

R, Databases and Docker

*Dipti Muni, Ian Frantz, John David Smith, Mary Anne Thygesen, M. Edward (Ed)
Borasky, Scott Case, and Sophie Yang*

2018-09-20

Contents

1	Introduction	5
1.1	Using R to query a DBMS in your organization	5
1.2	Docker's role	5
1.3	Docker and R on your machine	5
1.4	Who are we?	6
1.5	Prerequisites	6
1.6	Install Docker	6
1.7	Download the repo	7
2	Docker Hosting for Windows	9
2.1	Hardware requirements	9
2.2	Software requirements	9
2.3	Docker for Windows settings	9
2.4	Git, GitHub and line endings	11
3	Learning Goals and Use Cases	13
3.1	Context: Why integrate R with databases using Docker?	13
3.2	Learning Goals	13
3.3	Use cases	13
3.4	Environment	14
4	Docker, Postgres, and R	15
4.1	Verify that Docker running	15
4.2	Clean up if appropriate	16
4.3	Put the database password in an environment file	16
4.4	Connect, read and write to Postgres from R	17
4.5	Clean up	19
5	A persistent database in Postgres in Docker - all at once	21
5.1	Overview	21
5.2	First, verify that Docker is up and running:	21
5.3	Clean up if appropriate	22
5.4	Build the Docker Image	22
5.5	Run the Docker Image	22
5.6	Connect to Postgres with R	23
5.7	Stop and start to demonstrate persistence	24
5.8	Cleaning up	25
5.9	Using the <code>sql-pet</code> container in the rest of the book	25
6	A persistent database in Postgres in Docker - piecemeal	27
6.1	Overview	27
6.2	Retrieve the backup file	27

6.3	Now, verify that Docker is up and running:	28
6.4	Build the Docker Image	28
6.5	Stop and start to demonstrate persistence	30
6.6	Cleaning up	31
6.7	Using the <code>sql-pet</code> container in the rest of the book	31
7	Introduction to interacting with Postgres from R	33
7.1	Basics	33
7.2	Ask yourself about what you are aiming for?	33
7.3	Get some basic information about your database	33
8	Real work with real data	37
8.1	Some extra handy libraries	37
8.2	Basic investigation	37
8.3	Using Dplyr	37
8.4	What is dplyr sending to the server?	38
8.5	Writing your on SQL directly to the DBMS	38
8.6	Chosing between dplyr and native SQL	38
9	Real work with real data	39
9.1	Some extra handy libraries	39
9.2	More topics	39
9.3	Standards for production jobs	39
10	Other resources	41
10.1	Editing this book	41
10.2	Docker alternatives	41
10.3	Docker and R	41
10.4	Documentation Docker and Postgres	41
10.5	More Resources	41

Chapter 1

Introduction

1.1 Using R to query a DBMS in your organization

- Large data stores in organizations are stored in databases that have specific access constraints and structural characteristics. Data documentation may be incomplete, often emphasizes operational issues rather than analytical ones, and often needs to be confirmed on the fly. Data volumes and query performance are important design constraints.
- R users frequently need to make sense of complex data structures and coding schemes to address incompletely formed questions so that exploratory data analysis has to be fast. Exploratory techniques for the purpose should not be reinvented (and so would benefit from more public instruction or discussion).
- Learning to navigate the interfaces (passwords, packages, etc.) between R and a database is difficult to simulate outside corporate walls. Resources for interface problem diagnosis behind corporate walls may or may not address all the issues that R users face, so a simulated environment is needed.

1.2 Docker’s role

Noam Ross’s “Docker for the UseR” suggests that there are four distinct Docker use-cases for useRs.

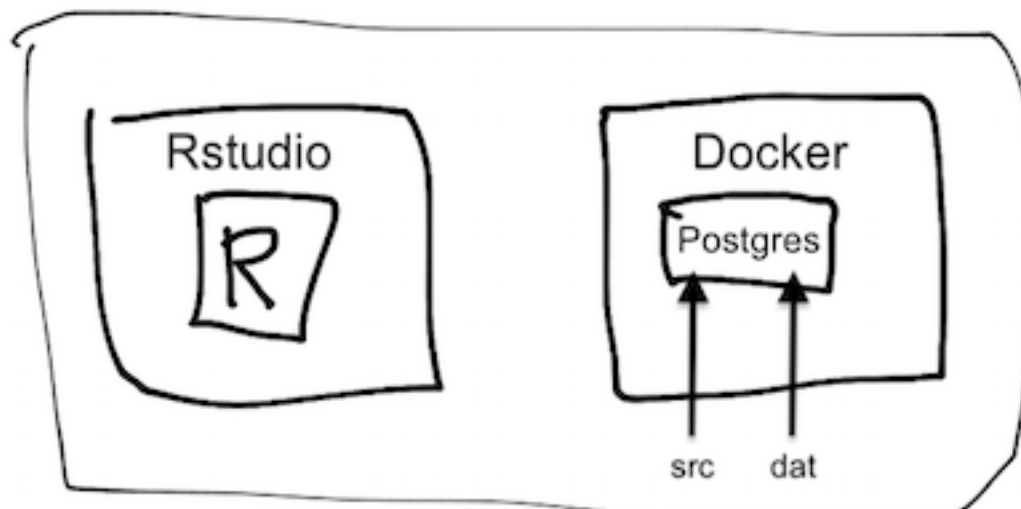
1. Make a fixed working environment for reproducible analysis
2. Access a service outside of R (**e.g., Postgres**)
3. Create an R based service (e.g., with **plumber**)
4. Send our compute jobs to the cloud with minimal reconfiguration or revision

This book explores #2 because it allows us to work on the database access issues described above and to practice on an industrial-scale DBMS.

- Docker is a relatively easy way to simulate the relationship between an R/Rstudio session and a database – all on on a single machine, provided you have Docker installed and running.
- You may want to run PostgreSQL on a Docker container, avoiding any OS or system dependencies that might come up.

1.3 Docker and R on your machine

Here is how R and Docker fit on your operating system in this tutorial:



needs to be updated as our directory structure evolves.)

(This diagram

1.4 Who are we?

- M. Edward (Ed) Borasky - @znmeb
- John David Smith - @smithjd
- Scott Came - @scottcame
- Ian Franz - @ianfrantz
- Sophie Yang - @SophieMYang
- Jim Tyhurst - @jimtyhurst
- Paul Refalo - @paulrefalo

1.5 Prerequisites

You will need

- A computer running Windows, MacOS, or Linux (Any Linux distro that will run Docker Community Edition, R and RStudio will work),
- R, and Rstudio and
- Docker hosting.

The database we use is PostgreSQL 10, but you do not need to install that - it's installed via a Docker image. RStudio 1.2 is highly recommended but not required.

In addition to the current version of R and Rstudio, you will need the following packages:

- tidyverse
- DBI
- RPostgres
- glue

1.6 Install Docker

Install Docker. Installation depends on your operating system:

- On a Mac
- On UNIX flavors
- For Windows, consider these issues and follow these instructions.

