

Table of contents

P	refac	e	1							
	Is this book for me?									
	How	is the book organised?	1							
	Cita	tion	1							
	Lice	nse	2							
	Cod	e	2							
	renv		2							
Ι	Ge	tting started with working databases from R	3							
1	A fi	rst analysis using data in a database	7							
	1.1	Getting set up	7							
	1.2	Taking a peek at the data	8							
	1.3	Inserting data into a database	9							
	1.4	Translation from R to SQL	10							
	1.5	Example analysis	11							
	1.6	Disconnecting from the database	16							
	1.7	Further reading	16							
2	Cor	e verbs for analytic pipelines utilising a database	17							
		2.0.1 Tidyverse functions	20							
	2.1	Getting to an analytic dataset	23							

3	Sup	apported expressions for database queries 27					
	3.1	Data	types	28			
		3.1.1	duckdb	28			
		3.1.2	Redshift	29			
		3.1.3	Postgres	30			
		3.1.4	Snowflake	31			
		3.1.5	Spark	32			
		3.1.6	SQL Server	33			
	3.2	Comp	varison and logical operators	35			
		3.2.1	duckdb	35			
		3.2.2	Redshift	36			
		3.2.3	Postgres	37			
		3.2.4	Snowflake	38			
		3.2.5	Spark	39			
		3.2.6	SQL Server	40			
	3.3	Condi	tional statements	42			
		3.3.1	duckdb	42			
		3.3.2	Redshift	43			
		3.3.3	Postgres	44			
		3.3.4	Snowflake	45			
		3.3.5	Spark	46			
		3.3.6	SQL Server	47			
	3.4	Worki	ing with strings	48			
		3.4.1	duckdb	48			
		3.4.2	Redshift	51			
		3.4.3	Postgres	53			
		3.4.4	Snowflake	56			
		3.4.5	Spark	58			
		3.4.6	SQL Server	61			
	3.5	Worki	ing with dates	64			
		3.5.1	duckdb	64			

TA	ABLE	OF CO	ONTENTS	v				
		3.5.2	Redshift	65				
		3.5.3	Postgres	66				
		3.5.4	Snowflake	67				
		3.5.5	Spark	68				
		3.5.6	SQL Server	69				
	3.6	Data a	aggregation	70				
		3.6.1	duckdb	70				
		3.6.2	postgres	71				
		3.6.3	${\it redshift} \dots \dots \dots \dots \dots \dots \dots \dots \dots $	71				
		3.6.4	Snowflake	72				
		3.6.5	Spark	73				
		3.6.6	SQL Server	73				
	3.7	Windo	ow functions	74				
		3.7.1	${\rm duckdb} \dots \dots \dots \dots \dots \dots \dots \dots \dots $	75				
		3.7.2	postgres	76				
		3.7.3	${\it redshift} \dots \dots \dots \dots \dots \dots \dots \dots \dots $	77				
		3.7.4	Snowflake	78				
		3.7.5	Spark	79				
		3.7.6	SQL Server	80				
	3.8	Calcul	ating quantiles, including the median	81				
		3.8.1	duckdb	81				
		3.8.2	postgres	81				
		3.8.3	${\it redshift} \dots \dots \dots \dots \dots \dots \dots \dots \dots $	82				
		3.8.4	Snowflake	82				
		3.8.5	Spark	83				
		3.8.6	SQL Server	83				
4	Bui	lding a	analytic pipelines for a data model	85				
	4.1	Defini	ng a data model	86				
	4.2	Creati	ng functions for the data model	88				
	4.3	Building efficient analytic pipelines						

		4.3.1 The risk of "clean" R code	96
		4.3.2 Piping and SQL \dots	98
		4.3.3 Computing intermediate queries	100
II	W	Forking with the OMOP CDM from R 1	.05
5	\mathbf{Cre}	ating a CDM reference	109
	5.1	The OMOP common data model (CDM) layout	109
	5.2	Creating a reference to the OMOP CDM	110
	5.3	CDM attributes	113
		5.3.1 CDM name	113
		5.3.2 CDM version	115
	5.4	Including cohort tables in the cdm reference	115
	5.5	Including achilles tables in the cdm reference	116
	5.6	Adding other tables to the cdm reference	117
	5.7	Mutability of the cdm reference	120
	5.8	Working with temporary and permanent tables	122
6	Dis	connecting 1	125
7	Fur	ther reading 1	127
8	Exp	oloring the OMOP CDM	129
	8.1	Counting people	130
	8.2	Summarising observation periods	132
	8.3	Summarising clinical records	133
9	Ide	ntifying patient characteristics	139
	9.1	Adding specific demographics	140
	9.2	Adding multiple demographics simultaneously	143
	9.3	Creating categories	146
	9.4	Adding custom variables	148

TABLE OF CONTENTS	vii
10 Further reading 1	.53
11 Adding cohorts to the CDM	.55
11.1 What is a cohort?	155
11.2 Set up	155
11.3 General concept based cohort	156
11.4 Applying inclusion criteria	159
11.4.1 Only include first cohort entry per person	159
11.4.2 Restrict to study period	159
11.4.3 Applying demographic inclusion criteria	159
11.4.4 Applying cohort-based inclusion criteria	159
11.5 Cohort attributes	160
12 Further reading 1	.63
13 Working with cohorts	65
13.1 Cohort intersections	165
13.2 Intersection between two cohorts	165
13.3 Set up	165
13.3.1 Flag	166
13.3.2 Count	166
13.3.3 Date and times	166
13.4 Intersection between a cohort and tables with patient data	166
14 Further reading 1	67

Preface

Is this book for me?

We've written this book for anyone interested in a working with databases using a tidyverse style approach. That is, human centered, consistent, composable, and inclusive (see https://design.tidyverse.org/unifying.html for more details on these principles).

New to R? We recommend you compliment the book with R for data science

New to databases? We recommend you take a look at some web tutorials on SQL, such as SQLBolt or SQLZoo

New to the OMOP CDM? We'd recommend you pare 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 Common Data Model (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

TO ADD

2 Preface

Packages	Version	Link
bit64	4.6.0-1	
CDMConnector	2.0.0	
cli	3.6.4	
clock	0.7.2	
CodelistGenerator	3.4.1	
CohortCharacteristics	0.5.1	
CohortConstructor	0.3.5	
DBI	1.2.3	
dbplyr	2.5.0	
dm	1.0.11	
dplyr	1.1.4	
duckdb	1.2.1	
ggplot2	3.5.1	
here	1.0.1	
Lahman	12.0-0	
omock	0.3.2	
omopgenerics	1.1.1	
palmerpenguins	0.1.1	
PatientProfiles	1.3.1	
purrr	1.0.4	
sloop	1.0.1	
stringr	1.5.1	
tidyr	1.3.1	

License

Code

The source code for the book can be found at this Github repository

renv

This book is rendered using the following version of packages:

Note here only the packages called explicitly are mentioned for the full list of packages and versions used see the book renv file in github.

Part I

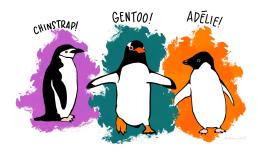
Getting started with working databases from R

In this first half of the book we will see how we can work with databases from R. In the following chapters we'll see that when working with data held in a relational database we can leverage various open-source R packages to help us 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 which we can then bring into R for further analysis.
- 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.

Chapter 1

A first analysis using data in a database



Artwork by @allison_horst

Before we start thinking about 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 simpler 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

Assuming that you have R and RStudio already set up1, first we need to install a few packages not included in base R if we don't already have them.

```
install.packages("dplyr")
install.packages("dbplyr")
```

```
install.packages("ggplot2")
install.packages("DBI")
install.packages("duckdb")
install.packages("palmerpenguins")
```

Once installed, we can load them like so.

```
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
                                                                                       <fct> Adelie, 
$ species
$ island
                                                                                       <fct> 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~
                                                                                       <int> 2007, 2007, 2007, 2007, 2007, 2007, 2007, 2007, 2007
$ year
```

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

39.1

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
                                             18.7
                                                                            3750
```

181

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 mor	# i 2 more variables: sex <fct>, year <int></int></fct>						

1.3 Inserting data into a database

Let's put our penguins data into a duckdb database. We need to first create the database and then add the penguins data to it.

```
db <- dbConnect(drv = duckdb())
dbWriteTable(db, "penguins", penguins)</pre>
```

We can see that our database now has one table:

```
dbListTables(db)
```

[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(db, "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
```

- male 2007
- 2 female 2007
- 3 female 2007
- 4 <NA> 2007
- 5 female 2007

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 snipets 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

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:

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(db, "penguins")
penguins_db</pre>
```

```
# Source: tabletabletablepenguins> [?? x 8]
# Database: DuckDB v1.2.1 [unknown@Linux 6.11.0-1012-azure:R 4.5.0/:memory:]
    species island         bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
```

<fct></fct>	<fct></fct>	<dbl></dbl>	<dbl></dbl>	<int></int>	<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	
n ·						

i more rows

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 v1.2.1 [unknown@Linux 6.11.0-1012-azure:R 4.5.0/: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 NANANANA36.7 19.3 193 3450 5 Adelie Torgersen # 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:

[#] i 2 more variables: sex <fct>, year <int>

```
penguins_db |>
  group_by(species) |>
  summarise(
   n = n(),
   min_bill_length_mm = min(bill_length_mm),
   mean_bill_length_mm = mean(bill_length_mm),
   max_bill_length_mm = max(bill_length_mm)
  ) |>
  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"
```

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),
    mean_bill_length_mm = mean(bill_length_mm),
    max_bill_length_mm = max(bill_length_mm)
) |>
  mutate(min_max_bill_length_mm = pasteO(min, " to ", max)) |>
  select(
    "species",
    "mean_bill_length_mm",
    "min_max_bill_length_mm"
) |>
  show_query()
```

penguins_db |>

3 Adelie

4 Gentoo

5 Chinstrap

```
SELECT
   species,
   mean_bill_length_mm,
   CONCAT_WS('', .Primitive("min"), ' to ', .Primitive("max")) 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 may well be more familiar if coming from an R programming background.

Not having to worry about the SQL translation behind our queries allows us to interrogate 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.

```
mutate(missing_body_mass_g = if_else(
   is.na(body_mass_g), 1, 0
  )) |>
  group by(species, missing body mass g) |>
 tally()
            SQL [?? x 3]
# Source:
# Database: DuckDB v1.2.1 [unknown@Linux 6.11.0-1012-azure:R 4.5.0/:memory:]
  species
            missing_body_mass_g
                           <dbl> <dbl>
  <fct>
1 Adelie
                              0
                                  151
2 Gentoo
                              0
                                  123
```

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

1

1

68

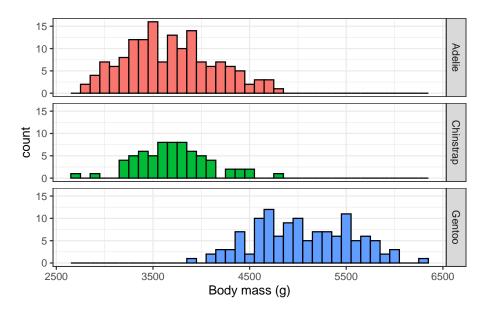
1

1

```
penguins_db |>
  group_by(species) |>
  summarise(mean_body_mass_g = round(mean(body_mass_g, na.rm = TRUE)))
            SQL [?? x 2]
# Source:
# Database: DuckDB v1.2.1 [unknown@Linux 6.11.0-1012-azure:R 4.5.0/:memory:]
  species
            mean_body_mass_g
  <fct>
                       <dbl>
1 Adelie
                         3701
2 Chinstrap
                         3733
                        5076
3 Gentoo
```

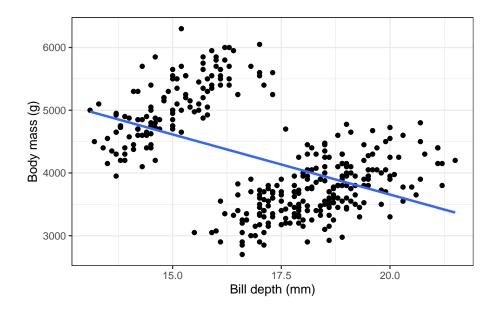
We could also make a histogram of values for each of the species. Here we would collect our data back into R before creating our plot.

```
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")
```



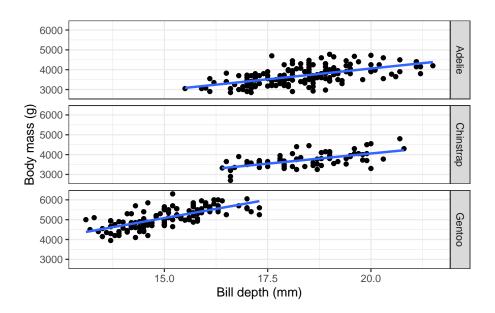
Now let's look at the relationship between body mass and bill depth.

```
penguins |>
  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 |>
  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 using the DBI package.

dbDisconnect(db)

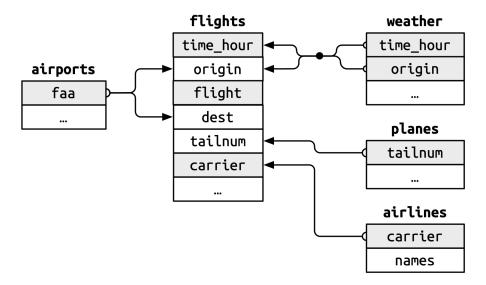
1.7 Further reading

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

Chapter 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 though 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, now we have a set of related tables with data on flights departing from New York City airports in 2013.



