  
  name  
  <chr>  
1 Boston Braves  
2 Boston Red Sox
```

3 New York Yankees

 Show query

```
<SQL>
SELECT DISTINCT q01.*
FROM (
  SELECT "name"
  FROM (
    SELECT DISTINCT q01.*
    FROM (
      SELECT teamID, yearID
      FROM Batting
      INNER JOIN (
        SELECT playerID
        FROM (
          SELECT People.*, CONCAT_WS(' ', nameFirst, ' ', nameLast) AS full_name
          FROM People
        ) q01
        WHERE (full_name IN ('Babe Ruth'))
      ) RHS
      ON (Batting.playerID = RHS.playerID)
    ) q01
  ) LHS
  LEFT JOIN Teams
    ON (LHS.teamID = Teams.teamID AND LHS.yearID = Teams.yearID)
) q01
```

```
getTeams(lahman = lahman, name = "Barry Bonds")
```

```
# Source:   SQL [?? x 1]
# Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-azure:R 4.4.1/:memory:]
  name
<chr>
1 San Francisco Giants
2 Pittsburgh Pirates
```

 Show query

```
<SQL>
SELECT DISTINCT q01.*
FROM (
  SELECT "name"
  FROM (
    SELECT DISTINCT q01.*
    FROM (
      SELECT teamID, yearID
      FROM Batting
      INNER JOIN (
        SELECT playerID
        FROM (
          SELECT People.*, CONCAT_WS(' ', nameFirst, ' ', nameLast) AS full_name
          FROM People
        ) q01
        WHERE (full_name IN ('Barry Bonds'))
      ) RHS
      ON (Batting.playerID = RHS.playerID)
    ) q01
  ) LHS
  LEFT JOIN Teams
    ON (LHS.teamID = Teams.teamID AND LHS.yearID = Teams.yearID)
) q01
```

Choosing the right time to collect data into R

The function `collect()` brings data out of the database and into R. When working with large datasets, as is often the case when interacting with a database, we typically want to keep as much computation as possible on the database side. In the case of our `getTeams()` function, for example, it does everything on the database side and so collecting will just bring out the result of the teams the person played for. In this case, we could also use `pull()` to get our result out as a vector rather than a data frame.

```
getTeams(lahman = lahman, name = "Barry Bonds") |>
  collect()
```

```
# A tibble: 2 x 1
  name
<chr>
1 San Francisco Giants
```

2 Pittsburgh Pirates

```
getTeams(lahman = lahman, name = "Barry Bonds") |>
  pull()
```

```
[1] "San Francisco Giants" "Pittsburgh Pirates"
```

In other cases however we may need to collect data so as to perform further analysis steps that are not possible using SQL. This might be the case for plotting or for other analytic steps like fitting statistical models. In such cases we should try to only bring out the data that we need (as we will likely have much less memory available on our local computer than is available for the database).

Similarly, we could make a function to add a player's year of birth to a table.

```
addBirthCountry <- function(x){
  x |>
    left_join(
      lahman$People |>
        select("playerID", "birthCountry"),
      by = "playerID"
    )
}
```

```
lahman$Batting |>
  addBirthCountry()
```

```
# Source:   SQL [?? x 23]
# Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-azure:R 4.4.1/:memory:]
  playerID  yearID stint teamID lgID      G    AB    R    H   X2B   X3B   HR
   <chr>      <int> <int> <fct>  <fct> <int> <int> <int> <int> <int> <int> <int>
1 aardsda01  2004     1 SFN    NL      11     0     0     0     0     0     0
2 aardsda01  2006     1 CHN    NL      45     2     0     0     0     0     0
3 aardsda01  2007     1 CHA    AL      25     0     0     0     0     0     0
4 aardsda01  2008     1 BOS    AL      47     1     0     0     0     0     0
5 aardsda01  2009     1 SEA    AL      73     0     0     0     0     0     0
6 aardsda01  2010     1 SEA    AL      53     0     0     0     0     0     0
7 aardsda01  2012     1 NYA    AL       1     0     0     0     0     0     0
8 aardsda01  2013     1 NYN    NL      43     0     0     0     0     0     0
9 aardsda01  2015     1 ATL    NL      33     1     0     0     0     0     0
10 aaronha01 1954     1 ML1    NL     122    468    58    131    27     6    13
```

```
# i more rows
# i 11 more variables: RBI <int>, SB <int>, CS <int>, BB <int>, SO <int>,
#   IBB <int>, HBP <int>, SH <int>, SF <int>, GIDP <int>, birthCountry <chr>
```

i Show query

```
<SQL>
SELECT Batting.*, birthCountry
FROM Batting
LEFT JOIN People
  ON (Batting.playerID = People.playerID)
```

```
lahman$Pitching |>
  addBirthCountry()
```

```
# Source:   SQL [?? x 31]
# Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-azure:R 4.4.1/:memory:]
  playerID  yearID stint teamID lgID      W      L      G      GS      CG      SHO      SV
  <chr>      <int> <int> <fct>  <fct> <int> <int> <int> <int> <int> <int> <int>
1 aardsda01  2004     1 SFN    NL      1      0     11      0      0      0      0
2 aardsda01  2006     1 CHN    NL      3      0     45      0      0      0      0
3 aardsda01  2007     1 CHA    AL      2      1     25      0      0      0      0
4 aardsda01  2008     1 BOS    AL      4      2     47      0      0      0      0
5 aardsda01  2009     1 SEA    AL      3      6     73      0      0      0     38
6 aardsda01  2010     1 SEA    AL      0      6     53      0      0      0     31
7 aardsda01  2012     1 NYA    AL      0      0      1      0      0      0      0
8 aardsda01  2013     1 NYN    NL      2      2     43      0      0      0      0
9 aardsda01  2015     1 ATL    NL      1      1     33      0      0      0      0
10 aasedo01  1977     1 BOS    AL      6      2     13     13      4      2      0
# i more rows
# i 19 more variables: IPouts <int>, H <int>, ER <int>, HR <int>, BB <int>,
#   SO <int>, BAOpp <dbl>, ERA <dbl>, IBB <int>, WP <int>, HBP <int>, BK <int>,
#   BFP <int>, GF <int>, R <int>, SH <int>, SF <int>, GIDP <int>,
#   birthCountry <chr>
```

i Show query

```
<SQL>
SELECT Pitching.*, birthCountry
FROM Pitching
```



```
LEFT JOIN People
  ON (Pitching.playerID = People.playerID)
```

We could then use our `addBirthCountry()` function as part of a larger query to summarise the proportion of players from each country over time (based on their presence in the batting table).

```
plot_data <- lahman$Batting |>
  select("playerID", "yearID") |>
  addBirthCountry() |>
  filter(yearID > 1960) |>
  mutate(birthCountry = case_when(
    birthCountry == "USA" ~ "USA",
    birthCountry == "D.R." ~ "Dominican Republic",
    birthCountry == "Venezuela" ~ "Venezuela",
    birthCountry == "P.R." ~ "Puerto Rico ",
    birthCountry == "Cuba" ~ "Cuba",
    birthCountry == "CAN" ~ "Canada",
    birthCountry == "Mexico" ~ "Mexico",
    .default = "Other"
  )) |>
  group_by(yearID, birthCountry) |>
  summarise(n = n(), .groups = "drop") |>
  group_by(yearID) |>
  mutate(percentage = n / sum(n) * 100) |>
  ungroup() |>
  collect()
```

i Show query

```
<SQL>
SELECT q01.*, (n / SUM(n) OVER (PARTITION BY yearID)) * 100.0 AS percentage
FROM (
  SELECT yearID, birthCountry, COUNT(*) AS n
  FROM (
    SELECT
      playerID,
      yearID,
      CASE
        WHEN (birthCountry = 'USA') THEN 'USA'
        WHEN (birthCountry = 'D.R.') THEN 'Dominican Republic'
        WHEN (birthCountry = 'Venezuela') THEN 'Venezuela'
```

```

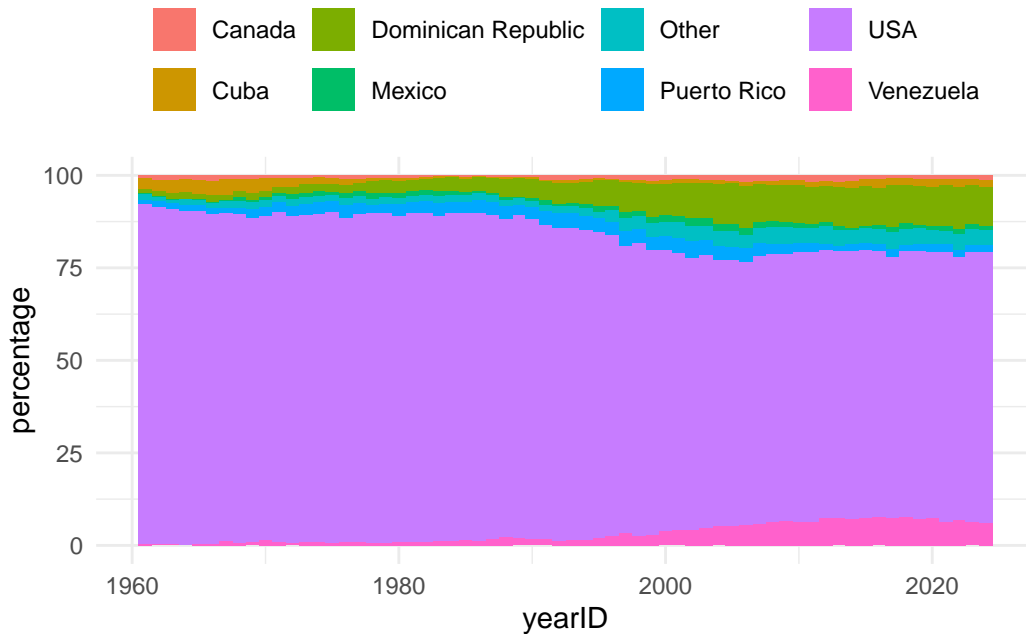
WHEN (birthCountry = 'P.R.') THEN 'Puerto Rico '
WHEN (birthCountry = 'Cuba') THEN 'Cuba'
WHEN (birthCountry = 'CAN') THEN 'Canada'
WHEN (birthCountry = 'Mexico') THEN 'Mexico'
ELSE 'Other'
END AS birthCountry
FROM (
  SELECT Batting.playerID AS playerID, yearID, birthCountry
  FROM Batting
  LEFT JOIN People
    ON (Batting.playerID = People.playerID)
) q01
WHERE (yearID > 1960.0)
) q01
GROUP BY yearID, birthCountry
) q01

```

```

library(ggplot2)
plot_data |>
  ggplot() +
  geom_col(mapping = aes(yearID, percentage, fill = birthCountry), width = 1) +
  theme_minimal() +
  theme(legend.title = element_blank(), legend.position = "top")

```



i Defining methods for the data model

As part of our `lahmanFromCon()` function our data model object has the class “`lahman_ref`”. Therefore as well as creating user-facing functions to work with our lahman data model, we can also define methods for this object.

```
class(lahman)
```

```
[1] "lahman_ref" "list"
```

With this we can make some specific methods for a “`lahman_ref`” object. For example, we can define a print method like so:

```
print.lahman_ref <- function(x, ...) {
  len <- length(names(x))
  cli_h1("# Lahman reference - {len} tables")
  cli_li(paste("{.strong tables:}", paste(names(x), collapse = ", ")))
  invisible(x)
}
```

Now we can see a summary of our lahman data model when we print the object.

```
lahman
```

```
-- # Lahman reference - 28 tables -----

* tables: AllstarFull, Appearances, AwardsManagers, AwardsPlayers,
AwardsManagers, AwardsShareManagers, Batting, BattingPost, CollegePlaying,
Fielding, FieldingOF, FieldingOFsplit, FieldingPost, HallOfFame, HomeGames,
LahmanData, Managers, ManagersHalf, Parks, People, Pitching, PitchingPost,
Salaries, Schools, SeriesPost, Teams, TeamsFranchises, TeamsHalf
```

And we can see that this print is being done by the method we defined.

```
library(sloop)
s3_dispatch(print(lahman))
```

```
=> print.lahman_ref
    print.list
* print.default
```

4.3 Building efficient analytic pipelines

4.3.1 The risk of “clean” R code

Following on from the above approach, we might think it a good idea to make another function `addBirthYear()`. We can then use it along with our `addBirthCountry()` to get a summarised average salary by birth country and birth year.

```
addBirthYear <- function(lahmanTbl){
  lahmanTbl |>
    left_join(
      lahman$People |>
        select("playerID", "birthYear"),
      by = "playerID"
    )
}

lahman$Salaries |>
  addBirthCountry() |>
  addBirthYear() |>
  group_by(birthCountry, birthYear) |>
  summarise(average_salary = mean(salary), .groups = "drop")
```

```
# Source:   SQL [?? x 3]
# Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-azure:R 4.4.1/:memory:]
  birthCountry birthYear average_salary
  <chr>         <int>         <dbl>
1 USA          1981         3064605.
2 USA          1972         2253152.
3 D.R.         1985         1531438.
4 Cuba         1987         4932700.
5 USA          1988         2101223.
6 USA          1987         1947774.
7 USA          1973         2142680.
8 USA          1959         851332.
9 Venezuela    1989         667980.
10 D.R.        1992         513467.
# i more rows
```

Although the R code on the face of it looks fine, when we look at the SQL we can see that our query has two joins to the People table. One join gets information on the birth country and the other on the birth year.

 Show query

```
<SQL>
SELECT birthCountry, birthYear, AVG(salary) AS average_salary
FROM (
  SELECT
    Salaries.*,
    "People...2".birthCountry AS birthCountry,
    "People...3".birthYear AS birthYear
  FROM Salaries
  LEFT JOIN People "People...2"
    ON (Salaries.playerID = "People...2".playerID)
  LEFT JOIN People "People...3"
    ON (Salaries.playerID = "People...3".playerID)
) q01
GROUP BY birthCountry, birthYear
```

To improve performance, we could instead have a single function to get both of these, birth country and birth year, at the same time.

```
addCharacteristics <- function(lahmanTbl){
  lahmanTbl |>
  left_join(
```

```

    lahman$People |>
      select("playerID", "birthYear", "birthCountry"),
      by = "playerID"
    )
  }

lahman$Salaries |>
  addCharacteristics() |>
  group_by(birthCountry, birthYear) |>
  summarise(average_salary = mean(salary), .groups = "drop")

```

```

# Source:   SQL [?? x 3]
# Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-azure:R 4.4.1/:memory:]
  birthCountry birthYear average_salary
    <chr>          <int>          <dbl>
1 USA             1981           3064605.
2 USA             1972           2253152.
3 D.R.            1985           1531438.
4 Cuba            1987           4932700.
5 USA             1988           2101223.
6 USA             1987           1947774.
7 USA             1973           2142680.
8 USA             1959            851332.
9 Venezuela       1989           667980.
10 D.R.           1992           513467.
# i more rows

```

 Show query

```

<SQL>
SELECT birthCountry, birthYear, AVG(salary) AS average_salary
FROM (
  SELECT Salaries.*, birthYear, birthCountry
  FROM Salaries
  LEFT JOIN People
    ON (Salaries.playerID = People.playerID)
) q01
GROUP BY birthCountry, birthYear

```

Now this query outputs the same result but is simpler than the previous one, thus lowering the computational cost of the analysis. All this is to show that when working with databases

we should keep in mind what is going on behind the scenes in terms of the SQL code actually being executed.

4.3.2 Piping and SQL

Although piping functions has little impact on performance when using R with data in memory, when working with a database the SQL generated will differ when using multiple function calls (with a separate operation specified in each) instead of multiple operations within a single function call.

For example, a single mutate function creating two new variables would generate the below SQL.

```
lahman$People |>
  mutate(
    birthDatePlus1 = add_years(x = birthDate, n = 1L),
    birthDatePlus10 = add_years(x = birthDate, n = 10L)
  ) |>
  select("playerID", "birthDatePlus1", "birthDatePlus10") |>
  show_query()
```

```
<SQL>
SELECT
  playerID,
  DATE_ADD(birthDate, INTERVAL (1) year) AS birthDatePlus1,
  DATE_ADD(birthDate, INTERVAL (10) year) AS birthDatePlus10
FROM People
```

Whereas the SQL will be different if these were created using multiple mutate calls (with now one being created in a sub-query).