1.7 Download the repo

First step: download this repo. It contains source code to build a Docker container that has the dvdrental database in Postgress and shows how to interact with the database from R.

Chapter 2

Docker Hosting for Windows

Skip these instructions if your computer has either OSX or a Unix variant.

2.1 Hardware requirements

You will need an Intel or AMD processor with 64-bit hardware and the hardware virtualization feature. Most machines you buy today will have that, but older ones may not. You will need to go into the BIOS / firmware and enable the virtualization feature. You will need at least 4 gigabytes of RAM!

2.2 Software requirements

You will need Windows 7 64-bit or later. If you can afford it, I highly recommend upgrading to Windows 10 Pro.

2.2.1 Windows 7, 8, 8.1 and Windows 10 Home (64 bit)

Install Docker Toolbox. The instructions are here: https://docs.docker.com/toolbox/toolbox_install_windows/. Make sure you try the test cases and they work!

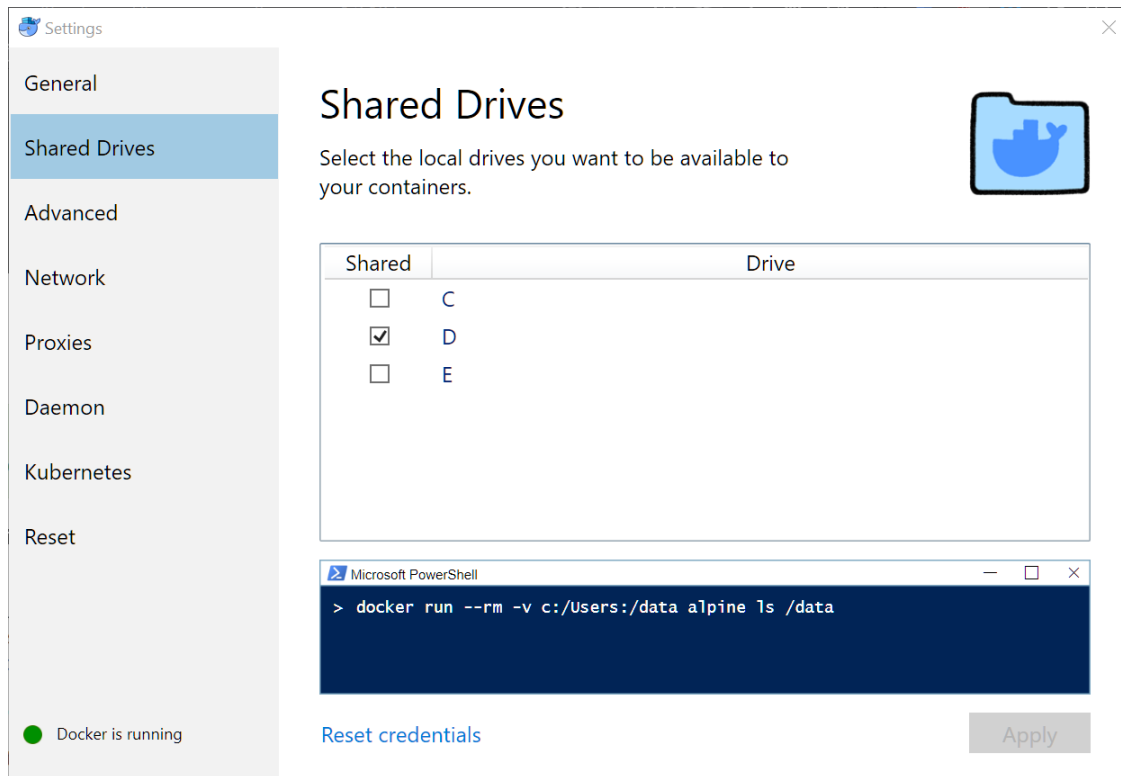
2.2.2 Windows 10 Pro

Install Docker for Windows *stable*. The instructions are here: <https://docs.docker.com/docker-for-windows/install/#start-docker-for-windows>. Again, make sure you try the test cases and they work.

2.3 Docker for Windows settings

2.3.1 Shared drives

If you're going to mount host files into container filesystems (as we do in the following chapters), you need to set up shared drives. Open the Docker settings dialog and select **Shared Drives**. Check the drives you want to share. In this screenshot, the D: drive is my 1 terabyte hard drive.



2.3.2 Kubernetes

Kubernetes is a container orchestration / cloud management package that's a major DevOps tool. It's heavily supported by Red Hat and Google, and as a result is becoming a required skill for DevOps.

However, it's overkill for this project at the moment. So you should make sure it's not enabled.

Go to the **Kubernetes** dialog and make sure the **Enable Kubernetes** checkbox is cleared.



2.4 Git, GitHub and line endings

Git was originally developed for Linux - in fact, it was created by Linus Torvalds to manage hundreds of different versions of the Linux kernel on different machines all around the world. As usage has grown, it's achieved a huge following and is the version control system used by most large open source projects, including this one.

If you're on Windows, there are some things about Git and GitHub you need to watch. First of all, there are quite a few tools for running Git on Windows, but the RStudio default and recommended one is Git for Windows (<https://git-scm.com/download/win>).

By default, text files on Linux end with a single linefeed (`\n`) character. But on Windows, text files end with a carriage return and a line feed (`\r\n`). See <https://en.wikipedia.org/wiki/Newline> for the gory details.

Git defaults to checking files out in the native mode. So if you're on Linux, a text file will show up with the Linux convention, and if you're on Windows, it will show up with the Windows convention.

Most of the time this doesn't cause any problems. But Docker containers usually run Linux, and if you have files from a repository on Windows that you've sent to the container, the container may malfunction or give weird results. *This kind of situation has caused a lot of grief for contributors to this project, so beware.*

In particular, executable `sh` or `bash` scripts will fail in a Docker container if they have Windows line endings. You may see an error message with `\r` in it, which means the shell saw the carriage return (`\r`) and gave up. But often you'll see no hint at all what the problem was.

So you need a way to tell Git that some files need to be checked out with Linux line endings. See <https://help.github.com/articles/dealing-with-line-endings/> for the details. Summary:

1. You'll need a `.gitattributes` file in the root of the repository.
2. In that file, all text files (scripts, program source, data, etc.) that are destined for a Docker container will need to have the designator `<spec> text eol=lf`, where `<spec>` is the file name specifier, for

example, `*.sh`.

This repo includes a sample: `.gitattributes`

Chapter 3

Learning Goals and Use Cases

3.1 Context: Why integrate R with databases using Docker?

- Large data stores in organizations are stored in databases that have specific access constraints and structural characteristics.
- Learning to navigate the gap between R and the database is difficult to simulate outside corporate walls.
- R users frequently need to make sense of complex data structures using diagnostic techniques that should not be reinvented (and so would benefit from more public instruction and commentary).
- Docker is a relatively easy way to simulate the relationship between an R/Rstudio session and database – all on a single machine.