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

```
library(dplyr)
library(dbplyr)
library(tidyr)
library(duckdb)
library(DBI)

db <- dbConnect(duckdb(), dbdir = ":memory:")
copy_nycflights13(db)

airports_db <- tbl(db, "airports")
airports_db |> glimpse()
```

```
Rows: ??
Columns: 8
Database: DuckDB v1.2.1 [unknown@Linux 6.11.0-1012-azure:R 4.5.0/:memory:]
      <chr> "04G", "06A", "06C", "06N", "09J", "0A9", "0G6", "0G7", "0P2", "~
$ faa
$ name <chr>> "Lansdowne Airport", "Moton Field Municipal Airport", "Schaumbur~
      <dbl> 41.13047, 32.46057, 41.98934, 41.43191, 31.07447, 36.37122, 41.4~
$ lat
$ lon
      <dbl> -80.61958, -85.68003, -88.10124, -74.39156, -81.42778, -82.17342~
      <dbl> 1044, 264, 801, 523, 11, 1593, 730, 492, 1000, 108, 409, 875, 10~
$ alt
$ tz
      $ dst
$ tzone <chr> "America/New_York", "America/Chicago", "America/Chicago", "Ameri~
```

```
flights_db <- tbl(db, "flights")</pre>
flights_db |> glimpse()
Rows: ??
Columns: 19
Database: DuckDB v1.2.1 [unknown@Linux 6.11.0-1012-azure:R 4.5.0/:memory:]
                <int> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2~
$ year
$ month
                $ day
                $ 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~
                <int> 830, 850, 923, 1004, 812, 740, 913, 709, 838, 753, 849,~
$ arr time
$ sched_arr_time <int> 819, 830, 850, 1022, 837, 728, 854, 723, 846, 745, 851,~
                <dbl> 11, 20, 33, -18, -25, 12, 19, -14, -8, 8, -2, -3, 7, -1~
$ arr delay
$ carrier
                <chr> "UA", "UA", "AA", "B6", "DL", "UA", "B6", "EV", "B6", "~
$ flight
                <int> 1545, 1714, 1141, 725, 461, 1696, 507, 5708, 79, 301, 4~
                <chr> "N14228", "N24211", "N619AA", "N804JB", "N668DN", "N394~
$ tailnum
                <chr> "EWR", "LGA", "JFK", "JFK", "LGA", "EWR", "EWR", "LGA",~
$ origin
                <chr> "IAH", "IAH", "MIA", "BQN", "ATL", "ORD", "FLL", "IAD",~
$ dest
                <dbl> 227, 227, 160, 183, 116, 150, 158, 53, 140, 138, 149, 1~
$ air_time
                <dbl> 1400, 1416, 1089, 1576, 762, 719, 1065, 229, 944, 733, ~
$ distance
                <dbl> 5, 5, 5, 5, 6, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6
$ hour
$ minute
                <dbl> 15, 29, 40, 45, 0, 58, 0, 0, 0, 0, 0, 0, 0, 0, 0, 59, 0~
$ time_hour
                <dttm> 2013-01-01 10:00:00, 2013-01-01 10:00:00, 2013-01-01 1~
```

```
weather_db <- tbl(db, "weather")
weather_db |> glimpse()
```

Rows: ?? Columns: 15 Database: DuckDB v1.2.1 [unknown@Linux 6.11.0-1012-azure:R 4.5.0/:memory:] <chr> "EWR", "EWR" \$ origin <int> 2013, \$ year \$ month \$ day <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 14, 15, 16, 17, 18, ~ \$ hour <dbl> 39.02, 39.02, 39.02, 39.92, 39.02, 37.94, 39.02, 39.92, 39.~ \$ temp <dbl> 26.06, 26.96, 28.04, 28.04, 28.04, 28.04, 28.04, 28.04, 28.04 \$ dewp \$ 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 \$ precip

```
$ pressure
           <dbl> 1012.0, 1012.3, 1012.5, 1012.2, 1011.9, 1012.4, 1012.2, 101~
$ visib
           $ time_hour
           <dttm> 2013-01-01 06:00:00, 2013-01-01 07:00:00, 2013-01-01 08:00~
planes_db <- tbl(db, "planes")</pre>
planes_db |> glimpse()
Rows: ??
Columns: 9
Database: DuckDB v1.2.1 [unknown@Linux 6.11.0-1012-azure:R 4.5.0/:memory:]
             <chr> "N10156", "N102UW", "N103US", "N104UW", "N10575", "N105UW~
$ tailnum
$ year
             <int> 2004, 1998, 1999, 1999, 2002, 1999, 1999, 1999, 1999, 1999
             <chr> "Fixed wing multi engine", "Fixed wing multi engine", "Fi~
$ type
$ manufacturer <chr> "EMBRAER", "AIRBUS INDUSTRIE", "AIRBUS INDUSTRIE", "AIRBU~
             <chr> "EMB-145XR", "A320-214", "A320-214", "A320-214", "EMB-145~
$ model
$ engines
             $ seats
             <int> 55, 182, 182, 182, 55, 182, 182, 182, 182, 182, 55, 55, 5~
             $ speed
             <chr> "Turbo-fan", "Turbo-fan", "Turbo-fan", "Turbo-fan", "Turb~
$ engine
airlines_db <- tbl(db, "airlines")</pre>
airlines_db |> glimpse()
Rows: ??
Columns: 2
Database: DuckDB v1.2.1 [unknown@Linux 6.11.0-1012-azure:R 4.5.0/:memory:]
$ carrier <chr> "9E", "AA", "AS", "B6", "DL", "EV", "F9", "FL", "HA", "MQ", "0~
```

<chr> "Endeavor Air Inc.", "American Airlines Inc.", "Alaska Airline~

2.0.1 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 below tidyverse functions (most of which coming 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

\$ name

Remember, 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.

Purpose	Functions	Description
Selecting rows	filter, distinct	To select rows in a table.
Ordering rows	arrange	To order rows in a table.
Column Transformation	mutate, select, relocate, rename	To create new columns or change existing ones.
Grouping and ungrouping	group_by, rowwise, ungroup	To group data by one or more variables and to
Aggregation	count, tally, summarise	remove grouping. These functions are used for summarising data.
Data merging and joining	inner_join, left_join, right_join, full_join, anti_join, semi_join, cross_join	These functions are used to combine data from different tables based on
Data reshaping	pivot_wider, pivot_longer	common columns. These functions are used to reshape data between wide and long formats.
Data union	union_all, union	This function combines two tables.
Randomly selects rows	slice_sample	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 col-
lected into r the object changes to have different attributes, one of which
being data.frame:
class(flights_db)
                                                       "tbl_sql"
[1] "tbl_duckdb_connection" "tbl_dbi"
[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>
                         <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.

2.1 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.

```
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 v1.2.1 [unknown@Linux 6.11.0-1012-azure:R 4.5.0/:memory:]
  dest distance carrier tailnum delay
```

<chr></chr>	<dbl></dbl>	<chr>></chr>	<chr></chr>	<dbl></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.

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.

```
i Show query

<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 v1.2.1 [unknown@Linux 6.11.0-1012-azure:R 4.5.0/:memory:]
  dest distance carrier tailnum delay seats
```

<chr< th=""><th>> <dbl></dbl></th><th><chr></chr></th><th><chr></chr></th><th><dbl></dbl></th><th><int></int></th></chr<>	> <dbl></dbl>	<chr></chr>	<chr></chr>	<dbl></dbl>	<int></int>
1 BQN	1576	B6	N804JB	-18	200
2 MCO	944	B6	N593JB	-8	200
3 PBI	1028	B6	N793JB	-2	200
4 BOS	187	B6	N708JB	-4	200
5 ATL	760	DL	N3739P	-8	189
6 SJU	1598	B6	N794JB	-21	200
7 PHX	2153	US	N535UW	0	379
8 BOS	187	B6	N805JB	-10	200
9 LAS	2248	B6	N558JB	-6	200
10 SLC	1990	DL	N3763D	-9	189
# i more	e rows				

Getting to this tidy dataset has been done in the database via R code translated to SQL. With this, we can now collect our analytic dataset into R and go from there (for example, to perform locally statistical analyses which might not be possible to run in a database).

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

delayed_flights |>
  glimpse()
```

 $26 CHAPTER\ 2.\ CORE\ VERBS\ FOR\ ANALYTIC\ PIPELINES\ UTILISING\ A\ DATABASE$

Chapter 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 pipleines 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 which 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 wish 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)

options(dplyr.strict_sql = TRUE) # force error if no known translation
```

3.1 Data types

Commonly used data types are consistently supported across database backends. We can use the base as.numeric(), as.integer(), as.charater(), 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
translate sql(as.numeric(var),
              con = simulate_duckdb())
<SQL> CAST(`var` AS NUMERIC)
translate_sql(as.integer(var),
              con = simulate_duckdb())
<SQL> CAST(`var` AS INTEGER)
translate_sql(as.integer64(var),
              con = simulate_duckdb())
<SQL> CAST(`var` AS BIGINT)
translate_sql(as.character(var),
              con = simulate_duckdb())
<SQL> CAST(`var` AS TEXT)
translate_sql(as.Date(var),
              con = simulate_duckdb())
<SQL> CAST(`var` AS DATE)
translate_sql(as_date(var),
              con = simulate_duckdb())
<SQL> CAST(`var` AS DATE)
translate_sql(as.POSIXct(var),
              con = simulate_duckdb())
```

3.1. DATA TYPES 29

```
<SQL> CAST(`var` AS TIMESTAMP)
translate_sql(as_datetime(var),
            con = simulate_duckdb())
<SQL> CAST(`var` AS TIMESTAMP)
translate_sql(as.logical(var),
            con = simulate_duckdb())
<SQL> CAST(`var` AS BOOLEAN)
3.1.2 Redshift
translate_sql(as.numeric(var),
            con = simulate_redshift())
<SQL> CAST(`var` AS FLOAT)
translate_sql(as.integer(var),
            con = simulate_redshift())
<SQL> CAST(`var` AS INTEGER)
translate_sql(as.integer64(var),
             con = simulate_redshift())
<SQL> CAST(`var` AS BIGINT)
translate_sql(as.character(var),
           con = simulate_redshift())
<SQL> CAST(`var` AS TEXT)
translate_sql(as.Date(var),
            con = simulate_redshift())
<SQL> CAST(`var` AS DATE)
translate_sql(as_date(var),
            con = simulate_redshift())
```

```
<SQL> CAST(`var` AS DATE)
translate_sql(as.POSIXct(var),
           con = simulate_redshift())
<SQL> CAST(`var` AS TIMESTAMP)
translate_sql(as_datetime(var),
            con = simulate_redshift())
<SQL> CAST(`var` AS TIMESTAMP)
translate_sql(as.logical(var),
           con = simulate_redshift())
<SQL> CAST(`var` AS BOOLEAN)
3.1.3 Postgres
translate_sql(as.numeric(var),
              con = simulate_postgres())
<SQL> CAST(`var` AS NUMERIC)
translate_sql(as.integer(var),
             con = simulate_postgres())
<SQL> CAST(`var` AS INTEGER)
translate_sql(as.integer64(var),
              con = simulate_postgres())
<SQL> CAST(`var` AS BIGINT)
translate_sql(as.character(var),
              con = simulate_postgres())
<SQL> CAST(`var` AS TEXT)
translate_sql(as.Date(var),
              con = simulate_postgres())
```

3.1. DATA TYPES 31

```
<SQL> CAST(`var` AS DATE)
translate_sql(as_date(var),
             con = simulate_postgres())
<SQL> CAST(`var` AS DATE)
translate_sql(as.POSIXct(var),
             con = simulate_postgres())
<SQL> CAST(`var` AS TIMESTAMP)
translate_sql(as_datetime(var),
             con = simulate_postgres())
<SQL> CAST(`var` AS TIMESTAMP)
translate_sql(as.logical(var),
          con = simulate_postgres())
<SQL> CAST(`var` AS BOOLEAN)
3.1.4 Snowflake
translate_sql(as.numeric(var),
             con = simulate_snowflake())
<SQL> CAST(`var` AS DOUBLE)
translate_sql(as.integer(var),
             con = simulate_snowflake())
<SQL> CAST(`var` AS INT)
translate_sql(as.integer64(var),
             con = simulate_snowflake())
<SQL> CAST(`var` AS BIGINT)
translate_sql(as.character(var),
            con = simulate_snowflake())
```

```
<SQL> CAST(`var` AS STRING)
translate_sql(as.Date(var),
           con = simulate_snowflake())
<SQL> CAST(`var` AS DATE)
translate_sql(as_date(var),
            con = simulate_snowflake())
<SQL> CAST(`var` AS DATE)
translate_sql(as.POSIXct(var),
        con = simulate_snowflake())
<SQL> CAST(`var` AS TIMESTAMP)
translate_sql(as_datetime(var),
            con = simulate_snowflake())
<SQL> CAST(`var` AS TIMESTAMP)
translate_sql(as.logical(var),
             con = simulate_snowflake())
<SQL> CAST(`var` AS BOOLEAN)
3.1.5 Spark
translate_sql(as.numeric(var),
             con = simulate_spark_sql())
<SQL> CAST(`var` AS DOUBLE)
translate_sql(as.integer(var),
             con = simulate_spark_sql())
<SQL> CAST(`var` AS INT)
translate_sql(as.integer64(var),
             con = simulate_spark_sql())
```

3.1. DATA TYPES 33

```
<SQL> CAST(`var` AS BIGINT)
translate_sql(as.character(var),
             con = simulate_spark_sql())
<SQL> CAST(`var` AS STRING)
translate_sql(as.Date(var),
             con = simulate_spark_sql())
<SQL> CAST(`var` AS DATE)
translate_sql(as_date(var),
            con = simulate_spark_sql())
<SQL> CAST(`var` AS DATE)
translate_sql(as.POSIXct(var),
     con = simulate_spark_sql())
<SQL> CAST(`var` AS TIMESTAMP)
translate_sql(as_datetime(var),
          con = simulate_spark_sql())
<SQL> CAST(`var` AS TIMESTAMP)
translate_sql(as.logical(var),
            con = simulate_spark_sql())
<SQL> CAST(`var` AS BOOLEAN)
3.1.6 SQL Server
translate_sql(as.numeric(var),
             con = simulate_mssql())
<SQL> TRY_CAST(`var` AS FLOAT)
translate_sql(as.integer(var),
           con = simulate_mssql())
```

```
<SQL> TRY_CAST(TRY_CAST(`var` AS NUMERIC) AS INT)
translate_sql(as.integer64(var),
              con = simulate_mssql())
<SQL> TRY_CAST(TRY_CAST(`var` AS NUMERIC(38, 0)) AS BIGINT)
translate_sql(as.character(var),
             con = simulate_mssql())
<SQL> TRY_CAST(`var` AS VARCHAR(MAX))
translate_sql(as.Date(var),
             con = simulate_mssql())
<SQL> TRY_CAST(`var` AS DATE)
translate_sql(as_date(var),
       con = simulate_mssql())
<SQL> TRY_CAST(`var` AS DATE)
translate_sql(as.POSIXct(var),
           con = simulate_mssql())
<SQL> TRY_CAST(`var` AS DATETIME2)
translate_sql(as_datetime(var),
           con = simulate_mssql())
<SQL> TRY_CAST(`var` AS DATETIME2)
translate_sql(as.logical(var),
             con = simulate_mssql())
<SQL> TRY_CAST(`var` AS BIT)
```

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 was in R.

