R, Databases and Docker

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

2018-12-22

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

1.1.1 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.
  * 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 and diagnostic techniques for the purpose should not be reinvented and would benefit from more public instruction or discussion.

- Learning to navigate the interfaces (passwords, packages, etc.) or gap 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.

- Docker is a relatively easy way to simulate the relationship between an R/Rstudio session and database – all on a single machine.

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.

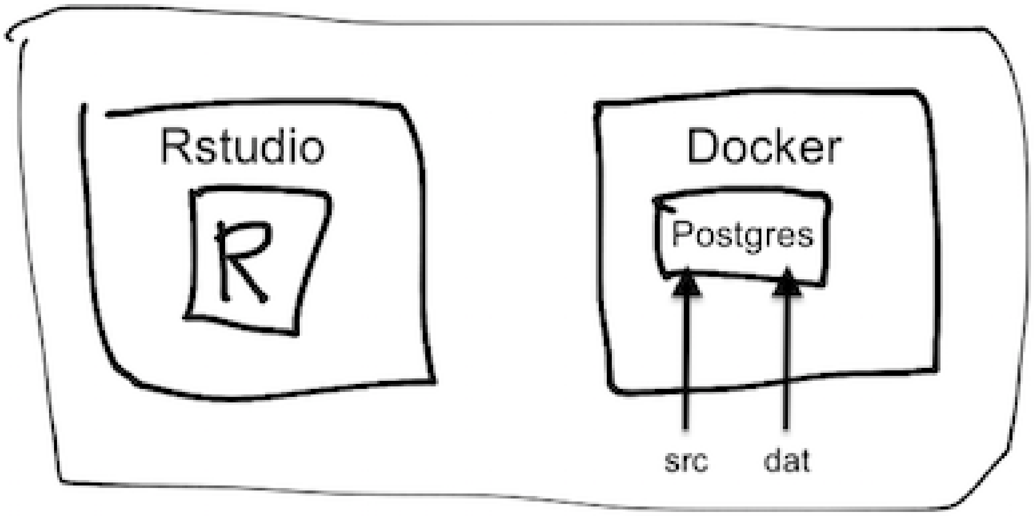
1. Make a fixed working environment for reproducible analysis
2. Access a service outside of R **(e.g., Postgres)**
3. Create an R based service (e.g., with `plumber`)
4. Send our compute jobs to the cloud with minimal reconfiguration or revision

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 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.)

1.4 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
- Maryanne Thygesen @maryannet
- Scott Came - @scottcame
- Sophie Yang - @SophieMYang

1.5 How did this project come about?

We trace this book back to the June 2, 2018 Cascadia R Conf where Aaron Makubuya gave a presentation using Vagrant hosting. After that John Smith, Ian Franz, and Sophie Yang had discussions after the monthly Data Discussion Meetups about the difficulties around setting up Vagrant, (a virtual environment), connecting to a corporate database and having realistic **public** environment to demo or practice the issues that come up behind corporate firewalls. Scott Came's tutorial on R and Docker (an alternative to Vagrant) at the 2018 UseR Conference in Melbourne was provocative and it turned out he lived nearby. We reconnected with M. Edward (Ed) Borasky who had done extensive development for a Hack Oregon data science containerization project.

Chapter 2

Setup instructions (00)

This chapter explains:

- What you need to run the code in this book
- Where to get documentation for Docker
- How you can contribute to the book project

2.1 R, RStudio and Git

Most of you will probably have these already, but if you don't:

1. If you do not have R:
   - Go to https://cran.rstudio.com/.
   - Select the download link for your system. For Linux, choose your distro. We recommend Ubuntu 18.04 LTS "Bionic Beaver". It's much easier to find support answers on the web for Ubuntu than other distros.
   - Follow the instructions.
   - Note: if you already have R, make sure it's upgraded to R 3.5.1. We don't test on older versions!
2. If you do not have RStudio: go to https://www.rstudio.com/products/rstudio/download/#download. Make sure you have version 1.1.463 or later.
3. If you do not have Git:
   - On Windows, go to https://git-scm.com/download/win and follow instructions. There are a lot of options. Just pick the defaults!!!
   - On MacOS, go to https://sourceforge.net/projects/git-osx-installer/files/ and follow instructions.
   - On Linux, install Git from your distribution.

2.2 Docker

You will need Docker Community Edition (Docker CE).

- Windows: Go to https://store.docker.com/editions/community/docker-ce-desktop-windows. If you don't have a Docker Store login, you'll need to create one. Then:
  - If you have Windows 10 Pro, download and install Docker for Windows.
  - If you have an older version of Windows, download and install Docker Toolbox (https://docs.docker.com/toolbox/overview/).
  - Note that both versions require 64-bit hardware and the virtualization needs to be enabled in the firmware.

- MacOS: Go to https://store.docker.com/editions/community/docker-ce-desktop-mac. If you don't have a Docker Store login, you'll need to create one. Then download and install Docker for Mac. Your MacOS must be at least release Yosemite (10.10.3).
- Linux: note that, as with Windows and MacOS, you'll need a Docker Store login. Although most Linux distros ship with some version of Docker, chances are it's not the same as the official Docker CE version.
  - Ubuntu: https://store.docker.com/editions/community/docker-ce-server-ubuntu,
  - Fedora: https://store.docker.com/editions/community/docker-ce-server-fedora,
  - CentOS: https://store.docker.com/editions/community/docker-ce-server-centos,
  - Debian: https://store.docker.com/editions/community/docker-ce-server-debian.

Note that on Linux, you will need to be a member of the `docker` group to use Docker. To do that, execute `sudo usermod -aG docker ${USER}`. Then, log out and back in again.

2.3 Defining the PostgreSQL connection parameters

We use a PostgreSQL database server running in a Docker container for the database functions. To connect to it, you have to define some parameters. These parameters are used in two places:

1. When the Docker container is created, they're used to initialize the database, and
2. Whenever we connect to the database, we need to specify them to authenticate.

We define the parameters in an environment file that R reads when starting up. The file is called `.Renviron`, and is located in your home directory.

The easiest way to make this file is to copy the following R code and paste it into the R console:

```
cat(
  "\nDEFAULT_POSTGRES_USER_NAME=postgres",
  file = "~/.Renviron",
  sep = "",
  append = TRUE
)
cat(
  "\nDEFAULT_POSTGRES_PASSWORD=postgres\n",
  file = "~/.Renviron",
  sep = "",
  append = TRUE
)
```

2.4 Next steps

2.4.1 Browsing the book

If you just want to read the book and copy / paste code into your working environment, simply browse to https://smithjd.github.io/sql-pet. If you get stuck, or find things aren't working, open an issue at https://github.com/smithjd/sql-pet/issues/new/.

2.4.2 Diving in

If you want to experiment with the code in the book, run it in RStudio and interact with it, you'll need to do two more things:

1. Install the `sqlpetr` R package. See https://smithjd.github.io/sqlpetr for the package documentation. This will take some time; it is installing a number of packages.
2. Clone the Git repository https://github.com/smithjd/sql-pet.git and open the project file `sql-pet.Rproj` in RStudio.

Onward!

Chapter 3

How to use this book (01)

> This chapter explains:
>
> - The prerequisites for running the code in this book
> - What R packages are used in the book

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

3.1 Prerequisites

You will need:

- A computer running
  - Windows (Windows 7 64-bit or late - Windows 10-Pro is recommended)
  - MacOS
  - Linux (any Linux distro that will run Docker Community Edition, R and RStudio will work)
- Current versions of R and RStudio required.
- Docker (instructions below)
- Our companion package `sqlpetr` installs with: `devtools::install_github("smithjd/sqlpetr")`. Note that when you install the package it will ask you to update the packages it uses and that can take some time.

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

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

- `DBI`
- `DiagrammeR`
- `RPostgres`
- `dbplyr`
- `devtools`
- `downloader`
- `glue`
- `here`
- `knitr`
- `skimr`
- `tidyverse`

- `bookdown` (for compiling the book, if you want to)

3.2 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.

3.3 Download the repo

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

3.4 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 4

Docker Hosting for Windows (02)

This chapter explains:

- How to setup your environment for Windows
- How to use Git and GitHub effectively on Windows

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

4.1 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!

4.2 Software requirements

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

4.2.1 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!

4.2.2 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.

4.3 Additional technical details

See the Chapter on Additional technical details for Windows users (95) for more information.

Chapter 5

Learning Goals and Use Cases (03)

This chapter sets the context for the book by:

- Challenging you to think about your goals and expectations
- Imagining the setting where our sample database would be used
- Posing some imaginary use cases that a data analyst might face
- Discussing the different elements involved in answering questions from an organization's database

5.1 Ask yourself, what are you 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.

5.2 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 advanced SQL techniques.
- Gain familiarity with the standard metadata that a 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.

5.3 Imagining 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 make better decision and solve problems.

5.4 Use cases

Imagine that you have one of following several roles at our fictional company **DVDs R Us** and you have a following need to be met:

- 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.

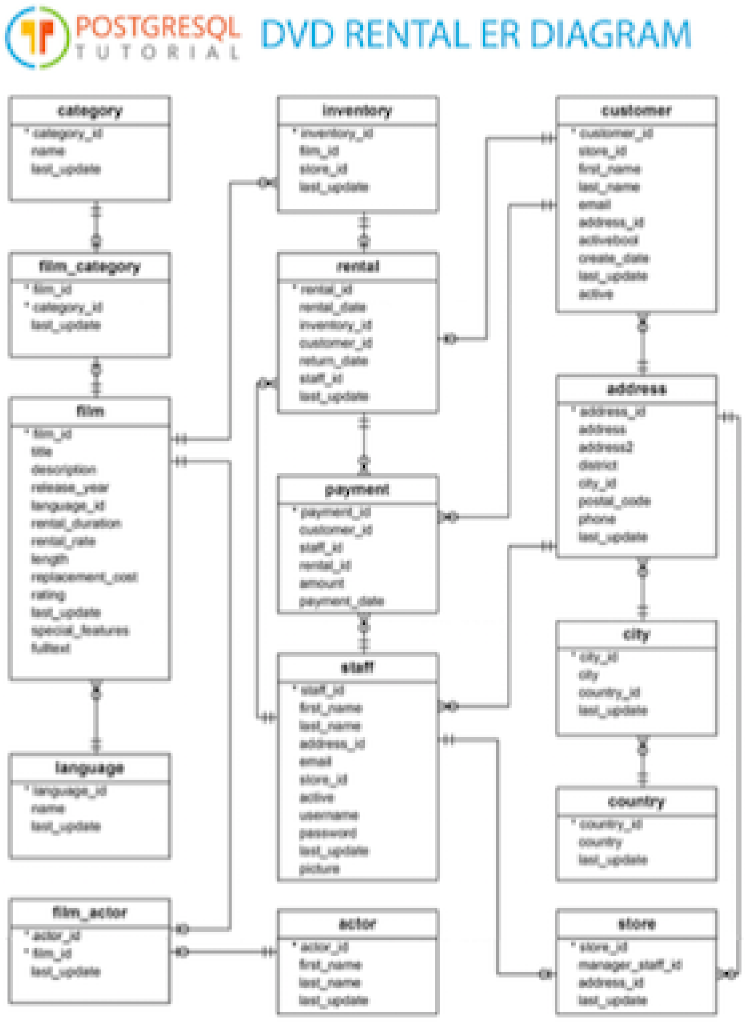


Figure 5.1: Entity Relationship diagram for the dvdrental database

- 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.

5.5 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 the 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 6

Connecting Docker, Postgres, and R (04)

This chapter demonstrates how to:

- Run, clean-up and close postgreSQL in docker containers.
- Keep necessary credentials secret while being available to R when it executes.
- Interact with PostgreSQL when it's running inside a Docker container.
- Read and write to PostgreSQL from R.

The following packages are used in this chapter:

```r
library(tidyverse)
library(DBI)
library(RPostgres)
require(knitr)
library(sqlpetr)
```

Please install the `sqlpetr` package if not already installed:

```r
library(devtools)
if (!require(sqlpetr)) devtools::install_github("smithjd/sqlpetr")
```

Note that when you install the package the first time, it will ask you to update the packages it uses and that can take some time.

6.1 Verify that Docker is running

Docker commands can be run from a terminal (e.g., the Rstudio Terminal pane) or with a `system()` command. We provide the necessary functions to start, stop Docker containers and do other busy work in the `sqlpetr` package. As time permits and curiosity dictates, feel free to look at those functions to see how they work.

Check that docker is up and running:

```r
sp_check_that_docker_is_up()
```

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

6.2 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")
```

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

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

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

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] "c22f41acb00c      postgres:10     \"docker-entrypoint.s…\"   Less than a second ago   Up Less than a
```

6.3 Connect, read and write to Postgres from R

6.3.1 Connect with Postgres

Connect to the postgrSQL using the `sp_get_postgres_connection` function:

```
con <- sp_get_postgres_connection(user = "postgres",
                                  password = "postgres",
                                  dbname = "postgres",
                                  seconds_to_test = 10)
```

Notice that we are using the postgreSQL default username and password at this point and that it's in plain text. That is bad practice because user credentials should not be shared in this way. In a subsequent chapter we'll demonstrate how to store and use credentials to access the dbms.