3.2 Learning Goals

After working through this tutorial, you can expect to be able to:

- Run queries against Postgres in an environment that simulates what you will find in a corporate setting.
- Understand some of the tradeoffs between queries aimed at exploration or informal investigation using dplyr and those where performance is important because of the size of the database or the frequency with which a query is run. You will be able to rewrite dplyr queries as SQL and submit them directly. You will have some understanding of techniques for assessing query structure and performance.
- Set up a Postgres database in a Docker environment and understand enough about Docker to swap databases, swap DBMS' (e.g., MySQL for Postgres, etc.)

3.3 Use cases

Imagine that you have one of several roles at **DVDs R Us** and that you need to:

- As a data scientist, I want to know the distribution of number of rentals per month per customer, so that the Marketing department can create incentives for customers in 3 segments: Frequent Renters, Average Renters, Infrequent Renters.
- As the Director of Sales, I want to see the total number of rentals per month for the past 6 months and I want to know how fast our customer base is growing/shrinking per month for the past 6 months.
- As the Director of Marketing, I want to know which categories of DVDs are the least popular, so that I can create a campaign to draw attention to rarely used inventory.
- As a shipping clerk, I want to add rental information when I fulfill a shipment order.



Figure 3.1: Entity Relationship diagram for the dvdrental database

- As the Director of Analytics, you want to test as much of the production R code in my shop against a new release of the DBMS that the IT department is implementing next month.
- etc.

3.4 Environment

This tutorial uses the Postgres version of “dvd rental” database, which can be downloaded here. Here’s a glimpse of it’s structure:

Chapter 4

Docker, Postgres, and R

We always load the tidyverse and some other packages, but don't show it unless we are using packages other than tidyverse, DBI, RPostgres, and glue.

4.1 Verify that Docker running

Docker commands can be run from a terminal (e.g., the Rstudio Terminal pane) or with a `system()` command. In this tutorial, we use `system2()` so that all the output that is created externally is shown. Note that `system2` calls are divided into several parts:

1. The program that you are sending a command to.
2. The parameters or commands that are being sent
3. `stdout = TRUE`, `stderr = TRUE` are two parameters that are standard in this book, so that the comand's full output is shown in the book.

The `docker version` command returns the details about the docker daemon that is running on your computer.

```
system2("docker", "version", stdout = TRUE, stderr = TRUE)
```

```
## [1] "Client:"
## [2] " Version:          18.06.1-ce"
## [3] " API version:      1.38"
## [4] " Go version:       go1.10.3"
## [5] " Git commit:       e68fc7a"
## [6] " Built:            Tue Aug 21 17:21:31 2018"
## [7] " OS/Arch:          darwin/amd64"
## [8] " Experimental:     false"
## [9] ""
## [10] "Server:"
## [11] " Engine:"
## [12] " Version:          18.06.1-ce"
## [13] " API version:      1.38 (minimum version 1.12)"
## [14] " Go version:       go1.10.3"
## [15] " Git commit:       e68fc7a"
## [16] " Built:            Tue Aug 21 17:29:02 2018"
## [17] " OS/Arch:          linux/amd64"
## [18] " Experimental:     true"
```

4.2 Clean up if appropriate

Remove the `cattle` and `sql-pet` containers if they exists (e.g., from a prior experiments).

```
if (system2("docker", "ps -a", stdout = TRUE) %>%
  grepl(x = ., pattern = 'cattle') %>%
  any()) {
  system2("docker", "rm -f cattle")
}
if (system2("docker", "ps -a", stdout = TRUE) %>%
  grepl(x = ., pattern = 'sql-pet') %>%
  any()) {
  system2("docker", "rm -f sql-pet")
}
```

The convention we use in this book is to assemble a command with `glue` so that the you can see all of its separate parts. The following chunk just constructs the command, but does not execute it. If you have problems executing a commnad, you can always copy the command and execute in your terminal session.

```
docker_cmd <- glue(
  "run ",          # Run is the Docker command. Everything that follows are `docker run` parameters.
  "--detach ",    # (or `-d`) tells Docker to disconnect from the terminal / program issuing the command
  "--name cattle ",      # tells Docker to give the container a name: `cattle`
  "--publish 5432:5432 ", # tells Docker to expose the Postgres port 5432 to the local network with 5432
  " postgres:10" # tells Docker the image that is to be run (after downloading if necessary)
)

# We name containers `cattle` for "throw-aways" and `pet` for ones we treasure and keep around. :-)
```

Submit the command constructed above:

```
# this is what you would submit from a terminal:
cat(glue(" docker ", docker_cmd))
```

```
## docker run --detach --name cattle --publish 5432:5432 postgres:10
```

```
# this is how R submits it to Docker:
```

```
system2("docker", docker_cmd, stdout = TRUE, stderr = TRUE)
```

```
## [1] "3122d34db771235ff4852ad5bbc8e5ed1b0d4c6564a0df504512dcd021e83bf2"
```

Docker returns a long string of numbers. If you are running this command for the first time, Docker is downloading the Postgres image and it takes a bit of time.

The following comand shows that `postgres:10` is running. `postgres` is waiting for a connection:

```
system2("docker", "ps", stdout = TRUE, stderr = TRUE)
```

```
## [1] "CONTAINER ID      IMAGE      COMMAND      CREATED      STATUS      PORTS
## [2] "3122d34db771     postgres:10  \"docker-entrypoint.s...\"  1 second ago  Up Less than a second
```

4.3 Put the database password in an environment file

The goal is to put the password in an untracked file that will **not** be committed in your source code repository. Your code can reference the name of the variable, but the value of that variable will not appear in open text in your source code.

We have chosen to call the file `dev_environment.csv` in the current working directory where you are executing this script. That file name appears in the `.gitignore` file, so that you will not accidentally commit it. We are going to create that file now.

You will be prompted for the database password. By default, a postgres database defines a database user named `postgres`, whose password is `postgres`. If you have changed the password or created a new user with a different password, then enter those new values when prompted. Otherwise, enter `postgres` and `postgres` at the two prompts.

In an interactive environment, you could execute a snippet of code that prompts the user for their username and password with the following snippet (which isn't run in the book):

```
wait_for_postgres <- function(seconds_to_test){
  for (i in 1:seconds_to_test) {
    db_ready <- DBI::dbCanConnect(RPostgres::Postgres(),
                                  host = "localhost",
                                  port = "5432",
                                  user = dplyr::filter(environment_variables, variable == "username")[, "username"],
                                  password = dplyr::filter(environment_variables, variable == "password")[, "password"],
                                  dbname = "postgres")

    if ( !db_ready ) {Sys.sleep(1)}
    else {con <- DBI::dbConnect(RPostgres::Postgres(),
                                host = "localhost",
                                port = "5432",
                                user = dplyr::filter(environment_variables, variable == "username")[, "username"],
                                password = dplyr::filter(environment_variables, variable == "password")[, "password"],
                                dbname = "postgres")
  }
  if (i == seconds_to_test & !db_ready) {con <- "there is no connection "}
}
con

DB_USERNAME <- trimws(readline(prompt = "username: "), which = "both")
DB_PASSWORD <- getPass::getPass(msg = "password: ")
environment_variables = data.frame(
  variable = c("username", "password"),
  value = c(DB_USERNAME, DB_PASSWORD),
  stringsAsFactors = FALSE)
write.csv(environment_variables, "./dev_environment.csv", row.names = FALSE)
```

Your password is still in plain text in the file, `dev_environment.csv`, so you should protect that file from exposure. However, you do not need to worry about committing that file accidentally to your git repository, because the name of the file appears in the `.gitignore` file.