```
    Show SQL

3.2.1 duckdb
translate_sql(var_1 == var_2,
              con = simulate_duckdb())
<SQL> `var_1` = `var_2`
translate_sql(var_1 >= var_2,
              con = simulate_duckdb())
<SQL> `var_1` >= `var_2`
translate_sql(var_1 < 100,
              con = simulate_duckdb())
<SQL> `var_1` < 100.0
translate_sql(var_1 %in% c("a", "b", "c"),
              con = simulate_duckdb())
<SQL> `var_1` IN ('a', 'b', 'c')
translate_sql(!var_1 %in% c("a", "b", "c"),
              con = simulate_duckdb())
<SQL> NOT(`var_1` IN ('a', 'b', 'c'))
translate_sql(is.na(var_1),
              con = simulate_duckdb())
<SQL> (`var_1` IS NULL)
translate_sql(!is.na(var_1),
              con = simulate_duckdb())
```

```
<SQL> NOT(('var_1' IS NULL))
translate_sql(var_1 >= 100 & var_1 < 200,
           con = simulate_duckdb())
<SQL> `var_1` >= 100.0 AND `var_1` < 200.0
translate_sql(var_1 >= 100 \mid var_1 < 200,
            con = simulate_duckdb())
<SQL> `var_1` >= 100.0 OR `var_1` < 200.0
3.2.2 Redshift
translate_sql(var_1 == var_2,
            con = simulate_redshift())
<SQL> `var_1` = `var_2`
translate_sql(var_1 >= var_2,
            con = simulate_redshift())
<SQL> `var 1` >= `var 2`
translate_sql(var_1 < 100,</pre>
              con = simulate_redshift())
<SQL> `var_1` < 100.0
translate_sql(var_1 %in% c("a", "b", "c"),
           con = simulate_redshift())
<SQL> `var_1` IN ('a', 'b', 'c')
translate_sql(!var_1 %in% c("a", "b", "c"),
          con = simulate_redshift())
<SQL> NOT(`var_1` IN ('a', 'b', 'c'))
translate_sql(is.na(var_1),
              con = simulate_redshift())
```

```
<SQL> (`var_1` IS NULL)
translate_sql(!is.na(var_1),
            con = simulate_redshift())
<SQL> NOT(('var_1' IS NULL))
translate_sql(var_1 >= 100 \& var_1 < 200,
             con = simulate_redshift())
<SQL> `var_1` >= 100.0 AND `var_1` < 200.0
translate_sql(var_1 >= 100 | var_1 < 200,</pre>
             con = simulate_redshift())
<SQL> `var_1` >= 100.0 OR `var_1` < 200.0
3.2.3 Postgres
translate_sql(var_1 == var_2,
             con = simulate_postgres())
<SQL> `var_1` = `var_2`
translate_sql(var_1 >= var_2,
             con = simulate_postgres())
<SQL> `var_1` >= `var_2`
translate_sql(var_1 < 100,
             con = simulate_postgres())
<SQL> `var_1` < 100.0
translate_sql(var_1 %in% c("a", "b", "c"),
             con = simulate_postgres())
<SQL> `var_1` IN ('a', 'b', 'c')
translate_sql(!var_1 %in% c("a", "b", "c"),
             con = simulate_postgres())
```

```
<SQL> NOT(`var_1` IN ('a', 'b', 'c'))
translate_sql(is.na(var_1),
              con = simulate_postgres())
<SQL> (`var_1` IS NULL)
translate_sql(!is.na(var_1),
             con = simulate_postgres())
<SQL> NOT(('var_1' IS NULL))
translate_sql(var_1 >= 100 & var_1 < 200,
              con = simulate_postgres())
<SQL> `var_1` >= 100.0 AND `var_1` < 200.0
translate_sql(var_1 >= 100 | var_1 < 200,
             con = simulate_postgres())
<SQL> `var_1` >= 100.0 OR `var_1` < 200.0
3.2.4 Snowflake
translate_sql(var_1 == var_2,
             con = simulate_snowflake())
<SQL> `var_1` = `var_2`
translate_sql(var_1 >= var_2,
             con = simulate_snowflake())
<SQL> `var 1` >= `var 2`
translate_sql(var_1 < 100,</pre>
              con = simulate_snowflake())
<SQL> `var_1` < 100.0
translate_sql(var_1 %in% c("a", "b", "c"),
              con = simulate_snowflake())
```

```
<SQL> `var_1` IN ('a', 'b', 'c')
translate_sql(!var_1 %in% c("a", "b", "c"),
            con = simulate_snowflake())
<SQL> NOT(`var_1` IN ('a', 'b', 'c'))
translate_sql(is.na(var_1),
              con = simulate_snowflake())
<SQL> (`var_1` IS NULL)
translate_sql(!is.na(var_1),
            con = simulate_snowflake())
<SQL> NOT(('var_1' IS NULL))
translate_sql(var_1 >= 100 & var_1 < 200,
             con = simulate_snowflake())
<SQL> `var_1` >= 100.0 AND `var_1` < 200.0
translate_sql(var_1 >= 100 | var_1 < 200,
             con = simulate_snowflake())
<SQL> `var_1` >= 100.0 OR `var_1` < 200.0
3.2.5 Spark
translate_sql(var_1 == var_2,
            con = simulate_spark_sql())
<SQL> `var_1` = `var_2`
translate_sql(var_1 >= var_2,
             con = simulate_spark_sql())
<SQL> `var_1` >= `var_2`
translate_sql(var_1 < 100,</pre>
             con = simulate_spark_sql())
```

```
<SQL> `var_1` < 100.0
translate_sql(var_1 %in% c("a", "b", "c"),
              con = simulate_spark_sql())
<SQL> `var_1` IN ('a', 'b', 'c')
translate_sql(!var_1 %in% c("a", "b", "c"),
             con = simulate_spark_sql())
<SQL> NOT(`var_1` IN ('a', 'b', 'c'))
translate_sql(is.na(var_1),
             con = simulate_spark_sql())
<SQL> (`var_1` IS NULL)
translate_sql(!is.na(var_1),
             con = simulate_spark_sql())
<SQL> NOT(('var_1' IS NULL))
translate_sql(var_1 >= 100 & var_1 < 200,
            con = simulate_spark_sql())
<SQL> `var_1` >= 100.0 AND `var_1` < 200.0
translate_sql(var_1 >= 100 \mid var_1 < 200,
             con = simulate_spark_sql())
<SQL> `var_1` >= 100.0 OR `var_1` < 200.0
3.2.6 SQL Server
translate_sql(var_1 == var_2,
              con = simulate_mssql())
<SQL> `var_1` = `var_2`
translate_sql(var_1 >= var_2,
            con = simulate_mssql())
```

```
<SQL> `var_1` >= `var_2`
translate_sql(var_1 < 100,</pre>
             con = simulate_mssql())
<SQL> `var_1` < 100.0
translate_sql(var_1 %in% c("a", "b", "c"),
             con = simulate_mssql())
<SQL> `var_1` IN ('a', 'b', 'c')
translate_sql(!var_1 %in% c("a", "b", "c"),
             con = simulate mssql())
<SQL> NOT('var_1' IN ('a', 'b', 'c'))
translate_sql(is.na(var_1),
           con = simulate_mssql())
<SQL> (`var_1` IS NULL)
translate_sql(!is.na(var_1),
           con = simulate_mssql())
<SQL> NOT(('var_1' IS NULL))
translate_sql(var_1 >= 100 & var_1 < 200,</pre>
             con = simulate_mssql())
<SQL> `var_1` >= 100.0 AND `var_1` < 200.0</pre>
translate sql(var 1 \ge 100 \mid var 1 < 200,
             con = simulate_mssql())
<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
translate sql(ifelse(var == "a", 1L, 2L),
              con = simulate_duckdb())
<SQL> CASE WHEN ('var' = 'a') THEN 1 WHEN NOT ('var' = 'a') THEN 2 END
translate_sql(if_else(var == "a", 1L, 2L),
              con = simulate duckdb())
<SQL> CASE WHEN ('var' = 'a') THEN 1 WHEN NOT ('var' = 'a') THEN 2 END
translate_sql(case_when(var == "a" ~ 1L, .default = 2L),
              con = simulate_duckdb())
<SQL> CASE WHEN ('var' = 'a') THEN 1 ELSE 2 END
translate_sql(case_when(var == "a" ~ 1L,
                        var == "b" \sim 2L,
                        var == "c" \sim 3L,
                        .default = NULL),
              con = simulate_duckdb())
<SQL> CASE
WHEN ('var' = 'a') THEN 1
WHEN ('var' = 'b') THEN 2
WHEN ('var' = 'c') THEN 3
translate sql(case when(var == "a" ~ 1L,
                        var == "b" \sim 2L,
                        var == "c" ~ 3L,
                        .default = "something else"),
              con = simulate_duckdb())
```

```
<SQL> CASE
WHEN ('var' = 'a') THEN 1
WHEN ('var' = 'b') THEN 2
WHEN ('var' = 'c') THEN 3
ELSE 'something else'
END
3.3.2 Redshift
translate_sql(ifelse(var == "a", 1L, 2L),
             con = simulate_redshift())
<SQL> CASE WHEN ('var' = 'a') THEN 1 WHEN NOT ('var' = 'a') THEN 2 END
translate_sql(if_else(var == "a", 1L, 2L),
            con = simulate_redshift())
<SQL> CASE WHEN ('var' = 'a') THEN 1 WHEN NOT ('var' = 'a') THEN 2 END
translate_sql(case_when(var == "a" ~ 1L, .default = 2L),
             con = simulate_redshift())
<SQL> CASE WHEN ('var' = 'a') THEN 1 ELSE 2 END
translate_sql(case_when(var == "a" ~ 1L,
                       var == "b" \sim 2L,
                        var == "c" ~ 3L,
                        .default = NULL),
              con = simulate_redshift())
<SQL> CASE
WHEN ('var' = 'a') THEN 1
WHEN ('var' = 'b') THEN 2
WHEN ('var' = 'c') THEN 3
translate_sql(case_when(var == "a" ~ 1L,
                       var == "b" ~ 2L,
                       var == "c" ~ 3L,
                        .default = "something else"),
              con = simulate_redshift())
```

```
<SQL> CASE
WHEN ('var' = 'a') THEN 1
WHEN ('var' = 'b') THEN 2
WHEN ('var' = 'c') THEN 3
ELSE 'something else'
END
3.3.3 Postgres
translate_sql(ifelse(var == "a", 1L, 2L),
              con = simulate_postgres())
<SQL> CASE WHEN ('var' = 'a') THEN 1 WHEN NOT ('var' = 'a') THEN 2 END
translate_sql(if_else(var == "a", 1L, 2L),
             con = simulate_postgres())
<SQL> CASE WHEN ('var' = 'a') THEN 1 WHEN NOT ('var' = 'a') THEN 2 END
translate_sql(case_when(var == "a" ~ 1L, .default = 2L),
              con = simulate_postgres())
<SQL> CASE WHEN ('var' = 'a') THEN 1 ELSE 2 END
translate_sql(case_when(var == "a" ~ 1L,
                        var == "b" \sim 2L,
                        var == "c" \sim 3L,
                        .default = NULL),
              con = simulate postgres())
<SQL> CASE
WHEN ('var' = 'a') THEN 1
WHEN ('var' = 'b') THEN 2
WHEN ('var' = 'c') THEN 3
translate sql(case when(var == "a" ~ 1L,
                        var == "b" \sim 2L,
                        var == "c" \sim 3L,
                        .default = "something else"),
              con = simulate_postgres())
```

```
<SQL> CASE
WHEN ('var' = 'a') THEN 1
WHEN ('var' = 'b') THEN 2
WHEN ('var' = 'c') THEN 3
ELSE 'something else'
END
3.3.4 Snowflake
translate_sql(ifelse(var == "a", 1L, 2L),
             con = simulate_snowflake())
<SQL> CASE WHEN ('var' = 'a') THEN 1 WHEN NOT ('var' = 'a') THEN 2 END
translate_sql(if_else(var == "a", 1L, 2L),
            con = simulate_snowflake())
<SQL> CASE WHEN ('var' = 'a') THEN 1 WHEN NOT ('var' = 'a') THEN 2 END
translate_sql(case_when(var == "a" ~ 1L, .default = 2L),
             con = simulate_snowflake())
<SQL> CASE WHEN ('var' = 'a') THEN 1 ELSE 2 END
translate_sql(case_when(var == "a" ~ 1L,
                        var == "b" \sim 2L,
                        var == "c" \sim 3L,
                        .default = NULL),
              con = simulate_snowflake())
<SQL> CASE
WHEN ('var' = 'a') THEN 1
WHEN ('var' = 'b') THEN 2
WHEN ('var' = 'c') THEN 3
translate_sql(case_when(var == "a" ~ 1L,
                       var == "b" ~ 2L,
                        var == "c" \sim 3L,
                        .default = "something else"),
              con = simulate_snowflake())
```

```
<SQL> CASE
WHEN ('var' = 'a') THEN 1
WHEN ('var' = 'b') THEN 2
WHEN ('var' = 'c') THEN 3
ELSE 'something else'
END
3.3.5 Spark
translate_sql(ifelse(var == "a", 1L, 2L),
              con = simulate_spark_sql())
<SQL> CASE WHEN ('var' = 'a') THEN 1 WHEN NOT ('var' = 'a') THEN 2 END
translate_sql(if_else(var == "a", 1L, 2L),
             con = simulate_spark_sql())
<SQL> CASE WHEN (`var` = 'a') THEN 1 WHEN NOT (`var` = 'a') THEN 2 END
translate_sql(case_when(var == "a" ~ 1L, .default = 2L),
              con = simulate_spark_sql())
<SQL> CASE WHEN ('var' = 'a') THEN 1 ELSE 2 END
translate_sql(case_when(var == "a" ~ 1L,
                        var == "b" \sim 2L
                        var == "c" \sim 3L,
                        .default = NULL),
              con = simulate_spark_sql())
<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" \sim 2L,
                        var == "c" \sim 3L,
                        .default = "something else"),
              con = simulate_spark_sql())
```

```
<SQL> CASE
WHEN ('var' = 'a') THEN 1
WHEN ('var' = 'b') THEN 2
WHEN ('var' = 'c') THEN 3
ELSE 'something else'
END
3.3.6 SQL Server
translate_sql(ifelse(var == "a", 1L, 2L),
              con = simulate_mssql())
<SQL> IIF(`var` = 'a', 1, 2)
translate_sql(if_else(var == "a", 1L, 2L),
             con = simulate_mssql())
<SQL> IIF(`var` = 'a', 1, 2)
translate_sql(case_when(var == "a" ~ 1L, .default = 2L),
             con = simulate_mssql())
<SQL> CASE WHEN ('var' = 'a') THEN 1 ELSE 2 END
translate_sql(case_when(var == "a" ~ 1L,
                        var == "b" \sim 2L,
                        var == "c" ~ 3L,
                        .default = NULL),
              con = simulate_mssql())
<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" \sim 3L,
                        .default = "something else"),
              con = simulate_mssql())
```

```
<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
translate_sql(nchar(var),
              con = simulate duckdb())
<SQL> LENGTH(`var`)
translate_sql(nzchar(var),
              con = simulate_duckdb())
<SQL> (('var' IS NULL) OR 'var' != '')
translate_sql(substr(var, 1, 2),
              con = simulate_duckdb())
<SQL> SUBSTR(`var`, 1, 2)
translate_sql(trimws(var),
              con = simulate duckdb())
<SQL> LTRIM(RTRIM(`var`))
translate_sql(tolower(var),
              con = simulate_duckdb())
```

```
<SQL> LOWER('var')
translate_sql(str_to_lower(var),
           con = simulate_duckdb())
<SQL> LOWER('var')
translate_sql(toupper(var),
            con = simulate_duckdb())
<SQL> UPPER(`var`)
translate_sql(str_to_upper(var),
            con = simulate_duckdb())
<SQL> UPPER(`var`)
translate_sql(str_to_title(var),
            con = simulate_duckdb())
<SQL> INITCAP(`var`)
translate_sql(str_trim(var),
            con = simulate_duckdb())
<SQL> LTRIM(RTRIM(`var`))
translate_sql(str_squish(var),
          con = simulate_duckdb())
<SQL> TRIM(REGEXP_REPLACE(`var`, '\s+', ' ', 'g'))
translate_sql(str_detect(var, "b"),
       con = simulate_duckdb())
<SQL> REGEXP_MATCHES(`var`, 'b')
translate_sql(str_detect(var, "b", negate = TRUE),
        con = simulate_duckdb())
<SQL> (NOT(REGEXP_MATCHES(`var`, 'b')))
```

```
translate_sql(str_detect(var, "[aeiou]"),
              con = simulate_duckdb())
<SQL> REGEXP_MATCHES(`var`, '[aeiou]')
translate_sql(str_replace(var, "a", "b"),
              con = simulate duckdb())
<SQL> REGEXP_REPLACE(`var`, 'a', 'b')
translate_sql(str_replace_all(var, "a", "b"),
              con = simulate duckdb())
<SQL> REGEXP_REPLACE(`var`, 'a', 'b', 'g')
translate_sql(str_remove(var, "a"),
              con = simulate_duckdb())
<SQL> REGEXP_REPLACE(`var`, 'a', '')
translate_sql(str_remove_all(var, "a"),
             con = simulate duckdb())
<SQL> REGEXP_REPLACE(`var`, 'a', '', 'g')
translate_sql(str_like(var, "a"),
            con = simulate_duckdb())
<SQL> `var` LIKE 'a'
translate_sql(str_starts(var, "a"),
              con = simulate_duckdb())
<SQL> REGEXP_MATCHES(`var`,'^(?:'||'a'))
translate_sql(str_ends(var, "a"),
           con = simulate_duckdb())
<SQL> REGEXP_MATCHES((?:`var`,'a'||')$')
```

3.4.2 Redshift

