

R, Databases and Docker

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

2018-12-06

Contents

1	Introduction	7
1.1	Using R to query a DBMS in your organization	7
1.2	Docker as a tool for Users	7
1.3	Docker and R on your machine	8
1.4	Who are we?	8
1.5	How did this project come about?	8
2	Setup instructions (00)	11
2.1	R, RStudio and Git	11
2.2	Docker	11
2.3	Defining the PostgreSQL connection parameters	12
2.4	Next steps	12
3	How to use this book (01)	15
3.1	Prerequisites	15
3.2	Installing Docker	16
3.3	Download the repo	16
3.4	Read along, experiment as you go	16
4	Docker Hosting for Windows (02)	17
4.1	Hardware requirements	17
4.2	Software requirements	17
4.3	Additional technical details	17
5	Learning Goals and Use Cases (03)	19
5.1	Ask yourself, what are you aiming for?	19
5.2	Learning Goals	19
5.3	Imagining a DVD rental business	20
5.4	Use cases	20
5.5	Investigating a question using with an organization's database	22
6	Docker, Postgres, and R (04)	23
6.1	Verify that Docker is running	23
6.2	Clean up if appropriate	23
6.3	Connect, read and write to Postgres from R	24
6.4	Clean up	25
7	The dvdrental database in Postgres in Docker (05)	27
7.1	Overview	27
7.2	Verify that Docker is up and running	27
7.3	Clean up if appropriate	28
7.4	Build the pet-sql Docker Image	28
7.5	Run the pet-sql Docker Image	28

7.6	Connect to Postgres with R	29
7.7	Stop and start to demonstrate persistence	30
7.8	Cleaning up	30
7.9	Using the <code>sql-pet</code> container in the rest of the book	30
8	Docker, Postgres, and R (04)	31
8.1	Set up the <code>sql-pet</code> docker container	31
8.2	Storing your dbms credentials	32
8.3	Clean up	33
9	Mapping your local environment (10)	35
9.1	Get some basic information about your database	35
9.2	Tutorial Environment	35
9.3	Communicating with Docker Applications	36
9.4	command structure	38
9.5	Exercises	40
10	Introduction to DBMS queries (11a)	43
10.1	Getting data from the database	43
10.2	Examining a single table with R	49
10.3	Additional reading	51
11	Lazy Evaluation and Lazy Queries (11b)	53
11.1	This chapter:	53
11.2	R is lazy and comes with guardrails	54
11.3	Lazy evaluation and lazy queries	55
11.4	SQL in R Markdown	61
11.5	DBI Package	61
11.6	Dividing the work between R on your machine and the DBMS	63
11.7	Other resources	64
12	Joins and complex queries (13)	65
12.1	Database constraints	65
12.2	Making up data for Join Examples	66
12.3	Joins	67
12.4	Natural Join Time Bomb	69
12.5	Join Templates	69
12.6	SQL anti join Costs	76
12.7	<code>dplyr</code> Anti joins	77
12.8	Exercises	79
12.9	Store analysis	83
12.10	Different strategies for interacting with the database	90
13	SQL Quick start - simple retrieval (15)	95
13.1	Intro	95
13.2	Databases and Third Normal Form - 3NF	96
13.3	SQL Commands	97
13.4	SQL SELECT Quick Start	99
13.5	Paradigm Shift from R-Dplyr to SQL	106
14	Getting metadata about and from the database (21)	109
14.1	Database contents and structure	109
14.2	What columns do those tables contain?	113
14.3	Characterizing how things are named	115
14.4	Database keys	116