```
lahman$People |>
  mutate(birthDatePlus1 = add_years(x = birthDate, n = 1L)) |>
  mutate(birthDatePlus10 = add_years(x = birthDate, n = 10L)) |>
  select("playerID", "birthDatePlus1", "birthDatePlus10") |>
  show_query()
```

```
<SQL>
SELECT
  playerID,
  birthDatePlus1,
```

```

    DATE_ADD(birthDate, INTERVAL (10) year) AS birthDatePlus10
FROM (
    SELECT People.*, DATE_ADD(birthDate, INTERVAL (1) year) AS birthDatePlus1
    FROM People
) q01

```

4.3.3 Computing intermediate queries

Let's say we want to summarise home runs in the batting table and strike outs in the pitching table by the college players attended and their birth year. We could do this like so:

```

players_with_college <- lahman$People |>
  select("playerID", "birthYear") |>
  inner_join(
    lahman$CollegePlaying |>
      filter(!is.na(schoolID)) |>
      distinct(playerID, schoolID),
    by = "playerID"
  )

lahman$Batting |>
  left_join(players_with_college, by = "playerID") |>
  group_by(schoolID, birthYear) |>
  summarise(home_runs = sum(H, na.rm = TRUE), .groups = "drop") |>
  collect()

```

```

# A tibble: 6,205 x 3
  schoolID birthYear home_runs
  <chr>      <int>      <dbl>
1 kentucky    1972        157
2 michigan    1967         2
3 texas       1958        10
4 nmstate     1968         0
5 sliprock    1988       624
6 unc         1980       218
7 stanford    1972         55
8 beloitwi    1872         2
9 upenn       1964         0
10 grambling  1942       999
# i 6,195 more rows

```



```
lahman$Pitching |>
  left_join(players_with_college, by = "playerID") |>
  group_by(schoolID, birthYear) |>
  summarise(strike_outs = sum(SO, na.rm = TRUE), .groups = "drop")|>
  collect()
```

```
# A tibble: 3,663 x 3
  schoolID  birthYear strike_outs
  <chr>      <int>      <dbl>
1 rice      1981        340
2 ucsd      1968        124
3 cacerri   1971        327
4 usc       1947        275
5 pepperdine 1969         4
6 lsu       1978        162
7 miamidade 1982         56
8 upperiowa 1918         11
9 jamesmad  1966         4
10 ucla     1984        323
# i 3,653 more rows
```

Looking at the SQL, we can see, however, that there is some duplication, because as part of each full query, we have run our `players_with_college` query.

 Show query

```
<SQL>
SELECT schoolID, birthYear, SUM(H) AS home_runs
FROM (
  SELECT Batting.*, birthYear, schoolID
  FROM Batting
  LEFT JOIN (
    SELECT People.playerID AS playerID, birthYear, schoolID
    FROM People
    INNER JOIN (
      SELECT DISTINCT playerID, schoolID
      FROM CollegePlaying
      WHERE (NOT((schoolID IS NULL)))
    ) RHS
    ON (People.playerID = RHS.playerID)
  ) RHS
)
```

```

        ON (Batting.playerID = RHS.playerID)
    ) q01
GROUP BY schoolID, birthYear

<SQL>
SELECT schoolID, birthYear, SUM(SO) AS strike_outs
FROM (
    SELECT Pitching.*, birthYear, schoolID
    FROM Pitching
    LEFT JOIN (
        SELECT People.playerID AS playerID, birthYear, schoolID
        FROM People
        INNER JOIN (
            SELECT DISTINCT playerID, schoolID
            FROM CollegePlaying
            WHERE (NOT((schoolID IS NULL)))
        ) RHS
        ON (People.playerID = RHS.playerID)
    ) RHS
    ON (Pitching.playerID = RHS.playerID)
) q01
GROUP BY schoolID, birthYear

```

To avoid this we could instead make use of the `compute()` function to force the computation of this first, intermediate, query to a temporary table in the database.

```

players_with_college <- players_with_college |>
  compute()

```

Now we have a temporary table with the result of our `players_with_college` query, and we can use this in both of our aggregation queries.

```

players_with_college |>
  show_query()

```

```

<SQL>
SELECT *
FROM dbplyr_w98K8TSbkP

```

```
lahman$Batting |>
  left_join(players_with_college, by = "playerID") |>
  group_by(schoolID, birthYear) |>
  summarise(home_runs = sum(H, na.rm = TRUE), .groups = "drop") |>
  collect()
```

```
# A tibble: 6,205 x 3
  schoolID birthYear home_runs
  <chr>      <int>      <dbl>
1 vermont      1869         38
2 michigan     1967          2
3 nmstate      1968          0
4 cacerri      1971          3
5 chicago      1874          2
6 byu          1961         28
7 pepperdine   1969          1
8 lsu          1978          2
9 miamidade    1982          0
10 stanford    1961        1611
# i 6,195 more rows
```

```
lahman$Pitching |>
  left_join(players_with_college, by = "playerID") |>
  group_by(schoolID, birthYear) |>
  summarise(strike_outs = sum(SO, na.rm = TRUE), .groups = "drop") |>
  collect()
```

```
# A tibble: 3,663 x 3
  schoolID birthYear strike_outs
  <chr>      <int>      <dbl>
1 vermont      1869        161
2 michigan     1967        888
3 nmstate      1968         98
4 cacerri      1971        327
5 byu          1961       1030
6 pepperdine   1969          4
7 lsu          1978        162
8 miamidade    1982         56
9 stanford     1961          0
10 incante     1893        526
# i 3,653 more rows
```

Show query

```
<SQL>
SELECT schoolID, birthYear, SUM(H) AS home_runs
FROM (
  SELECT Batting.*, birthYear, schoolID
  FROM Batting
  LEFT JOIN dbplyr_w98K8TSbkP
    ON (Batting.playerID = dbplyr_w98K8TSbkP.playerID)
) q01
GROUP BY schoolID, birthYear

<SQL>
SELECT schoolID, birthYear, SUM(SO) AS strike_outs
FROM (
  SELECT Pitching.*, birthYear, schoolID
  FROM Pitching
  LEFT JOIN dbplyr_w98K8TSbkP
    ON (Pitching.playerID = dbplyr_w98K8TSbkP.playerID)
) q01
GROUP BY schoolID, birthYear
```

In this case, the SQL from our initial approach was not so complicated. However, you can imagine that without using computation to intermediate tables, the SQL associated with a series of data manipulations could quickly become unmanageable. Moreover, we can end up with inefficient code that repeatedly gets the same result as part of a larger query. Therefore, although we don't want to overuse computation of intermediate queries, it is often a necessity when creating our analytic pipelines.

Indexes

Some SQL dialects use indexes for more efficient 'joins' performance. Briefly speaking, indexes store the location of the different values of a column. Every time that you create a new table with `compute()`, the indexes won't be carried over, so if you want your new table to keep some indexes, you will have to add them manually. That's why sometimes it won't be more efficient to add a `compute()` in between, because the new table generated won't have the indexes that make your query to be executed faster.

4.4 Disconnecting from the database

Now that we've reached the end of this example, we can close our connection to the database.

```
dbDisconnect(conn = con)
```

Part II

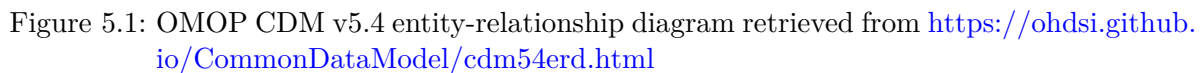
Working with the OMOP CDM from R

In this second half of the book, we will focus on how we can work with data in the OMOP CDM format from R.

- In Chapter 5 we will see how to create a `cdm_reference` in R, a data model that contains references to the OMOP CDM tables and provides the foundation for analysis.
- The OMOP CDM is a person-centric model, and the person and observation period tables are two key tables for any analysis. In Chapter 6 we will see more on how these tables can be used as the starting point for identifying your study participants.
- In Chapter 7 we will see how to add demographics information to different tables of interest and summarise it using `dplyr` code. Finally, we will also see how to use tidyverse verbs to add some custom features.
- In Chapter 8 we will have a look at the cohort object, how it is defined and what are their attributes. We will also see how to create some simple base cohorts and apply some inclusion criteria to them.
- Finally, in Chapter 9 we will learn how to intersect cohorts with one another, extracting counts, presence indicators, specific dates, or time differences to obtain the information of interest for our study population.

5.1 The OMOP CDM layout

OMOP Common Data Model 5.4



5.2 Creating a reference to the OMOP CDM

As we saw in Chapter 4, creating a data model in R to represent the OMOP CDM can provide a basis for analytic pipelines using the data. Luckily for us, we won't have to create functions and methods for this ourselves. Instead, we will use the [omopgenerics](#) package which defines a data model for OMOP CDM data and the [CDMConnector](#) package which provides functions for connecting to OMOP CDM data held in a database.

To see how this works, we will use the [omock](#) package to create example data in the format of the OMOP CDM, which we will then copy to a DuckDB database created by the [duckdb](#) package.

```
library(duckdb)
library(dplyr)
library(omock)
library(CDMConnector)
library(palmerpenguins)

cdm_local <- mockCdmReference() |>
  mockPerson(nPerson = 100) |>
  mockObservationPeriod() |>
  mockConditionOccurrence() |>
  mockDrugExposure() |>
  mockObservation() |>
  mockMeasurement() |>
  mockVisitOccurrence() |>
  mockProcedureOccurrence()

con <- dbConnect(drv = duckdb())
src <- dbSource(con = con, writeSchema = "main")

cdm <- insertCdmTo(cdm = cdm_local, to = src)
```

Note that [insertCdmTo\(\)](#) output is already a `<cdm_reference>` object. But how would we create this cdm reference from the connection? We can use the function [cdmFromCon\(\)](#) from [CDMConnector](#) to create our cdm reference. Note that as well as specifying the schema containing our OMOP CDM tables, we will also specify a write schema where any database tables we create during our analysis will be stored. Often, our OMOP CDM tables will be in a schema that we only have read-access to, and we'll have another schema where we can have write-access and where intermediate tables can be created for a given study.

```
cdm <- cdmFromCon(con = con,
                  cdmSchema = "main",
                  writeSchema = "main",
                  cdmName = "example_data")
```

```
cdm
```

```
-- # OMOP CDM reference (duckdb) of example_data -----
```

```
* omop tables: cdm_source, concept, concept_ancestor, concept_relationship,
concept_synonym, condition_occurrence, drug_exposure, drug_strength,
measurement, observation, observation_period, person, procedure_occurrence,
visit_occurrence, vocabulary
```

```
* cohort tables: -
```

```
* achilles tables: -
```

```
* other tables: -
```

Setting a write prefix

We can also specify a write prefix and this will be used whenever permanent tables are created in the write schema. This can be useful when we're sharing our write schema with others and want to avoid table name conflicts and easily drop tables created as part of a particular study.

```
cdm <- cdmFromCon(con = con,
                  cdmSchema = "main",
                  writeSchema = "main",
                  writePrefix = "my_study_",
                  cdmName = "example_data")
```

Note you only have to specify this writePrefix once at the connection stage, and then the cdm_reference object will store that and use it every time that you create a new table.

We can see that we now have an object that contains references to all the OMOP CDM tables. We can reference specific tables using the “\$” or “[[...]]

```
cdm$person
```

```
# Source:   table<person> [?? x 18]
# Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-azure:R 4.4.1/:memory:]
  person_id gender_concept_id year_of_birth month_of_birth day_of_birth
    <int>         <int>         <int>         <int>         <int>
1         1           8507           1961           4           12
2         2           8507           1996           1           8
3         3           8507           1961           4          19
4         4           8532           1969          11           6
5         5           8507           1964          10          22
6         6           8532           1963           5          18
7         7           8532           1987           4          27
8         8           8532           1958          11          13
9         9           8507           1965          12           9
10        10           8507           1963           2          12
# i more rows
# i 13 more variables: race_concept_id <int>, ethnicity_concept_id <int>,
#   birth_datetime <dtm>, location_id <int>, provider_id <int>,
#   care_site_id <int>, person_source_value <chr>, gender_source_value <chr>,
#   gender_source_concept_id <int>, race_source_value <chr>,
#   race_source_concept_id <int>, ethnicity_source_value <chr>,
#   ethnicity_source_concept_id <int>
```

```
cdm[["observation_period"]]
```

```
# Source:   table<observation_period> [?? x 5]
# Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-azure:R 4.4.1/:memory:]
  observation_period_id person_id observation_period_s~1 observation_period_e~2
          <int>         <int> <date>                <date>
1         1           1      1 1992-12-06            1999-12-13
2         2           2      2 2018-02-20            2019-08-04
3         3           3      3 2015-01-16            2017-11-20
4         4           4      4 1978-09-02            1990-01-15
5         5           5      5 1967-09-21            2000-09-12
6         6           6      6 2017-08-03            2018-09-13
7         7           7      7 2014-03-28            2016-03-25
8         8           8      8 1988-10-28            2007-12-30
9         9           9      9 2006-09-26            2018-10-20
10        10          10     10 2016-08-04            2018-07-14
# i more rows
```

```
# i abbreviated names: 1: observation_period_start_date,
#   2: observation_period_end_date
# i 1 more variable: period_type_concept_id <int>
```

Note that here we have first created a local version of the cdm with all the tables of interest with `omock` (`cdm_local`), then copied it to a DuckDB database, and finally created a reference to it with `CDMConnector`, so that we can work with the final `cdm` object as we normally would for one created with our own healthcare data. In that case, we would directly use `cdmFromCon()` with our own database information. Throughout this chapter, however, we will keep working with the mock dataset.

5.3 CDM attributes

5.3.1 CDM name

Our cdm reference will be associated with a name. By default, this name will be taken from the `cdm_source_name` field from the `cdm_source` table. We will use the function `cdmName()` from `omopgenerics` to get it.

```
cdm <- cdmFromCon(con = con,
                  cdmSchema = "main",
                  writeSchema = "main")
cdm$cdm_source
```

```
# Source:   table<cdm_source> [?? x 10]
# Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-azure:R 4.4.1/:memory:]
  cdm_source_name cdm_source_abbreviation cdm_holder source_description
  <chr>           <chr>                     <chr>      <chr>
1 mock           <NA>                     <NA>      <NA>
# i 6 more variables: source_documentation_reference <chr>,
#   cdm_etl_reference <chr>, source_release_date <date>,
#   cdm_release_date <date>, cdm_version <chr>, vocabulary_version <chr>
```

```
cdmName(cdm)
```

```
[1] "mock"
```

However, we can instead set this name to whatever else we want when creating our cdm reference.

```
cdm <- cdmFromCon(con = con,
                  cdmSchema = "main",
                  writeSchema = "main",
                  cdmName = "my_cdm")

cdmName(cdm)
```

```
[1] "my_cdm"
```

Note that we can also get our cdm name from any of the tables in our cdm reference.

```
cdmName(cdm$person)
```

```
[1] "my_cdm"
```

Behind the scenes

The class of the cdm reference itself is `<cdm_reference>`.

```
class(cdm)
```

```
[1] "cdm_reference"
```

Each of the tables has class `<cdm_table>`. If the table is one of the standard OMOP CDM tables, it will also have class `<omop_table>`. This latter class is defined so that we can allow different behavior for these core tables (person, condition_occurrence, observation_period, etc.) compared to other tables that are added to the cdm reference during the course of running a study.

```
class(cdm$person)
```

```
[1] "omop_table"          "cdm_table"           "tbl_duckdb_connection"
[4] "tbl_dbt"             "tbl_sql"             "tbl_lazy"
[7] "tbl"
```

We can see that `cdmName()` is a generic function, which works for both the cdm reference as a whole and individual tables.

```
library(sloop)
s3_dispatch(cdmName(cdm))
```

```
=> cdmName.cdm_reference
* cdmName.default
```

```
s3_dispatch(cdmName(cdm$person))
```

```
cdmName.omop_table  
=> cdmName.cdm_table  
cdmName.tbl_duckdb_connection  
cdmName.tbl_dbi  
cdmName.tbl_sql  
cdmName.tbl_lazy  
cdmName.tbl  
* cdmName.default
```

5.3.2 CDM version

We can also easily check the OMOP CDM version that is being used with the function `cdmVersion()` from `omopgenerics` like so:

```
cdmVersion(cdm)
```

```
[1] "5.3"
```

cdmVersion

Note, the `cdmVersion()` function also works for `<cdm_table>` objects:

```
cdmVersion(cdm$person)
```

```
[1] "5.3"
```

Methods functions

Although as stated, the `cdmName()` and `cdmVersion()` functions are defined by the `omop-generics` packages, these functions are re-exported in other packages and you won't need to load `omopgenerics` explicitly.

5.4 Including cohort tables in the cdm reference

A `cohort` is a fundamental piece in epidemiological studies. Later, we'll see how to create cohorts in more detail in Chapter 8. For the moment, let's just outline how we can include

the reference to an existing cohort in our cdm reference. For this, we'll use `omock` to add a cohort to our local cdm and upload that to a DuckDB database again.