```
translate_sql(nchar(var),
             con = simulate redshift())
<SQL> LENGTH(`var`)
translate_sql(nzchar(var),
            con = simulate_redshift())
<SQL> (('var' IS NULL) OR 'var' != '')
translate_sql(substr(var, 1, 2),
            con = simulate_redshift())
<SQL> SUBSTRING(`var`, 1, 2)
translate_sql(trimws(var),
            con = simulate_redshift())
<SQL> LTRIM(RTRIM(`var`))
translate_sql(tolower(var),
           con = simulate_redshift())
<SQL> LOWER('var')
translate_sql(str_to_lower(var),
          con = simulate_redshift())
<SQL> LOWER(`var`)
translate_sql(toupper(var),
       con = simulate_redshift())
<SQL> UPPER(`var`)
translate_sql(str_to_upper(var),
       con = simulate_redshift())
<SQL> UPPER('var')
```

```
translate_sql(str_to_title(var),
              con = simulate_redshift())
<SQL> INITCAP(`var`)
translate_sql(str_trim(var),
              con = simulate_redshift())
<SQL> LTRIM(RTRIM(`var`))
translate_sql(str_squish(var),
              con = simulate_redshift())
<SQL> LTRIM(RTRIM(REGEXP_REPLACE(`var`, '\s+', ' ', 'g')))
translate_sql(str_detect(var, "b"),
              con = simulate_redshift())
<SQL> `var` ~ 'b'
translate_sql(str_detect(var, "b", negate = TRUE),
              con = simulate_redshift())
<SQL> !(`var` ~ 'b')
translate_sql(str_detect(var, "[aeiou]"),
              con = simulate_redshift())
<SQL> `var` ~ '[aeiou]'
translate_sql(str_replace(var, "a", "b"),
              con = simulate redshift())
Error in `str_replace()`:
! `str_replace()` is not available in this SQL variant.
translate_sql(str_replace_all(var, "a", "b"),
              con = simulate_redshift())
<SQL> REGEXP_REPLACE(`var`, 'a', 'b')
translate_sql(str_remove(var, "a"),
             con = simulate_redshift())
```

```
<SQL> REGEXP_REPLACE(`var`, 'a', '')
translate_sql(str_remove_all(var, "a"),
              con = simulate_redshift())
<SQL> REGEXP_REPLACE(`var`, 'a', '', 'g')
translate_sql(str_like(var, "a"),
             con = simulate_redshift())
<SQL> `var` ILIKE 'a'
translate_sql(str_starts(var, "a"),
            con = simulate_redshift())
Error in `str_starts()`:
! Only fixed patterns are supported on database backends.
translate_sql(str_ends(var, "a"),
             con = simulate_redshift())
Error in `str_ends()`:
! Only fixed patterns are supported on database backends.
3.4.3 Postgres
translate_sql(nchar(var),
              con = simulate_postgres())
<SQL> LENGTH(`var`)
translate_sql(nzchar(var),
              con = simulate_postgres())
<SQL> (('var' IS NULL) OR 'var' != '')
translate_sql(substr(var, 1, 2),
              con = simulate_postgres())
<SQL> SUBSTR(`var`, 1, 2)
translate_sql(trimws(var),
            con = simulate_postgres())
```

```
<SQL> LTRIM(RTRIM(`var`))
translate_sql(tolower(var),
             con = simulate_postgres())
<SQL> LOWER(`var`)
translate_sql(str_to_lower(var),
             con = simulate_postgres())
<SQL> LOWER('var')
translate_sql(toupper(var),
              con = simulate_postgres())
<SQL> UPPER(`var`)
translate_sql(str_to_upper(var),
        con = simulate_postgres())
<SQL> UPPER(`var`)
translate_sql(str_to_title(var),
            con = simulate_postgres())
<SQL> INITCAP(`var`)
translate_sql(str_trim(var),
             con = simulate_postgres())
<SQL> LTRIM(RTRIM(`var`))
translate_sql(str_squish(var),
              con = simulate_postgres())
<SQL> LTRIM(RTRIM(REGEXP_REPLACE(`var`, '\s+', ' ', 'g')))
translate_sql(str_detect(var, "b"),
             con = simulate_postgres())
<SQL> `var` ~ 'b'
```

```
translate_sql(str_detect(var, "b", negate = TRUE),
              con = simulate_postgres())
<SQL> !(`var` ~ 'b')
translate_sql(str_detect(var, "[aeiou]"),
              con = simulate_postgres())
<SQL> `var` ~ '[aeiou]'
translate_sql(str_replace(var, "a", "b"),
             con = simulate postgres())
<SQL> REGEXP_REPLACE(`var`, 'a', 'b')
translate_sql(str_replace_all(var, "a", "b"),
              con = simulate_postgres())
<SQL> REGEXP REPLACE('var', 'a', 'b', 'g')
translate_sql(str_remove(var, "a"),
              con = simulate_postgres())
<SQL> REGEXP_REPLACE(`var`, 'a', '')
translate_sql(str_remove_all(var, "a"),
              con = simulate_postgres())
<SQL> REGEXP_REPLACE(`var`, 'a', '', 'g')
translate sql(str like(var, "a"),
             con = simulate_postgres())
<SQL> `var` ILIKE 'a'
translate_sql(str_starts(var, "a"),
              con = simulate_postgres())
Error in `str_starts()`:
! Only fixed patterns are supported on database backends.
translate_sql(str_ends(var, "a"),
             con = simulate_postgres())
Error in `str_ends()`:
! Only fixed patterns are supported on database backends.
```

3.4.4 Snowflake

```
translate_sql(nchar(var),
             con = simulate snowflake())
<SQL> LENGTH(`var`)
translate_sql(nzchar(var),
            con = simulate_snowflake())
<SQL> (('var' IS NULL) OR 'var' != '')
translate_sql(substr(var, 1, 2),
            con = simulate_snowflake())
<SQL> SUBSTR(`var`, 1, 2)
translate_sql(trimws(var),
            con = simulate_snowflake())
<SQL> LTRIM(RTRIM(`var`))
translate_sql(tolower(var),
           con = simulate_snowflake())
<SQL> LOWER('var')
translate_sql(str_to_lower(var),
         con = simulate_snowflake())
<SQL> LOWER('var')
translate_sql(toupper(var),
       con = simulate_snowflake())
<SQL> UPPER(`var`)
translate_sql(str_to_upper(var),
       con = simulate_snowflake())
<SQL> UPPER('var')
```

```
translate_sql(str_to_title(var),
              con = simulate_snowflake())
<SQL> INITCAP(`var`)
translate_sql(str_trim(var),
              con = simulate_snowflake())
<SQL> TRIM(`var`)
translate_sql(str_squish(var),
              con = simulate_snowflake())
<SQL> REGEXP_REPLACE(TRIM(`var`), '\\s+', ' ')
translate_sql(str_detect(var, "b"),
              con = simulate_snowflake())
Error in `REGEXP_INSTR()`:
! Don't know how to translate `REGEXP_INSTR()`
translate_sql(str_detect(var, "b", negate = TRUE),
             con = simulate_snowflake())
Error in `REGEXP_INSTR()`:
! Don't know how to translate `REGEXP_INSTR()`
translate_sql(str_detect(var, "[aeiou]"),
              con = simulate_snowflake())
Error in `REGEXP_INSTR()`:
! Don't know how to translate `REGEXP_INSTR()`
translate_sql(str_replace(var, "a", "b"),
              con = simulate_snowflake())
<SQL> REGEXP_REPLACE(`var`, 'a', 'b', 1.0, 1.0)
translate_sql(str_replace_all(var, "a", "b"),
             con = simulate_snowflake())
<SQL> REGEXP_REPLACE(`var`, 'a', 'b')
```

```
translate_sql(str_remove(var, "a"),
              con = simulate_snowflake())
<SQL> REGEXP_REPLACE(`var`, 'a', '', 1.0, 1.0)
translate_sql(str_remove_all(var, "a"),
              con = simulate snowflake())
<SQL> REGEXP_REPLACE(`var`, 'a')
translate_sql(str_like(var, "a"),
            con = simulate_snowflake())
<SQL> `var` LIKE 'a'
translate_sql(str_starts(var, "a"),
            con = simulate_snowflake())
Error in `REGEXP_INSTR()`:
! Don't know how to translate `REGEXP_INSTR()`
translate_sql(str_ends(var, "a"),
              con = simulate_snowflake())
Error in `REGEXP_INSTR()`:
! Don't know how to translate `REGEXP_INSTR()`
3.4.5 Spark
translate_sql(nchar(var),
              con = simulate_spark_sql())
<SQL> LENGTH(`var`)
translate_sql(nzchar(var),
              con = simulate_spark_sql())
<SQL> (('var' IS NULL) OR 'var' != '')
translate_sql(substr(var, 1, 2),
              con = simulate_spark_sql())
```

```
<SQL> SUBSTR(`var`, 1, 2)
translate_sql(trimws(var),
              con = simulate_spark_sql())
<SQL> LTRIM(RTRIM(`var`))
translate_sql(tolower(var),
             con = simulate_spark_sql())
<SQL> LOWER('var')
translate_sql(str_to_lower(var),
           con = simulate_spark_sql())
<SQL> LOWER(`var`)
translate_sql(toupper(var),
             con = simulate_spark_sql())
<SQL> UPPER('var')
translate_sql(str_to_upper(var),
             con = simulate_spark_sql())
<SQL> UPPER(`var`)
translate_sql(str_to_title(var),
             con = simulate_spark_sql())
<SQL> INITCAP('var')
translate_sql(str_trim(var),
             con = simulate_spark_sql())
<SQL> LTRIM(RTRIM(`var`))
translate_sql(str_squish(var),
             con = simulate_spark_sql())
Error in `str_squish()`:
! `str_squish()` is not available in this SQL variant.
```

```
translate_sql(str_detect(var, "b"),
              con = simulate_spark_sql())
Error in `str_detect()`:
! Only fixed patterns are supported on database backends.
translate_sql(str_detect(var, "b", negate = TRUE),
              con = simulate_spark_sql())
Error in `str detect()`:
! Only fixed patterns are supported on database backends.
translate_sql(str_detect(var, "[aeiou]"),
              con = simulate_spark_sql())
Error in `str_detect()`:
! Only fixed patterns are supported on database backends.
translate_sql(str_replace(var, "a", "b"),
              con = simulate_spark_sql())
Error in `str_replace()`:
! `str_replace()` is not available in this SQL variant.
translate_sql(str_replace_all(var, "a", "b"),
              con = simulate_spark_sql())
Error in `str_replace_all()`:
! `str_replace_all()` is not available in this SQL variant.
translate_sql(str_remove(var, "a"),
              con = simulate_spark_sql())
Error in `str_remove()`:
! `str_remove()` is not available in this SQL variant.
translate_sql(str_remove_all(var, "a"),
              con = simulate_spark_sql())
Error in `str_remove_all()`:
! `str_remove_all()` is not available in this SQL variant.
```

```
translate_sql(str_like(var, "a"),
              con = simulate_spark_sql())
<SQL> `var` LIKE 'a'
translate_sql(str_starts(var, "a"),
              con = simulate_spark_sql())
Error in `str_starts()`:
! Only fixed patterns are supported on database backends.
translate_sql(str_ends(var, "a"),
              con = simulate_spark_sql())
Error in `str_ends()`:
! Only fixed patterns are supported on database backends.
3.4.6 SQL Server
translate_sql(nchar(var),
              con = simulate_mssql())
<SQL> LEN('var')
translate_sql(nzchar(var),
              con = simulate_mssql())
<SQL> (('var' IS NULL) OR 'var' != '')
translate_sql(substr(var, 1, 2),
              con = simulate_mssql())
<SQL> SUBSTRING(`var`, 1, 2)
translate_sql(trimws(var),
              con = simulate_mssql())
<SQL> LTRIM(RTRIM(`var`))
translate_sql(tolower(var),
            con = simulate_mssql())
```

```
<SQL> LOWER('var')
translate_sql(str_to_lower(var),
              con = simulate mssql())
<SQL> LOWER('var')
translate_sql(toupper(var),
              con = simulate_mssql())
<SQL> UPPER(`var`)
translate_sql(str_to_upper(var),
             con = simulate_mssql())
<SQL> UPPER(`var`)
translate_sql(str_to_title(var),
              con = simulate mssql())
Error in `str_to_title()`:
! `str_to_title()` is not available in this SQL variant.
translate_sql(str_trim(var),
              con = simulate_mssql())
<SQL> LTRIM(RTRIM(`var`))
translate_sql(str_squish(var),
              con = simulate_mssql())
Error in `str_squish()`:
! `str_squish()` is not available in this SQL variant.
translate_sql(str_detect(var, "b"),
              con = simulate mssql())
Error in `str_detect()`:
! Only fixed patterns are supported on database backends.
translate_sql(str_detect(var, "b", negate = TRUE),
           con = simulate_mssql())
```

```
Error in `str_detect()`:
! Only fixed patterns are supported on database backends.
translate_sql(str_detect(var, "[aeiou]"),
              con = simulate_mssql())
Error in `str_detect()`:
! Only fixed patterns are supported on database backends.
translate_sql(str_replace(var, "a", "b"),
              con = simulate_mssql())
Error in `str_replace()`:
! `str_replace()` is not available in this SQL variant.
translate_sql(str_replace_all(var, "a", "b"),
             con = simulate_mssql())
Error in `str_replace_all()`:
! `str_replace_all()` is not available in this SQL variant.
translate_sql(str_remove(var, "a"),
              con = simulate_mssql())
Error in `str_remove()`:
! `str_remove()` is not available in this SQL variant.
translate_sql(str_remove_all(var, "a"),
              con = simulate_mssql())
Error in `str_remove_all()`:
! `str_remove_all()` is not available in this SQL variant.
translate_sql(str_like(var, "a"),
             con = simulate mssql())
<SQL> `var` LIKE 'a'
translate_sql(str_starts(var, "a"),
              con = simulate_mssql())
Error in `str_starts()`:
! 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
translate_sql(get_day(date_1),
              con = simulate_duckdb())
<SQL> DATE_PART('day', `date_1`)
translate_sql(get_month(date_1),
              con = simulate_duckdb())
<SQL> DATE_PART('month', `date_1`)
translate_sql(get_year(date_1),
              con = simulate_duckdb())
<SQL> DATE_PART('year', `date_1`)
translate_sql(add_days(date_1, 1),
              con = simulate_duckdb())
<SQL> DATE_ADD(`date_1`, INTERVAL (1.0) day)
translate_sql(add_years(date_1, 1),
              con = simulate_duckdb())
```

```
<SQL> DATE_ADD(`date_1`, INTERVAL (1.0) year)
translate_sql(difftime(date_1, date_2),
              con = simulate duckdb())
Error in `difftime()`:
! Don't know how to translate `difftime()`
translate_sql(date_count_between(date_1, date_2, "day"),
             con = simulate_duckdb())
<SQL> DATEDIFF('day', `date_1`, `date_2`)
translate_sql(date_count_between(date_1, date_2, "year"),
              con = simulate_duckdb())
Error in date_count_between(date_1, date_2, "year"): The only supported value for `precision` of
3.5.2 Redshift
translate_sql(get_day(date_1),
              con = simulate_redshift())
<SQL> DATE_PART('day', `date_1`)
translate_sql(get_month(date_1),
             con = simulate_redshift())
<SQL> DATE PART('month', `date 1`)
translate_sql(get_year(date_1),
             con = simulate_redshift())
<SQL> DATE_PART('year', `date_1`)
translate_sql(add_days(date_1, 1),
              con = simulate_redshift())
<SQL> DATEADD(DAY, 1.0, `date_1`)
translate_sql(add_years(date_1, 1),
             con = simulate_redshift())
```