14.5	Creating your own data dictionary	120
14.6	Save your work!	122
15	Drilling into your DBMS environment (22)	123
15.1	Which database?	123
15.2	How many databases reside in the Docker Container?	124
15.3	Which Schema?	125
15.4	Exercises	126
16	Explain queries (71)	137
16.1	Performance considerations	137
16.2	Clean up	139
17	SQL queries behind the scenes (72)	141
17.1	SQL Execution Steps	141
17.2	Passing values to SQL statements	142
17.3	Pass multiple sets of values with dbBind():	142
17.4	Clean up	143
18	Writing to the DBMS (73)	145
18.1	Create a new table	145
18.2	Modify an existing table	145
18.3	Remove table and Clean up	146
A	Other resources (89)	147
A.1	Editing this book	147
A.2	Docker alternatives	147
A.3	Docker and R	147
A.4	Documentation for Docker and Postgres	147
A.5	SQL and dplyr	147
A.6	More Resources	148
B	Mapping your local environment (92)	149
B.1	Environment Tools Used in this Chapter	149
B.2	Communicating with Docker Applications	152
C	Creating the sql-pet Docker container a step at a time	153
C.1	Overview	153
C.2	Download the dvdrental backup file	154
C.3	Build the Docker Container	154
C.4	Create the database and restore from the backup	155
C.5	Connect to the database with R	156
C.6	Cleaning up	157
D	APPENDIX D - Quick Guide to SQL (94)	159
E	Additional technical details for Windows users (95)	161
E.1	Docker for Windows settings	161
E.2	Git, GitHub and line endings	162
F	Dplyr functions and SQL cross-walk (96)	165
G	DBI package functions - coverage (96b)	169

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.

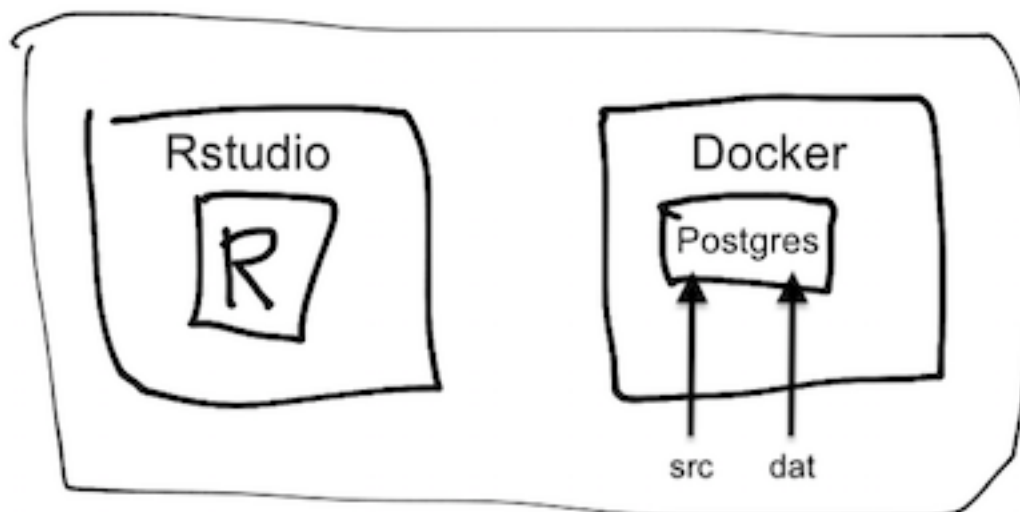
1. Make a fixed working environment for reproducible analysis
2. Access a service outside of R (**e.g., Postgres**)
3. Create an R based service (e.g., with **plumber**)
4. Send our compute jobs to the cloud with minimal reconfiguration or revision

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 a single machine, provided you have Docker installed and running.
- You may want to run PostgreSQL on a Docker container, avoiding any OS or system dependencies that might come up.

1.3 Docker and R on your machine

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



needs to be updated as our directory structure evolves.)

(This diagram

1.4 Who are we?

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 later - 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 you 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

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:

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

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

```
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:

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

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

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

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

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

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

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

```
sp_check_that_docker_is_up()
```

```
## [1] "Docker is up, running these containers:"
## [2] "CONTAINER ID      IMAGE          COMMAND                  CREATED          STATUS          PORTS
## [3] "8c047e891b7d     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 postgresQL 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")
```

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 (05)

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

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 56.07MB
```

```
## 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 5432
  "--mount ",        # tells Docker to mount a volume -- mapping Docker's internal file structure to the host
  "type=bind,",       # tells Docker that the mount command points to an actual file on the host system
  'source=',          # 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] "6f649719ecb282365646d3d31a12274cf69fdecd559708f3204a1b6d3ce2ae89"
```

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 = 10)
```

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 = 10)
```

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

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

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

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
## 6f649719ecb2      postgres-dvdrental  "docker-entrypoint.s..."  8 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

Docker, Postgres, and R (04)

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 credentials 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:

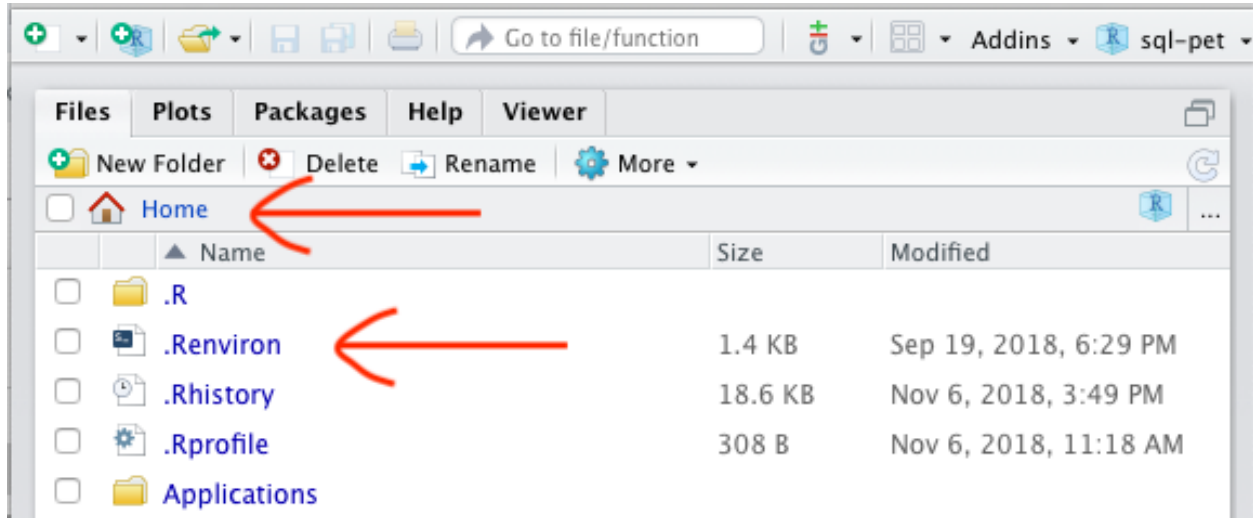
```
sp_docker_start("sql-pet")
```

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**:



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 upload 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 postgresQL 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 elements of your environment that are involved in running the examples in this book
- What the different roles that each element plays in your environment
- How those elements are connected and how you can send messages from one to another
- How each element has its own language / set of commands