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)
```

6.3.2 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

6.4 Clean up

Afterwards, always disconnect from the dbms:

```
dbDisconnect(con)
```

Tell Docker to stop the `cattle` container:

```
sp_docker_stop("cattle")
```

Tell Docker to remove the `cattle` container from it's library of active containers:

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

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

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

Chapter 7

The dvdrental database in Postgres in Docker (05a)

This chapter demonstrates how to:

- Setup the `dvdrental` database in Docker
- Stop and start Docker container to demonstrate persistence
- Connect to and disconnect R from the `dvdrental` database
- Set up the environment for subsequent chapters

7.1 Overview

In the last chapter we connected to PostgreSQL from R. Now we set up a "realistic" database named `dvdrental`. There are 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 look at an alternative approach in Creating the sql-pet Docker container a step at a time that breaks the process down into smaller chunks.

These packages are called in this Chapter:

```
library(tidyverse)
library(DBI)
library(RPostgres)
library(glue)
require(knitr)
library(dbplyr)
library(sqlpetr)
library(bookdown)
```

7.2 Verify that Docker is up and running

```
sp_check_that_docker_is_up()
```

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

7.3 Clean up if appropriate

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

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

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

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

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

7.4 Build the pet-sql Docker Image

Build an image that derives from postgres:10. The commands in `dvdrental.Dockerfile` creates a Docker container running PostgreSQL, and loads the `dvdrental` database. The dvdrental.Dockerfile is discussed below.

```
docker_messages <- 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)

cat(docker_messages, sep = "\n")
```

```
## Sending build context to Docker daemon  38.29MB


## Step 1/4 : FROM postgres:10
##  ---> ac25c2bac3c4
## Step 2/4 : WORKDIR /tmp
##  ---> Using cache
##  ---> 3f00a18e0bdf
## Step 3/4 : COPY init-dvdrental.sh /docker-entrypoint-initdb.d/
##  ---> Using cache
##  ---> 3453d61d8e3e
## Step 4/4 : RUN apt-get -qq update &&  apt-get install -y -qq curl zip  > /dev/null 2>&1 &&  curl -Os http:/
##  ---> Using cache
##  ---> f5e93aa64875
## Successfully built f5e93aa64875
## Successfully tagged postgres-dvdrental:latest
```

7.5 Run the pet-sql 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. Depending on how you look at it, that may be more or less difficult to understand than an alternative method.

```
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 `docker_cmd` into a terminal window after the command 'docker':

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

```
## [1] "2e3f033f6c1755108fdc71b435cd624c30b2cc846d88a8f9fa46e37e74010d61"
```

7.6 Connect to Postgres with R

Use the DBI package to connect to the `dvdrental` database in PostgreSQL. Remember the settings discussion about [keeping passwords hidden][Pause for some security considerations]

```
con <- sp_get_postgres_connection(password = "postgres",
                         user = "postgres",
                         dbname = "dvdrental",
                         seconds_to_test = 30)
```

List the tables in the database and the fields in one of those tables.

```
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"
```

Disconnect from the database:

```
dbDisconnect(con)
```

7.7 Stop and start to demonstrate persistence

Stop the container:

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

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

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

Connect to the `dvdrental` database in postgreSQL:

```
con <- sp_get_postgres_connection(user = "postgres",
                         password = "postgres",
                         dbname = "dvdrental",
                         seconds_to_test = 30)
```

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"
```

7.8 Cleaning up

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

```
dbDisconnect(con)
```

Stop the `sql-pet` container:

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

Show that the container still exists even though it's not running

```
sp_show_all_docker_containers()
```

```
## CONTAINER ID        IMAGE                COMMAND                  CREATED             STATUS                        PORTS
## 2e3f033f6c17        postgres-dvdrental   "docker-entrypoint.s…"   7 seconds ago       Exited (0) Less than a s
```

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")
```

7.9 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_start("sql-pet")
```

Chapter 8

Securing and using your dbms credentials (05b)

This chapter demonstrates how to:

- Keep necessary credentials secret while being available to R when it executes.
- Interact with PostgreSQL using your secret dbms credentials

Connecting to a dbms can be very frustrating at first. In many organizations, simply **getting** access credentials takes time and may involve jumping through multiple hoops.

In addition, a dbms is terse or deliberately inscrutable when your credetials are incorrect. That's a security strategy, not a limitation of your understanding or your software. When R can't log you on to a dbms, you will have no information as to what went wrong.

The following packages are used in this chapter:

```
library(tidyverse)
library(DBI)
library(RPostgres)
require(knitr)
library(sqlpetr)
```

8.1 Set up the sql-pet docker container

8.1.1 Verify that Docker is running

Check that docker is up and running:

```
sp_check_that_docker_is_up()
```

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

8.1.2 Start the docker container:

Start the sql-pet docker container:

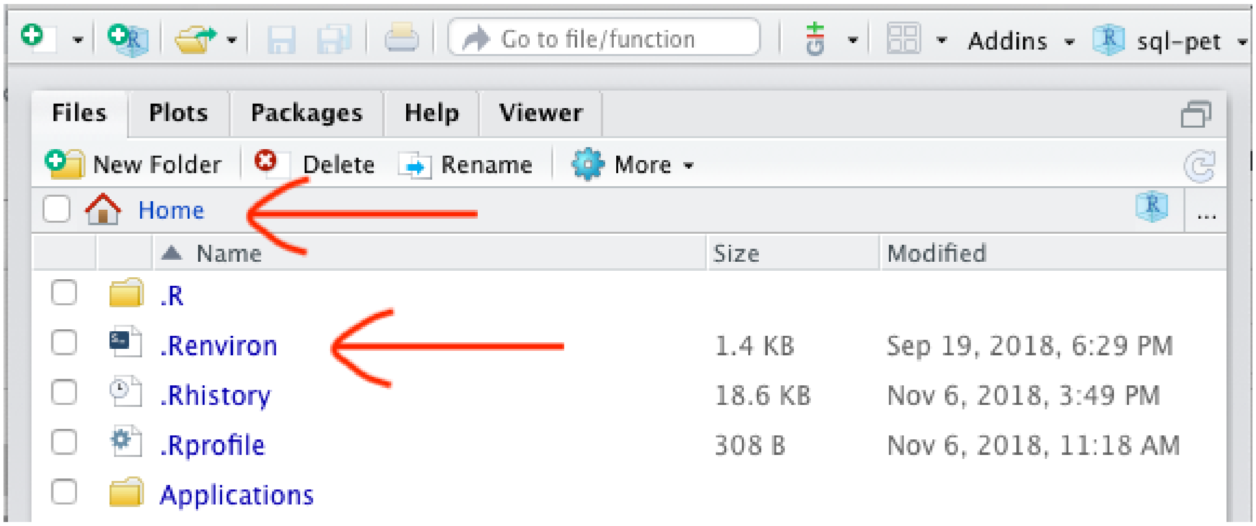
```
sp_docker_start("sql-pet")
```

8.2 Storing your dbms credentials

In previous chapters the connection string for connecting to the dbms has used default credentials specified in play text as follows:

`user= 'postgres', password = 'postgres'`

When we call `sp_get_postgres_connection` below we'll use environment variables that R obtains from reading the *.Renviron* file when R starts up. This approach has two benefits: that file is not uploaded to GitHub. R looks for it in your default directory every time it loads. To see whether you have already created that file, use the R Studio Files tab to look at your **home directory**:



Name	Size	Modified
.R		
.Renviron	1.4 KB	Sep 19, 2018, 6:29 PM
.Rhistory	18.6 KB	Nov 6, 2018, 3:49 PM
.Rprofile	308 B	Nov 6, 2018, 11:18 AM
Applications		

That file should contain lines that **look like** the example below. Although in this example it contains the PostgreSQL default values for the username and password, they are obviously not secret. But this approach demonstrates where you should put secrets that R needs while not risking accidental uploaded to GitHub or some other public location..

Open your `.Renviron` file with this command:

```
file.edit("~/.Renviron")
```

Or you can execute define_postgresql_params.R to create the file or you could copy / paste the following into your **.Renviron** file:

```
DEFAULT_POSTGRES_PASSWORD=postgres
DEFAULT_POSTGRES_USER_NAME=postgres
```

Once that file is created, restart R, and after that R reads it every time it comes up.

8.2.1 Connect with Postgres using the Sys.getenv function

Connect to the postgrSQL using the `sp_get_postgres_connection` function:

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

Once the connection object has been created, you can list all of the tables in 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"
```

8.3 Clean up

Afterwards, always disconnect from the dbms:

```
dbDisconnect(con)
```

Tell Docker to stop the `cattle` container:

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

Chapter 9

Mapping your local environment (10)

This chapter explores:

- The different entities involved in running the examples in this book's sandbox
- The different roles that each entity plays in the sandbox
- How those entities are connected and how communication between those entities happens
- Pointers to the commands that go with each entity

These packages are used in this chapter:

```
library(tidyverse)
library(DBI)
library(RPostgres)
require(knitr)
library(dbplyr)
library(sqlpetr)
library(DiagrammeR)
display_rows <- 5
```

9.1 Set up our standard pet-sql environment

Assume that the Docker container with PostgreSQL and the dvdrental database are ready to go. Start up the `docker-pet` container:

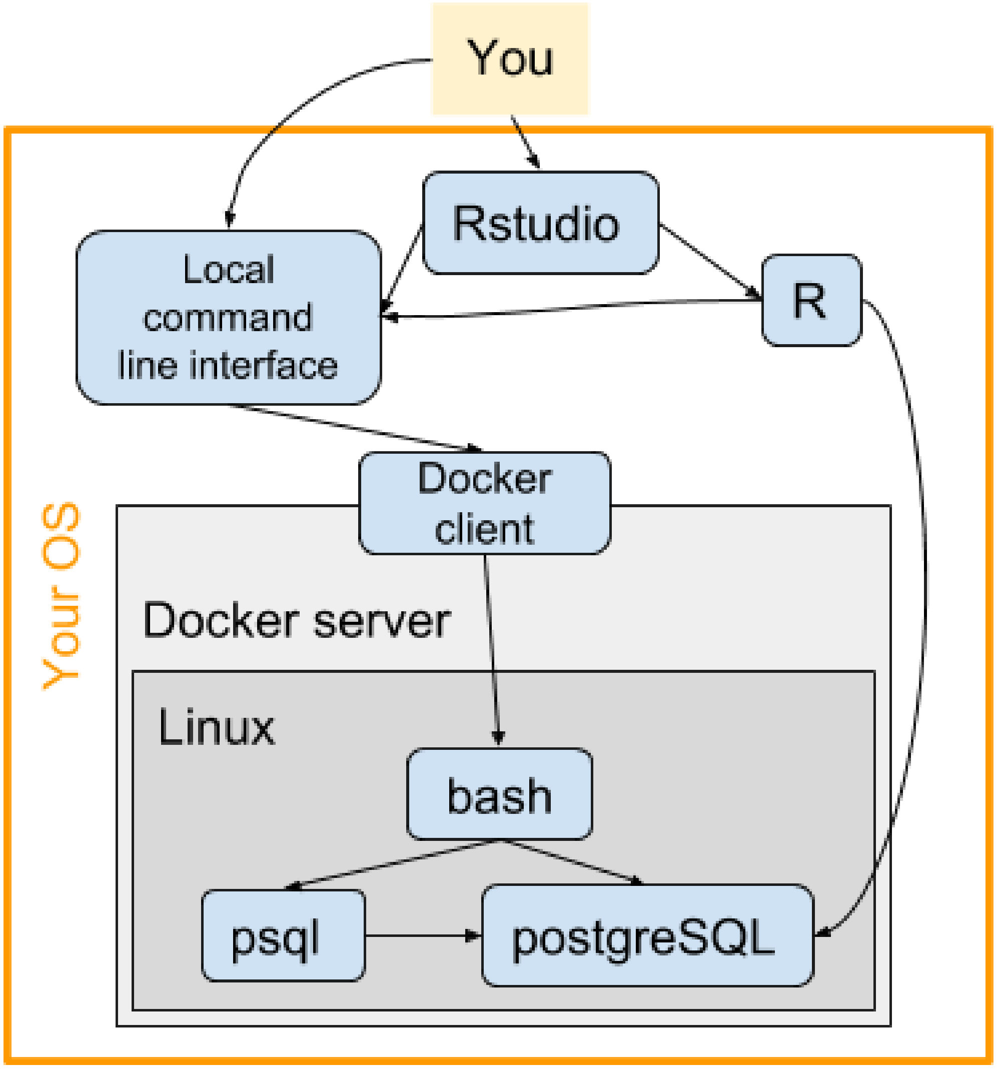
```
sp_docker_start("sql-pet")
```

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
)
```

9.2 Sandbox Environment

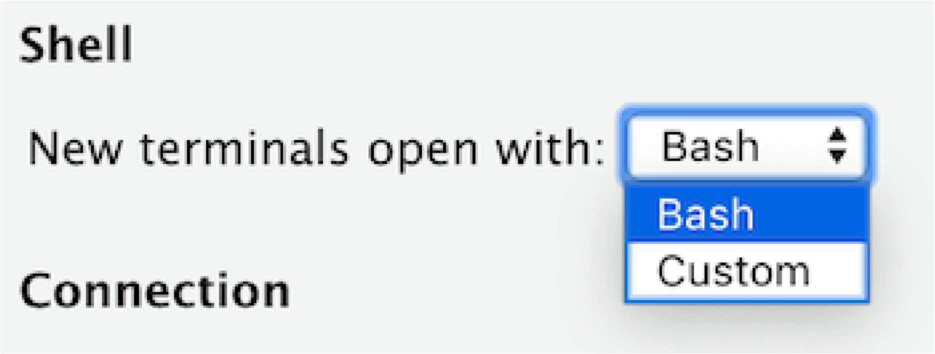
Here is an overview of our sandbox environment. In this chapter we explore each of the entities in the sandbox, how they are connected and how they communicate with each other. You can skip this chapter and come back later when you are curious about the setup that we're using in this book.



9.2.1 Sandbox entities and their roles

9.2.2 RStudio

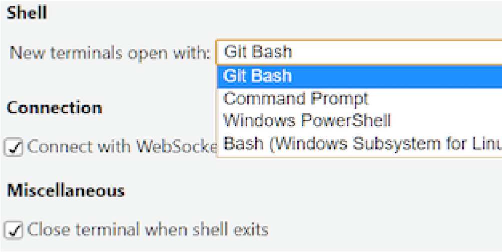
You communicate with Rstudio, which can send commands to both R and to Unix. Commands to your OS can be entered directly in the terminal pane or via an R function like `exec2()`. On a Unix or Mac computer, you typically communicate with `bash`, while you have several choices on a Windows computer.



Shell

New terminals open with: Bash

Connection



Shell

New terminals open with: Git Bash

Connection

☑ Connect with WebSocke

Miscellaneous

☑ Close terminal when shell exits

To check on the RStudio version you are using, enter this R command:

```
require(rstudioapi) versionInfo()
```

The RStudio cheat sheet is handy for learning your way around the IDE.

9.2.3 OS / local command line interface

You can type commands directly into a terminal window on your computer to communicate with your operating system (OS). It will be a `bash` prompt on a Unix or Mac, but could be one of several flavors on Windows. Our diagram conflates the operating system with the command line interface (CLI) which is a bit of a simplification as discussed below.

In addition to operating system commands, you can communicate with the Docker client through the CLI to start and stop the Docker server, load containers with programs such as Unix, postgreSQL, communicae with those programs, etc.

To check on the OS version you are using, enter this on your RStudio terminal or local CLI:

```
version -a
```

An OS can contain different comand line interfaces. Check on it with this on your RStudio terminal or local CLI:

```
echo $0
```

A Unix / Linux command line cheet sheet is a handy reference.

9.2.4 R

R processes instructions from Rstudio. It can send instructions to your OS via the `system2` function. R can also talks directly to postgreSQL through the DBI package.

R functions like `file.info("file.typ")` communicate with your operating system but do not visibly issue a command to your CLI. That's an example of an equivalence that can be useful or confusing (as in our environment diagram): you can get the same information from `ls -ql README.md` on a Unix command line as `file.info("README.md")` on the R console.

Although this sandbox seeks to make it easy, connecting to the database often involves technical and organizational hurdles like getting authorization. The main purpose of this book is to provide a sandbox for database queries to experiment with sending commands with the one of the *DBI* functions to the dbms directly from R. However, Docker and postreSQL commands are useful to know and may be necessary in extending the book's examples.

To check on the version of R that you are using, enter this on your R command line:

```
R.version
```

The growing collection of RStudio cheet sheets is indispensable.

9.2.5 Docker client

The docker client sets up the Docker server, loads containers, and passes instructions from your OS to the programs running in the Docker server. A Docker container will always contain a subset of the Linux operating system, so that it contains a second CLI in your sandbox. See more about the Docker environment.

In addition to interaction with docker through your computer's CLI or the RStudio terminal pane, the `docker` and `stevedore` packages can communicate with Docker from R. Both packages rely on the `reticulate` packaage and python.

For this book, we chose to send instructions to Docker through R's `system2()` function calls which do pass commands along to Docker through your computer's CLI. We chose that route in order to be as transparent as possible and because the book's sandbox environment is fairly simple. Although docker has different 44 commands, in this book we only use a subset: `ps`, `build`, `run`, `exec`, `start`, `stop`, and `rm`. We wrap all of these commands in `sqlpetr` package functions to encourage you to focus on R and postgreSQL.

To check on the Docker version you are using, enter this on your RStudio terminal or local CLI:

```
docker version
```

There are many Docker command line cheat sheets; this one is recommended.

9.2.6 In Docker: Linux

Docker runs a subset of the Linux operating system that in turn runs other programs like psql or postgreSQL. You may want to poke around the Linux environment inside Docker. To find what version of Linux Docker is running, enter the following command on your local CLI or in the RStudio terminal pane:

```
docker exec -ti sql-pet uname -a
```

As Linux can itself have different CLIs, enter the following command on your local CLI or in the RStudio terminal pane to find out which CLI is running inside Docker:

```
docker exec -ti sql-pet echo $0
```

To enter an interactive session inside Docker's Linux environment, enter the following command on your local CLI or in the RStudio terminal pane:

```
docker exec -ti sql-pet bash
```

To exit, enter:

```
exit
```

A Unix / Linux command line cheet sheet is a handy reference.

9.2.7 In Docker: `psql`

If you are comfortable executing SQL from a command line directly against the database, you can run the `psql` application in our Docker environment. To start up a `psql` session to investigate postgreSQL from a command line enter the following command on your computer's CLI or the RStudio terminal pane:

```
$ docker exec -ti sql-pet psql -a -p 5432 -d dvdrental -U postgres
```

Exit that environment with:

```
\q
```

Us this handy psql cheat sheet to get around.

9.2.8 In Docker: `postgreSQL`

The postgreSQL database is a whole environment unto itself. It can receive instructions through bash from `psql`, and it will respond to `DBI` queries from R on port 5282.

To check on the version of postgreSQL *client* (e.g., `psql`) you are using, enter this on your RStudio terminal or local command line interface:

```
docker exec -ti sql-pet psql --version
```

To check on the version of postgreSQL *server* you are running in Docker, enter this on your RStudio terminal or local command line interface:

```
docker exec -ti sql-pet psql -U postgres -c 'select version();'
```

Here's a recommended PostgreSQL cheat sheet.

9.3 Getting there from here: entity connections, equivalence, and commands

pathways, equivalences, command structures.

We use two trivial commands to explore the various *interfaces*. `ls -l` is the unix command for listing information about a file and `\du` is the psql command to list the users that exist in postgreSQL.

Your OS and the OS inside docker may be looking at the same file but they are in different time zones.

9.3.1 Get info on a local file from R code

```
file.info("README.md")
```

```
##           size isdir mode               mtime               ctime
## README.md 4973 FALSE  644 2018-12-22 17:12:51 2018-12-22 17:12:51
##                          atime uid gid uname grname
## README.md 2018-12-22 17:16:50 502  80   jds  admin
```

The equivalent information from executing a command on the CLI or terminal would be

```
system2("ls",  "-l README.md", stdout = TRUE, stderr = FALSE)
```

```
## [1] "-rw-r--r--  1 jds  admin  4973 Dec 22 17:12 README.md"
```

9.3.2 Get info on the same OS file inside Docker from R Code

```
system2("docker", "exec sql-pet ls -l petdir/README.md", stdout = TRUE, stderr = FALSE)
```

```
## Warning in system2("docker", "exec sql-pet ls -l petdir/README.md", stdout
## = TRUE, : running command ''docker' exec sql-pet ls -l petdir/README.md 2>/
## dev/null' had status 2
```

```
## character(0)
## attr(,"status")
## [1] 2
```

9.3.3 Docker and psql together from R or your CLI

As you become familiar with using docker, you'll see that there are various ways to do any given task. Here's an illustration of how to get a list of users who have access to the postegreSQL database.

```
system2("docker", "exec sql-pet psql -U postgres -c '\\du' ",
        stdout = TRUE, stderr = FALSE)
```

```
## [1] "                                   List of roles"
## [2] " Role name |                         Attributes                         | Member of "
## [3] "-----------+------------------------------------------------------------+-----------"
## [4] " postgres  | Superuser, Create role, Create DB, Replication, Bypass RLS | {}"
## [5] ""
```

From the RStudio terminal window, the equivalent would be a matter of dropping off some of the R code:

```
docker exec -it sql-pet psql -U postgres -c '\du'
```

9.3.4 Nesting commands illustrates how entities are connected

The following tables illustrates how the different entities communicate with each other by decomposing a command from the chapter on creating a docker container one step at a time:

```
system2("docker", "exec sql-pet pg_restore -U postgres -d dvdrental petdir/dvdrental.tar",
stdout = TRUE, stderr = TRUE)
```

Code element	Comment
`system2(`	R command to send instructions to your computer's CLI.
`"docker",`	The program (docker) on your computer that will interpret the commands passed from the `system2` function.
`"`	The entire string within the quotes is passed to docker
`exec sql-pet`	`exec` will pass a command to any program running in the `sql-pet` container.
`pg_restore`	`pg_restore` is the program inside the `sql-pet` container that processes instructions to restore a previously downloaded backup file.
`-U postgres -d dvdrental petdir/dvdrental.tar`	The `pg_restore` program requires a username, a database and a backup file to be restored.
`",`	End of the docker commands passed to the `system2` function in R.
`stdout = TRUE, stderr = TRUE)`	The `system2` function needs to know what to do with its output, which in this case is to print all of it.

9.4 Exercises

Docker containers have a small foot print. In our container, we are running a limited Linux kernel and a Postgres database. To show how tiny the docker environment is, we will look at all the processes running inside Docker and the top level file structure.

In the following exercies, use the `-i` option and the CONTAINER = `sql-pet`.

Start up R/RStudio and convert the CLI command to an R/RStudio command

#	Question	Docker CLI Command	R RStudio command	Local Command LINE
1	How many processes are running inside the Docker container?	docker exec -i sql-pet ps -eF		
1a	How many process are running on your local machine?			widows: tasklist Mac/Linux: ps -ef
2	What is the total number of files and directories in Docker?	docker exec -i sql-pet ls -al		
2a	What is the total number of files and directories on your local machine?			
3	Is Docker Running?	docker version		
3a	What are your Client and Server Versions?			
4	Does Postgres exist in the container?	docker ps -a		
4a	What is the status of Postgres?	docker ps -a		
4b	What is the size of Postgres?	docker images		
4c	What is the size of your laptop OS			https://www.quora.com/What-is-the-actual-size-of-Windows
5	If sql-pet status is Up, How do I stop it?	docker stop sql-pet		
5a	If sql-pet status is Exited, How do I start it?	docker start sql-pet		

Chapter 10

Introduction to DBMS queries (11a)

> This chapter demonstrates how to:
>
> - Get a glimpse of what tables are in the database and what fields a table contains
> - Download all or part of a table from the dbms
> - See how `dplyr` code is translated into `SQL` commands
> - Get acquainted with some useful tools for investigating a single table
> - Begin thinking about how to divide the work between your local R session and the dbms

The following packages are used in this chapter:

```r
library(tidyverse)
library(DBI)
library(RPostgres)
library(dbplyr)
require(knitr)
library(bookdown)
library(sqlpetr)
```

Assume that the Docker container with PostgreSQL and the dvdrental database are ready to go. If not go back to Chapter 7

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

Connect to the database:

```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
)
```

10.1 Getting data from the database

As we show later on, the database serves as a store of data and as an engine for sub-setting, joining, and computation on the data. We begin with getting data from the dbms, or “downloading” data.

10.1.1 Finding out what's there

We've already seen the simplest way of getting a list of tables in a database with `DBI` functions that list tables and fields. Generate a vector listing the (public) tables in the database:

```
tables <- DBI::dbListTables(con)
tables
```

```
##  [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"
```

Print a vector with all the fields (or columns or variables) in one specific table:

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

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

10.1.2 Listing all the fields for all the tables

The first example, `DBI::dbListTables(con)` returned 22 tables and the second example, `DBI::dbListFields(con, "rental")` returns 7 fields. Here we combine the two calls to return a list of tables which has a list of all the fields in the table. The code block just shows the first two tables.

```
table_columns <- lapply(tables, dbListFields, conn = con)

# or using purr:

table_columns <- map(tables, ~ dbListFields(.,conn = con) )

# rename each list [[1]] ... [[22]] to meaningful table name
names(table_columns) <- tables

head(table_columns)
```

```
## $actor_info
## [1] "actor_id"   "first_name" "last_name"  "film_info"
##
## $customer_list
## [1] "id"       "name"     "address"  "zip code" "phone"    "city"
## [7] "country"  "notes"    "sid"
##
## $film_list
## [1] "fid"         "title"       "description" "category"    "price"
## [6] "length"      "rating"      "actors"
##
## $nicer_but_slower_film_list
## [1] "fid"         "title"       "description" "category"    "price"
## [6] "length"      "rating"      "actors"
##
## $sales_by_film_category
## [1] "category"    "total_sales"
##
## $staff
##  [1] "staff_id"    "first_name"  "last_name"   "address_id"  "email"
##  [6] "store_id"    "active"      "username"    "password"    "last_update"
## [11] "picture"
```

Later on we'll discuss how to get more extensive data about each table and column from the database's own store of metadata using a similar technique. As we go further the issue of scale will come up again and again: you need to be careful about how much data a call to the dbms will return, whether it's a list of tables or a table that could have millions of rows.

It's improtant to connect with people who own, generate, or are the subjects of the data. A good chat with people who own the data, generate it, or are the subjects can generate insights and set the context for your investigation of the database. The purpose for collecting the data or circumsances where it was collected may be burried far afield in an organization, but *usually someone knows*. The metadata discussed in a later chapter is essential but will only take you so far.

There are different ways of just **looking at the data**, which we explore below.

10.1.3 Downloading an entire table

There are many different methods of getting data from a DBMS, and we'll explore the different ways of controlling each one of them.

`DBI::dbReadTable` will download an entire table into an R tibble.