```
<SQL> DATEADD(YEAR, 1.0, `date_1`)
translate_sql(difftime(date_1, date_2),
              con = simulate_redshift())
<SQL> DATEDIFF(DAY, `date_1`, `date_2`)
translate_sql(date_count_between(date_1, date_2, "day"),
              con = simulate_redshift())
Error in `date_count_between()`:
! Don't know how to translate `date count between()`
translate_sql(date_count_between(date_1, date_2, "year"),
              con = simulate_redshift())
Error in `date_count_between()`:
! Don't know how to translate `date_count_between()`
3.5.3 Postgres
translate_sql(get_day(date_1),
              con = simulate_postgres())
<SQL> DATE_PART('day', `date_1`)
translate_sql(get_month(date_1),
              con = simulate_postgres())
<SQL> DATE_PART('month', `date_1`)
translate_sql(get_year(date_1),
              con = simulate_postgres())
<SQL> DATE_PART('year', `date_1`)
translate_sql(add_days(date_1, 1),
              con = simulate_postgres())
<SQL> (`date_1` + 1.0*INTERVAL'1 day')
translate_sql(add_years(date_1, 1),
              con = simulate_postgres())
```

```
<SQL> (`date_1` + 1.0*INTERVAL'1 year')
translate_sql(difftime(date_1, date_2),
              con = simulate_postgres())
<SQL> (CAST(`date_2` AS DATE) - CAST(`date_1` AS DATE))
translate_sql(date_count_between(date_1, date_2, "day"),
              con = simulate_postgres())
Error in `date count between()`:
! Don't know how to translate `date_count_between()`
translate_sql(date_count_between(date_1, date_2, "year"),
              con = simulate_postgres())
Error in `date_count_between()`:
! Don't know how to translate `date_count_between()`
3.5.4 Snowflake
translate_sql(get_day(date_1),
              con = simulate_snowflake())
<SQL> DATE_PART(DAY, `date_1`)
translate_sql(get_month(date_1),
              con = simulate_snowflake())
<SQL> DATE_PART(MONTH, `date_1`)
translate_sql(get_year(date_1),
              con = simulate_snowflake())
<SQL> DATE_PART(YEAR, `date_1`)
translate_sql(add_days(date_1, 1),
              con = simulate_snowflake())
<SQL> DATEADD(DAY, 1.0, `date_1`)
translate_sql(add_years(date_1, 1),
            con = simulate_snowflake())
```

```
<SQL> DATEADD(YEAR, 1.0, `date_1`)
translate_sql(difftime(date_1, date_2),
              con = simulate_snowflake())
<SQL> DATEDIFF(DAY, `date_1`, `date_2`)
translate_sql(date_count_between(date_1, date_2, "day"),
              con = simulate_snowflake())
Error in `date_count_between()`:
! Don't know how to translate `date_count_between()`
translate_sql(date_count_between(date_1, date_2, "year"),
              con = simulate_snowflake())
Error in `date_count_between()`:
! Don't know how to translate `date_count_between()`
3.5.5 Spark
translate_sql(get_day(date_1),
              con = simulate_spark_sql())
<SQL> DATE PART('DAY', `date 1`)
translate_sql(get_month(date_1),
              con = simulate_spark_sql())
<SQL> DATE_PART('MONTH', `date_1`)
translate_sql(get_year(date_1),
              con = simulate_spark_sql())
<SQL> DATE_PART('YEAR', `date_1`)
translate_sql(add_days(date_1, 1),
              con = simulate_spark_sql())
<SQL> DATE_ADD(`date_1`, 1.0)
translate_sql(add_years(date_1, 1),
              con = simulate_spark_sql())
```

```
<SQL> ADD_MONTHS('`date_1`', 1.0 * 12.0)
translate_sql(difftime(date_1, date_2),
              con = simulate_spark_sql())
<SQL> DATEDIFF(`date_2`, `date_1`)
translate_sql(date_count_between(date_1, date_2, "day"),
              con = simulate_spark_sql())
Error in `date_count_between()`:
! Don't know how to translate `date_count_between()`
translate_sql(date_count_between(date_1, date_2, "year"),
              con = simulate_spark_sql())
Error in `date_count_between()`:
! Don't know how to translate `date_count_between()`
3.5.6 SQL Server
translate_sql(get_day(date_1),
              con = simulate_mssql())
<SQL> DATEPART(DAY, `date_1`)
translate_sql(get_month(date_1),
              con = simulate mssql())
<SQL> DATEPART(MONTH, `date_1`)
translate_sql(get_year(date_1),
              con = simulate_mssql())
<SQL> DATEPART(YEAR, `date_1`)
translate_sql(add_days(date_1, 1),
              con = simulate_mssql())
<SQL> DATEADD(DAY, 1.0, `date_1`)
translate_sql(add_years(date_1, 1),
            con = simulate_mssql())
```

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.

```
<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 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`
3.6.4 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`
3.6.5 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`
3.6.6 SQL Server
```

```
lazy_frame(x = c(1,2), a = "a", 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(*) 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`
```

3.7 Window functions

In the previous section we saw how aggregate functions can be used to perform operations across 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 using summarise().

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() to specify the order of rows. This will translate into a 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 window function operates on each group independently.

```
    Show SQL

3.7.1 duckdb
lazy_frame(x = c(10, 20, 30),
           z = c(1, 2, 3),
          con = simulate_duckdb()) %>%
 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, 30),
          y = c("a", "a", "b"),
           z = c(1, 2, 3),
          con = simulate_duckdb()) %>%
  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

```
lazy_frame(x = c(10, 20, 30),
           z = c(1, 2, 3),
           con = simulate_postgres()) %>%
  window_order(z) |>
  mutate(sum_x = cumsum(x),
         mean x = cummean(x),
         lag_x = lag(x),
         lead_x = lead(x)) \mid >
  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, 30),
           y = c("a", "a", "b"),
           z = c(1, 2, 3),
           con = simulate_postgres()) %>%
  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 redshift
lazy_frame(x = c(10, 20, 30),
          z = c(1, 2, 3),
          con = simulate_redshift()) %>%
 window_order(z) |>
  mutate(sum x = cumsum(x),
        mean_x = cummean(x),
        lag_x = lag(x),
        lead_x = lead(x)) >
show query()
<SQL>
SELECT
  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, 30),
          y = c("a", "a", "b"),
          z = c(1, 2, 3),
          con = simulate_redshift()) %>%
 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`
```

3.7.4 Snowflake

```
lazy_frame(x = c(10, 20, 30),
           z = c(1, 2, 3),
           con = simulate_snowflake()) %>%
  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, 30),
           y = c("a", "a", "b"),
           z = c(1, 2, 3),
           con = simulate_snowflake()) %>%
  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
  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`
```

3.7.5 Spark

```
lazy_frame(x = c(10, 20, 30),
           z = c(1, 2, 3),
           con = simulate_spark_sql()) %>%
 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, 30),
           y = c("a", "a", "b"),
           z = c(1, 2, 3),
           con = simulate_spark_sql()) %>%
  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`)
3.7.6 SQL Server
lazy_frame(x = c(10, 20, 30),
          z = c(1, 2, 3),
          con = simulate_mssql()) %>%
 window_order(z) |>
 mutate(sum x = cumsum(x),
        mean_x = cummean(x),
        lag_x = lag(x),
        lead_x = lead(x)) >
  show query()
<SQL>
SELECT
 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, 30),
          y = c("a", "a", "b"),
          z = c(1, 2, 3),
          con = simulate_mssql()) %>%
 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`
```

3.8 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 the case is when we are interested in summarising 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
lazy_frame(x = c(1,2), con = simulate_duckdb()) %>%
  summarise(median = median(x, na.rm = TRUE)) |>
  show_query()
<SQL>
SELECT MEDIAN(`x`) AS `median`
FROM `df`
lazy_frame(x = c(1,2), con = simulate_duckdb()) %>%
  mutate(median = median(x, na.rm = TRUE)) |>
  show_query()
<SQL>
SELECT `df`.*, MEDIAN(`x`) OVER () AS `median`
FROM `df`
3.8.2 postgres
lazy_frame(x = c(1,2), con = simulate_postgres()) %>%
  summarise(median = median(x, na.rm = TRUE)) |>
  show_query()
SELECT PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY `x`) AS `median`
FROM `df`
```

```
lazy_frame(x = c(1,2), con = simulate_postgres()) %>%
  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 redshift
lazy frame(x = c(1,2), con = simulate redshift()) %>%
  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 = simulate_redshift()) %>%
  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))) .
3.8.4 Snowflake
lazy_frame(x = c(1,2), con = simulate_snowflake()) %>%
  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 = simulate_snowflake()) %>%
  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`
3.8.5 Spark
lazy_frame(x = c(1,2), con = simulate_spark_sql()) %>%
  summarise(median = median(x, na.rm = TRUE)) |>
  show_query()
<SQL>
SELECT MEDIAN(`x`) AS `median`
FROM `df`
lazy_frame(x = c(1,2), con = simulate_spark_sql()) %>%
  mutate(median = median(x, na.rm = TRUE)) |>
 show_query()
<SQL>
SELECT `df`.*, MEDIAN(`x`) OVER () AS `median`
FROM `df`
3.8.6 SQL Server
lazy_frame(x = c(1,2), con = simulate_mssql()) %>%
  summarise(median = median(x, na.rm = TRUE)) |>
  show_query()
Error in `median()`:
! Translation of `median()` in `summarise()` is not supported for SQL
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 = simulate_mssql()) %>%
  mutate(median = median(x, na.rm = TRUE)) |>
  show_query()
<SQL>
SELECT
  PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY `x`) OVER () AS `median`
FROM `df`
```

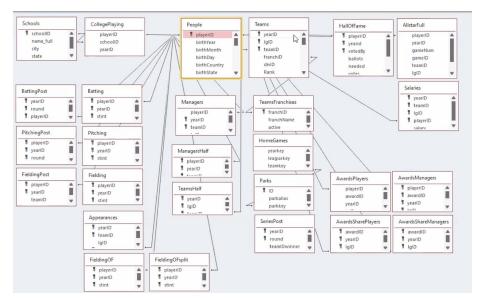
$84CHAPTER\ 3.\ \ SUPPORTED\ EXPRESSIONS\ FOR\ DATABASE\ QUERIES$

Chapter 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've interested in it and write bespoke 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 are often performing.

To see how we can develop a data model with associated methods and functions we'll use the Lahman baseball data. We can see below how the data is stored across various related tables.



4.1 Defining a data model

```
library(DBI)
library(duckdb)
library(dplyr)
library(tidyr)
library(purrr)
library(cli)
library(dbplyr)
library(Lahman)

db <- dbConnect(duckdb(), dbdir = ":memory:")
copy_lahman(db)</pre>
```

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(con, x))
    class(lahmanRef) <- c("lahman_ref", class(lahmanRef))
  lahmanRef
}</pre>
```

With this function we can now easily get references to all our lahman tables in one go using our lahmanFromCon() function.

```
lahman <- lahmanFromCon(db)

lahman$People |>
  glimpse()
```

Rows: ?? Columns: 26

Database: DuckDB v1.2.1 [unknown@Linux 6.11.0-1012-azure:R 4.5.0/:memory:]

```
$ playerID
                              <chr> "aardsda01", "aaronha01", "aaronto01", "aasedo01", "abada~
$ birthYear
                              <int> 1981, 1934, 1939, 1954, 1972, 1985, 1850, 1877, 1869, 186~
                              <int> 12, 2, 8, 9, 8, 12, 11, 4, 11, 10, 9, 3, 10, 2, 8, 9, 6, ~
$ birthMonth
                              <int> 27, 5, 5, 8, 25, 17, 4, 15, 11, 14, 20, 16, 22, 16, 17, 1~
$ birthDay
                              <chr> "Denver", "Mobile", "Mobile", "Orange", "Palm Beach", "La~
$ birthCity
$ birthCountry <chr> "USA", "USA", "USA", "USA", "USA", "D.R.", "USA", "USA", ~
                              <chr> "CO", "AL", "AL", "CA", "FL", "La Romana", "PA", "PA", "V~
$ birthState
                              <int> NA, 2021, 1984, NA, NA, NA, 1905, 1957, 1962, 1926, NA, 1~
$ deathYear
                              <int> NA, 1, 8, NA, NA, NA, 5, 1, 6, 4, NA, 2, 6, NA, NA, NA, N~
$ deathMonth
                              <int> NA, 22, 16, NA, NA, NA, 17, 6, 11, 27, NA, 13, 11, NA, NA~
$ deathDay
$ deathCountry <chr> NA, "USA", "USA", NA, NA, NA, "USA", 
$ deathState
                              <chr> NA, "GA", "GA", NA, NA, NA, "NJ", "FL", "VT", "CA", NA, "~
                              <chr> NA, "Atlanta", "Atlanta", NA, NA, NA, "Pemberton", "Fort ~
$ deathCity
                              <chr> "David", "Hank", "Tommie", "Don", "Andy", "Fernando", "Jo~
$ nameFirst
                              <chr> "Aardsma", "Aaron", "Aaron", "Aase", "Abad", "Abad", "Aba~
$ nameLast
                              <chr> "David Allan", "Henry Louis", "Tommie Lee", "Donald Willi~
$ nameGiven
$ weight
                              <int> 215, 180, 190, 190, 184, 235, 192, 170, 175, 169, 220, 19~
$ height
                              <int> 75, 72, 75, 75, 73, 74, 72, 71, 71, 68, 74, 71, 70, 78, 7~
                              <fct> R, R, R, R, L, L, R, R, R, L, R, R, R, R, R, R, L, R, L, L, ~
$ bats
                              <fct> R, R, R, R, L, L, R, R, R, L, R, R, R, R, R, L, L, R, R, ~
$ throws
                              <chr> "2004-04-06", "1954-04-13", "1962-04-10", "1977-07-26", "~
$ debut
                              <chr> "aardsda01", "aaronha01", "aaronto01", "aasedo01", "abada~
$ bbrefID
                              <chr> "2015-08-23", "1976-10-03", "1971-09-26", "1990-10-03", "~
$ finalGame
                              <chr> "aardd001", "aaroh101", "aarot101", "aased001", "abada001~
$ retroID
                              <date> NA, 2021-01-22, 1984-08-16, NA, NA, NA, 1905-05-17, 1957~
$ deathDate
$ birthDate
                              <date> 1981-12-27, 1934-02-05, 1939-08-05, 1954-09-08, 1972-08-~
```

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.

```
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(people, playerID) %>%
  dm_add_fk(batting, playerID, people)
lahman dm
-- Table source -----
src: DuckDB v1.2.1 [unknown@Linux 6.11.0-1012-azure:R 4.5.0/:memory:]
-- Metadata ------
Tables: `batting`, `people`
Columns: 48
Primary keys: 1
Foreign keys: 1
dm_examine_constraints(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 = join_by(playerID)
) |>
    select(
        "teamID",
        "yearID"
) |>
    distinct() |>
    left_join(lahman$Teams,
        by = join_by(teamID, yearID)
) |>
    select("name") |>
    distinct()
}
```

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, "Babe Ruth")

```
SQL [?? x 1]
# Source:
# Database: DuckDB v1.2.1 [unknown@Linux 6.11.0-1012-azure:R 4.5.0/:memory:]
 name
  <chr>
1 Boston Braves
2 Boston Red Sox
3 New York Yankees
  i Show query
  <SQL>
  SELECT DISTINCT q01.*
  FROM (
   SELECT "name"
   FROM (
     SELECT DISTINCT q01.*
        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'))
          ON (Batting.playerID = RHS.playerID)
      ) q01
    ) LHS
   LEFT JOIN Teams
     ON (LHS.teamID = Teams.teamID AND LHS.yearID = Teams.yearID)
 ) q01
getTeams(lahman, "Barry Bonds")
# Source:
           SQL [?? x 1]
# Database: DuckDB v1.2.1 [unknown@Linux 6.11.0-1012-azure:R 4.5.0/:memory:]
 name
  <chr>
1 Pittsburgh Pirates
2 San Francisco Giants
  i 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'))
          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 that a data frame.