```
cdm_local <- cdm_local |>
  mockCohort(name = "my_study_cohort")
con <- dbConnect(drv = duckdb())
src <- dbSource(con = con, writeSchema = "main")
cdm <- insertCdmTo(cdm = cdm_local, to = src)
```

Now we can specify we want to include this existing cohort table to our cdm object when creating our cdm reference.

```
cdm <- cdmFromCon(con = con,
  cdmSchema = "main",
  writeSchema = "main",
  cohortTables = "my_study_cohort",
  cdmName = "example_data")
```

```
cdm
```

```
cdm$my_study_cohort |>
  glimpse()
```

```
Rows: ??
```

```
Columns: 4
```

```
Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-azure:R 4.4.1/:memory:]
```

```
$ cohort_definition_id <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1~
```

```
$ subject_id <int> 3, 6, 8, 9, 10, 10, 11, 11, 12, 12, 14, 14, 16, 1~
```

```
$ cohort_start_date <date> 2017-03-25, 2017-08-18, 1996-11-23, 2012-05-29, ~
```

```
$ cohort_end_date <date> 2017-09-02, 2017-09-28, 2002-04-25, 2014-05-08, ~
```

i Tables included in the cdm reference

Note that by default the cohort table won't be included in the `cdm_reference` object.

```
cdm <- cdmFromCon(con = con,
  cdmSchema = "main",
  writeSchema = "main",
  cdmName = "example_data")
```

```
cdm
```

```
-- # OMOP CDM reference (duckdb) of example_data -----

* omop tables: cdm_source, concept, concept_ancestor, concept_relationship,
concept_synonym, condition_occurrence, drug_exposure, drug_strength,
measurement, observation, observation_period, person, procedure_occurrence,
visit_occurrence, vocabulary

* cohort tables: -

* achilles tables: -

* other tables: -
```

Even if the cohort exists in the database:

```
dbListTables(conn = con)
```

```
[1] "cdm_source"           "concept"
[3] "concept_ancestor"     "concept_relationship"
[5] "concept_synonym"      "condition_occurrence"
[7] "drug_exposure"        "drug_strength"
[9] "measurement"          "my_study_cohort"
[11] "my_study_cohort_attrition" "my_study_cohort_codelist"
[13] "my_study_cohort_set"   "observation"
[15] "observation_period"    "person"
[17] "procedure_occurrence"  "visit_occurrence"
[19] "vocabulary"
```

By default, only the default omop tables `omopTables()` will be included (if they exist) into the `cdm_reference` object.

5.5 Including achilles tables in the cdm reference

If we have the results tables from the [Achilles](#) package in our database, we can also include these in our cdm reference.

Just to show how this can be done, let's upload some empty results tables in the Achilles format.


```

dbWriteTable(conn = con,
  name = "achilles_analysis",
  value = tibble(
    analysis_id = NA_integer_,
    analysis_name = NA_character_,
    stratum_1_name = NA_character_,
    stratum_2_name = NA_character_,
    stratum_3_name = NA_character_,
    stratum_4_name = NA_character_,
    stratum_5_name = NA_character_,
    is_default = NA_character_,
    category = NA_character_))

dbWriteTable(conn = con,
  name = "achilles_results",
  value = tibble(
    analysis_id = NA_integer_,
    stratum_1 = NA_character_,
    stratum_2 = NA_character_,
    stratum_3 = NA_character_,
    stratum_4 = NA_character_,
    stratum_5 = NA_character_,
    count_value = NA_character_))

dbWriteTable(conn = con,
  name = "achilles_results_dist",
  value = tibble(
    analysis_id = NA_integer_,
    stratum_1 = NA_character_,
    stratum_2 = NA_character_,
    stratum_3 = NA_character_,
    stratum_4 = NA_character_,
    stratum_5 = NA_character_,
    count_value = NA_character_,
    min_value = NA_character_,
    max_value = NA_character_,
    avg_value = NA_character_,
    stdev_value = NA_character_,
    median_value = NA_character_,
    p10_value = NA_character_,
    p25_value = NA_character_,
    p75_value = NA_character_,
    p90_value = NA_character_))

```

We can now include these achilles tables in our cdm reference as in the previous case.

```
cdm <- cdmFromCon(con = con,
                  cdmSchema = "main",
                  writeSchema = "main",
                  cohortTables = "my_study_cohort",
                  achillesSchema = "main",
                  cdmName = "example_data")

cdm
```

Note we specified the `achillesSchema` that in this case is the same as the `writeSchema` and `cdmSchema`, but each one of them can be different and point to a separate [schema](#) in our database.

5.6 Adding other tables to the cdm reference

Let's say we have some additional **local** data that we want to add to our cdm reference. We can add this both to the same source (in this case a database) and to our cdm reference using `insertTable()` from `omopgenerics` (`insertTable()` is also re-exported in `CDMConnector`). We will show this with the dataset `cars` built-in to R.

```
cars |>
  glimpse()
```

```
Rows: 50
Columns: 2
$ speed <dbl> 4, 4, 7, 7, 8, 9, 10, 10, 10, 11, 11, 12, 12, 12, 12, 13, 13, 13~
$ dist  <dbl> 2, 10, 4, 22, 16, 10, 18, 26, 34, 17, 28, 14, 20, 24, 28, 26, 34~
```

```
cdm <- insertTable(cdm = cdm, name = "cars", table = cars)
```

We can see that now this extra table has been uploaded to the database behind our cdm reference and also added to our reference.

```
cdm
```

```
-- # OMOP CDM reference (duckdb) of example_data -----
```

```

* omop tables: cdm_source, concept, concept_ancestor, concept_relationship,
concept_synonym, condition_occurrence, drug_exposure, drug_strength,
measurement, observation, observation_period, person, procedure_occurrence,
visit_occurrence, vocabulary

* cohort tables: my_study_cohort

* achilles tables: achilles_analysis, achilles_results, achilles_results_dist

* other tables: cars

```

```
cdm$cars
```

```

# Source:   table<cars> [?? x 2]
# Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-azure:R 4.4.1/:memory:]
  speed  dist
  <dbl> <dbl>
1      4     2
2      4    10
3      7     4
4      7    22
5      8    16
6      9    10
7     10    18
8     10    26
9     10    34
10     11    17
# i more rows

```

If we already had the table in the database we could have instead just assigned it to our existing cdm reference. To see this let's upload the `penguins` table to our DuckDB database.

```
dbWriteTable(conn = con, name = "penguins", value = penguins)
```

Once we have this table in the database, we can just read it using the `readSourceTable()` function.

```
cdm <- readSourceTable(cdm = cdm, name = "penguins")

cdm
```

```
-- # OMOP CDM reference (duckdb) of example_data -----
```

```
* omop tables: cdm_source, concept, concept_ancestor, concept_relationship,  
concept_synonym, condition_occurrence, drug_exposure, drug_strength,  
measurement, observation, observation_period, person, procedure_occurrence,  
visit_occurrence, vocabulary
```

```
* cohort tables: my_study_cohort
```

```
* achilles tables: achilles_analysis, achilles_results, achilles_results_dist
```

```
* other tables: cars, penguins
```

Note that omopgenerics provides these functions `readSourceTable()`, `listSourceTables()`, and `dropSourceTable()` for the easier management of the tables in the writeSchema.

```
listSourceTables(cdm = cdm)
```

```
[1] "achilles_analysis"      "achilles_results"  
[3] "achilles_results_dist"  "cars"  
[5] "cdm_source"             "concept"  
[7] "concept_ancestor"       "concept_relationship"  
[9] "concept_synonym"        "condition_occurrence"  
[11] "drug_exposure"          "drug_strength"  
[13] "measurement"            "my_study_cohort"  
[15] "my_study_cohort_attrition" "my_study_cohort_codelist"  
[17] "my_study_cohort_set"     "observation"  
[19] "observation_period"      "penguins"  
[21] "person"                 "procedure_occurrence"  
[23] "visit_occurrence"       "vocabulary"
```

```
dropSourceTable(cdm = cdm, name = "penguins")  
listSourceTables(cdm = cdm)
```

```

[1] "achilles_analysis"      "achilles_results"
[3] "achilles_results_dist"  "cars"
[5] "cdm_source"             "concept"
[7] "concept_ancestor"       "concept_relationship"
[9] "concept_synonym"        "condition_occurrence"
[11] "drug_exposure"          "drug_strength"
[13] "measurement"            "my_study_cohort"
[15] "my_study_cohort_attrition" "my_study_cohort_codelist"
[17] "my_study_cohort_set"     "observation"
[19] "observation_period"     "person"
[21] "procedure_occurrence"   "visit_occurrence"
[23] "vocabulary"

```

i Difference between `insertTable` and `dbWriteTable`

- `dbWriteTable()` is a function from the DBI package that writes a local R data frame to a database. You need to manually specify the schema and table name and it does not update the cdm reference object.
- `insertTable()` is a function from the `omopgenerics` package designed for use with cdm reference objects. It writes a local table to the database and adds it to the list of tables in the cdm reference. Internally, it uses `dbWriteTable()` but also handles the schema and table name automatically using the `writeSchema` and `writePrefix` from the cdm reference.

In general, for studies using OMOP CDM data, you should use `insertTable()` rather than `dbWriteTable()`. It ensures the table is both written to the correct location in the database and accessible through the cdm reference object. Only use `dbWriteTable()` if you are confident working directly with the database and understand its structure.

Note `insertTable()` would also work for a local cdm reference or any other defined cdm reference source, whereas the `dbWriteTable()` is a database specific function.

TODO reference to omopgenerics supported sources.

5.7 Mutability of the cdm reference

An important characteristic of our cdm reference is that we can alter the tables in R, but the OMOP CDM data will not be affected. We will therefore only be transforming the data in our cdm object but the original datasets behind it will remain intact.

For example, let's say we want to perform a study with only people born in 1970. For this we could filter our person table to only people born in this year.

```
cdm$person <- cdm$person |>
  filter(year_of_birth == 1970)

cdm$person
```

```
# Source:   SQL [?? x 18]
# Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-azure:R 4.4.1/:memory:]
  person_id gender_concept_id year_of_birth month_of_birth day_of_birth
      <int>          <int>          <int>          <int>          <int>
1         17         8532         1970             6             7
2         89         8532         1970             1            11
# i 13 more variables: race_concept_id <int>, ethnicity_concept_id <int>,
#   birth_datetime <dtm>, location_id <int>, provider_id <int>,
#   care_site_id <int>, person_source_value <chr>, gender_source_value <chr>,
#   gender_source_concept_id <int>, race_source_value <chr>,
#   race_source_concept_id <int>, ethnicity_source_value <chr>,
#   ethnicity_source_concept_id <int>
```

From now on, when we work with our cdm reference this restriction will continue to have been applied.

```
cdm$person |>
  tally()
```

```
# Source:   SQL [?? x 1]
# Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-azure:R 4.4.1/:memory:]
      n
  <dbl>
1     2
```

The original OMOP CDM data itself however will remain unaffected. We can see that, indeed, if we create our reference again the underlying data is unchanged.

```
cdm <- cdmFromCon(con = con,
                  cdmSchema = "main",
                  writeSchema = "main",
                  cdmName = "example_data")

cdm$person |>
  tally()
```

```
# Source:   SQL [?? x 1]
# Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-azure:R 4.4.1/:memory:]
      n
<dbl>
1    100
```

The mutability of our cdm reference is a useful feature for studies as it means we can easily tweak our OMOP CDM data if needed. Meanwhile, leaving the underlying data unchanged is essential so that other study code can run against the data, unaffected by any of our changes.

One thing we can't do, though, is alter the structure of OMOP CDM tables. For example, the following code would cause an error as the person table must always have the column `person_id`.

```
cdm$person <- cdm$person |>
  rename("new_id" = "person_id")
```

```
Error in `newOmopTable()` :
! person_id is not present in table person
```

In such a case we would have to call the table something else first, and then run the previous code:

```
cdm$person_new <- cdm$person |>
  rename("new_id" = "person_id") |>
  compute(name = "person_new")
```

Now we would be allowed to have this new table as an additional table in our cdm reference, knowing it was not in the format of one of the core OMOP CDM tables.

```
cdm
```

```
-- # OMOP CDM reference (duckdb) of example_data -----

* omop tables: cdm_source, concept, concept_ancestor, concept_relationship,
concept_synonym, condition_occurrence, drug_exposure, drug_strength,
measurement, observation, observation_period, person, procedure_occurrence,
visit_occurrence, vocabulary
```

```
* cohort tables: -  
  
* achilles tables: -  
  
* other tables: person_new
```

The package `omopgenerics` provides a comprehensive list of the required features of a valid cdm reference. You can read more about it [here](#).

i Name consistency

Note also that there must be a name consistency between the name of the table and the assignment in the `cdm_reference` object.

```
cdm$new_table <- cdm$person |>  
  compute(name = "not_new_table")
```

```
Error in `[<-`:  
x You can't assign a table named not_new_table to new_table.  
i You can change the name using compute:  
cdm[['new_table']] <- yourObject |>  
  dplyr::compute(name = 'new_table')  
i You can also change the name using the `name` argument in your function:  
  `name = 'new_table'`.
```

5.8 Working with temporary and permanent tables

When we create new tables and our cdm reference is in a database we have a choice between using temporary or permanent tables. In most cases we can work with these interchangeably. Below we create one temporary table and one permanent table. We can see that both of these tables have been added to our cdm reference and that we can use them in the same way. Note that any new computed table will by default be temporary unless otherwise specified.

```
cdm$person_new_temp <- cdm$person |>  
  head(5) |>  
  compute(temporary = TRUE)
```

```
cdm$person_new_permanent <- cdm$person |>  
  head(5) |>  
  compute(name = "person_new_permanent", temporary = FALSE)
```



```
cdm
```

```
cdm$person_new_temp
```

```
# Source:   table<og_001_1761230506> [?? x 18]
# Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-azure:R 4.4.1/:memory:]
  person_id gender_concept_id year_of_birth month_of_birth day_of_birth
      <int>         <int>         <int>         <int>         <int>
1           1           8507           1961             4            12
2           2           8507           1996             1             8
3           3           8507           1961             4            19
4           4           8532           1969            11             6
5           5           8507           1964            10            22
# i 13 more variables: race_concept_id <int>, ethnicity_concept_id <int>,
#   birth_datetime <dtm>, location_id <int>, provider_id <int>,
#   care_site_id <int>, person_source_value <chr>, gender_source_value <chr>,
#   gender_source_concept_id <int>, race_source_value <chr>,
#   race_source_concept_id <int>, ethnicity_source_value <chr>,
#   ethnicity_source_concept_id <int>
```

```
cdm$person_new_permanent
```

```
# Source:   table<person_new_permanent> [?? x 18]
# Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-azure:R 4.4.1/:memory:]
  person_id gender_concept_id year_of_birth month_of_birth day_of_birth
      <int>         <int>         <int>         <int>         <int>
1           1           8507           1961             4            12
2           2           8507           1996             1             8
3           3           8507           1961             4            19
4           4           8532           1969            11             6
5           5           8507           1964            10            22
# i 13 more variables: race_concept_id <int>, ethnicity_concept_id <int>,
#   birth_datetime <dtm>, location_id <int>, provider_id <int>,
#   care_site_id <int>, person_source_value <chr>, gender_source_value <chr>,
#   gender_source_concept_id <int>, race_source_value <chr>,
#   race_source_concept_id <int>, ethnicity_source_value <chr>,
#   ethnicity_source_concept_id <int>
```

One benefit of working with temporary tables is that they will be automatically dropped at the end of the session, whereas the permanent tables will be left in the database until explicitly dropped. This helps maintain the original database structure tidy and free of irrelevant data.

However, one disadvantage of using temporary tables is that we will generally accumulate more and more of them as we go (in a single R session), whereas we can overwrite permanent tables continuously. For example, if our study code contains a loop that requires a compute, we would either overwrite an intermediate permanent table 100 times or create 100 different temporary tables in the process. In the latter case we should be wary of consuming a lot of drive memory, which could lead to performance issues or even crashes.

i name argument in `compute()`

Note that in the previous examples we explicitly specified the name of the new table and whether it must be temporary or permanent (`temporary = FALSE`), but we do not need to populate the temporary field explicitly as if name is left as NULL (default behavior), then the table will be temporary (`temporary = TRUE`), and if the name argument is populated with a character (e.g., `name = "my_custom_table"`), then the created table will be permanent:

```
cdm$person_new_temp <- cdm$person |>
  compute()

cdm$person_new_permanent <- cdm$person |>
  compute(name = "person_new_permanent")
```

5.9 Disconnecting

Once we have finished our analysis we can close our connection to the database behind our `cdm` reference.

