R, Databases and Docker

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

2018-10-25

Contents

Chapter 1

Introduction

At the end of this chapter, you will be able to

- Understand the importance of using R and Docker to query a DBMS and access a service like Postgres outside of R.
- Setup your environment to explore the use-case for useRs.

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 analytic 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 as a tool for UseRs

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.

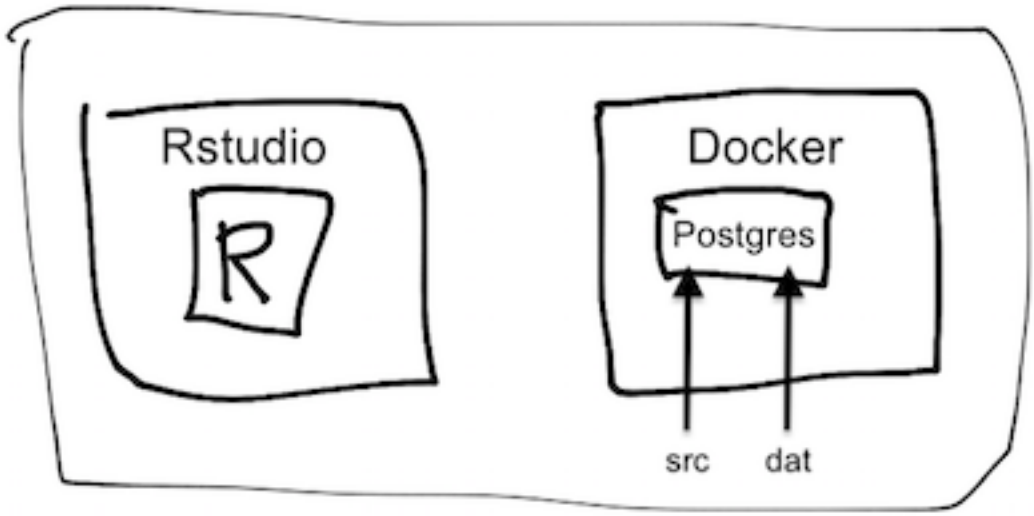
- Docker is a relatively easy way to simulate the relationship between an R/RStudio session and a database – all on on a single machine, provided you have Docker installed and running.
- You may want to run PostgreSQL on a Docker container, avoiding any OS or system dependencies that might come up.

1.3 Why write a book about DBMS access from R 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 on a single machine.

1.4 Docker and R on your machine

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



(This diagram needs to be updated as our directory structure evolves.)

1.5 Who are we?

We have been collaborating on this book since the Summer of 2018, each of us chipping into the project as time permits:

- Dipti Muni - @deemuni
- Ian Franz - @ianfrantz
- Jim Tyhurst - @jimtyhurst
- John David Smith - @smithjd
- M. Edward (Ed) Borasky - @znmeb
- Maryann Tygeson @maryannet
- Scott Came - @scottcame
- Sophie Yang - @SophieMYang

Chapter 2

How to use this book (01)

This book is full of examples that you can replicate on your computer.

2.1 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
- Docker
- Our companion package `sqlpetr` installs with: `devtools::install_github("smithjd/sqlpetr")`.

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 current versions of the following packages:

- tidyverse
- DBI
- RPostgres
- glue
- dbplyr
- knitr

2.2 Installing Docker

Install Docker. Installation depends on your operating system:

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

2.3 Download the repo

The code to generate the book and the exercises it contains can be downloaded from this repo.

2.4 Read along, experiment as you go

We have never been sure whether we're writing an expository book or a massive tutorial. You may use it either way.

After the introductory chapters and the chapter that creates the persistent database ("The dvdrental database in Postgres in Docker (05)), you can jump around and each chapter stands on its own.

Chapter 3

Docker Hosting for Windows (02)

At the end of this chapter, you will be able to

- Setup your environment for Windows.
- Use Git and GitHub effectively on Windows.

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

3.1 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!

3.2 Software requirements

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

3.2.1 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!