```
getTeams(lahman, "Barry Bonds") |>
  collect()

# A tibble: 2 x 1
  name
  <chr>
1 Pittsburgh Pirates
2 San Francisco Giants

getTeams(lahman, "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 the a player's year of birth to a table.

```
addBirthCountry <- function(lahmanTbl){
  lahmanTbl |>
   left_join(lahman$People |>
        select("playerID", "birthCountry"),
        join_by("playerID"))
}
```

```
lahman$Batting |>
  addBirthCountry()
# Source:
           SQL [?? x 23]
# Database: DuckDB v1.2.1 [unknown@Linux 6.11.0-1012-azure:R 4.5.0/:memory:]
  playerID yearID stint teamID lgID
                                         G
                                              AB
                                                     R
  <chr>
             <int> <int> <fct>
                               <fct> <int> <int> <int> <int> <int> <int> <int> <int>
1 aardsda01
              2004
                       1 SFN
                                               0
                                                     0
                                                           0
                                                                0
                                                                      0
                                        11
                                                                            0
                       1 CHN
2 aardsda01 2006
                               NL
                                        45
                                               2
                                                     0
                                                           0
                                                                0
                                                                      0
                                                                            0
3 aardsda01 2007
                     1 CHA
                                        25
                               AL
                                               0
                                                     0
                                                           0
                                                                            0
4 aardsda01
            2008
                      1 BOS
                               AL
                                        47
                                                     0
                                                           0
                                                                0
                                                                      0
                                                                            0
                                               1
             2009
5 aardsda01
                     1 SEA
                                        73
                                                     0
                                                                0
                               AL
                                               0
                                                           0
                                                                            0
                                                     0
6 aardsda01
            2010
                     1 SEA
                               AL
                                        53
                                               0
                                                           0
                                                                0
                                                                            0
                     1 NYA
7 aardsda01 2012
                               AL
                                        1
                                               0
                                                     0
                                                           0
                                                                0
                                                                      0
                                                                            0
8 aardsda01
              2013
                     1 NYN
                               NL
                                                                      0
                                                                            0
                                        43
                                               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
                                                         131
                                                                           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>
  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 v1.2.1 [unknown@Linux 6.11.0-1012-azure:R 4.5.0/:memory:]
  playerID yearID stint teamID lgID
                                         W
                                               L
                                                     G
                                                          GS
                                                                CG
                                                                    SHO
                                                                           SV
             <chr>
1 aardsda01
              2004
                       1 SFN
                                                           0
                                                                0
                                                                      0
                                         1
                                               0
                                                    11
                                                                            0
2 aardsda01 2006
                       1 CHN
                               NL
                                         3
                                                    45
                                                           0
                                                                      0
                                                                            0
                                               0
                       1 CHA
3 aardsda01 2007
                               AL
                                         2
                                                    25
                                                           0
                                                                      0
                                                                            0
                                               1
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
                     1 SEA
                                                                      0
6 aardsda01 2010
                               AL
                                         0
                                                    53
                                                           0
                                                                0
                                                                           31
```

0

AL

0

1

0

7 aardsda01 2012 1 NYA

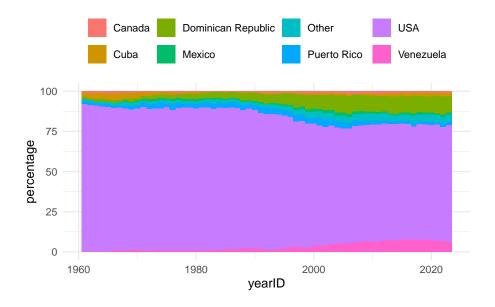
```
8 aardsda01
              2013
                       1 NYN
                                 NL
                                                     43
                                                            0
                                                                              0
 9 aardsda01
               2015
                       1 ATL
                                NL
                                          1
                                                 1
                                                      33
                                                            0
                                                                               0
                       1 BOS
                                          6
                                                     13
                                                           13
                                                                               0
10 aasedo01
              1977
                                 AL
# 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>
```

```
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"
  )) |>
  summarise(n = n(), .by = c("yearID", "birthCountry")) |>
  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 A$ percentage
  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)
   WHERE (yearID > 1960.0)
  GROUP BY yearID, birthCountry
) q01
```



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)
}</pre>
```

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 summarise average salary by birth country and birth year.

```
# Source:
            SQL [?? x 3]
# Database: DuckDB v1.2.1 [unknown@Linux 6.11.0-1012-azure:R 4.5.0/:memory:]
   birthCountry birthYear average_salary
   <chr>
                    <int>
                                    <dbl>
 1 USA
                                1030321.
                     1960
 2 USA
                     1952
                                 498378.
 3 USA
                     1956
                                 986760.
 4 USA
                     1961
                                  811250.
 5 USA
                                  625076.
                     1950
 6 Nicaragua
                     1954
                                2083440.
 7 Panama
                     1945
                                 875000
 8 CAN
                     1961
                                1080292.
9 Venezuela
                     1948
                                  632500
10 Cuba
                     1942
                                  250000
# 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.

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"),
        join_by("playerID"))
```

```
}
lahman$Salaries |>
  addCharacteristics() |>
  summarise(average_salary = mean(salary),
            .by = c("birthCountry", "birthYear"))
# Source:
            SQL [?? x 3]
# Database: DuckDB v1.2.1 [unknown@Linux 6.11.0-1012-azure:R 4.5.0/:memory:]
   birthCountry birthYear average_salary
   <chr>>
                     <int>
 1 D.R.
                     1985
                                 1531438.
 2 USA
                      1966
                                 1761151.
 3 Venezuela
                      1974
                                 4269365.
 4 Cuba
                      1987
                                 4932700.
 5 Panama
                                  555833.
                      1981
 6 USA
                     1978
                                 3133596.
 7 CAN
                     1985
                                  501000
 8 P.R.
                                  297786.
                     1959
9 USA
                                  811250.
                      1961
10 USA
                      1990
                                  728740.
# i more rows
  i Show query
  <SQL>
  SELECT birthCountry, birthYear, AVG(salary) AS average_salary
    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.

```
<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).

```
<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 stike 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)) |>
             select(playerID, schoolID) |>
             distinct(),
           by = join_by(playerID))
lahman$Batting |>
 left_join(players_with_college,
           by = join_by(playerID)) |>
  summarise(home_runs = sum(H, na.rm = TRUE),
                      .by = c(schoolID, birthYear)) |>
 collect()
# A tibble: 6,206 x 3
  schoolID birthYear home_runs
   <chr>
             <int> <dbl>
                1869
1 vermont
                            38
2 longbeach
                1968
                            19
 3 flgateway
                1980
                            86
4 elon
                 1921
                             1
5 swesterntx 1883
                              0
 6 ilparkl
                1970
                            6
7 unc
                 1988
                           518
8 okstate
                  1936
                           1022
                            306
9 ucla
                 1952
10 sprngfldma
               1947
                            452
# i 6,196 more rows
lahman$Pitching |>
 left_join(players_with_college,
           by = join by(playerID)) |>
  summarise(strike_outs = sum(SO, na.rm = TRUE),
                       .by = c(schoolID, birthYear))|>
  collect()
```

```
# A tibble: 3,662 x 3
    schoolID birthYear strike_outs
```

	<chr></chr>	<int></int>	<dbl></dbl>
1	pennst	1981	340
2	cacerri	1971	327
3	usc	1947	275
4	pepperdine	1969	4
5	lsu	1978	162
6	miamidade	1982	56
7	upperiowa	1918	11
8	jamesmad	1966	4
9	ucla	1984	323
10	gonzaga	1988	4
# -	i 3.652 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.

```
i Show query
SELECT schoolID, birthYear, SUM(H) AS home_runs
 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)
    ON (Batting.playerID = RHS.playerID)
GROUP BY schoolID, birthYear
<SQL>
SELECT schoolID, birthYear, SUM(SO) AS strike_outs
 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.

```
# A tibble: 6,206 x 3
  schoolID birthYear home_runs
  <chr>
           <int> <dbl>
1 kentucky
              1972
                       157
2 longbeach
              1968
                        19
3 elon
               1921
                          1
4 lehigh
              1901
                          1
5 ucla
              1952
                        306
6 usc
              1947
                         11
```

```
7 tamukvill
               1978
                             0
8 stanford
                 1972
                             55
9 lsu
                1927
                         1832
10 wake
                1915
                           72
# i 6,196 more rows
lahman$Pitching |>
 left_join(players_with_college,
           by = join_by(playerID)) |>
  summarise(strike_outs = sum(SO, na.rm = TRUE),
                       .by = c(schoolID, birthYear))|>
 collect()
# A tibble: 3,662 x 3
```

schoolID birthYear strike_outs <int> <dbl> <chr> 1968 1 longbeach 273 1921 1901 1947 2 elon 13 3 lehigh 1 4 usc 275 5 tamukvill 1978 6 stanford 1972 7 upenn 1964 8 arkansas 1962 409 218 14 537 9 kentucky 1985 91 571 10 txsjjcn 1983 # i 3,652 more rows

```
i Show query

<SQL>
SELECT schoolID, birthYear, SUM(H) AS home_runs
FROM (
    SELECT Batting.*, birthYear, schoolID
    FROM Batting
    LEFT JOIN dbplyr_tTKjic6s1y
        ON (Batting.playerID = dbplyr_tTKjic6s1y.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_tTKjic6s1y
    ON (Pitching.playerID = dbplyr_tTKjic6s1y.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.

Part II

Working with the OMOP CDM from R

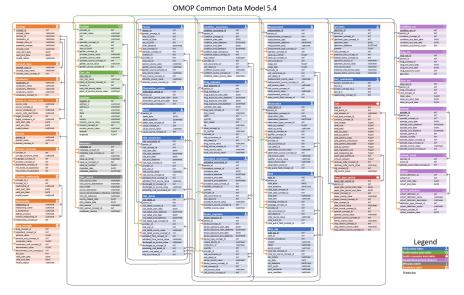
In this second half of the book we will see how we can work with data in the OMOP CDM format from R.

- 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 **?@secomop_person_obs_period** we will see more on how these tables can be used as the starting point for identifying your study participants.
- The OMOP CDM standarises the content of health care data via the OMOP CDM vocabulary tables, which provides a set of standard concepts to represent different clinical events. The vocabulary tables are described in **?@sec-omop_vocabularies**, with these tables playing a fundamental role when we identify the clinical events of interest for our study.
- Clinical records associated with individuals are spread across various OMOP CDM tables, covering various domains. In ?@secomop_clinical_tables we will see how these tables represent events and link back to the person and vocabulary tables.

Creating a CDM reference

5.1 The OMOP common data model (CDM) layout

The OMOP CDM standardises the structure of healthcare data. Data is stored across a system of tables with established relationships between them. In other words, the OMOP CDM provides a relational database structure, with version 5.4 of the OMOP CDM shown below.



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 a 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.

```
library(DBI)
library(duckdb)
library(here)
library(dplyr)
library(omock)
library(omopgenerics)
library(CDMConnector)
library(palmerpenguins)
cdm local <- mockCdmReference() |>
    mockPerson(nPerson = 100) |>
    mockObservationPeriod() |>
    mockConditionOccurrence() |>
    mockDrugExposure() |>
    mockObservation() |>
    mockMeasurement() |>
    mockVisitOccurrence() |>
    mockProcedureOccurrence()
con <- dbConnect(drv = duckdb())</pre>
cdm <- insertCdmTo(cdm = cdm local,</pre>
                    to = dbSource(con = con, writeSchema = "main"))
```

Note that <code>insertCdmTo()</code> output is already a <code><cdm_reference></code> object. But how we would create this cdm reference from the connection? We can use the function <code>cdmFromCon()</code> from <code>CDMConnector</code> 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

```
-- # OMOP CDM reference (duckdb) of example_data -----
```

- * omop tables: person, observation_period, visit_occurrence, condition_occurrence, drug_exposure, procedure_occurrence, measurement, observation, cdm_source, concept, vocabulary, concept_relationship, concept_synonym, concept_ancestor, drug_strength
- * 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.

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 "[[...]]" operators.

```
cdm$person
```

```
# Source: table<person> [?? x 18]
```

Database: DuckDB v1.2.1 [unknown@Linux 6.11.0-1012-azure:R 4.5.0/:memory:]
 person_id gender_concept_id year_of_birth month_of_birth day_of_birth

	<int></int>	<int></int>	<int></int>	<int></int>	<int></int>
1	1	8507	1977	7	30
2	2	8532	1998	3	11
3	3	8532	1983	12	22
4	4	8507	1994	4	17
5	5	8507	1976	2	10
6	6	8507	1973	2	6
7	7	8507	1953	7	11
8	8	8507	1980	4	29
9	9	8532	1982	12	22
10	10	8532	1981	7	17

- # i more rows
- # i 13 more variables: race_concept_id <int>, ethnicity_concept_id <int>,
- # birth_datetime <dttm>, 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 v1.2.1 [unknown@Linux 6.11.0-1012-azure:R 4.5.0/:memory:]
 observation_period_id person_id observation_period_s~1 observation_period_e~2

	<int></int>	<int></int>	<date></date>	<date></date>
1	1	1	1983-11-04	1990-03-16
2	2	2	2018-05-25	2018-10-12
3	3	3	2018-03-30	2019-01-11
4	4	4	2012-06-03	2018-01-14
5	5	5	1983-03-27	2006-01-09
6	6	6	1984-05-25	2019-07-08
7	7	7	1982-12-23	2006-12-11
8	8	8	2016-07-28	2019-09-11
9	9	9	2014-12-05	2016-06-29
10	10	10	1996-03-02	2013-03-14

- # 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 crated 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.

```
cdmName(cdm)
```

[1] "mock"

However, we can instead set this name to whatever else we want when creating our cdm reference.

```
[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 behaviour 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 dbi" "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"

i cdmVersion

Note, the cdmVersion() function also works for <cdm_table> objects:
    cdmVersion(cdm$person)

[1] "5.3"
```

5.4 Including cohort tables in the cdm reference

We'll be seeing how to create cohorts in more detail in **?@sec-creating_cohorts**. For the moment, let's just outline how we can include a 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.

Now we can specify we want to include this existing cohort table to our cdm object when creating our cdm reference.

<date> 1984-08-18, 1984-09-28, 1985-02-03, 1986-11-09, ~

\$ cohort_end_date

5.5 Including achilles tables in the cdm reference

If we have the results tables from the Achilles R 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 table in our cdm reference as in the previous case.

Note we specified the achillesSchema that in this case is the same than the writeSchema, but it can be different.

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 omogenerics (insertTable is also re-exported in CDMConnector). We will show this with the dataset cars in-built in R.

our cdm reference and also added to our reference.

```
cdm
```

```
-- # OMOP CDM reference (duckdb) of example_data -----

* omop tables: person, observation_period, visit_occurrence,
condition_occurrence, drug_exposure, procedure_occurrence, measurement,
observation, cdm_source, concept, vocabulary, concept_relationship,
concept_synonym, concept_ancestor, drug_strength

* cohort tables: my_study_cohort
```

* achilles tables: achilles_analysis, achilles_results, achilles_results_dist

* other tables: cars

```
cdm$cars
```

```
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.