```
cdmDisconnect(cdm)
```

5.10 Further reading

- Català M, Burn E (2025). *omopgenerics: Methods and Classes for the OMOP Common Data Model*. R package version 1.3.1, <https://darwin-eu.github.io/omopgenerics/>.
- Black A, Gorbachev A, Burn E, Català M, Nika I (2025). CDMConnector: Connect to an OMOP Common Data Model. R package version 2.2.0, <https://darwin-eu.github.io/CDMConnector/>.
- OmopOnPostgres (*in progress*)
- OmopOnSpark (*in progress*)
- OmopOnDuckDB (*in progress*)

6 Exploring the OMOP CDM

For this chapter, we'll use a synthetic COVID-19 dataset ("synthea-covid19-10k"). A characterisation of this dataset can be found [here](#).

```
library(dplyr)
library(omock)
library(ggplot2)
library(clock)
library(omopgenerics) # TODO https://github.com/OHDSI/omock/issues/189
```

You can download the dataset using the function `downloadMockDataset()`:

```
downloadMockDataset(datasetName = "synthea-covid19-10k")
```

Setup OMOP_DATA_FOLDER

The `downloadMockDataset` function checks if the database has already been downloaded, and if it is not, it is downloaded in a temporary directory. To avoid downloading the database every time we want to use it, we need to set up the `OMOP_DATA_FOLDER`. To do that, you need to create an environment variable named `OMOP_DATA_FOLDER`. You can add it in the R environment file (`usethis::edit_r_environ()`) or using `Sys.setenv(OMOP_DATA_FOLDER = "...")`. Either way, `OMOP_DATA_FOLDER` should point to a folder where the dataset will be downloaded. This way, the dataset will be stored permanently on your computer and you will not have to download it every time you want to use it.

Note that this folder is defined by `omopgenerics` and is also used by other packages to store OMOP-related data.

Once the dataset is downloaded you can create the cdm reference:

```
cdm <- mockCdmFromDataset(datasetName = "synthea-covid19-10k", source = "duckdb")
```

```
cdm
```

```
-- # OMOP CDM reference (duckdb) of synthea-covid19-10k -----

* omop tables: attribute_definition, care_site, cdm_source, cohort_definition,
concept, concept_ancestor, concept_class, concept_relationship,
concept_synonym, condition_era, condition_occurrence, cost, death,
device_exposure, domain, dose_era, drug_era, drug_exposure, drug_strength,
fact_relationship, location, measurement, metadata, note, note_nlp,
observation, observation_period, payer_plan_period, person,
procedure_occurrence, provider, relationship, source_to_concept_map, specimen,
visit_detail, visit_occurrence, vocabulary

* cohort tables: -

* achilles tables: -

* other tables: -
```

mockCdmFromDataset()

Note that if you call the function without downloading the database first you will be prompted to download it (or it will be downloaded directly if the session is not interactive).

By default, the mock cdm returns a local dataset that can be inserted in any source of interest using the `insertCdmTo()` function. Setting the source to `source = "duckdb"` will return an in-memory DuckDB cdm_reference with two schemas: `cdmSchema = "main"` and `writeSchema = "results"`.

6.1 Counting people

The OMOP CDM is person-centric, with the person table containing records to uniquely identify each person in the database. As each row refers to a unique person, we can quickly get a count of the number of individuals in the database like so

```
cdm$person |>
  count()
```

```
# Source:   SQL [?? x 1]
# Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-azure:R 4.4.1//tmp/RtmpVx6PGh/file21ea4a
      n
<dbl>
1 10754
```

The person table also contains some demographic information, including a gender concept for each person. We can get a count grouped by this variable, but as this uses a concept we'll also need to join to the concept table to get the corresponding concept name for each concept id.

```
cdm$person |>
  group_by(gender_concept_id) |>
  count() |>
  left_join(cdm$concept, by = c("gender_concept_id" = "concept_id")) |>
  select("gender_concept_id", "concept_name", "n") |>
  collect()
```

```
# A tibble: 2 x 3
# Groups:   gender_concept_id [2]
  gender_concept_id concept_name      n
          <int> <chr>          <dbl>
1             8532 FEMALE        5165
2             8507 MALE         5589
```

Vocabulary tables

Above we've got counts by specific concept IDs recorded in the condition occurrence table. What these IDs represent is described in the concept table. Here we have the name associated with the concept, along with other information such as its domain and vocabulary id.

```
cdm$concept |>
  glimpse()
```

```
Rows: ??
Columns: 10
Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-
azure:R 4.4.1//tmp/RtmpVx6PGh/file21ea4ad71eed.duckdb]
$ concept_id      <int> 45756805, 45756804, 45756803, 45756802, 45756801, 457~
$ concept_name    <chr> "Pediatric Cardiology", "Pediatric Anesthesiology", "~
$ domain_id      <chr> "Provider", "Provider", "Provider", "Provider", "Prov~
$ vocabulary_id   <chr> "ABMS", "ABMS", "ABMS", "ABMS", "ABMS", "ABMS", "ABMS~
$ concept_class_id <chr> "Physician Specialty", "Physician Specialty", "Physic~
$ standard_concept <chr> "S", "S", "S", "S", "S", "S", "S", "S", "S", "S", "S"~
$ concept_code     <chr> "OMOP4821938", "OMOP4821939", "OMOP4821940", "OMOP482~
$ valid_start_date <date> 1970-01-01, 1970-01-01, 1970-01-01, 1970-01-
01, 1970~
```

```
$ valid_end_date    <date> 2099-12-31, 2099-12-31, 2099-12-31, 2099-12-31, 2099~
$ invalid_reason    <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N~
```

Other vocabulary tables capture other information about concepts, such as the direct relationships between concepts (the concept relationship table) and hierarchical relationships between (the concept ancestor table).

```
cdm$concept_relationship |>
  glimpse()
```

```
Rows: ??
Columns: 6
Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-
azure:R 4.4.1//tmp/RtmpVx6PGh/file21ea4ad71eed.duckdb]
$ concept_id_1      <int> 35804314, 35804314, 35804314, 35804327, 35804327, 358~
$ concept_id_2      <int> 912065, 42542145, 42542145, 35803584, 42542145, 42542~
$ relationship_id   <chr> "Has modality", "Has accepted use", "Is current in", ~
$ valid_start_date  <date> 2021-01-26, 2019-08-29, 2019-08-29, 2019-05-27, 2019~
$ valid_end_date    <date> 2099-12-31, 2099-12-31, 2099-12-31, 2099-12-31, 2099~
$ invalid_reason    <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N~
```

```
cdm$concept_ancestor |>
  glimpse()
```

```
Rows: ??
Columns: 4
Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-
azure:R 4.4.1//tmp/RtmpVx6PGh/file21ea4ad71eed.duckdb]
$ ancestor_concept_id <int> 375415, 727760, 735979, 438112, 529411, 14196~
$ descendant_concept_id <int> 4335743, 2056453, 41070383, 36566114, 4326940~
$ min_levels_of_separation <int> 4, 1, 3, 2, 3, 3, 4, 3, 2, 5, 1, 3, 4, 2, 2, ~
$ max_levels_of_separation <int> 4, 1, 5, 3, 3, 6, 12, 3, 2, 10, 1, 3, 4, 2, 2~
```

More information on the vocabulary tables (as well as other tables in the OMOP CDM version 5.3) can be found at https://ohdsi.github.io/CommonDataModel/cdm53.html#Vocabulary_Tables.

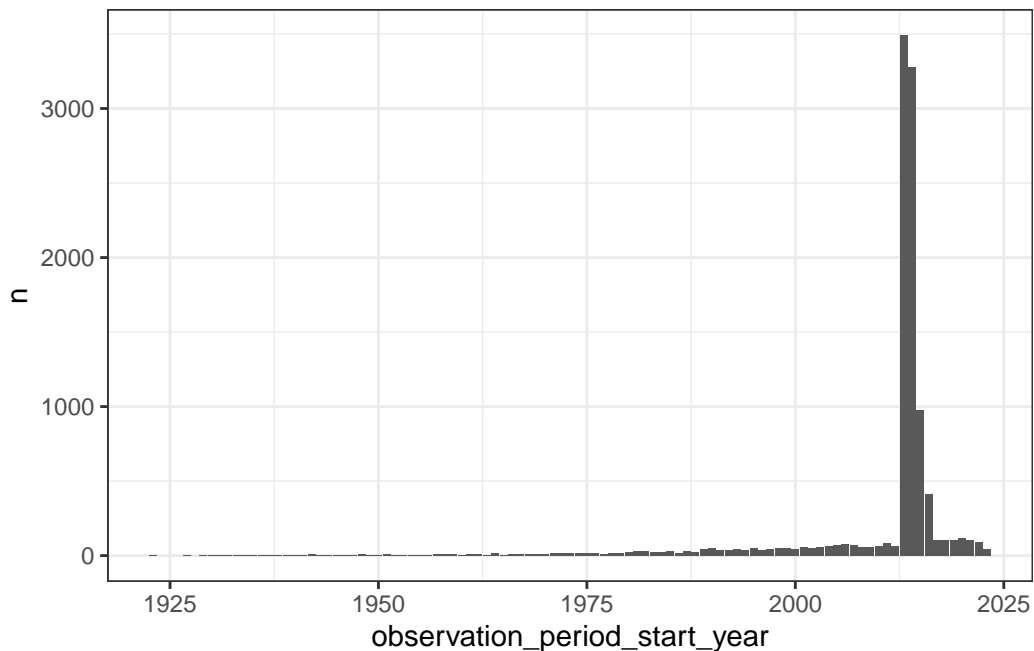
6.2 Summarising observation periods

The observation period table contains records indicating spans of time over which clinical events can be reliably observed for the people in the person table. Someone can potentially have multiple observation periods. So, say we wanted a count of people grouped by the year during which their first observation period started. We could do this like so:

```
first_observation_period <- cdm$observation_period |>
  group_by(person_id) |>
  arrange(observation_period_start_date) |>
  filter(row_number() == 1) |>
  compute()

first_records_per_year <- cdm$person |>
  left_join(first_observation_period, by = "person_id") |>
  mutate(observation_period_start_year = get_year(observation_period_start_date)) |>
  group_by(observation_period_start_year) |>
  count() |>
  collect()

ggplot(first_records_per_year) +
  geom_col(mapping = aes(x = observation_period_start_year, y = n)) +
  theme_bw()
```

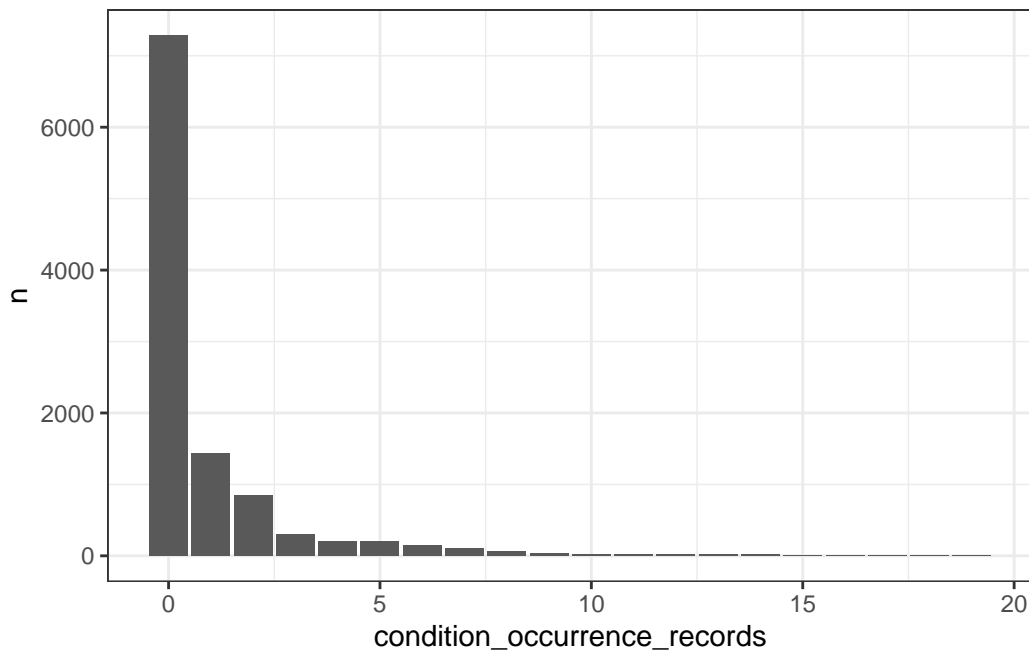


6.3 Summarising clinical records

What's the number of condition occurrence records per person in the database? We can find this out like so

```
number_condition_occurrence_records <- cdm$person |>
  left_join(
    cdm$condition_occurrence |>
      group_by(person_id) |>
      count(name = "condition_occurrence_records"),
    by="person_id"
  ) |>
  mutate(condition_occurrence_records = coalesce(condition_occurrence_records, 0)) |>
  group_by(condition_occurrence_records) |>
  count() |>
  collect()

ggplot(number_condition_occurrence_records) +
  geom_col(mapping = aes(x = condition_occurrence_records, y = n)) +
  theme_bw()
```



How about we were interested in getting record counts for some specific concepts related to Covid-19 symptoms?

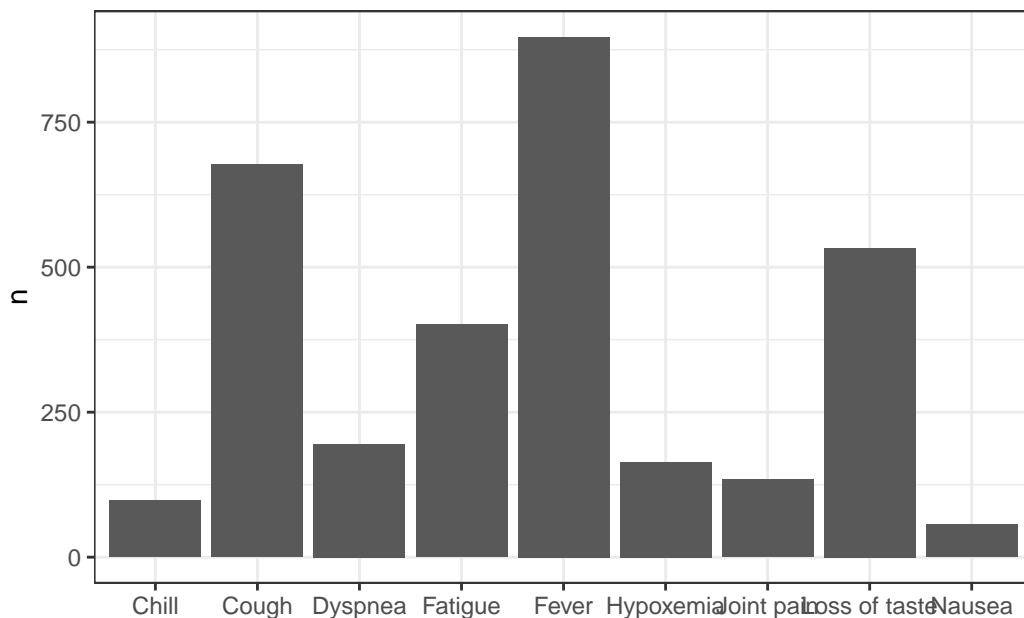

```

symptoms_records <- cdm$condition_occurrence |>
  filter(condition_concept_id %in% c(437663, 437390, 31967, 4289517, 4223659,
                                     312437, 434490, 254761, 77074)) |>

  group_by(condition_concept_id) |>
  count() |>
  left_join(cdm$concept, by = c("condition_concept_id" = "concept_id")) |>
  collect()

ggplot(symptoms_records) +
  geom_col(mapping = aes(x = concept_name, y = n)) +
  theme_bw()+
  xlab("")

```



We can also use summarise for various other calculations