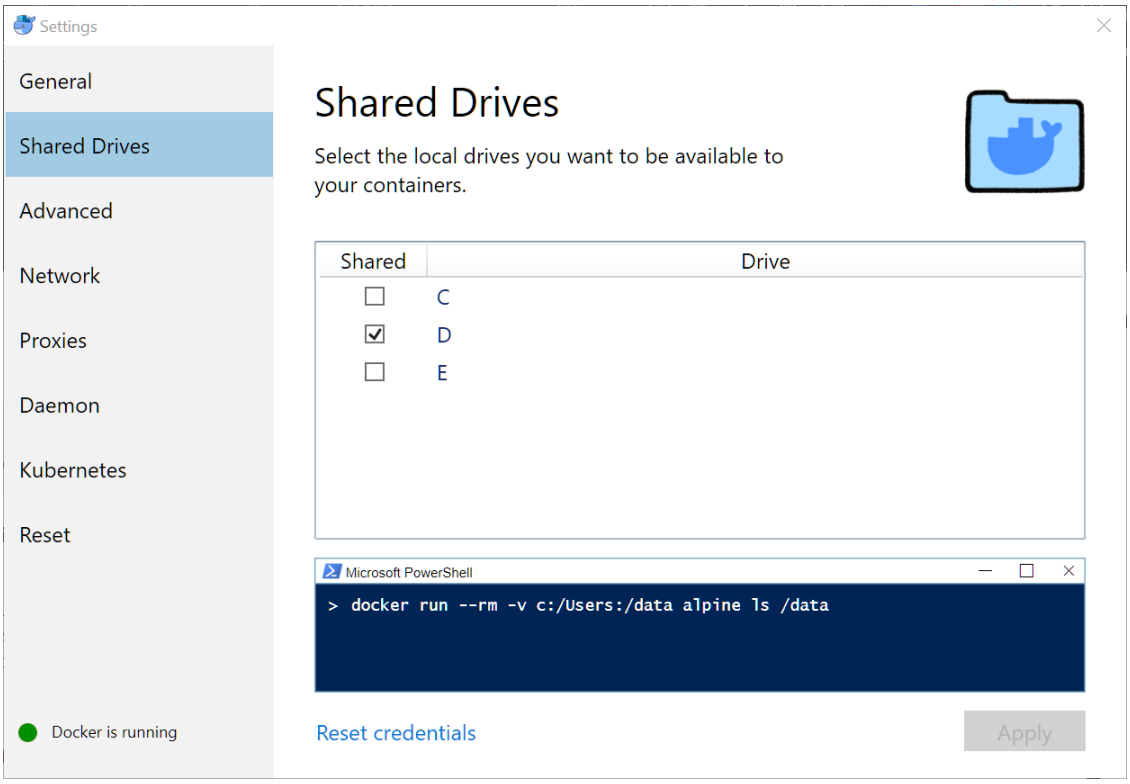
3.2.2 Windows 10 Pro

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.

3.3 Docker for Windows settings

3.3.1 Shared drives

If you're going to mount host files into container file systems (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.