Once we have this table in the database, we can just assign it to our cdm reference.

```
cdm$penguins <- tbl(src = con, "penguins")
cdm</pre>
```

```
-- # OMOP CDM reference (duckdb) of example_data -----
```

- * omop tables: person, observation_period, visit_occurrence, condition_occurrence, drug_exposure, procedure_occurrence, measurement, observation, cdm_source, concept, vocabulary, concept_relationship, concept_synonym, concept_ancestor, drug_strength
- * cohort tables: my_study_cohort
- * achilles tables: achilles_analysis, achilles_results, achilles_results_dist
- * other tables: cars, penguins

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.

5.7 Mutability of the cdm reference

cdm\$person <- cdm\$person |>

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.

```
filter(year_of_birth == 1970)

cdm$person

# Source: SQL [?? x 18]

# Database: DuckDB v1.2.1 [unknown@Linux 6.11.0-1012-azure:R 4.5.0/:memory:]

# i 18 variables: person_id <int>, gender_concept_id <int>,

# year_of_birth <int>, month_of_birth <int>, day_of_birth <int>,

# race_concept_id <int>, ethnicity_concept_id <int>, birth_datetime <dttm>,

# 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>,

#
```

From now on, when we work with our cdm reference this restriction will continue to have been applied.

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.

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:

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: person, observation_period, visit_occurrence,
condition_occurrence, drug_exposure, procedure_occurrence, measurement,
observation, cdm_source, concept, vocabulary, concept_relationship,
concept_synonym, concept_ancestor, drug_strength

* cohort tables: -

* achilles tables: -
```

The package omopgenerics provides a comprehensive list of the required features of a valid cdm reference. You can read more about it here.

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()
cdm$person_new_permanent <- cdm$person |>
  head(5) >
  compute(name = "person_new_permanent",
          temporary = FALSE)
cdm
cdm$person_new_temp
            table<og_001_1746263078> [?? x 18]
# Source:
# Database: DuckDB v1.2.1 [unknown@Linux 6.11.0-1012-azure:R 4.5.0/:memory:]
  person_id gender_concept_id year_of_birth month_of_birth day_of_birth
      <int>
                        <int>
                                      <int>
                                                      <int>
                         8507
                                        1977
                                                          7
                                                                      30
1
          1
2
          2
                         8532
                                        1998
                                                          3
                                                                      11
3
          3
                         8532
                                        1983
                                                         12
                                                                      22
                         8507
                                        1994
                                                          4
                                                                      17
          5
                         8507
                                        1976
                                                          2
                                                                      10
5
# i 13 more variables: race_concept_id <int>, ethnicity_concept_id <int>,
   birth_datetime <dttm>, location_id <int>, provider_id <int>,
   care_site_id <int>, person_source_value <chr>, gender_source_value <chr>,
#
    gender_source_concept_id <int>, race_source_value <chr>,
#
   race_source_concept_id <int>, ethnicity_source_value <chr>,
    ethnicity_source_concept_id <int>
cdm$person_new_permanent
# Source:
            table<person_new_permanent> [?? x 18]
# Database: DuckDB v1.2.1 [unknown@Linux 6.11.0-1012-azure:R 4.5.0/:memory:]
  person_id gender_concept_id year_of_birth month_of_birth day_of_birth
      <int>
                        <int>
                                       <int>
                                                      <int>
                                                                   <int>
                                                          7
                                                                      30
          1
                         8507
                                        1977
1
          2
                                        1998
                                                          3
2
                         8532
                                                                      11
3
          3
                         8532
                                        1983
                                                         12
                                                                      22
                                                          4
4
          4
                         8507
                                        1994
                                                                      17
                                                          2
5
          5
                         8507
                                        1976
                                                                      10
# i 13 more variables: race_concept_id <int>, ethnicity_concept_id <int>,
   birth datetime <dttm>, 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 over 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.

Disconnecting

Once we have finished our analysis we can close our connection to the database behind our cdm reference.

cdmDisconnect(cdm)

Further reading

- omopgenerics package
- CDMConnector package

Exploring the OMOP CDM

For this chapter, we'll use a synthetic Covid-19 dataset.

^{-- #} OMOP CDM reference (duckdb) of Synthea -----

^{*} omop tables: person, observation_period, visit_occurrence, visit_detail, condition_occurrence, drug_exposure, procedure_occurrence, device_exposure, measurement, observation, death, note, note_nlp, specimen, fact_relationship, location, care_site, provider, payer_plan_period, cost, drug_era, dose_era, condition_era, metadata, cdm_source, concept, vocabulary, domain, concept_class, concept_relationship, relationship, concept_synonym, concept_ancestor, source_to_concept_map, drug_strength, cohort_definition, attribute_definition

1

2

```
* cohort tables: -
* achilles tables: -
* other tables: -
```

8.1 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

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.

5165

5589

8532 FEMALE

8507 MALE

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 it's domain and vocabulary id.

```
cdm$concept |>
  glimpse()
```

```
Rows: ??
Columns: 10
```

Database: DuckDB v1.2.1 [unknown@Linux 6.11.0-1012-azure:R 4.5.0//tmp/RtmpCMm7xk/file1fa31cc39f

\$ vocabulary_id <chr> "ABMS", "ABMS",

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
```

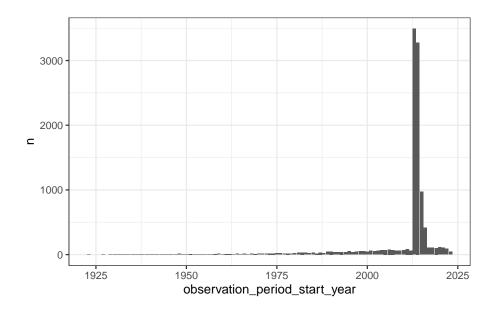
Database: DuckDB v1.2.1 [unknown@Linux 6.11.0-1012-azure:R 4.5.0//tmp/RtmpCMm7xk/file1fa31cc39f

\$ relationship_id <chr> "Has modality", "Has accepted use", "Is current in", ~

```
cdm$concept_ancestor |>
  glimpse()
```

8.2 Summarising observation periods

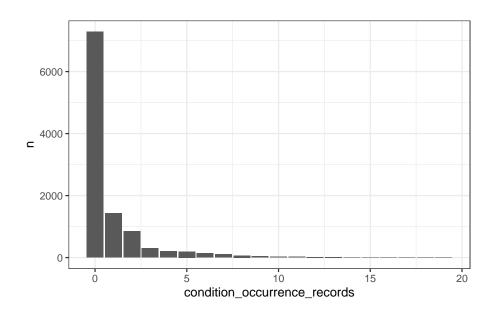
The observation period table contains records indicating spans of time over which clinical events can be reliably observed for the people in the person table. Someone can potentially have multiple observation periods. So say we wanted a count of people grouped by the year during which their first observation period started. We could do this like so:



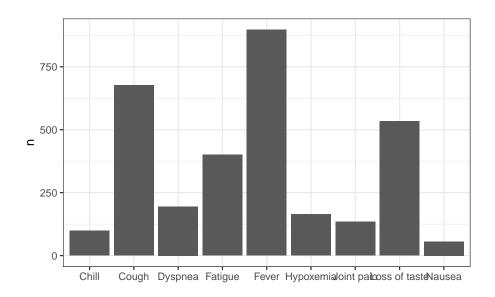
8.3 Summarising clinical records

What's the number of condition occurrence records per person in the database? We can find this out like so

```
cdm$person |>
  left_join(cdm$condition_occurrence |>
  group_by(person_id) |>
  count(name = "condition_occurrence_records"),
  by="person_id") |>
  mutate(condition_occurrence_records = if_else(
    is.na(condition_occurrence_records), 0,
    condition_occurrence_records)) |>
  group_by(condition_occurrence_records) |>
  count() |>
  collect() |>
  ggplot() +
  geom_col(aes(condition_occurrence_records, n)) +
  theme_bw()
```



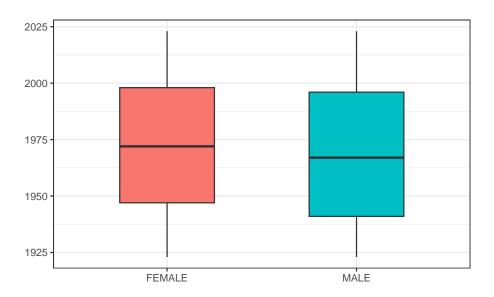
How about we were interested in getting record counts for some specific concepts related to Covid-19 symptoms?



We can also use summarise for various other calculations

As we've seen before, we can also quickly get results for various groupings or restrictions

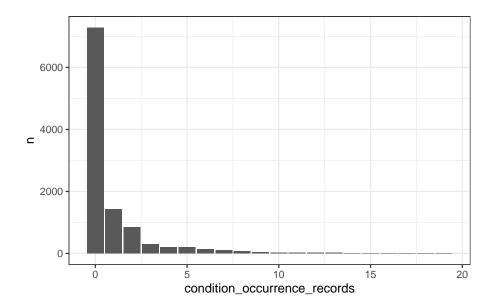
```
grouped_summary <- cdm$person |>
   group_by(gender_concept_id) |>
   summarise(min_year_of_birth = min(year_of_birth, na.rm=TRUE),
            q25_year_of_birth = quantile(year_of_birth, 0.25, na.rm=TRUE),
            median_year_of_birth = median(year_of_birth, na.rm=TRUE),
            q75_year_of_birth = quantile(year_of_birth, 0.75, na.rm=TRUE),
            max_year_of_birth = max(year_of_birth, na.rm=TRUE)) |>
 left_join(cdm$concept,
            by=c("gender_concept_id" = "concept_id")) |>
   collect()
grouped_summary |>
  ggplot(aes(x = concept_name, group = concept_name,
             fill = concept name)) +
  geom_boxplot(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("")
```



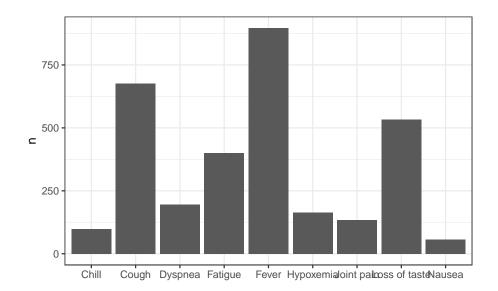
What's the number of condition occurrence records per person in the database?

We can find this out like so

```
cdm$person |>
  left_join(cdm$condition_occurrence |>
  group_by(person_id) |>
  count(name = "condition_occurrence_records"),
  by="person_id") |>
  mutate(condition_occurrence_records = if_else(
    is.na(condition_occurrence_records), 0,
    condition_occurrence_records)) |>
  group_by(condition_occurrence_records) |>
  count() |>
  collect() |>
  ggplot() +
  geom_col(aes(condition_occurrence_records, n)) +
  theme_bw()
```



How about we were interested in getting record counts for some specific concepts related to Covid-19 symptoms?



Identifying patient characteristics

For this chapter, we'll again use our example COVID-19 dataset.

```
library(DBI)
library(duckdb)
library(dbplyr)
library(dplyr)
library(here)
library(CDMConnector)
library(PatientProfiles)
library(ggplot2)
```

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 (ie a characteristic that does not change as time passes and a person ages) or time-varying.¹

¹In some datasets characteristics that could conceptually be considered as time-varying are encoded as time-invariant. One example for 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.

Rows: ??

9.1 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 are interested in individuals' age and sex at time of diagnosis with COVID-19. We can add these variables to the table like so (noting that because age is time-varying, we have to specify the variable with the date for which we want to calculate age relative to).

```
cdm$condition_occurrence <- cdm$condition_occurrence |>
   addSex() |>
   addAge(indexDate = "condition_start_date")

cdm$condition_occurrence |>
   glimpse()
```

```
Columns: 18
Database: DuckDB v1.2.1 [unknown@Linux 6.11.0-1012-azure:R 4.5.0//tmp/RtmpiyseqA/file1
$ 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~
                           <date> 1986-09-08, 2021-06-23, 2021-04-07, 202~
$ condition_start_date
$ condition_start_datetime
                           <dttm> 1986-09-08, 2021-06-23, 2021-04-07, 202~
$ condition_end_date
                           <date> 1986-09-08, 2021-06-23, 2021-04-07, 202~
                           <dttm> 1986-09-08, 2021-06-23, 2021-04-07, 202~
$ condition_end_datetime
                           <int> 38000175, 38000175, 38000175, 38000175, ~
$ condition_type_concept_id
$ condition_status_concept_id
                           $ stop_reason
                           $ provider_id
$ visit_occurrence_id
                           <int> 19, 55, 67, 79, 79, 79, 79, 168, 171, 19~
                           <int> 1000019, 1000055, 1000067, 1000079, 1000~
$ visit_detail_id
$ condition_source_value
                           <chr> "230690007", "410429000", "230690007", "~
                           <int> 381316, 321042, 381316, 37311061, 437663~
$ condition source concept id
$ sex
                           <chr> "Female", "Male", "Male", "Male", "Male"~
                           <int> 57, 25, 97, 2, 2, 2, 2, 75, 77, 57, 76, ~
$ age
```

²Although these manipulations can on the face of it seem quite simple, 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 |>
  addSexQuery() |>
  show_query()
```

```
Warning: ! The following columns will be overwritten: sex
<SQL>
SELECT
  condition_occurrence_id,
  og_002_1746263123.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_1746263123
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_1746263123.person_id = RHS.person_id)
```

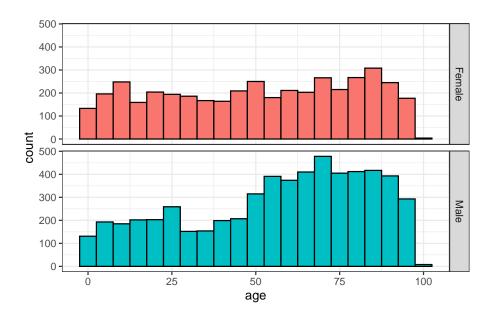
We now have two variables added containing values for age and sex.

```
cdm$condition_occurrence |>
  glimpse()
```

```
Rows: ??
Columns: 18
Database: DuckDB v1.2.1 [unknown@Linux 6.11.0-1012-azure:R 4.5.0//tmp/RtmpiyseqA/file1
$ condition_occurrence_id
                           <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 1~
                           <int> 2, 6, 7, 8, 8, 8, 8, 16, 16, 18, 18, 25,~
$ person id
$ condition_concept_id
                           <int> 381316, 321042, 381316, 37311061, 437663~
                           <date> 1986-09-08, 2021-06-23, 2021-04-07, 202~
$ condition_start_date
$ condition_start_datetime
                           <dttm> 1986-09-08, 2021-06-23, 2021-04-07, 202~
                           <date> 1986-09-08, 2021-06-23, 2021-04-07, 202~
$ condition end date
$ condition_end_datetime
                           <dttm> 1986-09-08, 2021-06-23, 2021-04-07, 202~
$ condition type concept id
                           <int> 38000175, 38000175, 38000175, 38000175, ~
$ condition_status_concept_id
                           $ stop_reason
                           $ provider_id
$ visit_occurrence_id
                           <int> 19, 55, 67, 79, 79, 79, 79, 168, 171, 19~
                           <int> 1000019, 1000055, 1000067, 1000079, 1000~
$ visit_detail_id
$ condition_source_value
                           <chr> "230690007", "410429000", "230690007", "~
$ condition_source_concept_id
                           <int> 381316, 321042, 381316, 37311061, 437663~
<chr> "Female", "Male", "Male", "Male", "Male"~
$ sex
                           <int> 57, 25, 97, 2, 2, 2, 2, 75, 77, 57, 76, ~
$ age
```

And with these now added it is straightforward to calculate mean age at condition start date by sex or even plot the distribution of age at diagnosis by sex.