9.1 Get some basic information about your database

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

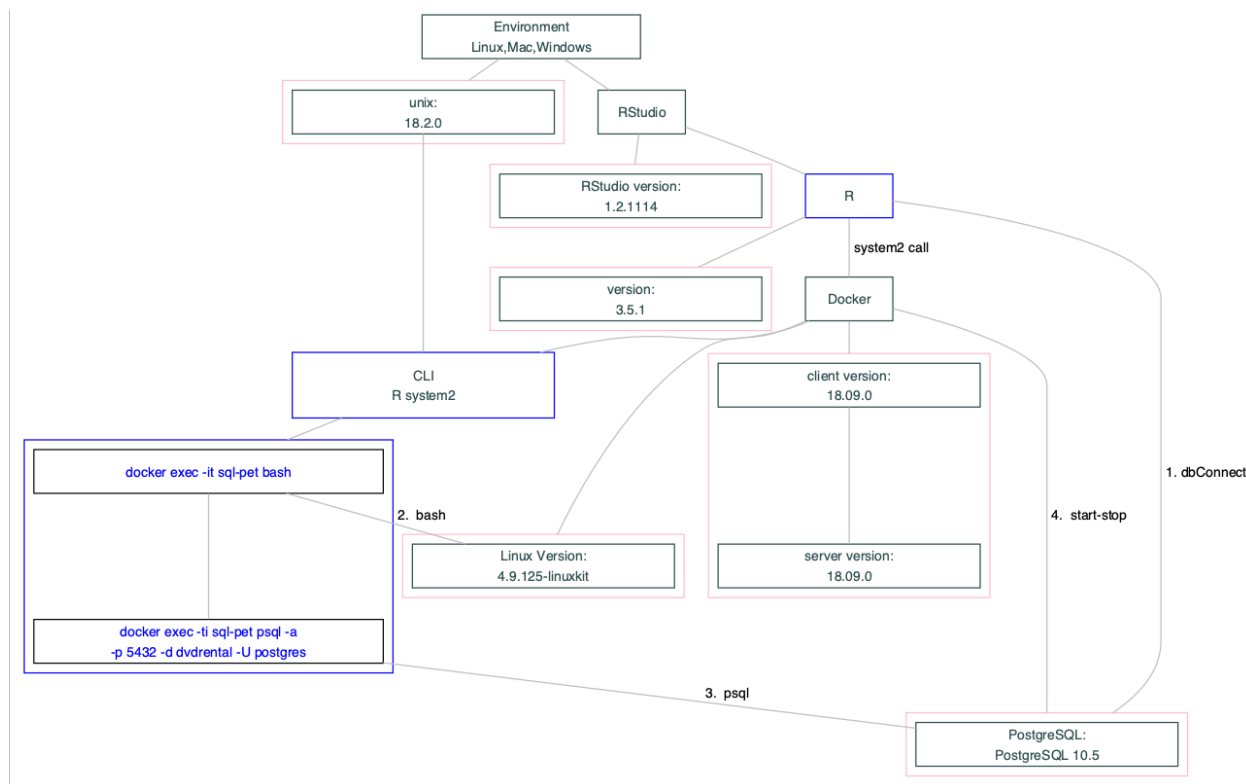
```
sp_docker_start("sql-pet")
```

Now connect to the `dvdrental` database with R.

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

9.2 Tutorial Environment

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



9.3 Communicating with Docker Applications

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

9.3.1 dbConnect

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

One focus of this tutorial is SQL processing through a **dbConnection** and we will come back to this in a future section. The remainder of this section focuses on some specific Docker commands.

9.3.2 bash

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

(In this and subsequent example, an initial **\$** represents your system prompt, which varies according to operating system and local environment.)

In your computer's command prompt:

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

Inside the **bash** (UNIX command) environment:

```
$ echo "'hello world'" talking to you live from a bash shell session within Docker!
```

```
'hello world' talking to you live from a bash shell session within Docker!
```

```
$ exit
```

Now back at your computer command prompt with:

```
$ exit
```

Note that the user in the example is root. Root has all privileges and can destroy the Docker environment.

9.3.3 psql

For users comfortable executing SQL from a command line directly against the database, can run the **psql** application directly. Below is the CLI command to start up **psql** session, execute a version of hello world, and quitting the **psql** version.

In your computer's command prompt:

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

That executes a postgresQL program in docker called **psql**, which is a command line interface to postgresQL. **psql** responds with:

```
psql (10.5 (Debian 10.5-1.pgdg90+1))
```

```
Type "help" for help.
```

You enter:

```
`select "'hello world" talking to you live from postgres inside Docker!' hello;`
```

All SQL commands need to end with a semi-colon. **psql** responds with:

```

                        hello
-----
"hello world" talking to you live from postgres inside Docker!
(1 row)
```

- To explore the PostgreSQL environment it's worth browsing around inside the Docker command with a shell.

- To run the Docker container that contains PostgreSQL, you can enter this from a command prompt:

```
$ docker exec -ti pet sh
```

- To exit Docker enter:

```
# exit
```

- Inside Docker, you can enter the PostgreSQL command-line utility **psql** by entering

```
# psql -U postgres
```

Handy **psql** commands include:

```

* postgres=# \h # psql help
* postgres=# \dt # list PostgreSQL tables
* postgres=# \c dbname # connect to database dbname
* postgres=# \l # list PostgreSQL databases
* postgres=# \conninfo # display information about current connection
* postgres=# \q # exit psql
```

To exit `psql`, enter:

```
\q
```

The `docker bash` and `psql` command options are optional for this tutorial, but open up a gateway to some very powerful programming techniques for future exploration.

9.3.4 start-stop

Docker has about 44 commands. We are interested in only those related to Postgres status, started, stopped, and available. In this tutorial, complete docker commands are printed out before being executed via a `system2` call. In the event that a code block fails, one can copy and paste the docker command into your local CLI and see if Docker is returning additional information.

9.4 command structure

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 postgresSQL.

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

9.4.1 Get info on a local file from R code

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

##           size isdir mode          mtime          ctime
## README.md 4973 FALSE  644 2018-11-01 14:23:56 2018-11-01 14:23:56
##
##           atime uid gid uname grname
## README.md 2018-12-06 15:15:26 502 80   jds  admin
# Or, using the system2 command:

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

## [1] "-rw-r--r--  1 jds  admin  4973 Nov  1 14:23 README.md"
```

9.4.2 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
```

9.4.3 List PostgreSQL users from Rstudio terminal (simulated)

We can't show exactly what a terminal session looks like in this R Markdown book, so we resort to representing it indirectly.