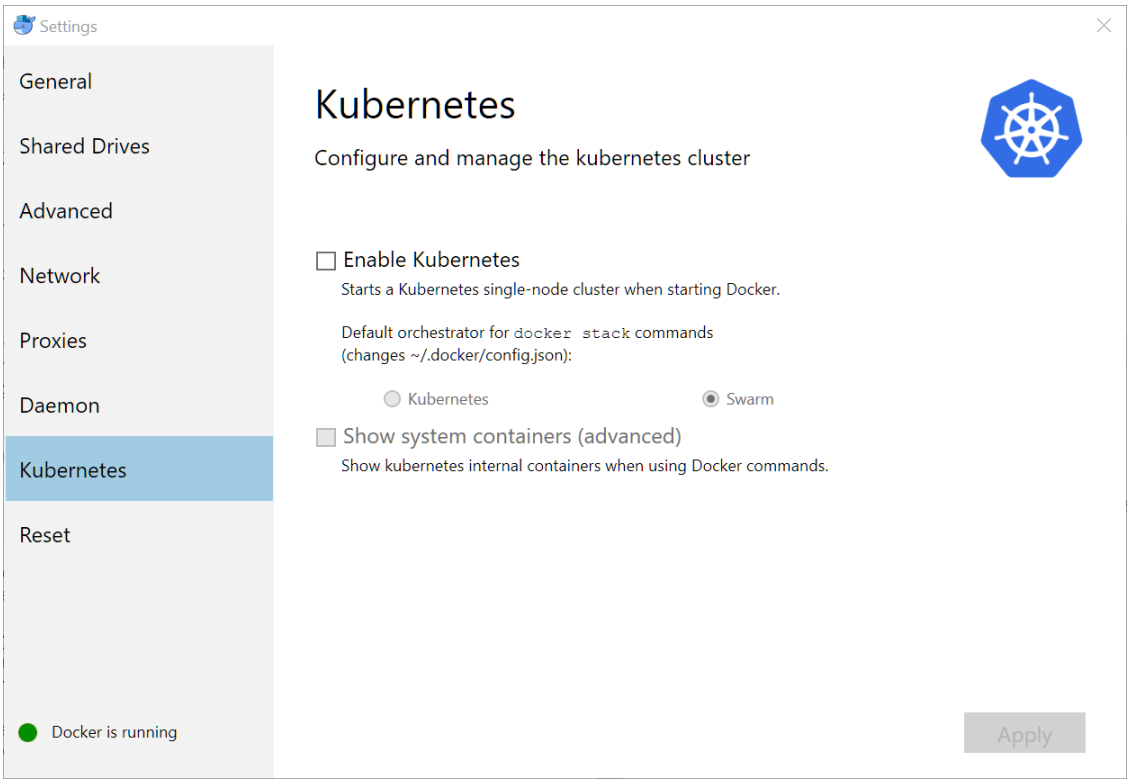


3.3.2 Kubernetes

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.



3.4 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, Git has 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 4

This Book's Learning Goals and Use Cases (03)

4.1 Learning Goals

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

- Set up a PostgreSQL database in a Docker environment.
- Run queries against PostgreSQL in an environment that simulates what you will find in a corporate setting.
- Understand techniques and some of the trade-offs between:
  1. queries aimed at exploration or informal investigation using dplyr; and
  2. those where performance is important because of the size of the database or the frequency with which a query is run.
- Understand the equivalence between `dplyr` and SQL queries and how R translates one into the other
- Understand some more advanced SQL techniques.
- Gain familiarity with the standard metadata that an SQL database contains to describe its own contents.
- Gain some understanding of techniques for assessing query structure and performance.
- Understand enough about Docker to swap databases, e.g. Sports DB for the DVD rental database used in this tutorial. Or swap the database management system (DBMS), e.g. MySQL for PostgreSQL.

4.2 Imaging a DVD rental business

- Years ago people rented videos on DVD disks and video stores were a big business.
- Imagine managing a video rental store like Movie Madness in Portland, Oregon.

- What data would be needed and what questions would you have to answer about the business?

This tutorial uses the Postgres version of "dvd rental" database which represents the transaction database for running a movie (e.g., dvd) rental business. The database can be downloaded here. Here's a glimpse of it's structure, which will be discussed in some detail:

A data analyst uses the database abstraction and the practical business questions to answer business questions.

4.3 Use cases

Imagine that you have one of several roles at our fictional company **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.
- As the Director of Analytics, I want to test as much of the production R code in my shop as possible against a new release of the DBMS that the IT department is implementing next month.
- etc.