4.4 Connect, read and write to Postgres from R

4.4.1 Pause for some security considerations

Create a connection to Postgres after waiting 3 seconds so that Docker has time to do its thing. For security, we use values from the `environment_variables` data.frame, rather than keeping the `username` and `password` in plain text in this source file.

```
wait_for_postgres <- function(user, password, dbname, seconds_to_test = 10) {
  for (i in 1:seconds_to_test) {
```

```

db_ready <- DBI::dbCanConnect(RPostgres::Postgres(),
                             host = "localhost",
                             port = "5432",
                             user = user,
                             password = password,
                             dbname = dbname)

if ( !db_ready ) {Sys.sleep(1)}
else {con <- DBI::dbConnect(RPostgres::Postgres(),
                             host = "localhost",
                             port = "5432",
                             user = user,
                             password = password,
                             dbname = dbname)

}
if (i == seconds_to_test & !db_ready) {con <- "There is no connection"}
}
con
}

```

This `wait_for_postgres` function uses environment variables that R obtains from reading a file named `.Rprofile` which is found in your default directory. To see whether you have already created that file, execute:

```
dir(path = "~", pattern = ".Rprofile", all.files = TRUE)
```

```
## [1] ".Rprofile"
```

It should contain lines such as:

```

DEFAULT_POSTGRES_PASSWORD=postgres
DEFAULT_POSTGRES_USER_NAME=postgres

con <- wait_for_postgres(user = Sys.getenv("DEFAULT_POSTGRES_USER_NAME"),
                        password = Sys.getenv("DEFAULT_POSTGRES_PASSWORD"),
                        dbname = "postgres",
                        seconds_to_test = 10)

```

Show that you can connect but that Postgres database doesn't contain any tables:

```
dbListTables(con)
```

```
## character(0)
```

Write `mtcars` to Postgres

```
dbWriteTable(con, "mtcars", mtcars)
```

List the tables in the Postgres database to show that `mtcars` is now there:

```
dbListTables(con)
```

```
## [1] "mtcars"
```

```
# list the fields in mtcars:
```

```
dbListFields(con, "mtcars")
```

```
## [1] "mpg" "cyl" "disp" "hp" "drat" "wt" "qsec" "vs" "am" "gear"
```

```
## [11] "carb"
```

Download the table from the DBMS to a local data frame:

```
mtcars_df <- tbl(con, "mtcars")

# Show a few rows:
knitr::kable(head(mtcars_df))
```

mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
21.0	6	160	110	3.90	2.620	16.46	0	1	4	4
21.0	6	160	110	3.90	2.875	17.02	0	1	4	4
22.8	4	108	93	3.85	2.320	18.61	1	1	4	1
21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
18.7	8	360	175	3.15	3.440	17.02	0	0	3	2
18.1	6	225	105	2.76	3.460	20.22	1	0	3	1

4.5 Clean up

Afterwards, always disconnect from the DBMS, stop the docker container and (optionally) remove it.

```
dbDisconnect(con)
```

```
# tell Docker to stop the container:
system2("docker", "stop cattle", stdout = TRUE, stderr = TRUE)
```

```
## [1] "cattle"
```

```
# tell Docker to remove the container from it's library of active containers:
system2("docker", "rm cattle", stdout = TRUE, stderr = TRUE)
```

```
## [1] "cattle"
```

If we **stop** the docker container but don't remove it (with the **rm cattle** command), the container will persist and we can start it up again later with **start cattle**. In that case, **mtcars** would still be there and we could retrieve it from R again. Since we have now removed the **cattle** container, the whole database has been deleted. (There are enough copies of **mtcars** in the world, so no great loss.)

Chapter 5

A persistent database in Postgres in Docker - all at once

5.1 Overview

You’ve already connected to Postgres with R, now you need a “realistic” (`dvdrental`) database. We’re going to demonstrate how to set one up, with two different approaches. This chapter and the next do the same job, illustrating the different approaches that you can take and helping you see the different points where you could swap what’s provided here with a different DBMS or a different backup file or something else.

The code in this first version is recommended because it is an “all in one” approach. Details about how it works and how you might modify it are included below. There is another version in the the next chapter that you can use to investigate Docker commands and components.

Note that this approach relies on two files that have quote that’s not shown here: `dvdrental.Dockerfile` and `init-dvdrental.sh`. They are discussed below.

Note that `tidyverse`, `DBI`, `RPostgres`, and `glue` are loaded.

5.2 First, verify that Docker is up and running:

```
system2("docker", "version", stdout = TRUE, stderr = TRUE)
```

```
## [1] "Client:"
## [2] " Version:          18.06.1-ce"
## [3] " API version:      1.38"
## [4] " Go version:       go1.10.3"
## [5] " Git commit:       e68fc7a"
## [6] " Built:            Tue Aug 21 17:21:31 2018"
## [7] " OS/Arch:          darwin/amd64"
## [8] " Experimental:     false"
## [9] ""
## [10] "Server:"
## [11] " Engine:"
## [12] " Version:          18.06.1-ce"
## [13] " API version:      1.38 (minimum version 1.12)"
## [14] " Go version:       go1.10.3"
## [15] " Git commit:       e68fc7a"
```

```
## [16] "   Built:           Tue Aug 21 17:29:02 2018"
## [17] "   OS/Arch:         linux/amd64"
## [18] "   Experimental:    true"
```

5.3 Clean up if appropriate

Remove the `sql-pet` container if it exists (e.g., from a prior run)

```
if (system2("docker", "ps -a", stdout = TRUE) %>%
  grepl(x = ., pattern = 'sql-pet') %>%
  any()) {
  system2("docker", "rm -f sql-pet")
}
```

5.4 Build the Docker Image

Build an image that derives from `postgres:10`, defined in `dvdreental.Dockerfile`, that is set up to restore and load the `dvdreental` db on startup. The `dvdreental.Dockerfile` is discussed below.

```
system2("docker",
  glue("build ", # tells Docker to build an image that can be loaded as a container
    "--tag postgres-dvdreental ", # (or -t) tells Docker to name the image
    "--file dvdreental.Dockerfile ", #(or -f) tells Docker to read `build` instructions from the d
    " . "), # tells Docker to look for dvdreental.Dockerfile in the current directory
  stdout = TRUE, stderr = TRUE)

## [1] "Sending build context to Docker daemon  2.974MB\r\r"
## [2] "Step 1/4 : FROM postgres:10"
## [3] " ---> ac25c2bac3c4"
## [4] "Step 2/4 : WORKDIR /tmp"
## [5] " ---> Using cache"
## [6] " ---> 3f00a18e0bdf"
## [7] "Step 3/4 : COPY init-dvdreental.sh /docker-entrypoint-initdb.d/"
## [8] " ---> Using cache"
## [9] " ---> 3453d61d8e3e"
## [10] "Step 4/4 : RUN apt-get -qq update && apt-get install -y -qq curl zip > /dev/null 2>&1 && curl -Os l
## [11] " ---> Using cache"
## [12] " ---> f5e93aa64875"
## [13] "Successfully built f5e93aa64875"
## [14] "Successfully tagged postgres-dvdreental:latest"
```

5.5 Run the Docker Image

Run `docker` to bring up `postgres`. The first time it runs it will take a minute to create the `Postgres` environment. There are two important parts to this that may not be obvious:

- The `source=` paramter points to `dvdreental.Dockerfile`, which does most of the heavy lifting. It has detailed, line-by-line comments to explain what it is doing.
- *Inside* `dvdreental.Dockerfile` the comand `COPY init-dvdreental.sh /docker-entrypoint-initdb.d/` copies `init-dvdreental.sh` from the local file system into the specified location in the `Docker` container. When the `Postgres` `Docker` container initializes, it looks for that file and executes it.