To get a unix command line *inside the sql-pet Docker environment*, enter this command:

```
$ docker exec -it sql-pet bash
```

The \$ represents the unix command prompt in your R Studio terminal session. The unix command prompt inside Docker in this case is `root@ce265d6e5361:/#`. A trivial Unix command to get the date therefore is:

```
root@ce265d6e5361:/# date
```

```
Fri Nov 30 00:29:04 UTC 2018
```

From that unix command prompt you can also enter the `psql` app (which has its own command line interface). Execute the PostgreSQL command line program `psql`, logging on as `postgres`, and show a list of users:

```
root@ce265d6e5361:/# psql -U postgres -c '\du' psql returns:
```

List of roles

Role name	Attributes	Member of
postgres	Superuser, Create role, Create DB, Replication, Bypass RLS	{}

To exit unix inside the Docker container, enter `exit`:

```
root@ce265d6e5361:/# exit
```

```
exit
```

We are then back to the R Studio terminal command:

```
$
```

9.4.4 List PostgreSQL users from R code

```
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] ""
```

Example	R	OS	Docker	Alpine container	pg_restore / ls	postgreSQL
-U postgres -d dvdrental					→	_____*
petdir/dvdrental.tar				_____>	_____*	
ls petdir grep "dvdrental.tar"			_____	_____	_____*	
exec sql-pet ls petdir grep "dvdrental.tar"			>			
docker exec sql-pet ls petdir grep "dvdrental.tar"		_____	_____	_____	_____*	
system2('docker', 'exec sql-pet ls petdir grep "dvdrental.tar"', stdout = TRUE, stderr = TRUE)	_____>	_____	_____	_____	_____*	

```
docker exec sql-pet ls -l petdir/README.md docker exec -it sql-pet psql -U postgres -c '\l'
```

9.5 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.

9.5.1 Docker Help

Typing `docker` at the command line will print up a summary of all available `docker` commands. Below are the docker commands used in the exercises.

Commands:

```
ps          List containers
start       Start one or more stopped containers
stop        Stop one or more running containers
```

The general format for a Docker command is

```
docker [OPTIONS] COMMAND ARGUMENTS
```

Below is the output for the Docker `exec` help command which was used in the `bash` and `psql` command examples above and for an exercise below.

```
$ docker help exec
```

```
Usage:  docker exec [OPTIONS] CONTAINER COMMAND [ARG...]
```

Run a command in a running container

Options:

```
-d, --detach           Detached mode: run command in the background
    --detach-keys string Override the key sequence for detaching a
                        container
-e, --env list         Set environment variables
-i, --interactive      Keep STDIN open even if not attached
    --privileged       Give extended privileges to the command
-t, --tty             Allocate a pseudo-TTY
-u, --user string      Username or UID (format:
                        <name|uid>[:<group|gid>])
-w, --workdir string   Working directory inside the container
```

In the following exercises, 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?	<code>docker exec -i sql-pet ps -eF</code>		
1a	How many process are running on your local machine?			windows: tasklist Mac/Linux: <code>ps -ef</code>

#	Question	Docker CLI Command	R RStudio command	Local Command LINE
2	What is the total number of files and directories in Docker?	<code>docker exec -i sql-pet ls -al</code>		
2a	What is the total number of files and directories on your local machine?			
3	Is Docker Running?	<code>docker version</code>		
3a	What are your Client and Server Versions?			
4	Does Postgres exist in the container?	<code>docker ps -a</code>		
4a	What is the status of Postgres?	<code>docker ps -a</code>		
4b	What is the size of Postgres?	<code>docker images</code>		
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?	<code>docker stop sql-pet</code>		
5a	If sql-pet status is Exited, How do I start it?	<code>docker start sql-pet</code>		

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:

```
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

