# SQL databases and R

### Data Carpentry contributors

## Learning Objectives

* Access a database from R.
* Run SQL queries in R using **RSQLite** and **dplyr**.
* Describe the lazy behavior of dplyr on data stored in a database outside of R.
* Prototype queries and retrieve all final results.
* Create complex queries across one or multiple database tables.
* Create an SQLite database from existing .csv files.

## Introduction

So far, we have dealt with small datasets that easily fit into your computer’s memory. But what about datasets that are too large for your computer to handle as a whole? In this case, storing the data outside of R and organizing it in a database is helpful. Connecting to the database allows you to retrieve only the chunks needed for the current analysis.

Even better, many large datasets are already available in public or private databases. You can query them without having to download the data first.

R can connect to almost any existing database type. Most common database types have R packages that allow you to connect to them (e.g., **RSQLite**, RMySQL, etc). Furthermore, the **[dplyr](https://cran.r-project.org/web/packages/dplyr/index.html)** package you used in the previous chapter, in conjunction with **[dbplyr](https://cran.r-project.org/package=dbplyr)** supports connecting to the widely-used open source databases [sqlite](https://sqlite.org/), [mysql](https://www.mysql.com/) and [postgresql](https://www.postgresql.org/), as well as Google’s [bigquery](https://cloud.google.com/bigquery/), and it can also be extended to other database types (a [vignette](https://cran.r-project.org/web/packages/dbplyr/vignettes/new-backend.html) in the **dplyr** package explains how to do it). RStudio has created [a website](http://db.rstudio.com/) that provides documentation and best practices to work on database interfaces.

Interfacing with databases using **dplyr** focuses on retrieving and analyzing datasets by generating SELECT SQL statements, but it doesn’t modify the database itself. **dplyr** does not offer functions to UPDATE or DELETE entries. If you need these functionalities, you will need to use additional R packages (e.g., **RSQLite**). Here we will demonstrate how to interact with a database using **dplyr**, using both the **dplyr**’s verb syntax and the SQL syntax.

## The portal\_mammals database

We will continue to explore the surveys data you are already familiar with from previous lessons. First, we are going to install the **dbplyr** package:

**install.packages**(**c**("dbplyr", "RSQLite"))

The SQLite database is contained in a single file portal\_mammals.sqlite that you generated during [the SQL lesson](https://datacarpentry.org/sql-ecology-lesson/00-sql-introduction/index.html). If you don’t have it, you can download it from Figshare into the data\_raw subdirectory using:

**dir.create**("data\_raw", showWarnings = FALSE)

**download.file**(url = "https://ndownloader.figshare.com/files/2292171",

destfile = "data\_raw/portal\_mammals.sqlite", mode = "wb")

## Connecting to databases

We can point R to this database using:

**library**(dplyr)

**library**(dbplyr)

#>

#> Attaching package: 'dbplyr'

#> The following objects are masked from 'package:dplyr':

#>

#> ident, sql

mammals <- DBI::**dbConnect**(RSQLite::**SQLite**(), "data\_raw/portal\_mammals.sqlite")

This command uses 2 packages that helps **dbplyr** and **dplyr** talk to the SQLite database. **DBI** is not something that you’ll use directly as a user. It allows R to send commands to databases irrespective of the database management system used. The **RSQLite** package allows R to interface with SQLite databases.

This command does not load the data into the R session (as the read\_csv() function did). Instead, it merely instructs R to connect to the SQLite database contained in the portal\_mammals.sqlite file.

Using a similar approach, you could connect to many other database management systems that are supported by R including MySQL, PostgreSQL, BigQuery, etc.

Let’s take a closer look at the mammals database we just connected to:

**src\_dbi**(mammals)

#> src: sqlite 3.35.5 [/home/runner/work/R-ecology-lesson/R-ecology-lesson/data\_raw/portal\_mammals.sqlite]

#> tbls: plots, species, surveys

Just like a spreadsheet with multiple worksheets, a SQLite database can contain multiple tables. In this case three of them are listed in the tbls row in the output above:

* plots
* species
* surveys

Now that we know we can connect to the database, let’s explore how to get the data from its tables into R.

## Querying the database with the SQL syntax

To connect to tables within a database, you can use the tbl() function from **dplyr**. This function can be used to send SQL queries to the database. To demonstrate this functionality, let’s select the columns “year”, “species\_id”, and “plot\_id” from the surveys table:

**tbl**(mammals, **sql**("SELECT year, species\_id, plot\_id FROM surveys"))

With this approach you can use any of the SQL queries we have seen in the database lesson.