```

cdm$person |>
  summarise(min_year_of_birth = min(year_of_birth, na.rm=TRUE),
            q05_year_of_birth = quantile(year_of_birth, 0.05, na.rm=TRUE),
            mean_year_of_birth = round(mean(year_of_birth, na.rm=TRUE),0),
            median_year_of_birth = median(year_of_birth, na.rm=TRUE),
            q95_year_of_birth = quantile(year_of_birth, 0.95, na.rm=TRUE),
            max_year_of_birth = max(year_of_birth, na.rm=TRUE)) |>
  glimpse()

```

Rows: ??

Columns: 6

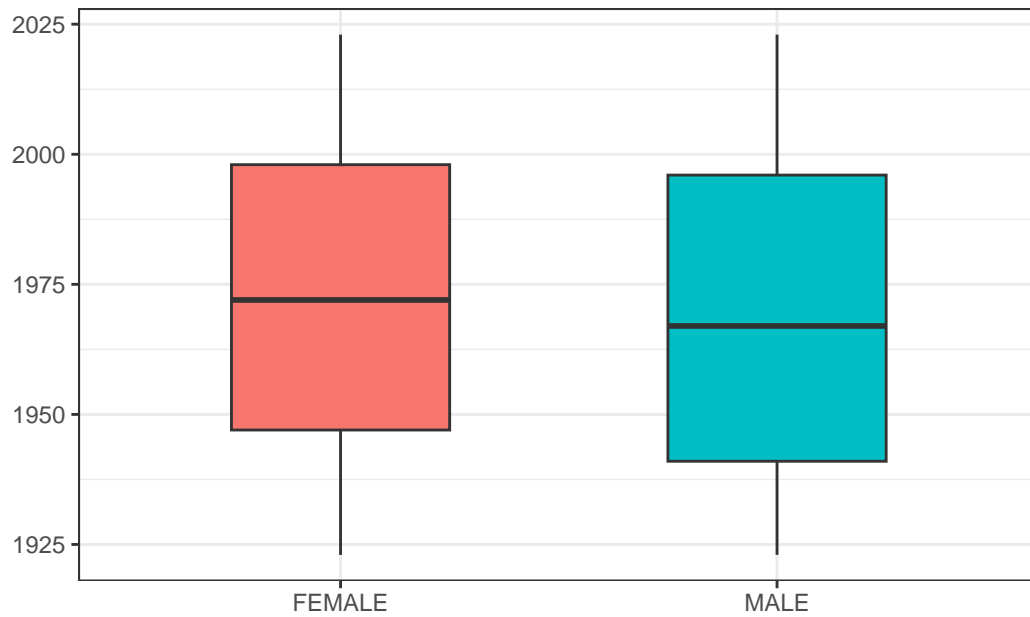
Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-azure:R 4.4.1//tmp/RtmpVx6PGh/file21ea4ad7

```
$ min_year_of_birth    <int> 1923
$ q05_year_of_birth    <dbl> 1927
$ mean_year_of_birth   <dbl> 1971
$ median_year_of_birth <dbl> 1970
$ q95_year_of_birth    <dbl> 2018
$ max_year_of_birth    <int> 2023
```

As we've seen before, we can also quickly get results for various groupings or restrictions

```
grouped_summary <- cdm$person |>
  group_by(gender_concept_id) |>
  summarise(min_year_of_birth = min(year_of_birth, na.rm=TRUE),
            q25_year_of_birth = quantile(year_of_birth, 0.25, na.rm=TRUE),
            median_year_of_birth = median(year_of_birth, na.rm=TRUE),
            q75_year_of_birth = quantile(year_of_birth, 0.75, na.rm=TRUE),
            max_year_of_birth = max(year_of_birth, na.rm=TRUE)) |>
  left_join(cdm$concept, by = c("gender_concept_id" = "concept_id")) |>
  collect()

grouped_summary |>
  ggplot(mapping = aes(x = concept_name, group = concept_name, fill = concept_name)) +
  geom_boxplot(mapping = aes(
    lower = q25_year_of_birth,
    upper = q75_year_of_birth,
    middle = median_year_of_birth,
    ymin = min_year_of_birth,
    ymax = max_year_of_birth),
    stat = "identity", width = 0.5) +
  theme_bw() +
  theme(legend.position = "none") +
  xlab("")
```



6.4 Disconnecting

Once we have finished our analysis we can close our connection to the database behind our cdm reference.

```
cdmDisconnect(cdm)
```

7 Identifying patient characteristics

For this chapter, we'll again use our example COVID-19 dataset.

```
library(dplyr)
library(omock)
library(PatientProfiles)
library(ggplot2)
library(omopgenerics) # TODO https://github.com/OHDSI/omock/issues/189

cdm <- mockCdmFromDataset(datasetName = "synthea-covid19-10k", source = "duckdb")
```

```
i Reading synthea-covid19-10k tables.
i Adding drug_strength table.
```

As part of an analysis, we almost always have a need to identify certain characteristics related to the individuals in our data. These characteristics might be time-invariant (i.e. a characteristic that does not change as time passes and a person ages) or time-varying.¹

7.1 Adding specific demographics

The `PatientProfiles` package makes it easy for us to add demographic information to tables in the OMOP CDM. Like the `CDMConnector` package we've seen previously, the fact that the structure of the OMOP CDM is known allows the `PatientProfiles` package to abstract away some common data manipulations required to do research with patient-level data.²

Let's say we're interested in individuals' age and sex at the time of COVID-19 diagnosis. We can add these variables to the table as follows (noting that, since age is time-varying, we need to specify the date relative to which it should be calculated).

¹In some datasets, characteristics that could conceptually be considered as time-varying are encoded as time-invariant. One example of the latter is that in some cases an individual may be associated with a particular socioeconomic status or nationality that for the purposes of the data is treated as time-invariant.

²Although these manipulations can seem quite simple on the face of it, their implementation across different database platforms with different data granularity (for example, whether day of birth has been filled in for all patients or not) presents challenges that the `PatientProfiles` package solves for us.

```

cdm$condition_occurrence <- cdm$condition_occurrence |>
  addSex() |>
  addAge(indexDate = "condition_start_date")

cdm$condition_occurrence |>
  glimpse()

```

Rows: ??

Columns: 18

Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-azure:R 4.4.1//tmp/RtmpsojHNg/file22512dbc]

```

$ condition_occurrence_id    <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 1~
$ person_id                  <int> 2, 6, 7, 8, 8, 8, 8, 16, 16, 18, 18, 25,~
$ condition_concept_id       <int> 381316, 321042, 381316, 37311061, 437663~
$ condition_start_date       <date> 1986-09-08, 2021-06-23, 2021-04-~
07, 202~
$ condition_start_datetime   <dtm> 1986-09-08, 2021-06-23, 2021-04-~
07, 202~
$ condition_end_date         <date> 1986-09-08, 2021-06-23, 2021-04-~
07, 202~
$ condition_end_datetime     <dtm> 1986-09-08, 2021-06-23, 2021-04-~
07, 202~
$ condition_type_concept_id  <int> 38000175, 38000175, 38000175, 38000175, ~
$ condition_status_concept_id <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0~
$ stop_reason                 <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, ~
$ provider_id                 <int> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, ~
$ visit_occurrence_id        <int> 19, 55, 67, 79, 79, 79, 79, 168, 171, 19~
$ visit_detail_id            <int> 1000019, 1000055, 1000067, 1000079, 1000~
$ condition_source_value     <chr> "230690007", "410429000", "230690007", "~
$ condition_source_concept_id <int> 381316, 321042, 381316, 37311061, 437663~
$ condition_status_source_value <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, ~
$ sex                         <chr> "Female", "Male", "Male", "Male", "Male"~
$ age                         <int> 57, 25, 97, 2, 2, 2, 2, 75, 77, 57, 76, ~

```

We have now added two variables containing values for age and sex.

```

cdm$condition_occurrence |>
  glimpse()

```

Rows: ??

Columns: 18

Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-azure:R 4.4.1//tmp/RtmpsojHNg/file22512dbc]

```

$ condition_occurrence_id    <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 1~
$ person_id                  <int> 2, 6, 7, 8, 8, 8, 8, 16, 16, 18, 18, 25,~
$ condition_concept_id       <int> 381316, 321042, 381316, 37311061, 437663~
$ condition_start_date       <date> 1986-09-08, 2021-06-23, 2021-04-
07, 202~
$ condition_start_datetime   <dtm> 1986-09-08, 2021-06-23, 2021-04-
07, 202~
$ condition_end_date         <date> 1986-09-08, 2021-06-23, 2021-04-
07, 202~
$ condition_end_datetime     <dtm> 1986-09-08, 2021-06-23, 2021-04-
07, 202~
$ condition_type_concept_id  <int> 38000175, 38000175, 38000175, 38000175, ~
$ condition_status_concept_id <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0~
$ stop_reason                 <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, ~
$ provider_id                 <int> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, ~
$ visit_occurrence_id        <int> 19, 55, 67, 79, 79, 79, 79, 168, 171, 19~
$ visit_detail_id            <int> 1000019, 1000055, 1000067, 1000079, 1000~
$ condition_source_value      <chr> "230690007", "410429000", "230690007", "~
$ condition_source_concept_id <int> 381316, 321042, 381316, 37311061, 437663~
$ condition_status_source_value <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, ~
$ sex                         <chr> "Female", "Male", "Male", "Male", "Male"~
$ age                         <int> 57, 25, 97, 2, 2, 2, 2, 75, 77, 57, 76, ~

```

With these now added, it is straightforward to calculate the mean age at condition start date by sex or even plot the distribution of age at diagnosis by sex.

```

cdm$condition_occurrence |>
  group_by(sex) |>
  summarise(mean_age = mean(age, na.rm=TRUE)) |>
  collect()

```

```

# A tibble: 2 x 2
  sex    mean_age
<chr>    <dbl>
1 Female    50.8
2 Male      56.5

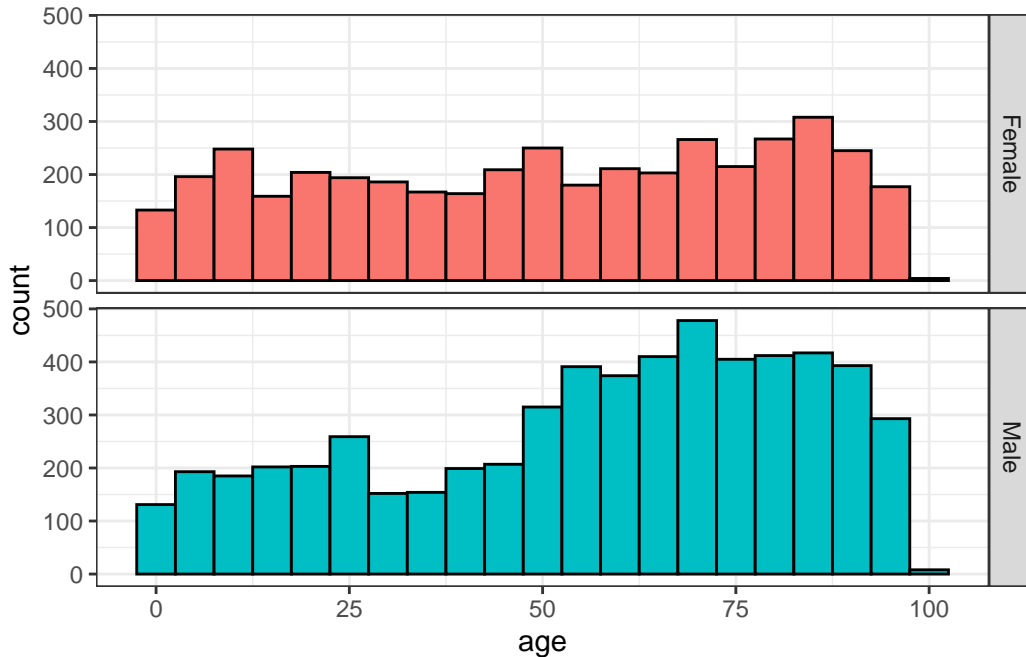
```

```

cdm$condition_occurrence |>
  select("person_id", "age", "sex") |>
  collect() |>
  ggplot(aes(fill = sex)) +

```

```
facet_grid(sex ~ .) +
geom_histogram(aes(age), colour = "black", binwidth = 5) +
theme_bw() +
theme(legend.position = "none")
```



[i](#) Show query

```
cdm$condition_occurrence |>
  addSexQuery() |>
  show_query()
```

Warning: ! The following columns will be overwritten: sex

```
<SQL>
SELECT
  condition_occurrence_id,
  og_002_1761230688.person_id AS person_id,
  condition_concept_id,
  condition_start_date,
  condition_start_datetime,
  condition_end_date,
  condition_end_datetime,
```

```

condition_type_concept_id,
condition_status_concept_id,
stop_reason,
provider_id,
visit_occurrence_id,
visit_detail_id,
condition_source_value,
condition_source_concept_id,
condition_status_source_value,
age,
RHS.sex AS sex
FROM og_002_1761230688
LEFT JOIN (
  SELECT
    person_id,
    CASE
  WHEN (gender_concept_id = 8507.0) THEN 'Male'
  WHEN (gender_concept_id = 8532.0) THEN 'Female'
  ELSE 'None'
  END AS sex
  FROM person
) RHS
ON (og_002_1761230688.person_id = RHS.person_id)

```

The difference between [addSexQuery\(\)](#) and [addSex\(\)](#) is explained in the next tip chunk.

7.2 Adding multiple demographics simultaneously

We've now seen individual functions from [PatientProfiles](#) that add specific patient characteristics such as age and sex. The package also includes functions to add other characteristics, such as the number of days of prior observation in the database (rather unimaginatively named [addPriorObservation\(\)](#)). In addition to these individual functions, the package also provides a more general function that retrieves all of these characteristics at the same time.³

```

cdm$drug_exposure <- cdm$drug_exposure |>
  addDemographics(
    indexDate = "drug_exposure_start_date",

```

³This function also provides a more time-efficient method than getting the characteristics one by one. This is because these characteristics are all derived from the OMOP CDM person and observation period tables, and so can be identified simultaneously.


```

    age = TRUE,
    sex = TRUE,
    priorObservation = TRUE,
    futureObservation = TRUE,
    dateOfBirth = TRUE
  )

cdm$drug_exposure |>
  glimpse()

```

Rows: ??

Columns: 28

Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-azure:R 4.4.1//tmp/RtmpsojHNg/file22512dbc]

```

$ drug_exposure_id      <int> 245761, 245762, 245763, 245764, 245765, 2~
$ person_id            <int> 7764, 7764, 7764, 7764, 7764, 7764, 7764, ~
$ drug_concept_id       <int> 40213227, 40213201, 40213198, 40213154, 4~
$ drug_exposure_start_date <date> 2015-02-08, 2010-01-10, 2010-01-10, 2017~
$ drug_exposure_start_datetime <dtm> 2015-02-08 22:40:04, 2010-01-10 22:40:04~
$ drug_exposure_end_date <date> 2015-02-08, 2010-01-10, 2010-01-10, 2017~
$ drug_exposure_end_datetime <dtm> 2015-02-08 22:40:04, 2010-01-10 22:40:04~
$ verbatim_end_date     <date> 2015-02-08, 2010-01-10, 2010-01-10, 2017~
$ drug_type_concept_id  <int> 32869, 32869, 32869, 32869, 32869, 32869, ~
$ stop_reason           <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N~
$ refills               <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ~
$ quantity              <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ~
$ days_supply           <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ~
$ sig                   <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N~
$ route_concept_id      <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ~
$ lot_number            <chr> "0", "0", "0", "0", "0", "0", "0", "0", "0", "0", ~
$ provider_id           <int> 14656, 14656, 14656, 14656, 14656, 14656, ~
$ visit_occurrence_id   <int> 80896, 80891, 80891, 80892, 80895, 80896, ~
$ visit_detail_id       <int> 1080896, 1080891, 1080891, 1080892, 10808~
$ drug_source_value     <chr> "113", "33", "133", "140", "140", "140", ~
$ drug_source_concept_id <int> 40213227, 40213201, 40213198, 40213154, 4~
$ route_source_value    <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N~
$ dose_unit_source_value <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, N~
$ age                   <int> 71, 66, 66, 73, 72, 71, 69, 67, 65, 70, 6~
$ sex                   <chr> "Male", "Male", "Male", "Male", "Male", "~
$ prior_observation     <int> 2597, 742, 742, 3339, 2968, 2597, 1855, 1~
$ future_observation    <int> 896, 2751, 2751, 154, 525, 896, 1638, 238~
$ date_of_birth         <date> 1943-10-10, 1943-10-10, 1943-10-10, 1943~

```

With these characteristics added, we can now calculate mean age, prior observation (the number of days have passed since each individual's most recent observation start date), and future observation (the number of days until the individual's nearest observation end date) at drug exposure start date, stratified by sex.