```
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
)
```

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 important 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 circumstances where it was collected may be buried 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	2251	2005-06-18 05:05:08	1458	143
## 2	2252	2005-06-18 05:05:18	1987	383
## 3	2253	2005-06-18 05:11:43	3719	122
## 4	2254	2005-06-18 05:15:14	1084	281
## 5	2255	2005-06-18 05:21:12	24	410
## 6	2256	2005-06-18 05:21:56	1863	93
## 7	2257	2005-06-18 05:29:52	2846	34
## 8	2258	2005-06-18 05:30:36	4573	292
## 9	2259	2005-06-18 05:37:45	4103	491
## 10	2260	2005-06-18 05:38:36	2773	297
## 11	2261	2005-06-18 05:46:15	1763	570
## 12	2262	2005-06-18 05:49:46	4172	218
## 13	2263	2005-06-18 05:57:47	3259	452
## 14	2264	2005-06-18 05:58:45	150	240
## 15	2265	2005-06-18 06:03:27	3069	267
## 16	2266	2005-06-18 06:05:02	2596	452
## 17	2267	2005-06-18 06:10:23	2086	218
## 18	2268	2005-06-18 06:13:41	4380	21
## 19	2269	2005-06-18 06:20:54	3088	431
## 20	2270	2005-06-18 06:29:01	3447	588

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
##   <dtm>           <dtm>
## 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 directly:

```
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> <dtm>                <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 shown 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
```

<<sm> We haven't investigated this, but it looks like `dplyr collect()` function triggers a call similar 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` in 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    <dtm> 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     <dtm> 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     <dtm> 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.

```
library(skimr)

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

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.

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

10.3 Additional reading

- (gre, 2018)
- [Baumer_2018]

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
- 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

11.1.1 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
)
```

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 a “*Structured Query Language (SQL) is a standard computer language for relational database management and data manipulation.*”¹ SQL has 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 Wickham 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.

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 arguments⁴ 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 actually fetch the data and load it into an object.*”⁶ The `collect()` function retrieves data into a local tibble.⁷

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/>

⁶<https://www.quora.com/What-is-a-lazy-query>

⁷<https://dplyr.tidyverse.org/reference/compute.html>

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/dvdfrental]
##   rental_id rental_date      inventory_id customer_id
##   <int>    <dtm>              <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 <dtm>,
## #   staff_id <int>, last_update <dtm>
```

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 ...
##   .. .. ..$ typename: 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" ...
##   ..- 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@typenames$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
```

11.3.2 Using a lazy query

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 %>%
  select(staff_id, customer_id, rental_date) %>%
```



```

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

```

Think of `Q` as a black box for the moment. The following examples will show how `Q` is interpreted differently by different functions.

R code	Result
<code>Q %>% print()</code>	Prints <code>x</code> rows; same as just entering <code>Q</code>
<code>Q %>% as.tibble()</code>	Forces <code>Q</code> to be a tibble
<code>Q %>% head()</code>	Prints the first 6 rows
<code>Q %>% length()</code>	Counts the rows in <code>Q</code>
<code>Q %>% str(max.level = 3)</code>	Shows the top 3 levels of the object <code>Q</code>
<code>Q %>% nrow()</code>	Attempts to determine the number of rows
<code>Q %>% tally()</code>	Counts all the rows – on the dbms side
<code>Q %>% collect (n = 20)</code>	Prints 20 rows
<code>Q %>% collect (n = 20) %>%</code> <code>head()</code>	Prints 6 rows
<code>Q %>% show_query()</code>	Translates the lazy query object into SQL
<code>Qc <- Q %>%</code> <code>count(customer_email, sort</code> <code>= TRUE) Qc</code>	Extends the lazy query object

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

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

```

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 <dtm> <chr> <chr>
```

It has only retrieved 10 rows but doesn't know how many rows are left to retrieve as it notes ... with more rows. 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
##   <dtm>          <chr>          <chr>
## 1 2005-05-24 22:54:33 Mike.Hillyer@sakilast~ tommy.collazo@sakilacustome~
## 2 2005-05-24 23:03:39 Mike.Hillyer@sakilast~ manuel.murrell@sakilacustom~
## 3 2005-05-24 23:04:41 Jon.Stephens@sakilast~ andrew.purdy@sakilacustomer~
## 4 2005-05-24 23:05:21 Mike.Hillyer@sakilast~ delores.hansen@sakilacustom~
## 5 2005-05-24 23:08:07 Mike.Hillyer@sakilast~ nelson.christenson@sakilacu~
## 6 2005-05-24 23:11:53 Jon.Stephens@sakilast~ cassandra.walters@sakilacus~
## 7 2005-05-24 23:31:46 Jon.Stephens@sakilast~ minnie.romero@sakilacustome~
## 8 2005-05-25 00:00:40 Mike.Hillyer@sakilast~ ellen.simpson@sakilacustome~
## 9 2005-05-25 00:02:21 Jon.Stephens@sakilast~ danny.isom@sakilacustomer.o~
## 10 2005-05-25 00:09:02 Jon.Stephens@sakilast~ april.burns@sakilacustomer.~
## # ... with 16,034 more rows
```

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
##   <dtm>          <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~
```