```
rental_tibble <- DBI::dbReadTable(con, "rental")
str(rental_tibble)
```

```
## 'data.frame':    16044 obs. of  7 variables:
##  $ rental_id   : int  2 3 4 5 6 7 8 9 10 11 ...
##  $ rental_date : POSIXct, format: "2005-05-24 22:54:33" "2005-05-24 23:03:39" ...
##  $ inventory_id: int  1525 1711 2452 2079 2792 3995 2346 2580 1824 4443 ...
##  $ customer_id : int  459 408 333 222 549 269 239 126 399 142 ...
##  $ return_date : POSIXct, format: "2005-05-28 19:40:33" "2005-06-01 22:12:39" ...
##  $ staff_id    : int  1 1 2 1 1 2 2 1 2 2 ...
##  $ last_update : POSIXct, format: "2006-02-16 02:30:53" "2006-02-16 02:30:53" ...
```

That's very simple, but if the table is large it may not be a good idea, since R is designed to keep the entire table in memory. Note that the first line of the str() output reports the total number of observations.

10.1.4 A table object that can be reused

The `dplyr::tbl` function gives us more control over access to a table by enabling control over which columns and rows to download. It creates an object that might **look** like a data frame, but it's actually a list object that `dplyr` uses for constructing queries and retrieving data from the DBMS.

```
rental_table <- dplyr::tbl(con, "rental")
```

10.1.5 Controlling the number of rows returned

The `collect` function triggers the creation of a tibble and controls the number of rows that the DBMS sends to R.

```
rental_table %>% collect(n = 3) %>% dim
```

```
## [1] 3 7
```

```
rental_table %>% collect(n = 500) %>% dim
```

```
## [1] 500   7
```

10.1.6 Random rows from the dbms

When the dbms contains many rows, a sample of the data may be plenty for your purposes. Although `dplyr` has nice functions to sample a data frame that's already in R (e.g., the `sample_n` and `sample_frac` functions), to get a sample from the dbms we have to use `dbGetQuery` to send native SQL to the database. To peak ahead, here is one example of a query that retrieves 20 rows from a 1% sample:

```
one_percent_sample <- DBI::dbGetQuery(
  con,
  "SELECT rental_id, rental_date, inventory_id, customer_id FROM rental TABLESAMPLE SYSTEM(1) LIMIT 20;
  "
)

one_percent_sample
```

```
##    rental_id         rental_date inventory_id customer_id
## 1       9314 2005-07-30 17:05:19         2358         390
## 2       9315 2005-07-30 17:05:29          444          96
## 3       9316 2005-07-30 17:11:58         4409         366
## 4       9317 2005-07-30 17:13:37         4138         217
## 5       9318 2005-07-30 17:14:30         2426         314
## 6       9319 2005-07-30 17:15:27         4066         136
## 7       9320 2005-07-30 17:16:39          909         221
## 8       9321 2005-07-30 17:19:44         3558         112
## 9       9322 2005-07-30 17:21:39          223         439
## 10      9323 2005-07-30 17:21:44         3749         131
## 11      9324 2005-07-30 17:28:52         1231         332
## 12      9325 2005-07-30 17:29:19         1938         476
## 13      9326 2005-07-30 17:30:03         3772         588
## 14      9327 2005-07-30 17:31:03          345         373
## 15      9328 2005-07-30 17:32:11         1087          89
## 16      9329 2005-07-30 17:42:38         1293         593
## 17      9330 2005-07-30 17:44:24         4227         232
## 18      9331 2005-07-30 17:46:50         2248         274
## 19      9332 2005-07-30 17:53:39         1156         480
## 20      9333 2005-07-30 17:53:45         1377         437
```

10.1.7 Sub-setting variables

A table in the dbms may not only have many more rows than you want and also many more columns. The `select` command controls which columns are retrieved.

```
rental_table %>% select(rental_date, return_date) %>% head()
```

```
## # Source:   lazy query [?? x 2]
## # Database: postgres [postgres@localhost:5432/dvdrental]
##   rental_date         return_date
##   <dttm>              <dttm>
## 1 2005-05-24 22:54:33 2005-05-28 19:40:33
## 2 2005-05-24 23:03:39 2005-06-01 22:12:39
## 3 2005-05-24 23:04:41 2005-06-03 01:43:41
## 4 2005-05-24 23:05:21 2005-06-02 04:33:21
## 5 2005-05-24 23:08:07 2005-05-27 01:32:07
## 6 2005-05-24 23:11:53 2005-05-29 20:34:53
```

That's exactly equivalent to submitting the following SQL commands dirctly:

```
DBI::dbGetQuery(
  con,
  'SELECT "rental_date", "return_date"
FROM "rental"
LIMIT 6')
```

```
##           rental_date         return_date
## 1 2005-05-24 22:54:33 2005-05-28 19:40:33
## 2 2005-05-24 23:03:39 2005-06-01 22:12:39
## 3 2005-05-24 23:04:41 2005-06-03 01:43:41
## 4 2005-05-24 23:05:21 2005-06-02 04:33:21
## 5 2005-05-24 23:08:07 2005-05-27 01:32:07
## 6 2005-05-24 23:11:53 2005-05-29 20:34:53
```

We won't discuss `dplyr` methods for sub-setting variables, deriving new ones, or sub-setting rows based on the values found in the table because they are covered well in other places, including:

- Comprehensive reference: https://dplyr.tidyverse.org/
- Good tutorial: https://suzan.rbind.io/tags/dplyr/

In practice we find that, **renaming variables** is often quite important because the names in an SQL database might not meet your needs as an analyst. In "the wild" you will find names that are ambiguous or overly specified, with spaces in them, and other problems that will make them difficult to use in R. It is good practice to do whatever renaming you are going to do in a predictable place like at the top of your code. The names in the `dvdrental` database are simple and clear, but if they were not, you might rename them for subsequent use in this way:

```
tbl(con, "rental") %>%
  rename(rental_id_number = rental_id, inventory_id_number = inventory_id) %>%
  select(rental_id_number, rental_date, inventory_id_number) %>%
  head()
```

```
## # Source:   lazy query [?? x 3]
## # Database: postgres [postgres@localhost:5432/dvdrental]
##   rental_id_number rental_date         inventory_id_number
##              <int> <dttm>                            <int>
## 1                2 2005-05-24 22:54:33                1525
## 2                3 2005-05-24 23:03:39                1711
## 3                4 2005-05-24 23:04:41                2452
## 4                5 2005-05-24 23:05:21                2079
## 5                6 2005-05-24 23:08:07                2792
## 6                7 2005-05-24 23:11:53                3995
```

That's equivalent to the following SQL code:

```
DBI::dbGetQuery(
  con,
  'SELECT "rental_id_number", "rental_date", "inventory_id_number"
FROM (SELECT "rental_id" AS "rental_id_number", "rental_date", "inventory_id" AS "inventory_id_number",
FROM "rental") "ihebfvnxvb"
LIMIT 6' )
```

```
##   rental_id_number         rental_date inventory_id_number
## 1                2 2005-05-24 22:54:33                1525
## 2                3 2005-05-24 23:03:39                1711
## 3                4 2005-05-24 23:04:41                2452
## 4                5 2005-05-24 23:05:21                2079
## 5                6 2005-05-24 23:08:07                2792
## 6                7 2005-05-24 23:11:53                3995
```

The one difference is that the SQL code returns a regular data frame and the dplyr code returns a tibble. Notice that the seconds are greyed out in the tibble display.

10.1.8 Translating dplyr code to SQL queries

Where did the translations we've showon above come from? The show_query function shows how dplyr is translating your query to the dialect of the target dbms:

```
rental_table %>%
  count(staff_id) %>%
  show_query()
```

```
## <SQL>
## SELECT "staff_id", COUNT(*) AS "n"
## FROM "rental"
## GROUP BY "staff_id"
```

Here is an extensive discussion of how dplyr code is translated into SQL:

- https://dbplyr.tidyverse.org/articles/sql-translation.html

The SQL code can submit the same query directly to the DBMS with the DBI::dbGetQuery function:

```
DBI::dbGetQuery(
  con,
  'SELECT "staff_id", COUNT(*) AS "n"
   FROM "rental"
   GROUP BY "staff_id";
  '
)
```

```
##   staff_id    n
## 1        2 8004
## 2        1 8040
```

<<smy We haven't investigated this, but it looks like dplyr collect() function triggers a call simmilar to the dbGetQuery call above. The default dplyr behavior looks like dbSendQuery() and dbFetch() model is used.>>

When you create a report to run repeatedly, you might want to put that query into R markdown. That way you can also execute that SQL code in a chunk with the following header:

{sql, connection=con, output.var = "query_results"}

```
SELECT "staff_id", COUNT(*) AS "n"
FROM "rental"
GROUP BY "staff_id";
```

Rmarkdown stores that query result in a tibble which can be printed by referring to it:

```
query_results
```

```
##   staff_id    n
## 1        2 8004
## 2        1 8040
```

10.2 Examining a single table with R

Dealing with a large, complex database highlights the utility of specific tools in R. We include brief examples that we find to be handy:

- Base R structure: `str`
- printing out some of the data: `datatable`, `kable`, and `View`
- summary statistics: `summary`
- `glimpse` oin the `tibble` package, which is included in the `tidyverse`
- `skim` in the `skimr` package

10.2.1 str - a base package workhorse

`str` is a workhorse function that lists variables, their type and a sample of the first few variable values.

```
str(rental_tibble)
```

```
## 'data.frame':    16044 obs. of  7 variables:
##  $ rental_id   : int  2 3 4 5 6 7 8 9 10 11 ...
##  $ rental_date : POSIXct, format: "2005-05-24 22:54:33" "2005-05-24 23:03:39" ...
##  $ inventory_id: int  1525 1711 2452 2079 2792 3995 2346 2580 1824 4443 ...
##  $ customer_id : int  459 408 333 222 549 269 239 126 399 142 ...
##  $ return_date : POSIXct, format: "2005-05-28 19:40:33" "2005-06-01 22:12:39" ...
##  $ staff_id    : int  1 1 2 1 1 2 2 1 2 2 ...
##  $ last_update : POSIXct, format: "2006-02-16 02:30:53" "2006-02-16 02:30:53" ...
```

10.2.2 Always look at your data with head, View, or kable

There is no substitute for looking at your data and R provides several ways to just browse it. The `head` function controls the number of rows that are displayed. Note that tail does not work against a database object. In every-day practice you would look at more than the default 6 rows, but here we wrap `head` around the data frame:

```
sp_print_df(head(rental_tibble))
```

rental_id	rental_date	inventory_id	customer_id	return_date	staff_id	last_update
2	2005-05-24 22:54:33	1525	459	2005-05-28 19:40:33	1	2006-02-16 02:30:53
3	2005-05-24 23:03:39	1711	408	2005-06-01 22:12:39	1	2006-02-16 02:30:53
4	2005-05-24 23:04:41	2452	333	2005-06-03 01:43:41	2	2006-02-16 02:30:53
5	2005-05-24 23:05:21	2079	222	2005-06-02 04:33:21	1	2006-02-16 02:30:53
6	2005-05-24 23:08:07	2792	549	2005-05-27 01:32:07	1	2006-02-16 02:30:53
7	2005-05-24 23:11:53	3995	269	2005-05-29 20:34:53	2	2006-02-16 02:30:53

10.2.3 The `summary` function in base

The basic statistics that the base package `summary` provides can serve a unique diagnostic purpose in this context. For example, the following output shows that `rental_id` is a sequential number from 1 to 16,049 with no gaps. The same is true of `inventory_id`. The number of NA's is a good first guess as to the number of dvd's rented out or lost on 2005-09-02 02:35:22.