```
cdm$drug_exposure |>
  group_by(sex) |>
  summarise(mean_age = mean(age, na.rm=TRUE),
            mean_prior_observation = mean(prior_observation, na.rm=TRUE),
            mean_future_observation = mean(future_observation, na.rm=TRUE)) |>
  collect()
```

```
# A tibble: 2 x 4
  sex      mean_age mean_prior_observation mean_future_observation
<chr>    <dbl>          <dbl>          <dbl>
1 Male      43.0            2455.            1768.
2 Female    39.4            2096.            1661.
```

💡 Returning a query from [PatientProfiles](#) rather than the result

In the above examples, the functions from [PatientProfiles](#) execute queries and write the results to a table in the database (either a temporary table if no name is provided when calling the function, or a permanent table). We might instead want to just get the underlying query back so that we have more control over how and when the query is executed.

```
cdm$visit_occurrence |>
  addSex() |>
  filter(sex == "Male") |>
  show_query()
```

```
<SQL>
SELECT og_004_1761230690.*
FROM og_004_1761230690
WHERE (sex = 'Male')
```

```
cdm$visit_occurrence |>
  addSex(name = "my_new_table") |>
  filter(sex == "Male") |>
  show_query()
```

```
<SQL>
```

```
SELECT test_my_new_table.*
FROM results.test_my_new_table
WHERE (sex = 'Male')
```

```
cdm$visit_occurrence |>
  addSexQuery() |>
  filter(sex == "Male") |>
  show_query()
```

```
<SQL>
SELECT q01.*
FROM (
  SELECT visit_occurrence.*, sex
  FROM visit_occurrence
  LEFT JOIN (
    SELECT
      person_id,
      CASE
        WHEN (gender_concept_id = 8507.0) THEN 'Male'
        WHEN (gender_concept_id = 8532.0) THEN 'Female'
        ELSE 'None'
      END AS sex
    FROM person
  ) RHS
  ON (visit_occurrence.person_id = RHS.person_id)
) q01
WHERE (sex = 'Male')
```

Query functions can be useful in some contexts where you don't want to generate multiple temporary tables or do not want to lose indexes of a certain table, but they can also generate large queries that could result in low performance.

7.3 Creating categories

When adding age, either via [addAge](#) or [addDemographics](#), we can also include an additional variable that groups individuals into age categories. These age groups must be specified in a list of vectors, each of which containing the lower and upper bounds.

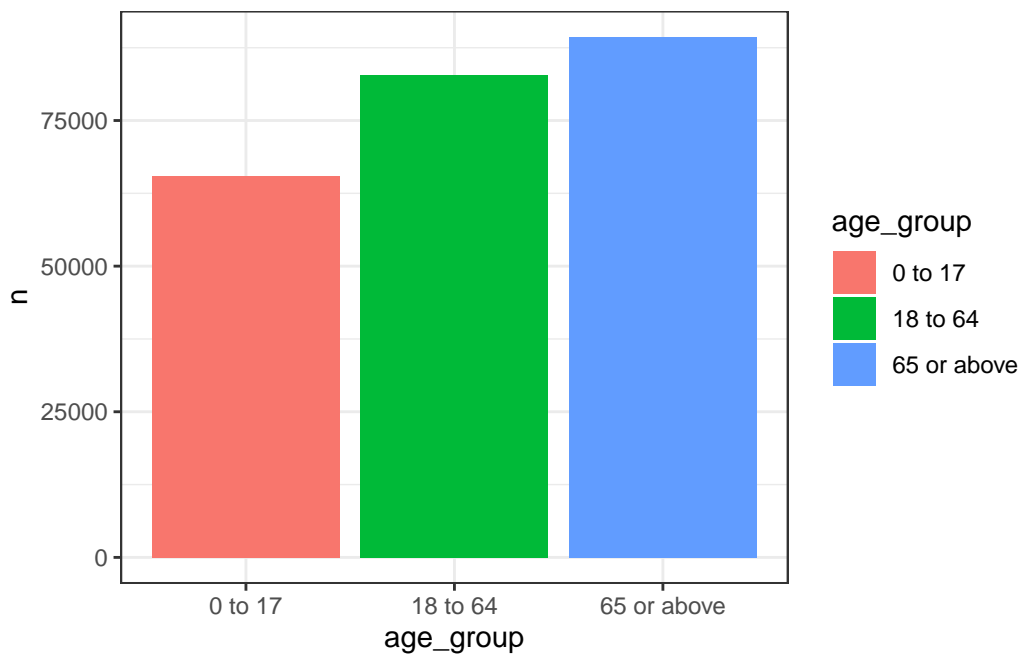
```
cdm$visit_occurrence <- cdm$visit_occurrence |>
  addAge(
```

```

    indexDate = "visit_start_date",
    ageGroup = list(c(0,17), c(18, 64), c(65, Inf))
  )

cdm$visit_occurrence |>
  # data quality issues with our synthetic data means we have
  # some negative ages so will drop these
  filter(age >= 0) |>
  group_by(age_group) |>
  tally() |>
  collect() |>
  ggplot() +
  geom_col(aes(x = age_group, y = n, fill = age_group)) +
  theme_bw()

```



💡 Naming age groups

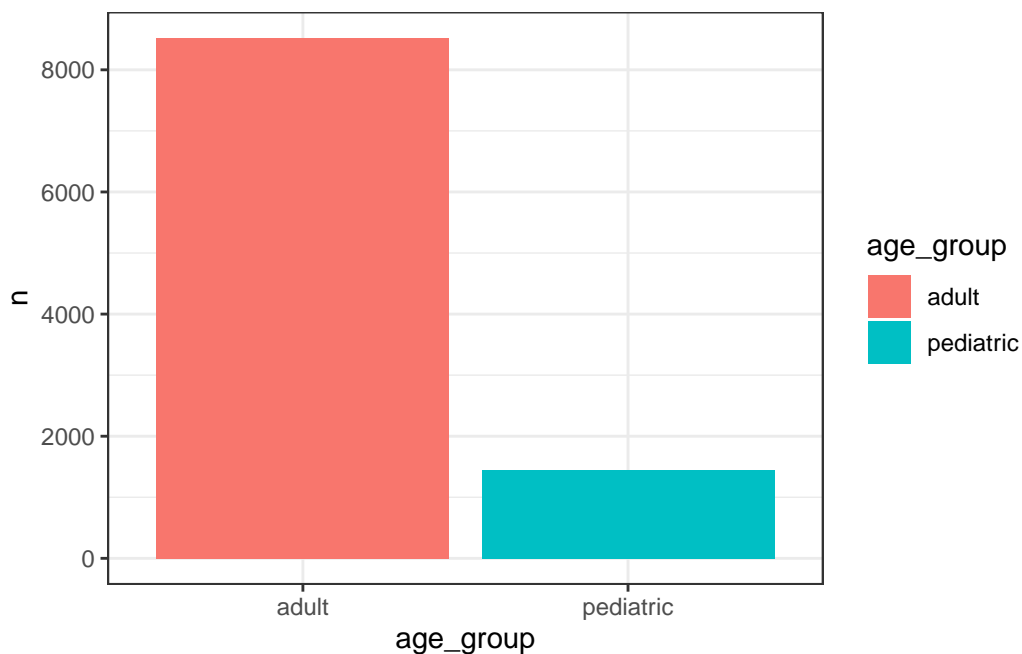
As we have seen, by default the age groups are named according to their lower and upper bounds ('0 to 17', '18 to 64', and '65 or above'). However, we can customise these labels by assigning names to the list of age groups:

```

cdm$condition_occurrence |>
  addAgeQuery(
    indexDate = "condition_start_date",
    ageGroup = list("pediatric" = c(0,17), "adult" = c(18, Inf))
  ) |>
  filter(age >= 0) |>
  group_by(age_group) |>
  tally() |>
  collect() |>
  ggplot() +
  geom_col(aes(x = age_group, y = n, fill = age_group)) +
  theme_bw()

```

Warning: ! The following columns will be overwritten: age



If you take a close look at the documentation of the function, you'll see that it also allows you to add multiple age groups and to control the name of the new column, which by default is 'age_group'.

[PatientProfiles](#) also provides a more general function for adding categories. Can you guess its name? That's right, we have `addCategories()` for this.

```

cdm$condition_occurrence |>
  addPriorObservation(indexDate = "condition_start_date") |>
  addCategories(
    variable = "prior_observation",
    categories = list("prior_observation_group" = list(
      c(0, 364), c(365, Inf)
    ))
  ) |>
  glimpse()

```

Rows: ??

Columns: 20

Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-azure:R 4.4.1//tmp/RtmpsojHNg/file22512dbc]

\$ condition_occurrence_id	<int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 1~
\$ person_id	<int> 2, 6, 7, 8, 8, 8, 8, 16, 16, 18, 18, 25,~
\$ condition_concept_id	<int> 381316, 321042, 381316, 37311061, 437663~
\$ condition_start_date	<date> 1986-09-08, 2021-06-23, 2021-04-07, 202~
\$ condition_start_datetime	<dtm> 1986-09-08, 2021-06-23, 2021-04-07, 202~
\$ condition_end_date	<date> 1986-09-08, 2021-06-23, 2021-04-07, 202~
\$ condition_end_datetime	<dtm> 1986-09-08, 2021-06-23, 2021-04-07, 202~
\$ condition_type_concept_id	<int> 38000175, 38000175, 38000175, 38000175, ~
\$ condition_status_concept_id	<int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0~
\$ stop_reason	<chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, ~
\$ provider_id	<int> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, ~
\$ visit_occurrence_id	<int> 19, 55, 67, 79, 79, 79, 79, 168, 171, 19~
\$ visit_detail_id	<int> 1000019, 1000055, 1000067, 1000079, 1000~
\$ condition_source_value	<chr> "230690007", "410429000", "230690007", "~
\$ condition_source_concept_id	<int> 381316, 321042, 381316, 37311061, 437663~
\$ condition_status_source_value	<chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, ~
\$ sex	<chr> "Female", "Male", "Male", "Male", "Male"~
\$ age	<int> 57, 25, 97, 2, 2, 2, 2, 75, 77, 57, 76, ~
\$ prior_observation	<int> 3437, 2898, 2842, 872, 872, 872, 872, 23~
\$ prior_observation_group	<chr> "365 or above", "365 or above", "365 or ~

7.4 Adding custom variables

While `PatientProfiles` provides a range of functions that can help you add characteristics of interest, you may also want to add other features. Obviously, we can't cover here all possible custom characteristics you may wish to add. However, two common groups of custom features are those that are derived from other variables in the same table and others that are taken from other tables and joined to our particular table of interest.

In the first case, where we want to add a new variable derived from existing variables within our table, we'll typically use `mutate()` (from the `dplyr` package). For example, perhaps we just want to add a new variable to our observation period table that contains the year of each individual's observation period start date. This is rather straightforward.

```
cdm$observation_period <- cdm$observation_period |>
  mutate(observation_period_start_year = get_year(observation_period_start_date))

cdm$observation_period |>
  glimpse()
```

Rows: ??

Columns: 6

Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-azure:R 4.4.1//tmp/RtmpsojHNg/file22512dbc]

\$ observation_period_id <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 1~

\$ person_id <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 1~

\$ observation_period_start_date <date> 2014-05-09, 1977-04-11, 2014-04-19, 201~

\$ observation_period_end_date <date> 2023-05-12, 1986-09-15, 2023-04-22, 202~

\$ period_type_concept_id <int> 44814724, 44814724, 44814724, 44814724, ~

\$ observation_period_start_year <dbl> 2014, 1977, 2014, 2014, 2013, 2013, 2013~

The second case is usually a more complex task, as adding a new variable involves joining to some other table following a certain logic. This table may have been created by some intermediate query that we wrote to derive the variable of interest. For example, let's say we want to add the number of condition occurrence records for each individual to the person table (remember that we saw how to calculate this in the previous chapter). To do this, we will need to perform a join between the person and condition occurrence tables (as some people might not have any records in the condition occurrence table). Here we'll create a table containing just the information we're interested in and compute it to a temporary table.

```

condition_summary <- cdm$person |>
  select("person_id") |>
  left_join(
    cdm$condition_occurrence |>
      group_by(person_id) |>
      count(name = "condition_occurrence_records"),
    by="person_id"
  ) |>
  select("person_id", "condition_occurrence_records") |>
  mutate(condition_occurrence_records = coalesce(condition_occurrence_records, 0)) |>
  compute()

condition_summary |>
  glimpse()

```

Rows: ??

Columns: 2

Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-azure:R 4.4.1//tmp/RtmpsojHNg/file22512dbc]

\$ person_id <int> 2, 6, 7, 8, 16, 18, 25, 36, 40, 42, 44, 4~

\$ condition_occurrence_records <dbl> 1, 1, 1, 4, 2, 2, 1, 4, 1, 3, 2, 5, 1, 3,~

We can see what goes on behind the scenes by viewing the associated SQL.

```

cdm$person |>
  select("person_id") |>
  left_join(
    cdm$condition_occurrence |>
      group_by(person_id) |>
      count(name = "condition_occurrence_records"),
    by="person_id"
  ) |>
  select("person_id", "condition_occurrence_records") |>
  mutate(condition_occurrence_records = coalesce(condition_occurrence_records, 0)) |>
  show_query()

```

<SQL>

SELECT

```

  person_id,
  COALESCE(condition_occurrence_records, 0.0) AS condition_occurrence_records
FROM (
  SELECT person.person_id AS person_id, condition_occurrence_records

```



```

FROM person
LEFT JOIN (
  SELECT person_id, COUNT(*) AS condition_occurrence_records
  FROM og_002_1761230688
  GROUP BY person_id
) RHS
ON (person.person_id = RHS.person_id)
) q01

```

💡 Taking care with joins

When adding variables through joins we need to pay particular attention to the dimensions of the resulting table. While sometimes we may want to have additional rows added as well as new columns, this is often not desired. For example, if we have a table with one row per person, performing a left join to another table containing multiple rows per person will result in multiple rows per person in the output.

Examples where to be careful include when joining to the observation period table, as individuals can have multiple observation periods, and when working with cohorts (which are the focus of the next chapter) as individuals can also enter the same study cohort multiple times.

Just to underline how problematic joins can become if we don't take care, here we join the condition occurrence table and the drug exposure table, both of which have multiple records per person. Even with our small synthetic dataset, this produces an extremely large table. When working with real patient data, which is oftentimes much, much larger, this would be extremely problematic (and would unlikely be needed to answer any research question). In other words, don't try this at home!

```

cdm$condition_occurrence |>
  tally()

```

```

# Source:   SQL [?? x 1]
# Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-
azure:R 4.4.1//tmp/RtmpsojHNg/file22512dbc8104.duckdb]
      n
  <dbl>
1  9967

```

```

cdm$drug_exposure |>
  tally()

```

```

# Source:   SQL [?? x 1]

```

```
# Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-  
azure:R 4.4.1//tmp/RtmpsojHNg/file22512dbc8104.duckdb]
```

```
      n  
      <dbl>  
1 337509
```

```
cdm$condition_occurrence |>  
  select(person_id, condition_start_date) |>  
  left_join(  
    cdm$drug_exposure |>  
      select(person_id, drug_exposure_start_date),  
    by = "person_id"  
  ) |>  
  tally()
```

```
# Source:   SQL [?? x 1]  
# Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-  
azure:R 4.4.1//tmp/RtmpsojHNg/file22512dbc8104.duckdb]
```

```
      n  
      <dbl>  
1 410683
```

7.5 Disconnecting

Once we have finished our analysis we can close our connection to the database behind our cdm reference.

```
cdmDisconnect(cdm)
```

7.6 Further reading

- Català M, Guo Y, Du M, Lopez-Guell K, Burn E, Mercade-Besora N (2025). *Patient-Profiles: Identify Characteristics of Patients in the OMOP Common Data Model*. R package version 1.4.3, <https://darwin-eu.github.io/PatientProfiles/>.

8 Adding cohorts to the CDM

8.1 What is a cohort?

When performing research with the OMOP CDM we often want to identify groups of individuals who share some set of characteristics. The criteria for including individuals can range from the seemingly simple (e.g. people diagnosed with asthma) to the much more complicated (e.g. adults diagnosed with asthma who had a year of prior observation time in the database prior to their diagnosis, had no prior history of chronic obstructive pulmonary disease, and no history of use of short-acting beta-antagonists).

The set of people we identify are cohorts, and the OMOP CDM has a specific structure by which they can be represented, with a cohort table having four required fields:

- 1) *Cohort definition id* a unique identifier for each cohort (multiple cohorts can be defined in the same cohort table)
- 2) *Subject id* a foreign key to the subject in the cohort - typically referring to records in the person table
- 3) *Cohort start date* date that indicates the start date of the record.
- 4) *Cohort end date* date that indicates the end date of the record.

Individuals must be defined in the person table and have an ongoing record in the observation period table to be part of a cohort. Individuals can enter a cohort multiple times, but the time periods in which they are in the cohort cannot overlap.

It is beyond the scope of this book to describe all the different ways cohorts could be created, however in this chapter we provide a summary of some of the key building blocks for cohort creation. Cohort-building pipelines can be created following these principles to create a wide range of study cohorts.

8.2 Set up

We'll use the same Covid-19 synthetic dataset that we used before for demonstrating how cohorts can be constructed.