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
```

Looking inside shows some of what's going on:

```
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: 0x7fc2b1f2ba68>
## .. ..$ : language ~staff_email
## .. ..- attr(*, ".Environment")=<environment: 0x7fc2b1f2ba68>
## .. ..$ : language ~customer_email
## .. ..- attr(*, ".Environment")=<environment: 0x7fc2b1f2ba68>
## .. ..- 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"
```

Notice the difference between `nrow()` and `tally()`:

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

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

```
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`.

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

```
## 18 2005-05-25 01:17:24 Mike.Hillyer@sakilast~ ronnie.ricketts@sakilacusto~
## 19 2005-05-25 01:48:41 Jon.Stephens@sakilast~ roberta.harper@sakilacustom~
## 20 2005-05-25 01:59:46 Jon.Stephens@sakilast~ craig.morrell@sakilacustome~
```

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

```
## # A tibble: 6 x 3
##   rental_date      staff_email      customer_email
##   <dtm>           <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~
```

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.

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

```
## <SQL>
## SELECT "rental_date", "staff_email", "customer_email"
## FROM (SELECT "staff_id", "customer_id", "rental_date", "staff_email", "store_id", "first_name", "last_n
## FROM (SELECT "TBL_LEFT"."staff_id" AS "staff_id", "TBL_LEFT"."customer_id" AS "customer_id", "TBL_LEFT"
##   FROM (SELECT "staff_id", "customer_id", "rental_date", "staff_email"
## FROM (SELECT "staff_id", "customer_id", "rental_date", "first_name", "last_name", "address_id", "email"
## FROM (SELECT "TBL_LEFT"."staff_id" AS "staff_id", "TBL_LEFT"."customer_id" AS "customer_id", "TBL_LEFT"
##   FROM (SELECT "staff_id", "customer_id", "rental_date"
## FROM "rental") "TBL_LEFT"
##   LEFT JOIN "staff" AS "TBL_RIGHT"
##   ON ("TBL_LEFT"."staff_id" = "TBL_RIGHT"."staff_id")
## ) "npghbogapn") "vapgdctabt") "TBL_LEFT"
##   LEFT JOIN "customer" AS "TBL_RIGHT"
##   ON ("TBL_LEFT"."customer_id" = "TBL_RIGHT"."customer_id")
## ) "jtjuanvogi") "jlxuthlhsv"
```

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.

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.

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

11.4 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
```

11.5 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")
```

11.5.1 Retrieve the whole table

SQL code that is submitted to a database is evaluated all at once⁸. 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 ...
```

⁸From R's perspective. Actually there are 4 steps behind the scenes.

```
dbClearResult(res)
```

11.5.2 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)')

chunk_num <- 0
while(!dbHasCompleted(res)){
  chunk_num <- chunk_num + 1
  chunk <- dbFetch(res, n = 5)
  print(nrow(chunk))
  if (!chunk_num %% 7) {print(chunk)}
}
```

```
## [1] 5
## [1] 5
## [1] 5
## [1] 5
## [1] 5
## [1] 5
## [1] 5
##
##           title rental_duration length
## 1 Christmas Moonshine           7    150
## 2      Citizen Shrek           7    165
## 3    Cleopatra Devil           6    150
## 4 Clockwork Paradise           7    143
## 5    Clones Pinocchio           6    124
## [1] 5
## [1] 5
## [1] 5
## [1] 5
## [1] 5
## [1] 5
## [1] 5
##
##           title rental_duration length
## 1 Extraordinary Conquerer         6    122
## 2      Flight Lies              7    179
## 3      Floats Garden             6    145
## 4    Forever Candidate           7    131
## 5  Frankenstein Stranger         7    159
## [1] 5
## [1] 5
## [1] 5
## [1] 5
## [1] 5
## [1] 5
## [1] 5
##
##           title rental_duration length
## 1    Jungle Closer              6    134
## 2  Killer Innocent             7    161
## 3  Lambs Cincinatti            6    144
```

```
## 4   Lawless Vision           6   181
## 5   Lawrence Love           7   175
## [1] 5
## [1] 5
## [1] 5
## [1] 5
## [1] 5
## [1] 5
## [1] 5
##
##           title rental_duration length
## 1   Outbreak Divine           6   169
## 2     Outlaw Hanky           7   148
## 3     Paris Weekend           7   121
## 4 Philadelphia Wife           7   137
## 5   Pianist Outfield         6   136
## [1] 5
## [1] 5
## [1] 5
## [1] 5
## [1] 5
## [1] 5
## [1] 5
##
##           title rental_duration length
## 1     Spinal Rocky            7   138
## 2 Spirit Flintstones         7   149
## 3   Steers Armageddon        6   140
## 4     Stock Glass            7   160
## 5     Story Side             7   163
## [1] 5
## [1] 5
## [1] 5
## [1] 5
## [1] 5
## [1] 2
dbClearResult(res)
```