Doing all of that work behind the scenes involves two layers of complexity. Depending on how you look at it, that may be more or less difficult to understand than the method shown in the next Chapter.

```
wd <- getwd()

docker_cmd <- glue(
  "run ",          # Run is the Docker command. Everything that follows are `run` parameters.
  "--detach ",    # (or `-d`) tells Docker to disconnect from the terminal / program issuing the command
  " --name sql-pet ",      # tells Docker to give the container a name: `sql-pet`
  "--publish 5432:5432 ",  # tells Docker to expose the Postgres port 5432 to the local network with 5432
  "--mount ",           # tells Docker to mount a volume -- mapping Docker's internal file structure to the host
  "type=bind,",         # tells Docker that the mount command points to an actual file on the host system
  "source=",            # tells Docker where the local file will be found
  wd, "/",              # the current working directory, as retrieved above
  "target=/petdir",     # tells Docker to refer to the current directory as "/petdir" in its file system
  " postgres-dvdrental" # tells Docker to run the image was built in the previous step
)

# if you are curious you can paste this string into a terminal window after the command 'docker':
docker_cmd

## run --detach --name sql-pet --publish 5432:5432 --mount type=bind,source='/Users/jds/Documents/Library
system2("docker", docker_cmd, stdout = TRUE, stderr = TRUE)

## [1] "204b0fdac0c79ba0b58c8f5a575fadb27494d0311ed64a270abe177a8e96f5c2"
```

5.6 Connect to Postgres with R

Use the DBI package to connect to Postgres. But first, wait for Docker & Postgres to come up before connecting.

We have loaded the `wait_for_postgres` function behind the scenes.

```
con <- wait_for_postgres(user = Sys.getenv("DEFAULT_POSTGRES_USER_NAME"),
                        password = Sys.getenv("DEFAULT_POSTGRES_PASSWORD"),
                        dbname = "dvdrental",
                        seconds_to_test = 10)

# if (con == "it's not there") {stop()}

dbListTables(con)

## [1] "actor_info"          "customer_list"
## [3] "film_list"           "nicer_but_slower_film_list"
## [5] "sales_by_film_category" "staff"
## [7] "sales_by_store"      "staff_list"
## [9] "category"            "film_category"
## [11] "country"             "actor"
## [13] "language"            "inventory"
## [15] "payment"             "rental"
## [17] "city"                "store"
## [19] "film"                "address"
## [21] "film_actor"          "customer"
```

```
dbListFields(con, "rental")

## [1] "rental_id"      "rental_date"    "inventory_id"   "customer_id"
## [5] "return_date"    "staff_id"       "last_update"

dbDisconnect(con)

Sys.sleep(2) # Can take a moment to disconnect.
```

5.7 Stop and start to demonstrate persistence

Stop the container

```
system2('docker', 'stop sql-pet',
        stdout = TRUE, stderr = TRUE)

## [1] "sql-pet"
Sys.sleep(3) # can take a moment for Docker to stop the container.
```

Restart the container and verify that the dvdrental tables are still there

```
system2("docker", "start sql-pet", stdout = TRUE, stderr = TRUE)

## [1] "sql-pet"

con <- wait_for_postgres(user = Sys.getenv("DEFAULT_POSTGRES_USER_NAME"),
                        password = Sys.getenv("DEFAULT_POSTGRES_PASSWORD"),
                        dbname = "dvdrental",
                        seconds_to_test = 10)

glimpse(dbReadTable(con, "film"))

## Observations: 1,000
## Variables: 13
## $ film_id      <int> 133, 384, 8, 98, 1, 2, 3, 4, 5, 6, 7, 9, 10, ...
## $ title        <chr> "Chamber Italian", "Grosse Wonderful", "Airpo...
## $ description  <chr> "A Fateful Reflection of a Moose And a Husban...
## $ release_year <int> 2006, 2006, 2006, 2006, 2006, 2006, 2006, 200...
## $ language_id  <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
## $ rental_duration <int> 7, 5, 6, 4, 6, 3, 7, 5, 6, 3, 6, 3, 6, 6, ...
## $ rental_rate  <dbl> 4.99, 4.99, 4.99, 4.99, 0.99, 4.99, 2.99, 2.9...
## $ length       <int> 117, 49, 54, 73, 86, 48, 50, 117, 130, 169, 6...
## $ replacement_cost <dbl> 14.99, 19.99, 15.99, 12.99, 20.99, 12.99, 18...
## $ rating       <chr> "NC-17", "R", "R", "PG-13", "PG", "G", "NC-17...
## $ last_update  <dtm> 2013-05-26 14:50:58, 2013-05-26 14:50:58, 20...
## $ special_features <chr> "{Trailers}", "{\\\"Behind the Scenes\\\"}", "{Tr...
## $ fulltext     <chr> "'chamber':1 'fate':4 'husband':11 'italian':...
```

It's always good to have R disconnect from the database

```
dbDisconnect(con)

Stop the container & show that the container is still there, so can be started again.

system2('docker', 'stop sql-pet',
        stdout = TRUE, stderr = TRUE)
```



```
## [1] "sql-pet"
```

```
# show that the container still exists even though it's not running  
psout <- system2("docker", "ps -a", stdout = TRUE)  
psout[grepl(x = psout, pattern = 'sql-pet')]
```

```
## [1] "204b0fdac0c7      postgres-dvdrental  \"docker-entrypoint.s...\"  21 seconds ago      Exited (137) Les
```

5.8 Cleaning up

Next time, you can just use this command to start the container:

```
system2("docker", "start sql-pet", stdout = TRUE, stderr = TRUE)
```

And once stopped, the container can be removed with:

```
system2("docker", "rm sql-pet", stdout = TRUE, stderr = TRUE)
```

5.9 Using the sql-pet container in the rest of the book

After this point in the book, we assume that Docker is up and that we can always start up our *sql-pet* database with:

```
system2("docker", "start sql-pet", stdout = TRUE, stderr = TRUE)
```


Chapter 6

A persistent database in Postgres in Docker - piecemeal

6.1 Overview

This chapter essentially repeats what was presented in the previous one, but does it in a step-by-step way that might be useful to understand how each of the steps involved in setting up a persistent Postgres database works. If you are satisfied with the method shown in that chapter, skip this one for now.