```
library(omock)
library(CohortConstructor)
library(CohortCharacteristics)
library(dplyr)

cdm <- mockCdmFromDataset(datasetName = "synthea-covid19-10k", source = "duckdb")
```

8.3 General concept based cohort

Often study cohorts will be based around a specific clinical event identified by some set of clinical codes. Here, for example, we use the `CohortConstructor` package to create a cohort of people with Covid-19. For this we are identifying any clinical records with the code `37311061`.

```
cdm$covid <- conceptCohort(cdm = cdm,
                          conceptSet = list("covid" = 37311061),
                          name = "covid")

cdm$covid
```

```
# Source:   table<results.test_covid> [?? x 4]
# Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-azure:R 4.4.1//tmp/RtmpD3rZGd/file22caccc]
  cohort_definition_id subject_id cohort_start_date cohort_end_date
            <int>         <int> <date>             <date>
1                 1           400 2021-01-18         2021-02-03
2                 1          3596 2021-06-05         2021-07-08
3                 1          4211 2020-07-30         2020-08-28
4                 1          5028 2020-10-31         2020-11-24
5                 1          8056 2020-10-05         2020-11-04
6                 1          1681 2020-11-25         2020-11-25
7                 1          2926 2020-04-28         2020-05-26
8                 1          4684 2020-08-09         2020-08-28
9                 1          8485 2020-11-18         2020-12-22
10                1          3209 2020-12-27         2021-01-09
# i more rows
```

i Name consistency

Note that the `name` argument determines the name of the permanent table written in the database and as we have seen before, we have to be consistent assigning the tables to the `cdm` object. That's why we used `name = "covid"` and then we were able to assign it to `cdm$covid`. Otherwise, see this failing example:

```
cdm$not_covid <- conceptCohort(cdm = cdm,
                              conceptSet = list("covid" = 37311061),
                              name = "covid")
```

Warning: ! `codelist` casted to integers.

```
i Subsetting table condition_occurrence using 1 concept with domain: condition.
i Combining tables.
i Creating cohort attributes.
i Applying cohort requirements.
i Merging overlapping records.
v Cohort covid created.
```

Error in `[<-`:

x You can't assign a table named covid to not_covid.

i You can change the name using compute:

```
cdm[['not_covid']] <- yourObject |>
  dplyr::compute(name = 'not_covid')
```

i You can also change the name using the `name` argument in your function:
`name = 'not_covid'`.

Finding appropriate codes

In defining the cohorts above, we have needed to provide concept IDs to define our cohort. But where do these come from?

We can search for codes of interest using the [CodelistGenerator](#) package. This can be done using a text search with the function [getCandidateCodes\(\)](#). For example, we can have found the code we used above (and many others) like so:

```
library(CodelistGenerator)
getCandidateCodes(cdm = cdm,
                  keywords = c("coronavirus", "covid"),
                  domains = "condition",
                  includeDescendants = TRUE)
```

Limiting to domains of interest

Getting concepts to include

Adding descendants

Search completed. Finishing up.

v 37 candidate concepts identified

Time taken: 0 minutes and 1 seconds

```
# A tibble: 37 x 6
  concept_id found_from concept_name domain_id vocabulary_id standard_concept
    <int> <chr>      <chr>      <chr>      <chr>      <chr>
1   703446 From initia~ Moderate ri~ Condition SNOMED      S
2   756031 From initia~ Bronchitis ~ Condition OMOP Extensi~ S
3   3661748 From initia~ Acute kidne~ Condition SNOMED      S
4   3656667 From initia~ Cardiomyopa~ Condition SNOMED      S
5   3661885 From initia~ Fever cause~ Condition SNOMED      S
6   37310286 From initia~ Infection o~ Condition SNOMED      S
7    703447 From initia~ High risk c~ Condition SNOMED      S
8   1340294 From initia~ Exacerbatio~ Condition OMOP Extensi~ S
9   3661631 From initia~ Lymphocytop~ Condition SNOMED      S
10  3661632 From initia~ Thrombocyto~ Condition SNOMED      S
# i 27 more rows
```

We can also do automated searches that make use of the hierarchies in the vocabularies. Here, for example, we find the code for the drug ingredient Acetaminophen and all of its descendants.

```
getDrugIngredientCodes(cdm = cdm, name = "acetaminophen")
```

```
-- 1 codelist -----
- 161_acetaminophen (25747 codes)
```

Note that in practice clinical expertise is vital in the identification of appropriate codes so as to decide which the codes are in line with the clinical idea at hand.

We can see that as well as having the cohort entries above, our cohort table is associated with several attributes.

First, we can see the settings associated with cohort.

```
settings(cdm$covid) |>
  glimpse()
```

```
Rows: 1
Columns: 4
$ cohort_definition_id <int> 1
$ cohort_name          <chr> "covid"
```

```
$ cdm_version          <chr> "5.3"
$ vocabulary_version   <chr> "v5.0 22-JUN-22"
```

In settings, we can see the cohort name that by default is the name of the codelist used, in this case 'covid' as we used `conceptSet = list(covid = 37311061)`. Also, the cdm and vocabulary versions are recorded in the settings by the CohortConstructor package.

Second, we can get counts of each cohort.

```
cohortCount(cdm$covid) |>
  glimpse()
```

```
Rows: 1
Columns: 3
$ cohort_definition_id <int> 1
$ number_records      <int> 964
$ number_subjects     <int> 964
```

Where you can see the number of records and number of subjects for each cohort. In this case, there are no multiple records per subject.

Attrition can also be retrieved from any cohort.

```
attrition(cdm$covid) |>
  glimpse()
```

```
Rows: 6
Columns: 7
$ cohort_definition_id <int> 1, 1, 1, 1, 1, 1
$ number_records      <int> 964, 964, 964, 964, 964, 964
$ number_subjects     <int> 964, 964, 964, 964, 964, 964
$ reason_id          <int> 1, 2, 3, 4, 5, 6
$ reason              <chr> "Initial qualifying events", "Record in observati~
$ excluded_records    <int> 0, 0, 0, 0, 0, 0
$ excluded_subjects   <int> 0, 0, 0, 0, 0, 0
```

And finally, you can extract the codelists used to create a cohort table:

```
codelist <- cohortCodelist(cdm$covid, cohortId = 1)
codelist
```

```
-- 1 codelist -----
```

```
- covid (1 codes)
```

```
codelist$covid
```

```
[1] 37311061
```

Note that in this case, we had to provide the cohortId of the cohort of interest.

All these attributes can be retrieved because it is a `cohort_table` object, a class on top of the usual `cdm_table` class that we have seen before:

```
class(cdm$covid)
```

```
[1] "cohort_table"      "cdm_table"          "GeneratedCohortSet"
[4] "tbl_duckdb_connection" "tbl_dbi"            "tbl_sql"
[7] "tbl_lazy"          "tbl"
```

As we will see below, these attributes of the cohorts become particularly useful as we apply further restrictions on our cohort.

💡 Behind the scenes

All these attributes that we have seen are part of the attributes of the `cohort_table` object and are used by these utility functions:

```
names(attributes(cdm$covid))
```

```
[1] "names"          "class"          "tbl_source"     "tbl_name"
[5] "cohort_set"     "cohort_attrition" "cohort_codelist" "cdm_reference"
```

In particular, the `cohort_set` (contains the `settings()` source), `cohort_attrition` (contains the source for `cohortCount()` and `attrition()`) and `cohort_codelist` (contains the source for `cohortCodelist()`) attributes are the ones of interest. For database back-ends, these attributes are stored in the database so when we read them again, the attributes persist. See that even though apparently we only have one table ‘`cdm$covid`’, in fact four tables were written in the database:


```
library(omopgenerics) # TODO https://github.com/OHDSI/omock/issues/189
```

Attaching package: 'omopgenerics'

The following object is masked from 'package:stats':

filter

```
listSourceTables(cdm = cdm)
```

```
[1] "covid"          "covid_attrition" "covid_codelist"  "covid_set"
```

We do not have to worry about the attributes and the naming of the tables as CohortConstructor, CDMConnector and omopgenerics take care of that and if we create the cohorts with functions such as `conceptCohort()` then we will be able to read them back with the `cohortTables` argument of `cdmFromCon()` or the `readSourceTable()` function and all the attributes will be in place.

8.4 Applying inclusion criteria

8.4.1 Only include first cohort entry per person

Let's say we first want to restrict to the first entry.

```
cdm$covid <- cdm$covid |>  
  requireIsFirstEntry()
```

8.4.2 Restrict to study period

Then we are interested in records only after January 1st, 2020.

```
cdm$covid <- cdm$covid |>  
  requireInDateRange(dateRange = c(as.Date("2020-01-01"), NA))
```

8.4.3 Applying demographic inclusion criteria

Finally, we want to restrict our population of interest to only adult males under 65 years old. We can do that with the `requireDemographics()` function.

```
cdm$covid <- cdm$covid |>
  requireDemographics(ageRange = c(18, 64), sex = "Male")
```

Similarity of naming with PatientProfiles

Note that all these `require*()` functions that come from the `CohortConstructor` package use functionalities from `PatientProfiles` and the naming is consistent. For example, `requireDemographics()` uses `addDemographics()`, `requirePriorObservation()` uses `addPriorObservation()`, and so on...

8.4.4 Applying cohort-based inclusion criteria

As well as requirements about specific demographics, we may also want to use another cohort for inclusion criteria. Let's say we want to exclude anyone with a history of cardiac conditions before their Covid-19 cohort entry.

We can first generate this new cohort table with records of cardiac conditions.

```
cdm$cardiac <- conceptCohort(
  cdm = cdm,
  conceptSet = list("myocardial_infarction" = c(317576L, 313217L, 321042L, 4329847L)),
  name = "cardiac"
)
cdm$cardiac
```

```
# Source:   table<results.test_cardiac> [?? x 4]
# Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-azure:R 4.4.1//tmp/RtmpD3rZGd/file22caccc]
  cohort_definition_id subject_id cohort_start_date cohort_end_date
      <int>           <int> <date>              <date>
1             1         256 2017-12-01          2017-12-01
2             1         695 2000-01-27          2000-01-27
3             1         731 2007-02-26          2007-02-26
4             1        1194 1997-03-14          1997-03-14
5             1        1357 2011-08-28          2011-08-28
6             1        1664 2020-11-10          2020-11-10
7             1        2257 1978-02-04          1978-02-04
```

```

8           1      3452 2018-04-04      2018-04-04
9           1      5120 2021-08-16      2021-08-16
10          1      6278 1997-01-31      1997-01-31
# i more rows

```

And now we can apply the inclusion criteria that individuals have zero intersections with the table in the time prior to their Covid-19 cohort entry.

```

cdm$covid <- cdm$covid |>
  requireCohortIntersect(targetCohortTable = "cardiac",
    indexDate = "cohort_start_date",
    window = c(-Inf, -1),
    intersections = 0)

```

Note that if we had wanted to require that individuals did have a history of a cardiac condition, we would instead have set `intersections = c(1, Inf)` above.

i Use `requireConceptIntersect`

We could have applied the exact same inclusion criteria using the `requireConceptIntersect()` function, this code would be equivalent:

```

cdm$covid <- cdm$covid |>
  requireConceptIntersect(
    conceptSet = list("myocardial_infarction" = c(317576L, 313217L, 321042L, 4329847L)),
    indexDate = "cohort_start_date",
    window = c(-Inf, -1),
    intersections = 0
  )

```

In fact, this approach would be more efficient unless we want to re-use the `myocardial_infarction` cohort for another inclusion criteria or analysis. Note that the intersection with the cohort table is more flexible as it can have more complicated inclusion/exclusion criteria, but you have to be more careful with the order of inclusion criteria (e.g., if we would restrict the `myocardial_infarction` cohort to a certain time span when we would do the intersect, we would require to not have the inclusion criteria on that time span).

8.5 Cohort attributes

Using the `require*()` functions, the cohort attributes have been updated to reflect the applied inclusion criteria.

```
settings(cdm$covid) |>
  glimpse()
```

```
Rows: 1
Columns: 8
$ cohort_definition_id  <int> 1
$ cohort_name           <chr> "covid"
$ cdm_version           <chr> "5.3"
$ vocabulary_version    <chr> "v5.0 22-JUN-22"
$ age_range             <chr> "18_64"
$ sex                   <chr> "Male"
$ min_prior_observation <dbl> 0
$ min_future_observation <dbl> 0
```

```
cohortCount(cdm$covid) |>
  glimpse()
```

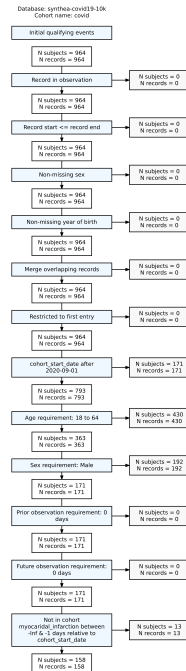
```
Rows: 1
Columns: 3
$ cohort_definition_id <int> 1
$ number_records       <int> 158
$ number_subjects      <int> 158
```

```
attrition(cdm$covid) |>
  glimpse()
```

```
Rows: 13
Columns: 7
$ cohort_definition_id <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
$ number_records       <int> 964, 964, 964, 964, 964, 964, 964, 964, 793, 363, 171, ~
$ number_subjects      <int> 964, 964, 964, 964, 964, 964, 964, 964, 793, 363, 171, ~
$ reason_id            <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13
$ reason               <chr> "Initial qualifying events", "Record in observati~
$ excluded_records     <int> 0, 0, 0, 0, 0, 0, 0, 0, 171, 430, 192, 0, 0, 13
$ excluded_subjects    <int> 0, 0, 0, 0, 0, 0, 0, 0, 171, 430, 192, 0, 0, 13
```

We can visualize the attrition with the CohortCharacteristics package. We can first extract it with `summariseCohortAttrition()` and then `plotCohortAttrition` to better view the impact of applying each inclusion criteria:

```
attrition_summary <- summariseCohortAttrition(cohort = cdm$covid)
plotCohortAttrition(result = attrition_summary, type = 'png')
```



Note that the `conceptCohort()` first step leads to several rows in the attrition table, whereas any other `require*()` function always adds just one record of attrition.

💡 Cohort naming utilities

As we have seen, by default the naming of the cohorts is the name of the codelist:

```
cdm$my_cohort <- conceptCohort(cdm = cdm,
                              conceptSet = list("concept_1" = 37311061L, "concept_2" = 31711061L),
                              name = "my_cohort")
```

- i Subsetting table `condition_occurrence` using 2 concepts with domain: `condition`.
- i Combining tables.
- i Creating cohort attributes.
- i Applying cohort requirements.

i Merging overlapping records.
v Cohort my_cohort created.