```
cdm$condition occurrence |>
  summarise(mean age = mean(age, na.rm=TRUE), .by = "sex") |>
  collect()
# A tibble: 2 x 2
         mean_age
            <dbl>
  <chr>
1 Female
             50.8
2 Male
             56.5
cdm$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")
```



9.2 Adding multiple demographics simultaneously

We've now seen individual functions from PatientProfiles to add age and sex, and the package has others to add other characteristics like days of prior observation in the database (rather unimaginatively named addPriorObservation()). In additional to these individuals functions, the package also provides a more general function to get all of these characteristics at the same time.³

```
cdm$drug_exposure <- cdm$drug_exposure |>
  addDemographics(indexDate = "drug_exposure_start_date")

cdm$drug_exposure |>
  glimpse()
```

Rows: ?? Columns: 27

³This function also provides a more time efficient method that 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.

```
$ drug_concept_id
                       <int> 40213260, 40213260, 40213260, 40213260, 4~
                       <date> 2021-04-30, 2020-04-24, 2021-04-30, 2020~
$ drug_exposure_start_date
$ drug_exposure_start_datetime <dttm> 2021-04-30 16:49:39, 2020-04-24 16:49:39~
$ drug_exposure_end_date
                       <date> 2021-04-30, 2020-04-24, 2021-04-30, 2020~
                       <dttm> 2021-04-30 16:49:39, 2020-04-24 16:49:39~
$ drug exposure end datetime
$ verbatim end date
                       <date> 2021-04-30, 2020-04-24, 2021-04-30, 2020~
$ drug_type_concept_id
                       <int> 32869, 32869, 32869, 32869, 32869, 32869,~
$ stop reason
                       $ refills
$ quantity
                       $ days_supply
                       $ sig
                       $ route_concept_id
                       $ lot_number
$ provider_id
                       <int> 12357, 12357, 12356, 12356, 12357, 12356,~
                       <int> 6, 8, 6, 8, 6, 6, 7, 2, 6, 8, 9, 1, 7, 2,~
$ visit_occurrence_id
$ visit_detail_id
                       <int> 1000006, 1000008, 1000006, 1000008, 10000~
                       <chr> "121", "121", "121", "121", "113", "113", "
$ drug_source_value
                       <int> 40213260, 40213260, 40213260, 40213260, 4~
$ drug_source_concept_id
                       $ route_source_value
                       $ dose_unit_source_value
$ age
                       <int> 51, 50, 51, 50, 51, 51, 53, 52, 51, 50, 4~
                       <chr> "Female", "Female", "Female", "Female", "~
$ sex
                       <int> 2548, 2177, 2548, 2177, 2548, 2548, 3290,~
$ prior_observation
$ future_observation
                       <int> 742, 1113, 742, 1113, 742, 742, 0, 371, 7~
```

With these characteristics now all added, we can now calculate mean age, prior observation (how many days have passed since the individual's most recent observation start date), and future observation (how many days until the individual's nearest observation end date) at drug exposure start date by sex.

Returning a query from PatientProfiles rather than the result

In the above examples the functions from PatientProfiles will execute queries with the results written to a table in the database (either temporary if no name is provided or a permanent table if one is given). We might though instead want to to instead just get the underlying query back so that we have more control over how and when the query will be executed.

```
cdm$visit_occurrence |>
 addSex() |>
 filter(sex == "Male") |>
 show_query()
<SQL>
SELECT og_004_1746263125.*
FROM og_004_1746263125
WHERE (sex = 'Male')
cdm$visit_occurrence |>
 addSex(name = "my_new_table") |>
 filter(sex == "Male") |>
 show_query()
<SQL>
SELECT my_new_table.*
FROM 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'
```

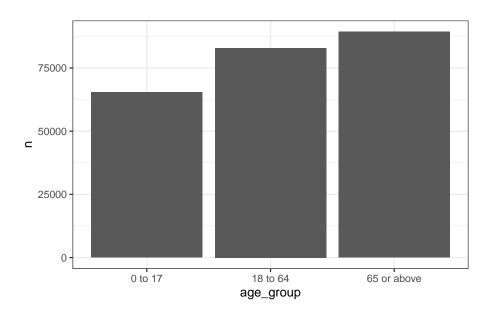
```
WHEN (gender_concept_id = 8532.0) THEN 'Female'
ELSE 'None'
END AS sex
   FROM person
) RHS
   ON (visit_occurrence.person_id = RHS.person_id)
) q01
WHERE (sex = 'Male')
```

9.3 Creating categories

When we add age, either via addAge or addDemographics, we can also add another variable containing age groups. These age groups are specified in a list of vectors, each of which contain the lower and upper bounds.

\$ condition status concept id

\$ stop_reason



PatientProfiles also provides a more general function for adding categories. Can you guess it's 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 v1.2.1 [unknown@Linux 6.11.0-1012-azure:R 4.5.0//tmp/RtmpiyseqA/file1fef1c723cb2
                                <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 1~
$ condition_occurrence_id
$ person_id
                                <int> 2, 6, 7, 8, 8, 8, 8, 16, 16, 18, 18, 25,~
$ condition_concept_id
                                <int> 381316, 321042, 381316, 37311061, 437663~
                                <date> 1986-09-08, 2021-06-23, 2021-04-07, 202~
$ condition_start_date
                                <dttm> 1986-09-08, 2021-06-23, 2021-04-07, 202~
$ condition_start_datetime
                                <date> 1986-09-08, 2021-06-23, 2021-04-07, 202~
$ condition_end_date
                                <dttm> 1986-09-08, 2021-06-23, 2021-04-07, 202~
$ condition_end_datetime
$ condition_type_concept_id
                                <int> 38000175, 38000175, 38000175, 38000175, ~
```

```
$ provider_id
                          $ visit_occurrence_id
                          <int> 19, 55, 67, 79, 79, 79, 79, 168, 171, 19~
$ visit_detail_id
                          <int> 1000019, 1000055, 1000067, 1000079, 1000~
                          <chr> "230690007", "410429000", "230690007", "~
$ condition_source_value
                          <int> 381316, 321042, 381316, 37311061, 437663~
$ condition source concept id
$ sex
                          <chr> "Female", "Male", "Male", "Male", "Male"~
$ age
                          <int> 57, 25, 97, 2, 2, 2, 2, 75, 77, 57, 76, ~
                          <int> 3437, 2898, 2842, 872, 872, 872, 872, 23~
$ prior_observation
$ prior_observation_group
                          <chr> "365 or above", "365 or above", "365 or ~
```

9.4 Adding custom variables

While PatientProfiles provides a range of functions that can help 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 other variables in our table we'll typically be using mutate() (from dplyr package). For example, perhaps we just want to add a new variable to our observation period table containing the year of individuals' 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()
```

The second case is normally a more complex task where adding a new variable involves joining to some other table. This table may well have been created by

some intermediate query that we wrote to derive the variable of interest. For example, lets say we want to add each number of condition occurrence records for each individual to the person table (remember that we saw how to calculate this in the previous chapter). For this we will need to do 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 save the create a table containing just the information we're interested in and compute 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 = if_else(
    is.na(condition_occurrence_records),
    0, condition_occurrence_records) |>
  compute()

condition_summary |>
  glimpse()
```

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 = if_else(
    is.na(condition_occurrence_records),
    O, condition_occurrence_records)) |>
  show_query()
```

```
SELECT
   person_id,
   CASE WHEN ((condition_occurrence_records IS NULL)) THEN 0.0 WHEN NOT ((condition_occurrence))
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_1746263123
        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. If we, for example, have a table with one row per person then a left join to a table with multiple rows per person can then result in a table with multiple rows per person.

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. Remember this is just with our small synthetic data, so 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!

```
# Source:
            SQL [?? x 1]
# Database: DuckDB v1.2.1 [unknown@Linux 6.11.0-1012-azure:R 4.5.0//tmp/RtmpiyseqA/file1fef1c72
   <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 v1.2.1 [unknown@Linux 6.11.0-1012-azure:R 4.5.0//tmp/RtmpiyseqA/file1fef1c72
   <dbl>
1 410683
```

Further reading

• PatientProfiles package

Adding cohorts to the CDM

11.1 What is a cohort?

When performing research with the OMOP common data model 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), 2) subject id (a foreign key to the subject in the cohort - typically referring to records in the person table), 3) cohort start date, and 4) cohort end date. Individuals can enter a cohort multiple times, but the time periods in which they are in the cohort cannot overlap. Individuals will only be considered in a cohort when they have have an ongoing observation period.

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.

11.2 Set up

We'll use our synthetic dataset for demonstrating how cohorts can be constructed.

i more rows

11.3 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.

	<int></int>	<int></int>	<date></date>	<date></date>
1	1	5083	2020-11-29	2020-12-18
2	1	7249	2020-12-09	2021-01-08
3	1	5780	2020-11-20	2020-12-14
4	1	6601	2020-06-07	2020-06-23
5	1	7616	2020-07-30	2020-08-14
6	1	1935	2021-09-03	2021-10-09
7	1	2646	2020-11-20	2020-12-18
8	1	3827	2020-11-10	2020-12-05
9	1	6230	2020-10-31	2020-11-24
10	1	937	2021-01-16	2021-02-11

```
• Finding appropriate codes
```

In the 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 CodelistGenerator::getCandidateCodes(). For example, we can have found the code we used above (and many others) like so:

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>
     3661406 From initia~ Acute respi~ Condition SNOMED
                                                                S
1
                                                                S
      439676 From initia~ Coronavirus~ Condition SNOMED
                                                                S
     3655973 From initia~ At increase~ Condition SNOMED
 3
                                                                S
    40479642 From initia~ Pneumonia d~ Condition SNOMED
5
     3656668 From initia~ Conjunctivi~ Condition SNOMED
                                                                S
                                                                S
    37310254 From initia~ Otitis medi~ Condition SNOMED
 7
    37310287 From initia~ Myocarditis~ Condition SNOMED
                                                                S
                                                                S
    37310283 From initia~ Gastroenter~ Condition SNOMED
                                                                S
9
     3661631 From initia~ Lymphocytop~ Condition SNOMED
      3661748 From initia~ Acute kidne~ 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 it's 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()
```

Second, we can get counts of the cohort.

```
cohortCount(cdm$covid) |>
  glimpse()
```

```
Rows: 1
Columns: 3
$ cohort_definition_id <int> 1
$ number_records <int> 964
$ number_subjects <int> 964
```

And last we can see attrition related to the cohort.

```
attrition(cdm$covid) |>
  glimpse()
```

As we will see below these attributes of the cohorts become particularly useful as we apply further restrictions on our cohort.

11.4 Applying inclusion criteria

11.4.1 Only include first cohort entry per person

Let's say we first want to restrict to first entry.

```
cdm$covid <- cdm$covid |>
    requireIsFirstEntry()
```

11.4.2 Restrict to study period

```
cdm$covid <- cdm$covid |>
  requireInDateRange(dateRange = c(as.Date("2020-09-01"), NA))
```

11.4.3 Applying demographic inclusion criteria

Say for our study we want to include people with a GI bleed who were aged 40 or over at the time. We can use the add variables with these characteristics as seen in chapter 4 and then filter accordingly. The function CDMConnector::record_cohort_attrition() will then update our cohort attributes as we can see below.

```
cdm$covid <- cdm$covid |>
  requireDemographics(ageRange = c(18, 64), sex = "Male")
```

11.4.4 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,
  list("myocaridal_infarction" = c(</pre>
```

```
317576, 313217, 321042, 4329847
)),
name = "cardiac"
)
cdm$cardiac
```

- # Source: table<cardiac> [?? x 4]
- # Database: DuckDB v1.2.1 [unknown@Linux 6.11.0-1012-azure:R 4.5.0//tmp/Rtmpd54jDp/filcohort_definition_id subject_id cohort_start_date cohort_end_date

	<int></int>	<int></int>	<date></date>	<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.

Note if we had wanted to have required that individuals did have a history of a cardiac condition we would instead have set intersections = c(1, Inf) above.

11.5 Cohort attributes

We can see that the attributes of the cohort were updated as we applied the inclusion criteria.

```
settings(cdm$covid) |>
glimpse()
```

\$ excluded_records

\$ excluded_subjects

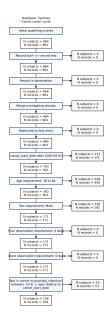
```
Rows: 1
Columns: 8
$ cohort_definition_id <int> 1
                         <chr> "covid"
$ cohort_name
                         <chr> "18 64"
$ age_range
$ sex
                         <chr> "Male"
$ min_prior_observation <dbl> 0
$ min_future_observation <dbl> 0
                         <chr> "5.3"
$ cdm_version
$ vocabulary_version
                       <chr> "v5.0 22-JUN-22"
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: 11
Columns: 7
$ cohort_definition_id <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
                       <int> 964, 964, 964, 964, 964, 793, 363, 171, 171, 171,~
$ number_records
$ number_subjects
                       <int> 964, 964, 964, 964, 964, 793, 363, 171, 171, 171,~
$ reason id
                       <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
                       <chr> "Initial qualifying events", "Record start <= rec~
$ reason
```

For attrition, we can use CohortConstructor::summariseCohortAttrition() and then CohortConstructor::tableCohortAttrition() to better view the impact of applying the additional inclusion criteria.

<int> 0, 0, 0, 0, 171, 430, 192, 0, 0, 13

<int> 0, 0, 0, 0, 0, 171, 430, 192, 0, 0, 13

```
attrition_summary <- summariseCohortAttrition(cdm$covid)
plotCohortAttrition(attrition_summary, type = 'png')</pre>
```



Further reading

• ...

Working with cohorts

13.1 Cohort intersections

PatientProfiles::addCohortIntersect()

13.2 Intersection between two cohorts

13.3 Set up

```
library(CDMConnector)
library(dplyr)
library(PatientProfiles)
# For this example we will use GiBleed data set
downloadEunomiaData(datasetName = "GiBleed")
db <- DBI::dbConnect(duckdb::duckdb(), eunomiaDir())</pre>
cdm <- cdmFromCon(db, cdmSchema = "main", writeSchema = "main")</pre>
# cdm <- cdm |>
    generate_concept_cohort_set(concept_set = list("gi_bleed" = 192671),
                              limit = "all",
#
                              end = 30,
#
                              name = "gi_bleed",
                              overwrite = TRUE) |>
    generate_concept_cohort_set(concept_set = list("acetaminophen" = c(1125315,
```

```
#
                                                                   1127078,
                                                                   1127433,
#
#
                                                                   40229134,
#
                                                                   40231925,
                                                                   40162522,
#
                                                                   19133768)),
                                 limit = "all",
#
                               # end = "event_end_date",
                               name = "acetaminophen",
                               overwrite = TRUE)
```

13.3.1 Flag

```
# cdm$gi_bleed <- cdm$gi_bleed |>
# addCohortIntersectFlag(targetCohortTable = "acetaminophen",
# window = list(c(-Inf, -1), c(0,0), c(1, Inf)))
#
# cdm$gi_bleed |>
# summarise(acetaminophen_prior = sum(acetaminophen_minf_to_m1),
# acetaminophen_index = sum(acetaminophen_0_to_0),
# acetaminophen_post = sum(acetaminophen_1_to_inf)) |>
# collect()
```

13.3.2 Count

13.3.3 Date and times

13.4 Intersection between a cohort and tables with patient data

Further reading

• ...