```
summary(rental_tibble)
```

```
##    rental_id      rental_date                   inventory_id
##  Min.   :    1   Min.   :2005-05-24 22:53:30   Min.   :   1
##  1st Qu.: 4014   1st Qu.:2005-07-07 00:58:40   1st Qu.:1154
##  Median : 8026   Median :2005-07-28 16:04:32   Median :2291
##  Mean   : 8025   Mean   :2005-07-23 08:13:34   Mean   :2292
##  3rd Qu.:12037   3rd Qu.:2005-08-17 21:16:23   3rd Qu.:3433
##  Max.   :16049   Max.   :2006-02-14 15:16:03   Max.   :4581
##
##   customer_id     return_date                     staff_id
##  Min.   :  1.0   Min.   :2005-05-25 23:55:21   Min.   :1.000
##  1st Qu.:148.0   1st Qu.:2005-07-10 15:49:36   1st Qu.:1.000
##  Median :296.0   Median :2005-08-01 19:45:29   Median :1.000
##  Mean   :297.1   Mean   :2005-07-25 23:58:03   Mean   :1.499
##  3rd Qu.:446.0   3rd Qu.:2005-08-20 23:35:55   3rd Qu.:2.000
##  Max.   :599.0   Max.   :2005-09-02 02:35:22   Max.   :2.000
##                  NA's   :183
##   last_update
##  Min.   :2006-02-15 21:30:53
##  1st Qu.:2006-02-16 02:30:53
##  Median :2006-02-16 02:30:53
##  Mean   :2006-02-16 02:31:31
##  3rd Qu.:2006-02-16 02:30:53
##  Max.   :2006-02-23 09:12:08
##
```

10.2.4 The `glimpse` function in the `tibble` package

The `tibble` package's `glimpse` function is a more compact version of `str`:

```
tibble::glimpse(rental_tibble)
```

```
## Observations: 16,044
## Variables: 7
## $ rental_id    <int> 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 1...
## $ rental_date  <dttm> 2005-05-24 22:54:33, 2005-05-24 23:03:39, 2005-0...
## $ inventory_id <int> 1525, 1711, 2452, 2079, 2792, 3995, 2346, 2580, 1...
## $ customer_id  <int> 459, 408, 333, 222, 549, 269, 239, 126, 399, 142,...
## $ return_date  <dttm> 2005-05-28 19:40:33, 2005-06-01 22:12:39, 2005-0...
## $ staff_id     <int> 1, 1, 2, 1, 1, 2, 2, 1, 2, 2, 2, 1, 1, 1, 2, 1, 2...
## $ last_update  <dttm> 2006-02-16 02:30:53, 2006-02-16 02:30:53, 2006-0...
```

10.2.5 The `skim` function in the `skimr` package

The `skimr` package has several functions that make it easy to examine an unknown data frame and assess what it contains. It is also extensible.

```r
library(skimr)
```

```
##
## Attaching package: 'skimr'
## The following object is masked from 'package:knitr':
##
##     kable
```

```r
skim(rental_tibble)
```

```
## Skim summary statistics
##  n obs: 16044
##  n variables: 7
##
## -- Variable type:integer -----------------------------------------------------------------------------
##      variable missing complete     n    mean       sd p0     p25    p50
##   customer_id       0    16044 16044  297.14  172.45  1  148     296
##  inventory_id       0    16044 16044 2291.84 1322.21  1 1154    2291
##     rental_id       0    16044 16044 8025.37 4632.78  1 4013.75 8025.5
##      staff_id       0    16044 16044    1.5     0.5   1    1       1
##       p75  p100     hist
##    446      599
##   3433     4581
##  12037.25 16049
##      2        2
##
## -- Variable type:POSIXct -----------------------------------------------------------------------------
##     variable missing complete     n        min        max     median
##  last_update       0    16044 16044 2006-02-15 2006-02-23 2006-02-16
##  rental_date       0    16044 16044 2005-05-24 2006-02-14 2005-07-28
##  return_date     183    15861 16044 2005-05-25 2005-09-02 2005-08-01
##  n_unique
##         3
##     15815
##     15836
```

```r
wide_rental_skim <- skim_to_wide(rental_tibble)
```

10.2.6 Close the connection and shut down sql-pet

Where you place the `collect` function matters.

```r
dbDisconnect(con)
sp_docker_stop("sql-pet")
```

10.3 Additional reading

- ?
- ?

Chapter 11

Lazy Evaluation and Lazy Queries (11b)

11.1 This chapter:

- Reviews lazy evaluation and discusses its interaction with remote query execution on a dbms
- Demonstrates how `dplyr` queries behave in connection with several different functions
- Offers some further resources on lazy loading, evaluation, execution, etc.

11.1.1 Setup

The following packages are used in this chapter:

```r
library(tidyverse)
library(DBI)
library(RPostgres)
library(dbplyr)
require(knitr)
library(bookdown)
library(sqlpetr)
```

Assume that the Docker container with PostgreSQL and the dvdrental database are ready to go. If not go back to the previous Chapter

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

Connect to the database:

```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
)
```

11.2 R is lazy and comes with guardrails

By design, R is both a language and an interactive development environment (IDE). As a language, R tries to be as efficient as possible. As an IDE, R creates some guardrails to make it easy and safe to work with your data. For example `getOption("max.print")` prevents R from printing more rows of data than you can handle, with a nice default of 99999, which may or may not suit you.

On the other hand SQL is a *"Structured Query Language (SQL): a standard computer language for relational database management and data manipulation."*.[1] SQL has has various database-specific Interactive Development Environments (IDEs): for postgreSQL it's pgAdmin. Roger Peng explains in R Programming for Data Science that:

> R has maintained the original S philosophy, which is that it provides a language that is both useful for interactive work, but contains a powerful programming language for developing new tools.

This is complicated when R interacts with SQL. In the vignette for dbplyr Hadley Wikham explains:

> The most important difference between ordinary data frames and remote database queries is that your R code is translated into SQL and executed in the database on the remote server, not in R on your local machine. When working with databases, dplyr tries to be as lazy as possible:
>
> - It never pulls data into R unless you explicitly ask for it.
> - It delays doing any work until the last possible moment: it collects together everything you want to do and then sends it to the database in one step.

Exactly when, which and how much data is returned from the dbms is the topic of this chapter. Exactly how the data is represented in the dbms and then translated to a data frame is discussed in the DBI specification.

Eventually, if you are interacting with a dbms from R you will need to understand the differences between lazy loading, lazy evaluation, and lazy queries.

11.2.1 Lazy loading

"*Lazy loading is always used for code in packages but is optional (selected by the package maintainer) for datasets in packages.*"[2] Lazy loading means that the code for a particular function doesn't actually get loaded into memory until the last minute – when it's actually being used.

11.2.2 Lazy evaluation

Essentially "Lazy evaluation is a programming strategy that allows a symbol to be evaluated only when needed."[3] That means that lazy evaluation is about **symbols** such as function arguments[4] when they are evaluated. Tidy evaluation complicates lazy evaluation.[5]

11.2.3 Lazy Queries

"*When you create a "lazy" query, you're creating a pointer to a set of conditions on the database, but the query isn't actually run and the data isn't actually loaded until you call "next" or some similar method to*

[1] https://www.techopedia.com/definition/1245/structured-query-language-sql

[2] https://cran.r-project.org/doc/manuals/r-release/R-ints.html#Lazy-loading

[3] https://colinfay.me/lazyeval/

[4] http://adv-r.had.co.nz/Functions.html#function-arguments

[5] https://colinfay.me/tidyeval-1/

actually fetch the data and load it into an object."[6] The `collect()` function retrieves data into a local tibble.[7]

11.3 Lazy evaluation and lazy queries

11.3.1 Dplyr connection objects

As introduced in the previous chapter, the `dplyr::tbl` function creates an object that might **look** like a data frame in that when you enter it on the command line, it prints a bunch of rows from the dbms table. But is actually a **list** object that `dplyr` uses for constructing queries and retrieving data from the DBMS.

The following code illustrates these issues. The `dplyr::tbl` function creates the connection object that we store in an object named `rental_table`:

```
rental_table <- dplyr::tbl(con, "rental")
```

At first glance, it kind of **looks** like a data frame although it only prints 10 of the table's 16,044 rows:

```
rental_table
```

```
## # Source:   table<rental> [?? x 7]
## # Database: postgres [postgres@localhost:5432/dvdrental]
##    rental_id rental_date         inventory_id customer_id
##        <int> <dttm>                     <int>       <int>
##  1         2 2005-05-24 22:54:33         1525         459
##  2         3 2005-05-24 23:03:39         1711         408
##  3         4 2005-05-24 23:04:41         2452         333
##  4         5 2005-05-24 23:05:21         2079         222
##  5         6 2005-05-24 23:08:07         2792         549
##  6         7 2005-05-24 23:11:53         3995         269
##  7         8 2005-05-24 23:31:46         2346         239
##  8         9 2005-05-25 00:00:40         2580         126
##  9        10 2005-05-25 00:02:21         1824         399
## 10        11 2005-05-25 00:09:02         4443         142
## # ... with more rows, and 3 more variables: return_date <dttm>,
## #   staff_id <int>, last_update <dttm>
```

But consider the structure of `rental_table`:

```
str(rental_table)
```

```
## List of 2
##  $ src:List of 2
##   ..$ con  :Formal class 'PqConnection' [package "RPostgres"] with 3 slots
##   .. .. ..@ ptr     :<externalptr>
##   .. .. ..@ bigint  : chr "integer64"
##   .. .. ..@ typnames:'data.frame':   437 obs. of  2 variables:
##   .. .. .. ..$ oid    : int [1:437] 16 17 18 19 20 21 22 23 24 25 ...
##   .. .. .. ..$ typname: chr [1:437] "bool" "bytea" "char" "name" ...
##   ..$ disco: NULL
##   ..- attr(*, "class")= chr [1:3] "src_dbi" "src_sql" "src"
##  $ ops:List of 2
##   ..$ x   : 'ident' chr "rental"
##   ..$ vars: chr [1:7] "rental_id" "rental_date" "inventory_id" "customer_id" ...
```

[6] https://www.quora.com/What-is-a-lazy-query

[7] https://dplyr.tidyverse.org/reference/compute.html

```
##   ..- attr(*, "class")= chr [1:3] "op_base_remote" "op_base" "op"
##  - attr(*, "class")= chr [1:4] "tbl_dbi" "tbl_sql" "tbl_lazy" "tbl"
```

It has two rows. The first row contains all the information in the `con` object, which contains information about all the tables and objects in the database:

```
rental_table$src$con@typnames$typname[380:437]
```

```
##  [1] "customer"                    "_customer"
##  [3] "actor_actor_id_seq"          "actor"
##  [5] "_actor"                      "category_category_id_seq"
##  [7] "category"                    "_category"
##  [9] "film_film_id_seq"            "film"
## [11] "_film"                       "pg_toast_16434"
## [13] "film_actor"                  "_film_actor"
## [15] "film_category"               "_film_category"
## [17] "actor_info"                  "_actor_info"
## [19] "address_address_id_seq"      "address"
## [21] "_address"                    "city_city_id_seq"
## [23] "city"                        "_city"
## [25] "country_country_id_seq"      "country"
## [27] "_country"                    "customer_list"
## [29] "_customer_list"              "film_list"
## [31] "_film_list"                  "inventory_inventory_id_seq"
## [33] "inventory"                   "_inventory"
## [35] "language_language_id_seq"    "language"
## [37] "_language"                   "nicer_but_slower_film_list"
## [39] "_nicer_but_slower_film_list" "payment_payment_id_seq"
## [41] "payment"                     "_payment"
## [43] "rental_rental_id_seq"        "rental"
## [45] "_rental"                     "sales_by_film_category"
## [47] "_sales_by_film_category"     "staff_staff_id_seq"
## [49] "staff"                       "_staff"
## [51] "pg_toast_16529"              "store_store_id_seq"
## [53] "store"                       "_store"
## [55] "sales_by_store"              "_sales_by_store"
## [57] "staff_list"                  "_staff_list"
```

The second row contains a list of the columns in the `rental` table, among other things:

```
rental_table$ops$vars
```

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

To illustrate the different issues involved in data retrieval, we create equivalent connection objects to link to two other tables.

```
staff_table <- dplyr::tbl(con, "staff")
# the 'staff' table has 2 rows

customer_table <- dplyr::tbl(con, "customer")
# the 'customer' table has 599 rows
```

11.4 When does a lazy query trigger data retrieval?

11.4.1 Create a black box query for experimentation

Here is a typical string of dplyr verbs strung together with the magrittr `%>%` command that will be used to tease out the several different behaviors that a lazy query has when passed to different R functions. This query joins three connection objects into a query we'll call `Q`:

```
Q <- rental_table %>%
  left_join(staff_table, by = c("staff_id" = "staff_id")) %>%
  rename(staff_email = email) %>%
  left_join(customer_table, by = c("customer_id" = "customer_id")) %>%
  rename(customer_email = email) %>%
  select(rental_date, staff_email, customer_email)
```

11.4.2 Experiment overview

Think of `Q` as a black box for the moment. The following examples will show how `Q` is interpreted differently by different functions. In this table, a single green check indicates that some rows are returned, two green checks indicates that all the rows are returned, and the red X indicates that no rows have are returned.

R code	Result
`Q %>% print()`	✓ Prints x rows; same as just entering `Q`
`Q %>% as.tibble()`	✓ ✓ Forces `Q` to be a tibble
`Q %>% head()`	✓ Prints the first 6 rows
`Q %>% length()`	✗ Counts the rows in `Q`
`Q %>% str()`	✗ Shows the top 3 levels of the **object** `Q`
`Q %>% nrow()`	✗ **Attempts** to determine the number of rows
`Q %>% tally()`	✓ ✓ Counts all the rows – on the dbms side
`Q %>% collect(n = 20)`	✓ Prints 20 rows

R code	Result
`Q %>% collect(n = 20) %>% head()`	✓ Prints 6 rows
`Q %>% show_query()`	✗ **Translates** the lazy query object into SQL
`Qc <- Q %>% count(customer_email, sort = TRUE) Qc`	✗ **Extends** the lazy query object

(The next chapter will discuss how to build queries and how to explore intermediate steps.)

11.4.3 Q %>% print()

Remember that `Q %>% print()` is equivalent to `print(Q)` and the same as just entering `Q` on the command line. We use the magrittr pipe operator here because chaining functions highlights how the same object behaves differently in each use.