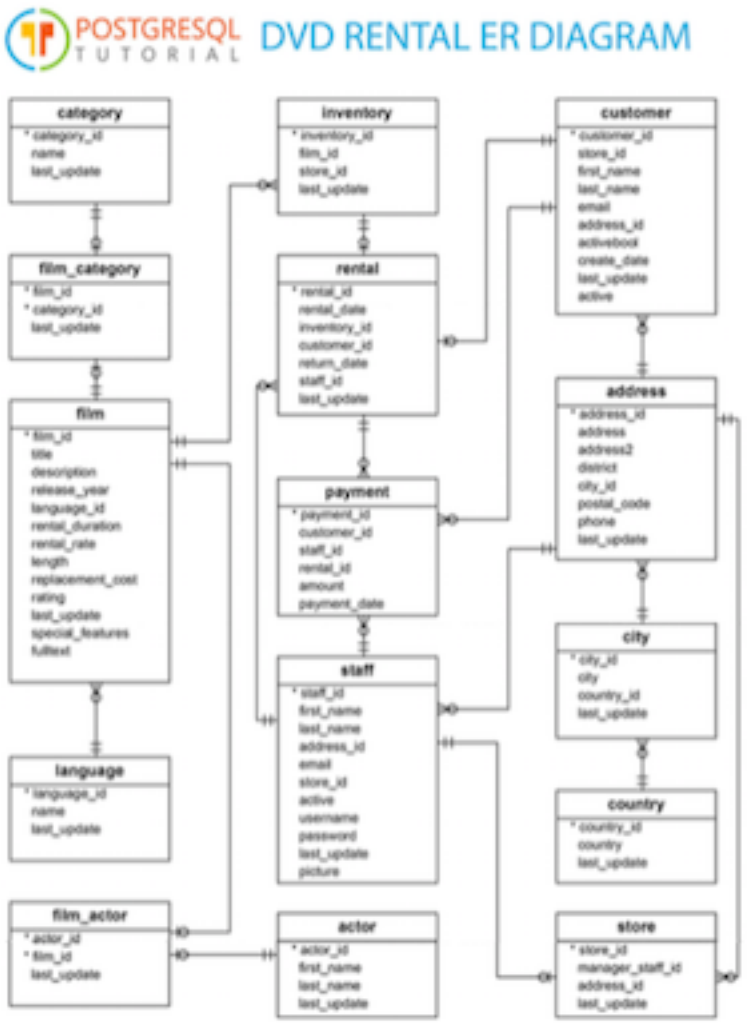


Figure 4.1: Entity Relationship diagram for the dvdrental database

4.4 Investigating a question using with an organization's database

- Need both familiarity with the data and a focus question
  - An iterative process where
    - the data resource can shape your understanding of the question
    - the question you need to answer will frame how your see the data resource
  - You need to go back and forth between the two, asking
    - do I understand the question?
    - do I understand the data?
- How well do you understand the data resource (in the DBMS)?
  - Use all available documentation and understand its limits
  - Use your own tools and skills to examine the data resource
  - what's *missing* from the database: (columns, records, cells)
  - why is there missing data?
- How well do you understand the question you seek to answer?
  - How general or specific is your question?
  - How aligned is it with the purpose for which the database was designed and is being operated?
  - How different are your assumptions and concerns from those of the people who enter and use the data on a day to day basis?

Chapter 5

Docker, Postgres, and R (04)

At the end of this chapter, you will be able to

- Run, clean-up and close Docker containers.
- See how to keep credentials secret in code that's visible to the world.
- Interact with Postgres using Rstudio inside Docker container. # Read and write to postgreSQL from 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`.

Devtools install of sqlpetr if not already installed

5.1 Verify that Docker is 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 command's full output is shown in the book.

Check that docker is up and running:

```
sp_check_that_docker_is_up()
```

```
## [1] "Docker is up, running these containers:"
## [2] "CONTAINER ID      IMAGE           COMMAND             CREATED         STATUS          PORTS
## [3] "49294fef82fb      postgres-dvdrental  \"docker-entrypoint.s…\"   4 minutes ago     Up 3 minutes
```

5.2 Clean up if appropriate

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

```
sp_docker_remove_container("cattle")
```

```
## Warning in system2("docker", docker_command, stdout = TRUE, stderr = TRUE):
## running command ''docker' rm -f cattle 2>&1' had status 1
```

```
## [1] "Error: No such container: cattle"
## attr(,"status")
## [1] 1
```

```
sp_docker_remove_container("sql-pet")
```

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

The convention we use in this book is to put docker commands in the `sqlpetr` package so that you can ignore them if you want. However, the functions are set up so that you can easily see how to do things with Docker and modify if you want.

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

```
sp_make_simple_pg("cattle")
```

```
## [1] 0
```

Docker returns a long string of numbers. If you are running this command for the first time, Docker downloads the PostgreSQL image, which takes a bit of time.

The following command shows that a container named `cattle` is running `postgres:10`. `postgres` is waiting for a connection:

```
sp_check_that_docker_is_up()
```

```
## [1] "Docker is up, running these containers:"
## [2] "CONTAINER ID      IMAGE           COMMAND             CREATED         STATUS              PORTS
## [3] "1a1991ec0377      postgres:10       \"docker-entrypoint.s…\"   1 second ago      Up Less than a secon
```

5.3 Connect, read and write to Postgres from R

5.3.1 Pause for some security considerations

We use the following `sp_get_postgres_connection` function, which will repeatedly try to connect to PostgreSQL. PostgreSQL can take different amounts of time to come up and be ready to accept connections from R, depending on various factors that will be discussed later on.

This is how the `sp_get_postgres_connection` function is used:

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

If you don't have an `.Rprofile` file that defines those passwords, you can just insert a string for the parameter, like:

```
password = 'whatever',
```

Make sure that you can connect to the PostgreSQL database that you started earlier. If you have been executing the code from this tutorial, the database will not contain any tables yet:

```
dbListTables(con)
```

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

5.3.2 Alternative: 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 PostgreSQL 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):

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.

For security, we use values from the `environment_variables` data.frame, rather than keeping the `username` and `password` in plain text in a source file.