Note that `tidyverse`, `DBI`, `RPostgres`, and `glue` are loaded.

6.2 Retrieve the backup file

The first step is to get a local copy of the `dvdrental` Postgres restore file. It comes in a zip format and needs to be un-zipped. Use the `downloader` and `here` packages to keep track of things.

```
if (!require(downloader)) install.packages("downloader")

## Loading required package: downloader
if (!require(here)) install.packages("here")

## Loading required package: here

## here() starts at /Users/jds/Documents/Library/R/r-system/sql-pet/r-database-docker
library(downloader, here)

download("http://www.postgresqltutorial.com/wp-content/uploads/2017/10/dvdrental.zip", destfile = here("dvdrental.zip"))
unzip(here("dvdrental.zip"), exdir = here()) # creates a tar archive named "dvdrental.tar"
file.remove(here("dvdrental.zip")) # the Zip file is no longer needed.

## [1] TRUE
```

6.3 Now, verify that Docker is up and running:

```
system2("docker", "version", stdout = TRUE, stderr = TRUE)
```

```
## [1] "Client:"
## [2] " Version:      18.06.1-ce"
## [3] " API version:   1.38"
## [4] " Go version:    go1.10.3"
## [5] " Git commit:    e68fc7a"
## [6] " Built:         Tue Aug 21 17:21:31 2018"
## [7] " OS/Arch:       darwin/amd64"
## [8] " Experimental:  false"
## [9] ""
## [10] "Server:"
## [11] " Engine:"
## [12] " Version:      18.06.1-ce"
## [13] " API version:   1.38 (minimum version 1.12)"
## [14] " Go version:    go1.10.3"
## [15] " Git commit:    e68fc7a"
## [16] " Built:         Tue Aug 21 17:29:02 2018"
## [17] " OS/Arch:       linux/amd64"
## [18] " Experimental:  true"
```

Remove the `sql-pet` container if it exists (e.g., from a prior run)

```
if (system2("docker", "ps -a", stdout = TRUE) %>%
  grepl(x = ., pattern = 'sql-pet') %>%
  any()) {
  system2("docker", "rm -f sql-pet")
}
```

6.4 Build the Docker Image

Build an image that derives from `postgres:10`. Connect the local and Docker directories that need to be shared. Expose the standard Postgres port 5432.

`" postgres-dvdrental"` # tells Docker the image that is to be run (after downloading if necessary)

```
wd <- getwd()
```

```
docker_cmd <- glue(
  "run ",          # Run is the Docker command. Everything that follows are `run` parameters.
  "--detach ",     # (or `-d`) tells Docker to disconnect from the terminal / program issuing the command
  " --name sql-pet ", # tells Docker to give the container a name: `sql-pet`
  "--publish 5432:5432 ", # tells Docker to expose the Postgres port 5432 to the local network with 5432
  "--mount ",       # tells Docker to mount a volume -- mapping Docker's internal file structure to the host
  "type=bind,source='", wd, "'/,target=/petdir",
  " postgres:10" # tells Docker the image that is to be run (after downloading if necessary)
)
```

```
system2("docker", docker_cmd, stdout = TRUE, stderr = TRUE)
```

```
## [1] "e18eac258acce9fe1272ab5cc0624ab561eaccae9b62d73b6dd97ef3634b1317"
```

Peek inside the docker container and list the files in the `petdir` directory. Notice that `dvdrental.tar` is in both.

```
system2('docker', 'exec sql-pet ls petdir | grep "dvdrental.tar" ',
        stdout = TRUE, stderr = TRUE)
```

```
## [1] "dvdrental.tar"
```

```
dir(wd, pattern = "dvdrental.tar")
```

```
## [1] "dvdrental.tar"
```

We can execute programs inside the Docker container with the `exec` command. In this case we tell Docker to execute the `psql` program inside the `sql-pet` container and pass it some commands.

```
Sys.sleep(2) # is this really needed?
# inside Docker, execute the postgres SQL command-line program to create the dvdrental database:
system2('docker', 'exec sql-pet psql -U postgres -c "CREATE DATABASE dvdrental;"',
        stdout = TRUE, stderr = TRUE)
```

```
## [1] "CREATE DATABASE"
```

The `psql` program repeats back to us what it has done, e.g., to create a database named `dvdrental`.

Next we execute a different program in the Docker container, `pg_restore`, and tell it where the restore file is located. If successful, the `pg_restore` just responds with a very laconic `character(0)`.

```
Sys.sleep(2) # is this really needed?
# restore the database from the .tar file
system2("docker", "exec sql-pet pg_restore -U postgres -d dvdrental petdir/dvdrental.tar", stdout = TRUE)
```

```
## character(0)
```

```
file.remove(here("dvdrental.tar")) # the tar file is no longer needed.
```

```
## [1] TRUE
```

Use the DBI package to connect to Postgres. But first, wait for Docker & Postgres to come up before connecting.

```
wait_for_postgres <- function(seconds_to_test){
  for (i in 1:seconds_to_test) {
    db_ready <- DBI::dbCanConnect(RPostgres::Postgres(),
                                  host = "localhost",
                                  port = "5432",
                                  user = "postgres",
                                  password = "postgres",
                                  dbname = "dvdrental" ) # note that the dbname is specified

    if ( !db_ready ) {Sys.sleep(1)}
    else {con <- DBI::dbConnect(RPostgres::Postgres(),
                                host = "localhost",
                                port = "5432",
                                user = "postgres",
                                password = "postgres",
                                dbname = "dvdrental" ) # note that the dbname is specified
    }
    if (i == seconds_to_test & !db_ready) {con <- "it's not there"}
  }
  con
}
```

```

con <- wait_for_postgres(10)

dbListTables(con)

## [1] "actor_info"           "customer_list"
## [3] "film_list"           "nicer_but_slower_film_list"
## [5] "sales_by_film_category" "staff"
## [7] "sales_by_store"      "staff_list"
## [9] "category"           "film_category"
## [11] "country"            "actor"
## [13] "language"           "inventory"
## [15] "payment"            "rental"
## [17] "city"               "store"
## [19] "film"               "address"
## [21] "film_actor"         "customer"

dbListFields(con, "film")

## [1] "film_id"           "title"           "description"
## [4] "release_year"      "language_id"      "rental_duration"
## [7] "rental_rate"       "length"          "replacement_cost"
## [10] "rating"            "last_update"     "special_features"
## [13] "fulltext"

dbDisconnect(con)

```

6.5 Stop and start to demonstrate persistence

Stop the container

```

system2('docker', 'stop sql-pet',
        stdout = TRUE, stderr = TRUE)

```

```
## [1] "sql-pet"
```

Restart the container and verify that the dvdrental tables are still there

```
system2("docker", "start sql-pet", stdout = TRUE, stderr = TRUE)
```

```
## [1] "sql-pet"
```

```
Sys.sleep(1) # need to wait for Docker & Postgres to come up before connecting.
```

```
con <- wait_for_postgres(10)
```

```
glimpse(dbReadTable(con, "film"))
```

```

## Observations: 1,000
## Variables: 13
## $ film_id      <int> 133, 384, 8, 98, 1, 2, 3, 4, 5, 6, 7, 9, 10, ...
## $ title        <chr> "Chamber Italian", "Grosse Wonderful", "Airpo...
## $ description  <chr> "A Fateful Reflection of a Moose And a Husban...
## $ release_year <int> 2006, 2006, 2006, 2006, 2006, 2006, 2006, 200...
## $ language_id  <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
## $ rental_duration <int> 7, 5, 6, 4, 6, 3, 7, 5, 6, 3, 6, 3, 6, 6, ...
## $ rental_rate  <dbl> 4.99, 4.99, 4.99, 4.99, 0.99, 4.99, 2.99, 2.9...
## $ length       <int> 117, 49, 54, 73, 86, 48, 50, 117, 130, 169, 6...