```
settings(cdm$my_cohort)
```

```
# A tibble: 2 x 4
  cohort_definition_id cohort_name cdm_version vocabulary_version
      <int> <chr>          <chr>          <chr>
1             1 concept_1    5.3            v5.0 22-JUN-22
2             2 concept_2    5.3            v5.0 22-JUN-22
```

But maybe we are interested in renaming a cohort (e.g., after applying the inclusion criteria). We can do that with the `renameCohort()` utility function:

```
cdm$my_cohort <- cdm$my_cohort |>
  requirePriorObservation(minPriorObservation = 365, cohortId = 1) |>
  renameCohort(cohortId = 1, newCohortName = "concept_1_365obs")
settings(cdm$my_cohort)
```

```
# A tibble: 2 x 5
  cohort_definition_id cohort_name      cdm_version vocabulary_version
      <int> <chr>          <chr>          <chr>
1             1 concept_1_365obs 5.3            v5.0 22-JUN-22
2             2 concept_2        5.3            v5.0 22-JUN-22
# i 1 more variable: min_prior_observation <dbl>
```

Note that for arguments such as `cohortId`, `targetCohortId`, etc., we are able to use the name of the cohort of interest. See for example:

```
cdm$my_cohort <- cdm$my_cohort |>
  requireSex(sex = "Female", cohortId = "concept_2") |>
  renameCohort(cohortId = "concept_2", newCohortName = "concept_2_female")
settings(cdm$my_cohort)
```

```
# A tibble: 2 x 6
  cohort_definition_id cohort_name      cdm_version vocabulary_version
      <int> <chr>          <chr>          <chr>
1             1 concept_1_365obs 5.3            v5.0 22-JUN-22
2             2 concept_2_female 5.3            v5.0 22-JUN-22
# i 2 more variables: min_prior_observation <dbl>, sex <chr>
```

This functionality also applies to other packages, such as CohortCharacteristics, PatientProfiles, DrugUtilisation, etc. Finally, in some cases, it is useful to add the cohort_name as a column to not have to check manually the equivalence between *cohort definition id* and *cohort name*. This can be done using the PatientProfiles utility function addCohortName():

```
library(PatientProfiles)
cdm$my_cohort |>
  addCohortName() |>
  glimpse()
```

```
Rows: ??
Columns: 5
Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-
azure:R 4.4.1//tmp/RtmpD3rZGd/file22cacc6c71a.duckdb]
$ cohort_definition_id <int> 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2~
$ subject_id          <int> 71, 82, 134, 151, 153, 183, 194, 196, 200, 215, 2~
$ cohort_start_date   <date> 1936-01-24, 1997-08-04, 2001-01-09, 2022-
01-13, ~
$ cohort_end_date     <date> 1936-01-24, 1997-08-04, 2001-01-09, 2022-
01-13, ~
$ cohort_name         <chr> "concept_2_female", "concept_2_female", "concept_~
```

Also other utility functions that can be useful are those provided by omopgenerics:

```
library(omopgenerics)
getCohortId(cohort = cdm$my_cohort, cohortName = "concept_2_female")
```

```
concept_2_female
      2
```

```
getCohortId(cohort = cdm$my_cohort)
```

```
concept_1_365obs concept_2_female
      1           2
```

```
getCohortName(cohort = cdm$my_cohort, cohortId = 1)
```

```
      1
"concept_1_365obs"
```

```
getCohortName(cohort = cdm$my_cohort, cohortId = c(2, 1))
```

```
      2      1  
"concept_2_female" "concept_1_365obs"
```

```
getCohortName(cohort = cdm$my_cohort)
```

```
      1      2  
"concept_1_365obs" "concept_2_female"
```

8.6 Disconnecting

Once we have finished our analysis we can close our connection to the database behind our cdm reference.

```
cdmDisconnect(cdm)
```

8.7 Further reading

- [Cohort tables](#)
- Burn E, Català M, Mercade-Besora N, Alcalde-Herraiz M, Du M, Guo Y, Chen X, Lopez-Guell K, Rowlands E (2025). *CohortConstructor: Build and Manipulate Study Cohorts Using a Common Data Model*. R package version 0.5.0, <https://ohdsi.github.io/CohortConstructor/>.

9 Working with cohorts

9.1 Cohort intersections

When conducting research, it is often necessary to study patients who meet multiple clinical criteria simultaneously. For example, we may be interested in analysing outcomes among patients who have both diabetes and hypertension. Using the OMOP CDM, this typically involves first creating two separate cohorts: one for patients with diabetes and another for those with hypertension. To identify patients who meet both conditions, the next step is to compute the intersection of these cohorts. This ensures that the final study population includes only individuals who satisfy all specified criteria. Hence, finding cohort intersections is a common and essential task when working with the OMOP CDM, enabling researchers to define precise target populations that align with their research objectives.

Depending on the research question, the definition of a cohort intersection may vary. For instance, you might require patients to have a diagnosis of hypertension before developing diabetes, or that both diagnoses occur within a specific time window. These additional temporal or clinical criteria can make cohort intersection more complex. The `PatientProfiles` R package addresses these challenges by providing a suite of flexible functions to support the calculation of cohort intersections under various scenarios.

9.2 Intersection between two cohorts

Suppose we are interested in studying patients with gastrointestinal (GI) bleeding who have also been exposed to acetaminophen. First, we would create two separate cohorts: one for patients with GI bleeding and another for patients with exposure to acetaminophen. Below is an example of the code used to create these cohorts within the GiBleed synthetic database. A characterisation of this dataset can be found [here](#).

```
library(omock)
library(dplyr)
library(PatientProfiles)
library(CohortConstructor)
library(omopgenerics) # TODO https://github.com/OHDSI/omock/issues/189
```

```

cdm <- mockCdmFromDataset(datasetName = "GiBleed", source = "duckdb")

# gi_bleed contains all records of gi bleed, end date is 30 days after index
# date
cdm$gi_bleed <- conceptCohort(
  cdm = cdm,
  conceptSet = list("gi_bleed" = 192671L),
  name = "gi_bleed",
  exit = "event_start_date"
) |>
  padCohortEnd(days = 30)

# drugs cohort contains records of acetaminophen using start and end dates of
# the drug records and collapsing records separated by less than 30 days
cdm$drugs <- conceptCohort(
  cdm = cdm,
  conceptSet = list("acetaminophen" = c(
    1125315L, 1127078L, 1127433L, 40229134L, 40231925L, 40162522L, 19133768L
  )),
  name = "drugs",
  exit = "event_end_date"
) |>
  collapseCohorts(gap = 30)

```

The `PatientProfiles` package contains functions to obtain the intersection flag, count, date, or number of days between cohorts. To get a binary indicator showing the presence of an intersection between the cohorts within a given time window, we can use `addCohortIntersectFlag()`.

9.2.1 Flag

```

x <- cdm$gi_bleed |>
  addCohortIntersectFlag(
    targetCohortTable = "drugs",
    window = list("prior" = c(-Inf, -1), "index" = c(0, 0), "post" = c(1, Inf))
  )

x |>
  summarise(
    acetaminophen_prior = sum(acetaminophen_prior, na.rm = TRUE),

```

```

    acetaminophen_index = sum(acetaminophen_index, na.rm = TRUE),
    acetaminophen_post = sum(acetaminophen_post, na.rm = TRUE)
  ) |>
  collect()

```

A tibble: 1 x 3

	acetaminophen_prior	acetaminophen_index	acetaminophen_post
	<dbl>	<dbl>	<dbl>
1	467	1	315

Window naming

Windows work very similarly to age groups that we have seen before. If a name is not provided, an automatic name will be obtained from the values of the window limits:

```

cdm$gi_bleed |>
  addCohortIntersectFlag(
    targetCohortTable = "drugs",
    window = list(c(-Inf, -1), c(0, 0), c(1, Inf))
  ) |>
  glimpse()

```

Rows: ??

Columns: 7

Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-

azure:R 4.4.1//tmp/RtmpYZAqIG/file235063834f39.duckdb]

```

$ cohort_definition_id    <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ~
$ subject_id              <int> 549, 1076, 2591, 2343, 4591, 2556, 3675, 3820~
$ cohort_start_date       <date> 1987-12-28, 2013-09-07, 1974-09-
16, 1996-02-~
$ cohort_end_date         <date> 1988-01-27, 2013-10-07, 1974-10-
16, 1996-03-~
$ acetaminophen_minf_to_m1 <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ~
$ acetaminophen_0_to_0    <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ~
$ acetaminophen_1_to_inf  <dbl> 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, ~

```

Note that to avoid conflicts with column naming, all names will be lower case, spaces are not allowed, and the - symbol for negative values is replaced by m. That's why it is usually nice to provide your own custom names:

```
cdm$gi_bleed |>
  addCohortIntersectFlag(
    targetCohortTable = "drugs",
    window = list("prior" = c(-Inf, -1), "index" = c(0, 0), "post" = c(1, Inf))
  ) |>
  glimpse()
```

Rows: ??

Columns: 7

Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-

azure:R 4.4.1//tmp/RtmpYZAqIG/file235063834f39.duckdb]

\$ cohort_definition_id <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1~

\$ subject_id <int> 549, 1076, 2591, 2343, 4591, 2556, 3675, 3820, 30~

\$ cohort_start_date <date> 1987-12-28, 2013-09-07, 1974-09-16, 1996-02-10, ~

\$ cohort_end_date <date> 1988-01-27, 2013-10-07, 1974-10-16, 1996-03-11, ~

\$ acetaminophen_post <dbl> 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1~

\$ acetaminophen_prior <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1~

\$ acetaminophen_index <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0~

New column naming

By default, the name of new columns is ‘{cohort_name}_{window_name}’ as we have seen in the prior examples. In some cases, you only have one cohort or one window and you might want to rename the column as you please. In that case, you can use the `nameStyle` argument to change the new naming of the columns:

```
cdm$gi_bleed |>
  addCohortIntersectFlag(
    targetCohortTable = "drugs",
    window = list("prior" = c(-Inf, -1), "index" = c(0, 0), "post" = c(1, Inf)),
    nameStyle = "my_column_{window_name}"
  ) |>
  glimpse()
```

Rows: ??

Columns: 7

Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-

azure:R 4.4.1//tmp/RtmpYZAqIG/file235063834f39.duckdb]

\$ cohort_definition_id <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1~

```

$ subject_id      <int> 549, 1076, 2591, 2343, 4591, 2556, 3675, 3820, 30~
$ cohort_start_date <date> 1987-12-28, 2013-09-07, 1974-09-16, 1996-
02-10, ~
$ cohort_end_date  <date> 1988-01-27, 2013-10-07, 1974-10-16, 1996-
03-11, ~
$ my_column_post   <dbl> 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1~
$ my_column_prior  <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1~
$ my_column_index  <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0~

```

If multiple windows are provided but ‘{window_name}’ is not included in `nameStyle`, then an error will prompt:

```

cdm$gi_bleed |>
  addCohortIntersectFlag(
    targetCohortTable = "drugs",
    window = list("prior" = c(-Inf, -1), "index" = c(0, 0), "post" = c(1, Inf)),
    nameStyle = "my_new_column"
  ) |>
  glimpse()

```

```

Error in `$.addIntersect()` :
! The following elements are not present in nameStyle:
* {window_name}

```

Many functions that create new columns (usually start with `add*()`) have this `nameStyle` functionality that allows you to control the naming of the new columns created.

9.2.2 Count

To get the count of occurrences of intersection between two cohorts, we can use `addCohortIntersectCount()`:

```

x <- cdm$gi_bleed |>
  addCohortIntersectCount(
    targetCohortTable = "drugs",
    window = list("prior" = c(-Inf, -1), "index" = c(0, 0), "post" = c(1, Inf)),
  )

x |>
  summarise(
    sum_prior = sum(acetaminophen_prior, na.rm = TRUE),

```

```

mean_prior = mean(acetaminophen_prior, na.rm = TRUE),
sum_index = sum(acetaminophen_index, na.rm = TRUE),
mean_index = mean(acetaminophen_index, na.rm = TRUE),
sum_post = sum(acetaminophen_post, na.rm = TRUE),
mean_post = mean(acetaminophen_post, na.rm = TRUE)
) |>
collect()

```

A tibble: 1 x 6

	sum_prior	mean_prior	sum_index	mean_index	sum_post	mean_post
	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
1	1669	3.48	1	0.00209	758	1.58

i Handling the observation period

Note that **only intersections in the current observation period are considered**. The count and flag new columns can also have NA values meaning that the individual was not in observation in that window of interest. If we see individual 2070, it has *3748* of future observation:

```

cdm$gi_bleed |>
  filter(subject_id == 2070) |>
  addFutureObservation() |>
  glimpse()

```

Rows: ??

Columns: 5

Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-
azure:R 4.4.1//tmp/RtmpYZAqIG/file235063834f39.duckdb]

```

$ cohort_definition_id <int> 1
$ subject_id          <int> 2070
$ cohort_start_date   <date> 2008-08-15
$ cohort_end_date     <date> 2008-09-14
$ future_observation  <int> 3748

```

Now we will perform the intersect with the following window of interest: `c(2000, 3000)`, `c(3000, 4000)`, `c(4000, 5000)`.

```

cdm$gi_bleed |>
  filter(subject_id == 2070) |>
  addCohortIntersectCount(
    targetCohortTable = "drugs",
    window = list(c(2000, 3000), c(3000, 4000), c(4000, 5000)),
  ) |>
  glimpse()

```

```

Rows: ??
Columns: 7
Database: DuckDB 1.4.1 [unknown@Linux 6.11.0-1018-
azure:R 4.4.1//tmp/RtmpYZAqIG/file235063834f39.duckdb]
$ cohort_definition_id      <int> 1
$ subject_id                <int> 2070
$ cohort_start_date         <date> 2008-08-15
$ cohort_end_date           <date> 2008-09-14
$ acetaminophen_2000_to_3000 <dbl> 0
$ acetaminophen_3000_to_4000 <dbl> 0
$ acetaminophen_4000_to_5000 <dbl> NA

```

See that for the window 2000 to 3000, where the individual is still in observation, a 0 is reported. The same happens for the window 3000 to 4000 even if the individual does not have complete observation in the window. But for the last window, as the individual is not in observation at any point of the window, NA is reported.

9.2.3 Date and times

To get the date of the intersection with a cohort within a given time window, we can use `addCohortIntersectDate()`. To get the number of days between the index date and intersection, we can use `addCohortIntersectDays()`.

Both functions allow the `order` argument to specify which value to return:

- `first` returns the first date/days that satisfy the window
- `last` returns the last date/days that satisfy the window

```

x <- cdm$gi_bleed |>
  addCohortIntersectDate(
    targetCohortTable = "drugs",
    window = list("post" = c(1, Inf)),
    order = "first"
  )

```

```

)

x |>
  summarise(acetaminophen_post = median(acetaminophen_post, na.rm = TRUE)) |>
  collect()

# A tibble: 1 x 1
  acetaminophen_post
  <dtm>
1 2004-02-01 00:00:00

x <- cdm$gi_bleed |>
  addCohortIntersectDays(
    targetCohortTable = "drugs",
    window = list("prior" = c(-Inf, -1)),
    order = "last"
  )

x |>
  summarise(acetaminophen_prior = median(acetaminophen_prior, na.rm = TRUE)) |>
  collect()

# A tibble: 1 x 1
  acetaminophen_prior
  <dbl>
1 -3159

```

Note that for the window in the future, we used `order = "first"` and for the window in the past, we used `order = "last"` as in both cases we wanted to get the intersection that was closer to the index date. Individuals with no intersection will have NA on the newly created columns.

9.3 Intersection between a cohort and tables with patient data

Sometimes we might want to get the intersection between a cohort and another OMOP table. `PatientProfiles` also includes several `addTableIntersect*` functions to obtain intersection flags, counts, days, or dates between a cohort and clinical tables.

For example, if we want to get the number of general practitioner (GP) visits for individuals in the cohort, we can use the `visit_occurrence` table:


```
x <- cdm$gi_bleed |>
  addTableIntersectCount(
    tableName = "visit_occurrence",
    window = list(c(-Inf, -1)),
    nameStyle = "number_visits"
  )

x |>
  summarise(visit_occurrence_prior = median(number_visits, na.rm = TRUE)) |>
  collect()
```

```
# A tibble: 1 x 1
  visit_occurrence_prior
                <dbl>
1                      0
```

9.4 Disconnecting

Once we have finished our analysis we can close our connection to the database behind our cdm reference.

```
cdmDisconnect(cdm)
```

9.5 Further reading

Full details on the intersection functions in `PatientProfiles` can be found on the package website: <https://darwin-eu.github.io/PatientProfiles/>.

Final remarks

Tidy R programming with the OMOP Common Data Model aims to (1) explain the main principles for working with databases from R and (2) how to apply these principles and use them with the OMOP CDM. Hopefully, after reading this book, you can understand how the `dplyr` and `dbplyr` packages interact with the databases, in particular with data formatted to the OMOP CDM; how the `cdm_reference` object can be used to extract and identify your population of interest; and add the desired features to your dataset. Note that in this book we always worked with relatively small synthetic data with unrealistic performance. Any analysis conducted with real-world data and big databases will take more time, that's why we would always recommend you test your code against synthetic data or subsets of your data to ensure good performance. Be careful, especially while writing custom code using join functions that can create some ugly SQL.

Tidy R in OMOP collection

This book is the first of the *Tidy OMOP CDM* collection. If you like it, please take a look at the full collection:

- 1 [Tidy R programming with the OMOP CDM](#)
- 2 [Tidy R packages with the OMOP CDM](#) *under construction*
- 3 [DrugUtilisation in the OMOP CDM](#) *under construction*

Support us

We encourage you to support this work by either citing the book in your papers or documentation, recommending it to your colleagues, or starring the [GitHub repository](#), or simply letting us know how it helped you. But on top of that, **use it** so it can help you and others in your research for patient benefit.

About the authors

- Edward Burn
- Martí Català