5.3.3 Interact with Postgres

Write `mtcars` to PostgreSQL

```
dbWriteTable(con, "mtcars", mtcars, overwrite = TRUE)
```

List the tables in the PostgreSQL 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

5.4 Clean up

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

```
dbDisconnect(con)

# tell Docker to stop the container:
sp_docker_stop("cattle")
```

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

```
# Tell Docker to remove the container from it's library of active containers:
sp_docker_remove_container("cattle")
```

```
## [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 6

The dvdrental database in Postgres in Docker (05)

At the end of this chapter, you will be able to

- Setup the `dvdrental` database
- Stop and start Docker container to demonstrate persistence
- Connect to and disconnect R from the `dvdrental` database
- Execute the code in subsequent chapters

6.1 Overview

In the last chapter we connected to PostgreSQL from R. Now we set up a "realistic" database named `dvdrental`. There are two different approaches to doing this: this chapter sets it up in a way that doesn't delve into the Docker details. If you are interested, you can examine the functions provided in `sqlpetr` to see how it works or look at an alternative approach in docker-detailed-postgres-setup-with-dvdrental.R)

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

6.2 Verify that Docker is up and running

```
sp_check_that_docker_is_up()
```

```
## [1] "Docker is up but running no containers"
```

6.3 Clean up if appropriate

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

```
sp_docker_remove_container("sql-pet")
```

```
## Warning in system2("docker", docker_command, stdout = TRUE, stderr = TRUE):
## running command ''docker' rm -f sql-pet 2>&1' had status 1
```

```
## [1] "Error: No such container: sql-pet"
## attr(,"status")
```

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

6.4 Build the Docker Image

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

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

```
##  [1] "Sending build context to Docker daemon  38.63MB\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-dvdrental.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
## [11] " ---> Using cache"
## [12] " ---> f5e93aa64875"
## [13] "Successfully built f5e93aa64875"
## [14] "Successfully tagged postgres-dvdrental:latest"
```

6.5 Run the Docker Image

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

- The `source=` parameter points to dvdrental.Dockerfile, which does most of the heavy lifting. It has detailed, line-by-line comments to explain what it is doing.

- *Inside* dvdrental.Dockerfile the command `COPY init-dvdrental.sh /docker-entrypoint-initdb.d/` copies init-dvdrental.sh from the local file system into the specified location in the Docker container. When the PostgreSQL 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 543
  "--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="', # specifies the directory on the host to mount into the container at the mount point spec
```

```
  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] "b53c8d66db9bc6edf853b24efe99ee9f6393213ec05dec0aa241152784ccd886"
```

6.6 Connect to Postgres with R

Use the DBI package to connect to PostgreSQL.

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

List the tables in the database and the fields in one of those tables. Then disconnect from the database.

```
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)
```

6.7 Stop and start to demonstrate persistence

Stop the container

```
sp_docker_stop("sql-pet")
```

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

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

```
sp_docker_start("sql-pet")

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

Check that you can still see the fields in the `rental` table:

```
dbListFields(con, "rental")
```

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

6.8 Cleaning up

Always have R disconnect from the database when you're done.

```
dbDisconnect(con)
```

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

```
sp_docker_stop("sql-pet")
```

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

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

```
## [1] "CONTAINER ID      IMAGE           COMMAND             CREATED         STATUS                       POR
## [2] "b53c8d66db9b      postgres-dvdrental  \"docker-entrypoint.s…\"  7 seconds ago     Exited (0) Less t
```

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

```
sp_docker_start("sql-pet")
```

And once stopped, the container can be removed with:

```
sp_check_that_docker_is_up("sql-pet)
```

6.9 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:

```
sp_docker_stop("sql-pet")
```

Chapter 7

Mapping your local environment (10)

Start up the `docker-pet` container