```

```
## $ replacement_cost <dbl> 14.99, 19.99, 15.99, 12.99, 20.99, 12.99, 18....
## $ rating           <chr> "NC-17", "R", "R", "PG-13", "PG", "G", "NC-17...
## $ last_update      <dtm> 2013-05-26 14:50:58, 2013-05-26 14:50:58, 20...
## $ special_features <chr> "{Trailers}", "{\\"Behind the Scenes\\"}", "{Tr...
## $ fulltext         <chr> "'chamber':1 'fate':4 'husband':11 'italian':...
```

Stop the container & show that the container is still there, so can be started again.

```
system2('docker', 'stop sql-pet',
        stdout = TRUE, stderr = TRUE)
```

```
## [1] "sql-pet"
```

```
# show that the container still exists even though it's not running
```

```
psout <- system2("docker", "ps -a", stdout = TRUE)
```

```
psout[grepl(x = psout, pattern = 'sql-pet')]
```

```
## [1] "e18eac258acc      postgres:10      \\"docker-entrypoint.s...\\" 29 seconds ago      Exited (137) Less t
```

6.6 Cleaning up

Next time, you can just use this command to start the container:

```
system2("docker", "start sql-pet", stdout = TRUE, stderr = TRUE)
```

And after disconnecting from it the container can be completely removed with:

```
system2("docker", "rm sql-pet -f", stdout = TRUE, stderr = TRUE)
```

6.7 Using the sql-pet container in the rest of the book

After this point in the book, we assume that Docker is up and that we can always start up our *sql-pet* database with:

```
system2("docker", "start sql-pet", stdout = TRUE, stderr = TRUE)
```


Chapter 7

Introduction to interacting with Postgres from R

Note that `tidyverse`, `DBI`, `RPostgres`, and `glue` are loaded.

7.1 Basics

- keeping passwords secure
- Coverage in this book. There are many SQL tutorials that are available. For example, we are drawing some materials from a tutorial we recommend. In particular, we will not replicate the lessons there, which you might want to complete. Instead, we are showing strategies that are recommended for R users. That will include some translations of queries that are discussed there.

7.2 Ask yourself about what you are aiming for?

- differences between production and data warehouse environments
- learning to keep your DBAs happy
 - You are your own DBA in this simulation, so you can wreak havoc and learn from it, but you can learn to be DBA-friendly here.
 - in the end it's the subject-matter experts that understand your data, but you have to work with your DBAs first

7.3 Get some basic information about your database

Assume that the Docker container with Postgres and the `dvdrental` database are ready to go.

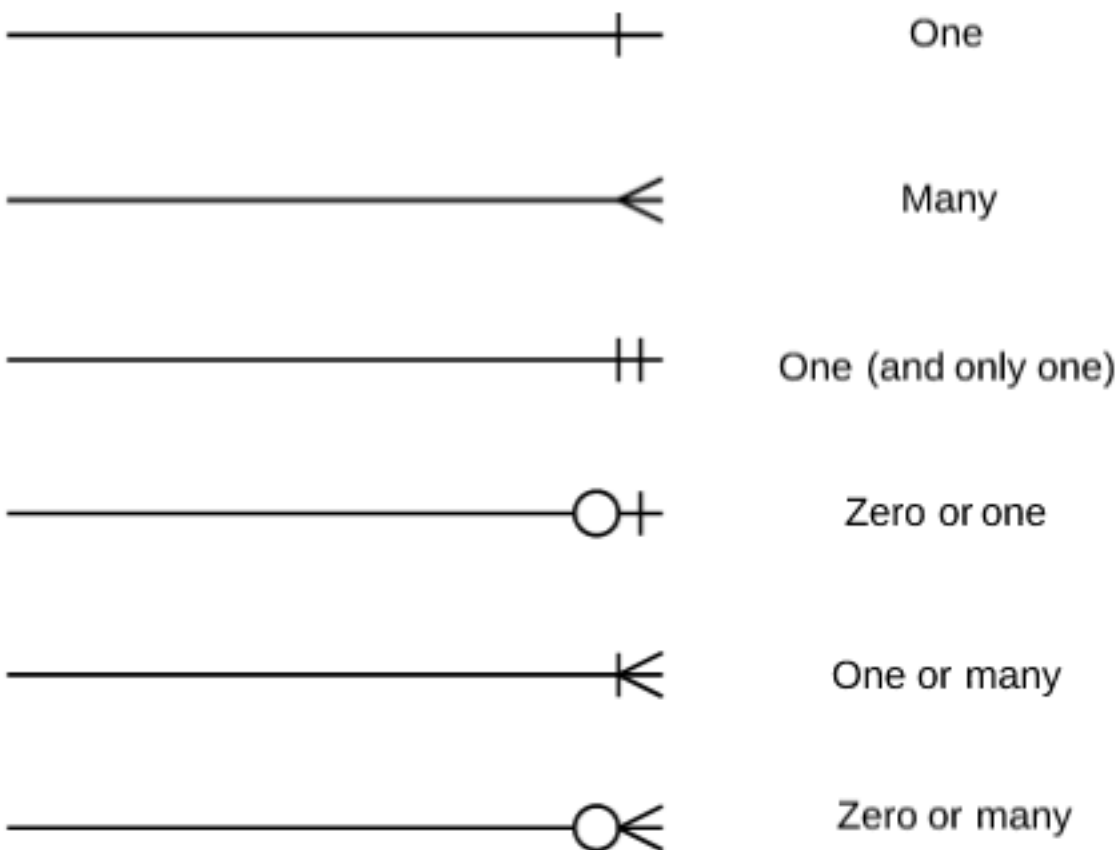
```
system2("docker", "start sql-pet", stdout = TRUE, stderr = TRUE)
```

```
## [1] "sql-pet"
```

```
Sys.sleep(2) # need to wait for Docker & Postgres to come up before connecting.
```

```
con <- DBI::dbConnect(RPostgres::Postgres(), host = "localhost",  
  port = "5432", user = "postgres",  
  password = "postgres", dbname = "dvdrental" ) # note that the dbname is specified
```