```
Q %>% print()
```

```
## # Source:   lazy query [?? x 3]
## # Database: postgres [postgres@localhost:5432/dvdrental]
##    rental_date         staff_email              customer_email
##    <dttm>              <chr>                    <chr>
##  1 2005-05-24 22:54:33 Mike.Hillyer@sakilasta~ tommy.collazo@sakilacustome~
##  2 2005-05-24 23:03:39 Mike.Hillyer@sakilasta~ manuel.murrell@sakilacustom~
##  3 2005-05-24 23:04:41 Jon.Stephens@sakilasta~ andrew.purdy@sakilacustomer~
##  4 2005-05-24 23:05:21 Mike.Hillyer@sakilasta~ delores.hansen@sakilacustom~
##  5 2005-05-24 23:08:07 Mike.Hillyer@sakilasta~ nelson.christenson@sakilacu~
##  6 2005-05-24 23:11:53 Jon.Stephens@sakilasta~ cassandra.walters@sakilacus~
##  7 2005-05-24 23:31:46 Jon.Stephens@sakilasta~ minnie.romero@sakilacustome~
##  8 2005-05-25 00:00:40 Mike.Hillyer@sakilasta~ ellen.simpson@sakilacustome~
##  9 2005-05-25 00:02:21 Jon.Stephens@sakilasta~ danny.isom@sakilacustomer.o~
## 10 2005-05-25 00:09:02 Jon.Stephens@sakilasta~ april.burns@sakilacustomer.~
## # ... with more rows
```

✓ R retrieves 10 observations and 3 columns. In its role as IDE, R has provided nicely formatted output that is similar to what it prints for a tibble, with descriptive information about the dataset and each column:

> # Source: lazy query [?? x 3] # Database: postgres [postgres@localhost:5432/dvdrental] rental_date staff_email customer_email <dttm> <chr> <chr>

R has not determined how many rows are left to retrieve as it notes `... with more rows`.

11.4.4 Q %>% as.tibble()

✓✓ In contrast to `print()`, the `as.tibble()` function causes R to download the whole table, using tibble's default of displaying only the first 10 rows.

```
Q %>% as.tibble()
```

```
## # A tibble: 16,044 x 3
##    rental_date         staff_email              customer_email
##    <dttm>              <chr>                    <chr>
##  1 2005-05-24 22:54:33 Mike.Hillyer@sakilasta~ tommy.collazo@sakilacustome~
##  2 2005-05-24 23:03:39 Mike.Hillyer@sakilasta~ manuel.murrell@sakilacustom~
##  3 2005-05-24 23:04:41 Jon.Stephens@sakilasta~ andrew.purdy@sakilacustomer~
##  4 2005-05-24 23:05:21 Mike.Hillyer@sakilasta~ delores.hansen@sakilacustom~
##  5 2005-05-24 23:08:07 Mike.Hillyer@sakilasta~ nelson.christenson@sakilacu~
##  6 2005-05-24 23:11:53 Jon.Stephens@sakilasta~ cassandra.walters@sakilacus~
##  7 2005-05-24 23:31:46 Jon.Stephens@sakilasta~ minnie.romero@sakilacustome~
##  8 2005-05-25 00:00:40 Mike.Hillyer@sakilasta~ ellen.simpson@sakilacustome~
##  9 2005-05-25 00:02:21 Jon.Stephens@sakilasta~ danny.isom@sakilacustomer.o~
## 10 2005-05-25 00:09:02 Jon.Stephens@sakilasta~ april.burns@sakilacustomer.~
## # ... with 16,034 more rows
```

11.4.5 Q %>% head()

✓ The `head()` function is very similar to print but has a different "`max.print`" value.

```
Q %>% head()
```

```
## # Source:   lazy query [?? x 3]
## # Database: postgres [postgres@localhost:5432/dvdrental]
##   rental_date         staff_email              customer_email
##   <dttm>              <chr>                    <chr>
## 1 2005-05-24 22:54:33 Mike.Hillyer@sakilasta~ tommy.collazo@sakilacustomer~
## 2 2005-05-24 23:03:39 Mike.Hillyer@sakilasta~ manuel.murrell@sakilacustome~
## 3 2005-05-24 23:04:41 Jon.Stephens@sakilasta~ andrew.purdy@sakilacustomer.~
## 4 2005-05-24 23:05:21 Mike.Hillyer@sakilasta~ delores.hansen@sakilacustome~
## 5 2005-05-24 23:08:07 Mike.Hillyer@sakilasta~ nelson.christenson@sakilacus~
## 6 2005-05-24 23:11:53 Jon.Stephens@sakilasta~ cassandra.walters@sakilacust~
```

11.4.6 Q %>% length()

✗ Because the `Q` object is relatively complex, using `str()` on it prints many lines. You can glimpse what's going on with `length()`:

```
Q %>% length()
```

```
## [1] 2
```

11.4.7 Q %>% str()

✗ Looking inside shows some of what's going on (three levels deep):

```
Q %>% str(max.level = 3)
```

```
## List of 2
##  $ src:List of 2
##   ..$ con  :Formal class 'PqConnection' [package "RPostgres"] with 3 slots
##   ..$ disco: NULL
##   ..- attr(*, "class")= chr [1:3] "src_dbi" "src_sql" "src"
##  $ ops:List of 4
##   ..$ name: chr "select"
##   ..$ x   :List of 4
##   .. ..$ name: chr "rename"
##   .. ..$ x   :List of 4
##   .. .. ..- attr(*, "class")= chr [1:3] "op_join" "op_double" "op"
##   .. ..$ dots:List of 1
##   .. ..$ args: list()
##   .. ..- attr(*, "class")= chr [1:3] "op_rename" "op_single" "op"
##   ..$ dots:List of 3
##   .. ..$ : language ~rental_date
##   .. .. ..- attr(*, ".Environment")=<environment: 0x7f9241118658>
##   .. ..$ : language ~staff_email
##   .. .. ..- attr(*, ".Environment")=<environment: 0x7f9241118658>
##   .. ..$ : language ~customer_email
##   .. .. ..- attr(*, ".Environment")=<environment: 0x7f9241118658>
##   .. ..- attr(*, "class")= chr "quosures"
##   ..$ args: list()
##   ..- attr(*, "class")= chr [1:3] "op_select" "op_single" "op"
##  - attr(*, "class")= chr [1:4] "tbl_dbi" "tbl_sql" "tbl_lazy" "tbl"
```

11.4.8 Q %>% nrow()

✗ Notice the difference between `nrow()` and `tally()`. The `nrow` functions returns `NA` and does not execute a query:

```
Q %>% nrow()
```

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

11.4.9 Q %>% tally()

✓ The `tally` function actually counts all the rows.

```
Q %>% tally()
```

```
## # Source:   lazy query [?? x 1]
## # Database: postgres [postgres@localhost:5432/dvdrental]
##   n
##   <S3: integer64>
## 1 16044
```

The `nrow()` function knows that `Q` is a list. On the other hand, the `tally()` function tells SQL to go count all the rows. Notice that `Q` results in 16,044 rows – the same number of rows as `rental`.

11.4.10 Q %>% collect()

The `dplyr::collect()` function triggers a dbFetch() function behind the scenes, which forces R to download a specified number of rows:

```
Q %>% collect(n = 20)
```

```
## # A tibble: 20 x 3
##    rental_date         staff_email              customer_email
##    <dttm>              <chr>                    <chr>
##  1 2005-05-24 22:54:33 Mike.Hillyer@sakilasta~ tommy.collazo@sakilacustome~
##  2 2005-05-24 23:03:39 Mike.Hillyer@sakilasta~ manuel.murrell@sakilacustom~
##  3 2005-05-24 23:04:41 Jon.Stephens@sakilasta~ andrew.purdy@sakilacustomer~
##  4 2005-05-24 23:05:21 Mike.Hillyer@sakilasta~ delores.hansen@sakilacustom~
##  5 2005-05-24 23:08:07 Mike.Hillyer@sakilasta~ nelson.christenson@sakilacu~
##  6 2005-05-24 23:11:53 Jon.Stephens@sakilasta~ cassandra.walters@sakilacus~
##  7 2005-05-24 23:31:46 Jon.Stephens@sakilasta~ minnie.romero@sakilacustome~
##  8 2005-05-25 00:00:40 Mike.Hillyer@sakilasta~ ellen.simpson@sakilacustome~
##  9 2005-05-25 00:02:21 Jon.Stephens@sakilasta~ danny.isom@sakilacustomer.o~
## 10 2005-05-25 00:09:02 Jon.Stephens@sakilasta~ april.burns@sakilacustomer.~
## 11 2005-05-25 00:19:27 Jon.Stephens@sakilasta~ deanna.byrd@sakilacustomer.~
## 12 2005-05-25 00:22:55 Mike.Hillyer@sakilasta~ raymond.mcwhorter@sakilacus~
## 13 2005-05-25 00:31:15 Mike.Hillyer@sakilasta~ theodore.culp@sakilacustome~
## 14 2005-05-25 00:39:22 Mike.Hillyer@sakilasta~ ronald.weiner@sakilacustome~
## 15 2005-05-25 00:43:11 Jon.Stephens@sakilasta~ steven.curley@sakilacustome~
## 16 2005-05-25 01:06:36 Mike.Hillyer@sakilasta~ isaac.oglesby@sakilacustome~
## 17 2005-05-25 01:10:47 Jon.Stephens@sakilasta~ ruth.martinez@sakilacustome~
## 18 2005-05-25 01:17:24 Mike.Hillyer@sakilasta~ ronnie.ricketts@sakilacusto~
## 19 2005-05-25 01:48:41 Jon.Stephens@sakilasta~ roberta.harper@sakilacustom~
## 20 2005-05-25 01:59:46 Jon.Stephens@sakilasta~ craig.morrell@sakilacustome~
```

```
Q %>% collect(n = 20) %>% head()
```

```
## # A tibble: 6 x 3
##   rental_date         staff_email              customer_email
##   <dttm>              <chr>                    <chr>
## 1 2005-05-24 22:54:33 Mike.Hillyer@sakilasta~ tommy.collazo@sakilacustomer~
## 2 2005-05-24 23:03:39 Mike.Hillyer@sakilasta~ manuel.murrell@sakilacustome~
## 3 2005-05-24 23:04:41 Jon.Stephens@sakilasta~ andrew.purdy@sakilacustomer.~
## 4 2005-05-24 23:05:21 Mike.Hillyer@sakilasta~ delores.hansen@sakilacustome~
## 5 2005-05-24 23:08:07 Mike.Hillyer@sakilasta~ nelson.christenson@sakilacus~
## 6 2005-05-24 23:11:53 Jon.Stephens@sakilasta~ cassandra.walters@sakilacust~
```

The `collect` function triggers the creation of a tibble and controls the number of rows that the DBMS sends to R. Notice that **head** only prints 6 of the 25 rows that R has retrieved.

11.4.11 Q %>% show_query()

```
Q %>% show_query()
```

```
## <SQL>
## SELECT "rental_date", "staff_email", "customer_email"
## FROM (SELECT "rental_id", "rental_date", "inventory_id", "customer_id", "return_date", "staff_id", "las
## FROM (SELECT "TBL_LEFT"."rental_id" AS "rental_id", "TBL_LEFT"."rental_date" AS "rental_date", "TBL_LEF
##  FROM (SELECT "rental_id", "rental_date", "inventory_id", "customer_id", "return_date", "staff_id", "la
## FROM (SELECT "TBL_LEFT"."rental_id" AS "rental_id", "TBL_LEFT"."rental_date" AS "rental_date", "TBL_LEF
##   FROM "rental" AS "TBL_LEFT"
##   LEFT JOIN "staff" AS "TBL_RIGHT"
##   ON ("TBL_LEFT"."staff_id" = "TBL_RIGHT"."staff_id")
## ) "qzdwvvvtzq") "TBL_LEFT"
##   LEFT JOIN "customer" AS "TBL_RIGHT"
##   ON ("TBL_LEFT"."customer_id" = "TBL_RIGHT"."customer_id")
## ) "rnitnthkfz") "ohetoyqsmw"
```

Hand-written SQL code to do the same job will probably look a lot nicer and could be more efficient, but functionally dplyr does the job.

11.4.12 Qc <- Q %>% count(customer_email)

✗ Until **Q** is executed, we can add to it. This behavior is the basis for a useful debugging and development process where queries are built up incrementally.

```
Qc <- Q %>% count(customer_email, sort = TRUE)
```

✓ When all the accumulated **dplyr** verbs are executed, they are submitted to the dbms and the number of rows that are returned follow the same rules as discussed above.