```
sp_docker_start("sql-pet")
```

Now connect to the `dvdrental` database with R

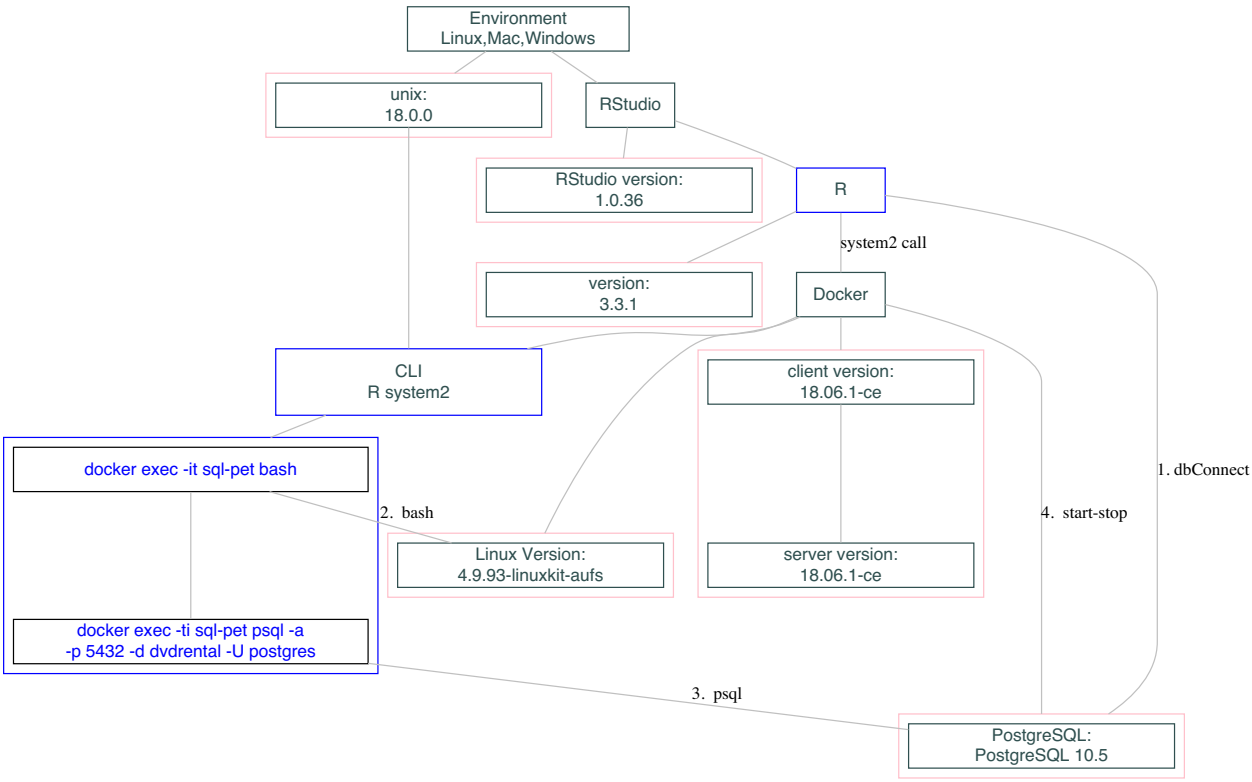
```
con <- sp_get_postgres_connection(
  user = Sys.getenv("DEFAULT_POSTGRES_USER_NAME"),
  password =  Sys.getenv("DEFAULT_POSTGRES_PASSWORD"),
  dbname = "dvdrental",
  seconds_to_test = 10)
con
```

```
## <PqConnection> dvdrental@localhost:5432
```

The following code block confirms that one can connect to the Postgres database. The connection is needed for some of the examples/exercises used in this section. If the connection is successful, the output is `<PostgreSQLConnection>`.

7.1 Tutorial Environment

Below is a high level diagram of our tutorial environment. The single black or blue boxed items are the apps running on your PC, (Linux, Mac, Windows), RStudio, R, Docker, and CLI, a command line interface. The red boxed items are the versions of the applications shown. The labels are to the right of the line.



7.2 Communicating with Docker Applications

One assumption we made is that most users use `RStudio` to interface with `R`. The four take aways from the diagram above are labeled:

1. dbConnect

R-SQL processing, the purpose of this tutorial, is performed via a database connection. This should be a simple task, but often turns out to take a lot of time to actually get it to work. We assume that your final write ups are done in some flavor of an Rmd document and others will have access to the database to confirm or further your analysis.

For this tutorial, the following are the hardcoded values used to make the Postgres database connection.

```
con <- dbConnect(drv = "PostgreSQL",
                 user = "postgres",
                 password = "postgres",
                 host = "localhost",
                 port = 5432,
                 dbname = "dvdrental"
                 )
```

The main focus of the entire tutorial is SQL processing through a dbConnection. The remainder of this section focuses on some specific Docker commands.

2. bash

The Docker container runs on top of a small Linux kernel foot print. Since Mac and Linux users run a version of Linux already, they may want to poke around the Docker environment. Below is the CLI command to start up a bash session, execute a version of hello world, and exit the `bash` session.