You usually need to use both the available documentation for your database and to be somewhat skeptical (e.g., empirical). It's worth learning to interpret the symbols in an Entity Relationship Diagram:



Depending on how skeptical you are about the documentation, you might want to get an overview of a database by pulling data from the database `information_schema`. Here's a selection of useful information although you may want more (or less). There is a lot to choose from a vast list of metadata. Note that information schemas are somewhat consistent across different DBMS' that you may encounter.

have we hidden “`in_schema()`” as in:

```
con %>% tbl(in_schema("aux", "df"))

table_schema_query <- glue("SELECT ",
  "table_name, column_name, data_type, ordinal_position, column_default, character_maximum_length",
  " FROM information_schema.columns ",
  "WHERE table_schema = 'public'")

rental_meta_data <- dbGetQuery(con, table_schema_query)

glimpse(rental_meta_data)

## Observations: 128
## Variables: 6
## $ table_name      <chr> "actor_info", "actor_info", "actor_in...
## $ column_name     <chr> "actor_id", "first_name", "last_name"...
## $ data_type        <chr> "integer", "character varying", "char...
## $ ordinal_position <int> 1, 2, 3, 4, 1, 2, 3, 4, 5, 6, 7, 8, 9...
```

```
## $ column_default      <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, N...
## $ character_maximum_length <int> NA, 45, 45, NA, NA, NA, 50, 10, 20, 5...
```

Pull out some rough-and-ready but useful statistics about your database. Since we are in SQL-land we talk about variables as columns.

Start with a list of tables names and a count of the number of columns that each one contains.

```
rental_meta_data %>% count(table_name) %>% rename(number_of_columns = n) %>% as.data.frame()

##           table_name number_of_columns
## 1             actor              4
## 2          actor_info              4
## 3             address              8
## 4            category              3
## 5              city              4
## 6             country              3
## 7           customer             10
## 8        customer_list              9
## 9              film             13
## 10         film_actor              3
## 11       film_category              3
## 12          film_list              8
## 13          inventory              4
## 14           language              3
## 15 nicer_but_slower_film_list      8
## 16            payment              6
## 17             rental              7
## 18 sales_by_film_category              2
## 19       sales_by_store              3
## 20              staff             11
## 21          staff_list              8
## 22             store              4
```

How many column names are shared across tables (or duplicated)?

```
rental_meta_data %>% count(column_name, sort = TRUE) %>% filter(n > 1)
```

```
## # A tibble: 34 x 2
##   column_name      n
##   <chr>         <int>
## 1 last_update     14
## 2 address_id       4
## 3 film_id          4
## 4 first_name       4
## 5 last_name        4
## 6 name             4
## 7 store_id         4
## 8 actor_id         3
## 9 address          3
## 10 category         3
## # ... with 24 more rows
```

How many column names are unique?

```
rental_meta_data %>% count(column_name) %>% filter(n > 1)
```

```
## # A tibble: 34 x 2
##   column_name      n
```

```
##      <chr>      <int>
## 1 active        2
## 2 actor_id      3
## 3 actors        2
## 4 address       3
## 5 address_id    4
## 6 category      3
## 7 category_id   2
## 8 city          3
## 9 city_id       2
## 10 country      3
## # ... with 24 more rows
```

What data types are found in the database?

```
rental_meta_data %>% count(data_type)
```

```
## # A tibble: 13 x 2
##   data_type      n
##   <chr>      <int>
## 1 ARRAY          1
## 2 boolean        2
## 3 bytea          1
## 4 character       1
## 5 character varying 36
## 6 date           1
## 7 integer        22
## 8 numeric         7
## 9 smallint       25
## 10 text          11
## 11 timestamp without time zone 17
## 12 tsvector        1
## 13 USER-DEFINED     3
```

Chapter 8

Real work with real data

8.1 Some extra handy libraries

Here are some packages that we find handy in the preliminary investigation of a database (or a problem that involves data from a database).

```
library(glue)
library(skimr)
```

8.2 Basic investigation

- R tools for data investigation
 - glimpse
 - str
 - View and kable
- overview investigation: do you understand your data
 - documentation and its limits
 - what's *missing* from the database: (columns, records, cells)
- find out how the data is used by those who enter it and others who've used it before
 - why is there missing data?

8.3 Using Dplyr

We already started, but that's OK.

8.3.1 finding out what's in the database

- DBI / RPostgres packages
- R tools like glimpse, skimr, kable.
- examining dplyr queries (show_query on the R side v EXPLAIN on the Postgres side)
- Tutorials like: <https://suzan.rbind.io/tags/dplyr/>
- Benjamin S. Baumer, A Grammar for Reproducible and Painless Extract-Transform-Load Operations on Medium Data: <https://arxiv.org/pdf/1708.07073>

8.3.2 sample query

- rental
- date subset
- left join staff
- left join customer

8.3.3 Subset: only retrieve what you need

- Columns
- Rows
 - number of row
 - specific rows
- dplyr joins in the R

8.3.4 Make the server do as much work as you can

discuss this simple example? <http://www.postgresqltutorial.com/postgresql-left-join/>

- dplyr joins on the server side
- Where you put (`collect(n = Inf)`) really matters

8.4 What is dplyr sending to the server?

- `show_query` as a first draft

8.5 Writing your on SQL directly to the DBMS

- `dbquery`
- Glue for constructing SQL statements
 - parameterizing SQL queries

8.6 Chosing between dplyr and native SQL

- performance considerations: first get the right data, then worry about performance
- Tradeoffs between leaving the data in Postgres vs what's kept in R:
 - browsing the data
 - larger samples and complete tables
 - using what you know to write efficient queries that do most of the work on the server

Chapter 9

Real work with real data

9.1 Some extra handy libraries

Here are some packages that we find handy in the preliminary investigation of a database (or a problem that involves data from a database).

```
library(glue)

##
## Attaching package: 'glue'
## The following object is masked from 'package:dplyr':
##
##      collapse
library(skimr)
```

9.2 More topics

- Check this against Aaron Makubuya's workshop at the Cascadia R Conf.

9.3 Standards for production jobs

- writing tests for your queries

Chapter 10

Other resources

10.1 Editing this book

- Here are instructions for editing this tutorial

10.2 Docker alternatives

- Choosing between Docker and Vagrant

10.3 Docker and R

- Noam Ross' talk on Docker for the UseR and his Slides give a lot of context and tips.
- Good Docker tutorials
 - An introductory Docker tutorial
 - A Docker curriculum
- Scott Came's materials about Docker and R on his website and at the 2018 UseR Conference focus on **R inside Docker**.
- It's worth studying the ROpenSci Docker tutorial

10.4 Documentation Docker and Postgres

- The Postgres image documentation
- Dockerize PostgreSQL
- Postgres & Docker documentation
- Usage examples of Postgres with Docker

10.5 More Resources

- David Severski describes some key elements of connecting to databases with R for MacOS users
- This tutorial picks up ideas and tips from Ed Borasky's Data Science pet containers, which creates a framework based on that Hack Oregon example and explains why this repo is named pet-sql.