## Querying the database with the dplyr syntax

One of the strengths of **dplyr** is that the same operation can be done using **dplyr**’s verbs instead of writing SQL. First, we select the table on which to do the operations by creating the surveys object, and then we use the standard **dplyr** syntax as if it were a data frame:

surveys <- **tbl**(mammals, "surveys")

surveys %>%

**select**(year, species\_id, plot\_id)

In this case, the surveys object behaves like a data frame. Several functions that can be used with data frames can also be used on tables from a database. For instance, the head() function can be used to check the first 10 rows of the table:

head(surveys, n = 10)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| record\_id  <int> | month  <int> | day  <int> | year  <int> | plot\_id  <int> | species\_id  <chr> | sex  <chr> | hindfoot\_length  <int> | weight  <int> |
| 1 | 7 | 16 | 1977 | 2 | NL | M | 32 | *NA* |
| 2 | 7 | 16 | 1977 | 3 | NL | M | 33 | *NA* |
| 3 | 7 | 16 | 1977 | 2 | DM | F | 37 | *NA* |
| 4 | 7 | 16 | 1977 | 7 | DM | M | 36 | *NA* |
| 5 | 7 | 16 | 1977 | 3 | DM | M | 35 | *NA* |
| 6 | 7 | 16 | 1977 | 1 | PF | M | 14 | *NA* |
| 7 | 7 | 16 | 1977 | 2 | PE | F | *NA* | *NA* |
| 8 | 7 | 16 | 1977 | 1 | DM | M | 37 | *NA* |
| 9 | 7 | 16 | 1977 | 1 | DM | F | 34 | *NA* |
| 10 | 7 | 16 | 1977 | 6 | PF | F | 20 | *NA* |

1-10 of 10 rows

This output of the head command looks just like a regular data.frame: The table has 9 columns and the head() command shows us the first 10 rows. Note that the columns plot\_type, taxa, genus, and species are missing. These are now located in the tables plots and species which we will join together in a moment.

However, some functions don’t work quite as expected. For instance, let’s check how many rows there are in total using nrow():

**nrow**(surveys)

#> [1] NA

That’s strange - R doesn’t know how many rows the surveys table contains - it returns NA instead. You might have already noticed that the first line of the head() output included ?? indicating that the number of rows wasn’t known.

The reason for this behavior highlights a key difference between using dplyr on datasets in memory (e.g. loaded into your R session via read\_csv()) and those provided by a database. To understand it, we take a closer look at how dplyr communicates with our SQLite database.

## SQL translation

Relational databases typically use a special-purpose language, Structured Query Language (SQL), to manage and query data.

For example, the following SQL query returns the first 10 rows from the surveys table:

SELECT \*

FROM `surveys`

LIMIT 10

Behind the scenes, dplyr:

* translates your R code into SQL
* submits it to the database
* translates the database’s response into an R data frame

To lift the curtain, we can use dplyr’s show\_query() function to show which SQL commands are actually sent to the database:

show\_query(head(surveys, n = 10))

The output shows the actual SQL query sent to the database; it matches our manually constructed SELECT statement above.

Instead of having to formulate the SQL query ourselves - and having to mentally switch back and forth between R and SQL syntax - we can delegate this translation to dplyr. (You don’t even need to know SQL to interact with a database via dplyr!)

dplyr, in turn, doesn’t do the real work of subsetting the table, either. Instead, it merely sends the query to the database, waits for its response and returns it to us.

That way, R never gets to see the full surveys table - and that’s why it could not tell us how many rows it contains. On the bright side, this allows us to work with large datasets - even too large to fit into our computer’s memory.

dplyr can translate many different query types into SQL allowing us to, e.g., select() specific columns, filter() rows, or join tables.

To see this in action, let’s compose a few queries with dplyr.

## Simple database queries

First, let’s only request rows of the surveys table in which weight is less than 5 and keep only the species\_id, sex, and weight columns.

surveys %>%

filter(weight < 5) %>%

select(species\_id, sex, weight)

Executing this command will return a table with 10 rows and the requested species\_id, sex and weight columns. Great!

… but wait, why are there only 10 rows?

The last line:

# ... with more rows

indicates that there are more results that fit our filtering criterion. Why was R lazy and only retrieved 10 of them?