```
Qc
```

```
## # Source:     lazy query [?? x 2]
## # Database:   postgres [postgres@localhost:5432/dvdrental]
## # Ordered by: desc(n)
##    customer_email                    n
##    <chr>                             <S3: integer64>
##  1 eleanor.hunt@sakilacustomer.org   46
##  2 karl.seal@sakilacustomer.org      45
##  3 clara.shaw@sakilacustomer.org     42
##  4 marcia.dean@sakilacustomer.org    42
##  5 tammy.sanders@sakilacustomer.org  41
##  6 wesley.bull@sakilacustomer.org    40
##  7 sue.peters@sakilacustomer.org     40
##  8 tim.cary@sakilacustomer.org       39
##  9 rhonda.kennedy@sakilacustomer.org 39
## 10 marion.snyder@sakilacustomer.org  39
## # ... with more rows
```

See more example of lazy execution can be found Here.

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

11.5 Other resources

- Benjamin S. Baumer, A Grammar for Reproducible and Painless Extract-Transform-Load Operations on Medium Data: https://arxiv.org/pdf/1708.07073

Chapter 12

DBI and SQL (11c)

12.1 This chapter:

- Introduces more DBI functions and demonstrates techniques for submitting SQL to the dbms
- Illustrates some of the differences between writing `dplyr` commands and SQL
- Suggests some strategies for dividing the work between your local R session and the dbms

12.1.1 Setup

The following packages are used in this chapter:

```
library(tidyverse)
library(DBI)
library(RPostgres)
library(dbplyr)
require(knitr)
library(bookdown)
library(sqlpetr)
```

Assume that the Docker container with PostgreSQL and the dvdrental database are ready to go. If not go back to the previous Chapter

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

Connect to the database:

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

12.2 SQL in R Markdown

When you create a report to run repeatedly, you might want to put that query into R markdown. See the discussion of multiple language engines in R Markdown. That way you can also execute that SQL code in a chunk with the following header:

{sql, connection=con, output.var = "query_results"}

```
SELECT "staff_id", COUNT(*) AS "n"
FROM "rental"
GROUP BY "staff_id";
```

Rmarkdown stored that query result in a tibble:

```
query_results
```

```
##   staff_id    n
## 1        2 8004
## 2        1 8040
```

12.3 DBI Package

In this chapter we touched on a number of functions from the DBI Package. The table in file 96b shows other functions in the package. The Chapter column references a section in the book if we have used it.

```
film_table <- tbl(con, "film")
```

12.3.1 Retrieve the whole table

SQL code that is submitted to a database is evaluated all at once[1]. To think through an SQL query, either use dplyr to build it up step by step and then convert it to SQL code or an IDE such as pgAdmin. DBI returns a data.frame, so you don't have dplyr's guardrails.

```
res <- dbSendQuery(con, 'SELECT "title", "rental_duration", "length"
FROM "film"
WHERE ("rental_duration" > 5.0 AND "length" > 117.0)')

res_output <- dbFetch(res)
str(res_output)
```

```
## 'data.frame':    202 obs. of  3 variables:
##  $ title          : chr  "African Egg" "Alamo Videotape" "Alaska Phantom" "Alley Evolution" ...
##  $ rental_duration: int  6 6 6 6 6 7 6 7 6 6 ...
##  $ length         : int  130 126 136 180 181 179 119 127 170 162 ...
```

```
dbClearResult(res)
```

12.3.2 Or a chunk at a time

```
res <- dbSendQuery(con, 'SELECT "title", "rental_duration", "length"
FROM "film"
WHERE ("rental_duration" > 5.0 AND "length" > 117.0)')

set.seed(5432)

chunk_num <- 0
while (!dbHasCompleted(res)) {
```

[1] From R's perspective. Actually there are 4 steps behind the scenes.

```
  chunk_num <- chunk_num + 1
  chunk <- dbFetch(res, n = sample(7:13,1))
  # print(nrow(chunk))
  chunk$chunk_num <- chunk_num
  if (!chunk_num %% 9) {print(chunk)}
}
```

```
##                 title rental_duration length chunk_num
## 1      Grinch Massage               7    150         9
## 2     Groundhog Uncut               6    139         9
## 3       Half Outfield               6    146         9
## 4       Hamlet Wisdom               7    146         9
## 5       Harold French               6    168         9
## 6        Hedwig Alter               7    169         9
## 7     Holes Brannigan               7    128         9
## 8     Hollow Jeopardy               7    136         9
## 9  Holocaust Highball               6    149         9
## 10          Home Pity               7    185         9
## 11     Homicide Peach               6    141         9
## 12    Hotel Happiness               6    181         9
##                      title rental_duration length chunk_num
## 1        Towers Hurricane               7    144        18
## 2                Town Ark               6    136        18
## 3       Trading Pinocchio               6    170        18
## 4 Trainspotting Strangers               7    132        18
## 5          Uncut Suicides               7    172        18
## 6    Unforgiven Zoolander               7    129        18
## 7         Uprising Uptown               6    174        18
## 8             Vanilla Day               7    122        18
## 9         Vietnam Smoochy               7    174        18
```

```
dbClearResult(res)
```

12.4 Dividing the work between R on your machine and the DBMS

They work together.

12.4.1 Make the server do as much work as you can

- show_query as a first draft of SQL. May or may not use SQL code submitted directly.

12.4.2 Criteria for choosing between `dplyr` and native SQL

This probably belongs later in the book.

- performance considerations: first get the right data, then worry about performance
- Trade offs between leaving the data in PostgreSQL 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

Where you place the `collect` function matters. Here is a typical string of dplyr verbs strung together with the magrittr `%>%` command that will be used to tease out the several different behaviors that a lazy query has when passed to different R functions. This query joins three connection objects into a query we'll call `Q`:

```
rental_table <- dplyr::tbl(con, "rental")
staff_table <- dplyr::tbl(con, "staff")
# the 'staff' table has 2 rows
customer_table <- dplyr::tbl(con, "customer")
# the 'customer' table has 599 rows

Q <- rental_table %>%
  left_join(staff_table, by = c("staff_id" = "staff_id")) %>%
  rename(staff_email = email) %>%
  left_join(customer_table, by = c("customer_id" = "customer_id")) %>%
  rename(customer_email = email) %>%
  select(rental_date, staff_email, customer_email)
```

```
Q %>% show_query()
```

```
## <SQL>
## SELECT "rental_date", "staff_email", "customer_email"
## FROM (SELECT "rental_id", "rental_date", "inventory_id", "customer_id", "return_date", "staff_id", "las
## FROM (SELECT "TBL_LEFT"."rental_id" AS "rental_id", "TBL_LEFT"."rental_date" AS "rental_date", "TBL_LEF
##  FROM (SELECT "rental_id", "rental_date", "inventory_id", "customer_id", "return_date", "staff_id", "la
## FROM (SELECT "TBL_LEFT"."rental_id" AS "rental_id", "TBL_LEFT"."rental_date" AS "rental_date", "TBL_LEF
##   FROM "rental" AS "TBL_LEFT"
##   LEFT JOIN "staff" AS "TBL_RIGHT"
##   ON ("TBL_LEFT"."staff_id" = "TBL_RIGHT"."staff_id")
## ) "tvnvuviyiw") "TBL_LEFT"
##   LEFT JOIN "customer" AS "TBL_RIGHT"
##   ON ("TBL_LEFT"."customer_id" = "TBL_RIGHT"."customer_id")
## ) "dkimtwhtoo") "dkadgsqpgd"
```

Here is the SQL query formatted for readability:

```
SELECT "rental_date",
       "staff_email",
       "customer_email"
FROM   (SELECT "rental_id",
               "rental_date",
               "inventory_id",
               "customer_id",
               "return_date",
               "staff_id",
               "last_update.x",
               "first_name.x",
               "last_name.x",
               "address_id.x",
               "staff_email",
               "store_id.x",
               "active.x",
               "username",
               "password",
               "last_update.y",
               "picture",
               "store_id.y",
               "first_name.y",
               "last_name.y",
               "email" AS "customer_email",
               "address_id.y",
               "activebool",
               "create_date",
               "last_update",
               "active.y"
        FROM   (SELECT "TBL_LEFT"."rental_id"     AS "rental_id",
                       "TBL_LEFT"."rental_date"   AS "rental_date",
                       "TBL_LEFT"."inventory_id"  AS "inventory_id",
                       "TBL_LEFT"."customer_id"   AS "customer_id",
                       "TBL_LEFT"."return_date"   AS "return_date",
                       "TBL_LEFT"."staff_id"      AS "staff_id",
                       "TBL_LEFT"."last_update.x" AS "last_update.x",
                       "TBL_LEFT"."first_name"    AS "first_name.x",
                       "TBL_LEFT"."last_name"     AS "last_name.x",
                       "TBL_LEFT"."address_id"    AS "address_id.x",
                       "TBL_LEFT"."staff_email"   AS "staff_email",
                       "TBL_LEFT"."store_id"      AS "store_id.x",
                       "TBL_LEFT"."active"        AS "active.x",
                       "TBL_LEFT"."username"      AS "username",
                       "TBL_LEFT"."password"      AS "password",
                       "TBL_LEFT"."last_update.y" AS "last_update.y",
                       "TBL_LEFT"."picture"       AS "picture",
                       "TBL_RIGHT"."store_id"     AS "store_id.y",
                       "TBL_RIGHT"."first_name"   AS "first_name.y",
                       "TBL_RIGHT"."last_name"    AS "last_name.y",
                       "TBL_RIGHT"."email"        AS "email",
                       "TBL_RIGHT"."address_id"   AS "address_id.y",
                       "TBL_RIGHT"."activebool"   AS "activebool",
                       "TBL_RIGHT"."create_date"  AS "create_date",
                       "TBL_RIGHT"."last_update"  AS "last_update",
                       "TBL_RIGHT"."active"       AS "active.y"
                FROM   (SELECT "rental_id",
                               "rental_date",
                               "inventory_id",
                               "customer_id",
                               "return_date",
                               "staff_id",
                               "last_update.x",
                               "first_name",
                               "last_name",
                               "address_id",
                               "email" AS "staff_email",
                               "store_id",
                               "active",
                               "username",
                               "password",
                               "last_update.y",
                               "picture"
                        FROM   (SELECT "TBL_LEFT"."rental_id"    AS "rental_id",
                                       "TBL_LEFT"."rental_date"  AS
                                       "rental_date",
                                       "TBL_LEFT"."inventory_id" AS
                                       "inventory_id",
                                       "TBL_LEFT"."customer_id"  AS
                                       "customer_id",
                                       "TBL_LEFT"."return_date"  AS
                                       "return_date",
                                       "TBL_LEFT"."staff_id"     AS "staff_id",
                                       "TBL_LEFT"."last_update"  AS
                                       "last_update.x",
                                       "TBL_RIGHT"."first_name"  AS "first_name"
                                       ,
                       "TBL_RIGHT"."last_name"   AS "last_name",
                       "TBL_RIGHT"."address_id"  AS "address_id",
                       "TBL_RIGHT"."email"       AS "email",
                       "TBL_RIGHT"."store_id"    AS "store_id",
                       "TBL_RIGHT"."active"      AS "active",
                       "TBL_RIGHT"."username"    AS "username",
                       "TBL_RIGHT"."password"    AS "password",
                       "TBL_RIGHT"."last_update" AS "last_update.y",
                       "TBL_RIGHT"."picture"     AS "picture"
                                FROM   "rental" AS "TBL_LEFT"
                                       LEFT JOIN "staff" AS "TBL_RIGHT"
                                              ON ( "TBL_LEFT"."staff_id" =
                                                   "TBL_RIGHT"."staff_id" ))
                               "ymdofxkiex") "TBL_LEFT"
                       LEFT JOIN "customer" AS "TBL_RIGHT"
                              ON ( "TBL_LEFT"."customer_id" =
                                 "TBL_RIGHT"."customer_id" ))
               "exddcnhait") "aohfdiedlb"
```

Hand-written SQL code to do the same job will probably look a lot nicer and could be more efficient, but functionally dplyr does the job.

```
GQ <- dbGetQuery(
  con,
  "select r.rental_date, s.email staff_email,c.email customer_email
     from rental r
          left outer join staff s on r.staff_id = s.staff_id
          left outer join customer c on r.customer_id = c.customer_id
  "
)
```

But because `Q` hasn't been executed, we can add to it. This behavior is the basis for a useful debugging and development process where queries are built up incrementally.

Where you place the `collect` function matters.

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

Chapter 13

Joins and complex queries (13)

This chapter demonstrates how to:

- Use primary and foreign keys to retrieve specific rows of a table
- do different kinds of join queries
- Exercises
- Query the database to get basic information about each dvdrental story
- How to interact with the database using different strategies

Verify Docker is up and running:

```
sp_check_that_docker_is_up()
```

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

Verify pet DB is available, it may be stopped.

```
sp_show_all_docker_containers()
```

```
## CONTAINER ID      IMAGE           COMMAND               CREATED         STATUS                 PORTS
## 2e3f033f6c17      postgres-dvdrental  "docker-entrypoint.s…"  31 seconds ago    Exited (0) 2 seconds ag
```

Start up the `docker-pet` container

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

Now connect to the database with R

```
# need to wait for Docker & Postgres to come up before connecting.

con <- sp_get_postgres_connection(
  user = Sys.getenv("DEFAULT_POSTGRES_USER_NAME"),
  password = Sys.getenv("DEFAULT_POSTGRES_PASSWORD"),
  dbname = "dvdrental",
  seconds_to_test = 30
)
```

13.1 Database Privileges

In the DVD rental database, you have all database privileges to perform any CRUD operaion, create, read, update, and delete on any database object. As a data analyst, you typically only get select privilege which allows you to read only a subset of the tables in a database. Occasionally, a proof of concept project may have a sandbox spun up where users are granted additional priviledges.

13.2 Database constraints

As a data analyst, you really do not need to worry about database constraints since you are primarily writing dplyr/SQL queries to pull data out of the database. Constraints can be enforced at multiple levels: column, table, multiple tables, or at the schema itself. The common database constraints are a column is `NOT NULL`, a column is `UNIQUE`, a column is a `PRIMARY KEY`, both `NOT NULL` and `UNIQUE`, or a column is a `FOREIGN KEY`, the `PRIMARY KEY` on another table. Constraints restrict column values to a set of defined values and help enforce referential integrity between tables.

13.2.1 DVD Rental Primary Foreign Key Constraints

For this tutorial, we are primarily concerned with primary and foreign key relationships between tables in order to correctly join the data between tables. If one looks at all the tables in the DVD Rental ERD, add link here, the first column is the name of the table followed by "_id". This is the primary key on the table. In some of the tables, there are other columns that begin with the name of a different table, the foreign table, and end in"_id". These are foreign keys and the foreign key value is the primary key value on the foreign table. The DBA will index the primary and foreign key columns to speed up query performanace.

In the table below, all the primary foreign key relationships are shown because the DVD rental system is small. Real world databases typically have hundreds or thousands of primary foreign key relationships. In the search box, enter 'PRIMARY' or 'FOREIGN' to see the table primary key or the table's foreign key relationships.

table_name	column_name	constraint_type	ref_table	ref_table_col
actor	actor_id	PRIMARY KEY		
address	address_id	PRIMARY KEY		
address	city_id	FOREIGN KEY	city	city_id
category	category_id	PRIMARY KEY		
city	city_id	PRIMARY KEY		
city	country_id	FOREIGN KEY	country	country_id
country	country_id	PRIMARY KEY		
customer	customer_id	PRIMARY KEY		
customer	address_id	FOREIGN KEY	address	address_id
film	film_id	PRIMARY KEY		
film	language_id	FOREIGN KEY	language	language_id
film_actor	actor_id	FOREIGN KEY	actor	actor_id
film_actor	actor_id	PRIMARY KEY		
film_actor	film_id	PRIMARY KEY		
film_actor	film_id	FOREIGN KEY	film	film_id
film_category	category_id	FOREIGN KEY	category	category_id
film_category	film_id	PRIMARY KEY		
film_category	category_id	PRIMARY KEY		
film_category	film_id	FOREIGN KEY	film	film_id
inventory	film_id	FOREIGN KEY	film	film_id
inventory	inventory_id	PRIMARY KEY		
language	language_id	PRIMARY KEY		
payment	staff_id	FOREIGN KEY	staff	staff_id
payment	customer_id	FOREIGN KEY	customer	customer_id
payment	rental_id	FOREIGN KEY	rental	rental_id
payment	payment_id	PRIMARY KEY		
rental	customer_id	FOREIGN KEY	customer	customer_id
rental	rental_id	PRIMARY KEY		
rental	staff_id	FOREIGN KEY	staff	staff_id
rental	inventory_id	FOREIGN KEY	inventory	inventory_id
staff	staff_id	PRIMARY KEY		
staff	address_id	FOREIGN KEY	address	address_id
store	store_id	PRIMARY KEY		
store	address_id	FOREIGN KEY	address	address_id
store	manager_staff_id	FOREIGN KEY	staff	staff_id

Searching for 'FOREIGN' in the table above, one sees that the `column_name` matches the `ref_table_col`. This is pretty typical, but not always the case. This can occur because of an inconsistent naming convention in the application design or a table contains multiple references to the same table foreign table and each reference indicates a different role. A non-DVD rental example of the latter is a patient transaction record that has a referring doctor and and performing doctor. The two columns will have different names, but may refer to the same or different doctors in the doctor table. In this case you may hear one say that the doctor table is performing two different roles.

13.3 Making up data for Join Examples

Each chapter in the book stands on its own. If you have worked through the code blocks in this chapter in a previous session, you created some new customer records and cloned the film table as smy_film in order to work through material in the rest of the chapter. In the next couple of code blocks, we delete the new data and recreate the smy_film table for the join examples below.

13.3.1 SQL Delete Data Syntax

```
    DELETE FROM <source> WHERE <where_clause>;
```

13.3.2 Delete New Practice Customers from the Customer table.

In the next code block we delete out the new customers that were added when the book was compliled or added working through the exercises. Out of the box, the DVD rental database's highest customer_id = 599.

`dbExecute() always returns a scalar numeric that specifies the number of rows affected by the statement.`

```
dbExecute(
  con,
  "delete from customer
   where customer_id >= 600;
  "
)
```

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

The number above tells us how many rows were actually deleted from the customer table.

13.3.3 SQL Single Row Insert Data Syntax

```
    INSERT INTO <target> <column_list> VALUES <values list>;
    <target> : target table/view
    <column list> : csv list of columns
    <values list> : values assoicated with the column list.
```

The `column list` is the list of column names on the table and the corresponding list of values have to have the correct data type. The following code block returns the `CUSTOMER` column names and data types.

```
customer_cols <- dbGetQuery(con,
       "select table_name, column_name, ordinal_position, data_type
          from information_schema.columns
         where table_catalog = 'dvdrental'
           and table_name = 'customer'
       ;")

sp_print_df(customer_cols)
```

table_name	column_name	ordinal_position	data_type
customer	customer_id	1	integer
customer	store_id	2	smallint
customer	first_name	3	character varying
customer	last_name	4	character varying
customer	email	5	character varying
customer	address_id	6	smallint
customer	activebool	7	boolean
customer	create_date	8	date
customer	last_update	9	timestamp without time zone
customer	active	10	integer

In the next code block, we insert Sophie as a new customer into the customer table via a SQL insert statement. The columns list clause has three id columns, customer_id, store_id, and address_id. The customer_id is a primary key column and the other two look like foreign key columns.

For now we are interested in getting some new customers into the customer table. We look at the relations between the customer table and the store and address tables later in this chapter.

```
dbExecute(
  con,
  "insert into customer
  (customer_id,store_id,first_name,last_name,email,address_id,activebool
  ,create_date,last_update,active)
  values(600,3,'Sophie','Yang','sophie.yang@sakilacustomer.org',1,TRUE,now(),now()::date,1)
  ")
```

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

```
new_customers <- dbGetQuery(con,'select * from customer where customer_id >= 600;')
sp_print_df(new_customers)
```

customer_id	store_id	first_name	last_name	email	address_id	activebool	create_
600	3	Sophie	Yang	sophie.yang@sakilacustomer.org	1	TRUE	2018-12

<<<<<<< HEAD The **film** table has a primary key, film_id, and a foreign key column, language_id. One cannot insert a new row into the film table with a language_id = 30 because of a constraint on the language_id column. The language_id value must already exist in the **language** table before the database will allow the new row to be inserted into the table.
======= ### Primary and Foreign Key Constraint Error Messages

For the new customers, we are concerned with not violating the PK and FK constraints.

If the customer_id = 600 value is changed to 599, the database throws the following error message.

```
Error in result_create(conn@ptr, statement) : Failed to fetch row: ERROR: duplicate key value violates uniqu
```

If the address_id value = 1 is changed to 611, the database throws the following error message:

```
Error in result_create(conn@ptr, statement) : Failed to fetch row: ERROR: insert or update on table "custome
```

13.3.4 R Exercise: Inserting a Single Row via a Dataframe

In the following code block replace Sophie Yang with your name where appropriate.
Note:

1. The last data frame parameter sets the stringsAsFactors is `FALSE`. Databases do not have a native `FACTOR` type.
2. The dataframe column names must match the table column names.
3. The dbWriteTable function needs `append` = true to actually insert the new row.

```
df <- data.frame(customer_id=601
                ,store_id=2
                ,first_name='Sophie'
                ,last_name='Yang'
                ,email='sophie.yang@sakilacustomer.org'
                ,address_id=1
                ,activebool=TRUE
                ,create_date=Sys.Date()
                ,last_update=Sys.time(),active=1
                ,stringsAsFactors = FALSE)
dbWriteTable(con, "customer", value = df, append = TRUE, row.names = FALSE)

new_customers <- dbGetQuery(con,'select * from customer where customer_id >= 600;')
sp_print_df(new_customers)
```

customer_id	store_id	first_name	last_name	email	address_id	activebool	create_
600	3	Sophie	Yang	sophie.yang@sakilacustomer.org	1	TRUE	2018-12
601	2	Sophie	Yang	sophie.yang@sakilacustomer.org	1	TRUE	2018-12

13.4 SQL Multi-Row Insert Data Syntax

```
INSERT INTO <target> <column_list> VALUES <values list1>, ... <values listn>;
<target> : target table/view
<column list> : csv list of columns
<values list> : values assoicated with the column list.
```

Postgres and some other flavors of SQL allow multiple rows to be inserted at a time. The syntax is identical to the Single Row syntax, but includes multiple `<values list>` clauses separated by commas. The following code block illustrates the SQL multi-row insert. Note that the customer_id column takes on sequential values to satisfy the PK constraint.

13.5 SQL Multi-Row Insert Data Example

```
#
dbExecute(
  con,
  "insert into customer
  (customer_id,store_id,first_name,last_name,email,address_id,activebool
  ,create_date,last_update,active)
   values(602,1,'John','Smith','john.smith@sakilacustomer.org',2,TRUE
         ,now()::date,now()::date,1)
         ,(603,1,'Ian','Frantz','ian.frantz@sakilacustomer.org',3,TRUE
         ,now()::date,now()::date,1)
         ,(604,1,'Ed','Borasky','ed.borasky@sakilacustomer.org',4,TRUE
         ,now()::date,now()::date,1)
         ;")
```

```
## [1] 3
```

The Postgres R multi-row insert is similar to the single row insert. The single column values are converted to a vector of values.

13.5.1 R Exercise: Inserting Multiple Rows via a Dataframe

Replace the two first_name, last_name, and email column values with your own made up values.

```
customer_id <- c(605,606)
store_id    <- c(3,4)
first_name  <- c('John','Ian')
last_name   <- c('Smith','Frantz')
email       <- c('john.smith@sakilacustomer.org'
                ,'ian.frantz@sakilacustomer.org')
address_id  <- c(3,4)
activebool  <- c(TRUE,TRUE)
create_date <- c(Sys.Date(),Sys.Date())
last_update <- c(Sys.time(),Sys.time())
active      <- c(1,1)
```

```
df2 <- data.frame(customer_id,store_id,first_name,last_name,email,address_id
            ,activebool,create_date,last_update,active,stringsAsFactors = FALSE)


dbWriteTable(con, "customer",
             value = df2, append = TRUE, row.names = FALSE)

new_customers <- dbGetQuery(con,'select * from customer where customer_id >= 600;')
sp_print_df(new_customers)
```

customer_id	store_id	first_name	last_name	email	address_id	activebool	create_
600	3	Sophie	Yang	sophie.yang@sakilacustomer.org	1	TRUE	2018-12
601	2	Sophie	Yang	sophie.yang@sakilacustomer.org	1	TRUE	2018-12
602	1	John	Smith	john.smith@sakilacustomer.org	2	TRUE	2018-12
603	1	Ian	Frantz	ian.frantz@sakilacustomer.org	3	TRUE	2018-12
604	1	Ed	Borasky	ed.borasky@sakilacustomer.org	4	TRUE	2018-12
605	3	John	Smith	john.smith@sakilacustomer.org	3	TRUE	2018-12
606	4	Ian	Frantz	ian.frantz@sakilacustomer.org	4	TRUE	2018-12

The `film` table has a primary key, film_id, and a foreign key column, language_id. In the next code bloock we see five sample rows from the film table.

```
films <- dbGetQuery(con,
       "select film_id,title, language_id
          from film
        order by film_id
        limit 5
       ;")

sp_print_df(films)
```

film_id	title	language_id
1	Academy Dinosaur	1
2	Ace Goldfinger	1
3	Adaptation Holes	1
4	Affair Prejudice	1
5	African Egg	1

The next code block all the rows in the language table.

```
languages <- dbGetQuery(con,
       "select language_id, name, last_update from language
       ;")

sp_print_df(languages)
```

language_id	name	last_update
1	English	2006-02-15 10:02:19
2	Italian	2006-02-15 10:02:19
3	Japanese	2006-02-15 10:02:19
4	Mandarin	2006-02-15 10:02:19
5	French	2006-02-15 10:02:19
6	German	2006-02-15 10:02:19

One cannot insert/update a new row into the film table with a language_id = 10 because of a constraint on the language_id column. The language_id value must already exist in the `language` table before the database will allow the new row to be inserted into the table.