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

They work together.

11.6.1 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.

11.6.2 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.

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

11.7 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

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 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
## 6f649719ecb2	postgres-dvdrental	"docker-entrypoint.s..."	29 seconds ago	Exited (0)	2 seconds ago

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 = 10  
)
```

12.1 Database constraints

As a data analyst, you really do not have 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.

For this tutorial, we are primarily concerned with primary and foreign key constraints. If one looks at all the tables in the DVD Rental ERD, 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 performance.

12.2 Making up data for Join Examples

12.2.1 insert yourself as a new customer

```
# Customer 600 should be the next customer.
# It gets deleted here just in case it was added in a different session.
dbExecute(
  con,
  "delete from customer
  where customer_id = 600;
  "
)
```

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

```
# Now add yourself as the next customer. Replace Sophie Yang with your name.
dbExecute(
  con,
  "insert into customer
  (customer_id,store_id,first_name,last_name,email,address_id
  ,activebool,create_date,last_update,active)
  values(600,2,'Sophie','Yang','email@email.com',1,TRUE,now()::date,now()::date,1)
  ;
  "
)
```

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

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 = 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.

To work around this inconvenience for the tutorial:

1. we drop the `smy_film` table if it exists from a previous session.

```
dbExecute(con, "drop table if exists smy_film;")
```

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

2. we create a new table `smy_film` from the `film` table and add a new row with a `language_id = 10`;

```
dbExecute(con, "create table smy_film as select * from film;")
```

```
## [1] 1000
```

3. We create a film with `language_id = 10`;

```
dbExecute(
  con,
```

```

"insert into smy_film
(film_id,title,description,release_year,language_id
,rental_duration,rental_rate,length,replacement_cost,rating
,last_update,special_features,fulltext)
values(1001,'Sophie''s Choice','orphaned language_id=10',2018,10
,7,4.99,120,14.99,'PG'
,now()::date,'{Trailers}','')
;
"
)

```

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

4. Confirm that the new record exists.

```

dbGetQuery(
  con,
  "select film_id,title,description,language_id from smy_film where film_id = 1001;"
)

```

```
##   film_id          title          description language_id
## 1    1001 Sophie's Choice orphaned language_id=10          10
```

12.3 Joins

In section ‘SQL Quick Start Simple Retrieval’, there is a brief discussion of databases and 3NF. The goal of normalization is to push the data into separate tables at a very granular level.

Bill Kent famously summarized 3NF as every non-key column “must provide a fact about the key,the whole key, and nothing but the key, so help me Codd.”

Normalization breaks data down and JOINS denormalizes the data and builds it back up.

The above diagram can be found here There are additional graphics at the link, but the explanations are poorly worded and hard to follow.

The diagram above shows nicely the hierarchy of different types of joins. For this tutorial, we can think of joins as either an Inner Join or an Outer Join.

Instead of showing standard Venn diagrams showing the different JOINS, we use an analogy. For those interested though, the typical Venn diagrams can be found here.

12.3.1 Valentines Party

Imagine you are at a large costume Valentine’s Day dance party. The hostess of the party, a data scientist, would like to learn more about the people attending her party. She interrupts the music to let everyone know it is time for the judges to evaluate the winners for best costumes and associated prizes.

She requests the following:

1. All the couples at the party to line up in front of her with the men on the left and the women on the right, (inner join)
2. All the remaining men to form a second line two feet behind the married men,(left outer join)
3. Right Outer Join: All the remaining women to form a third line two feet in front of the married women, (right outer join, all couples + unattached women)

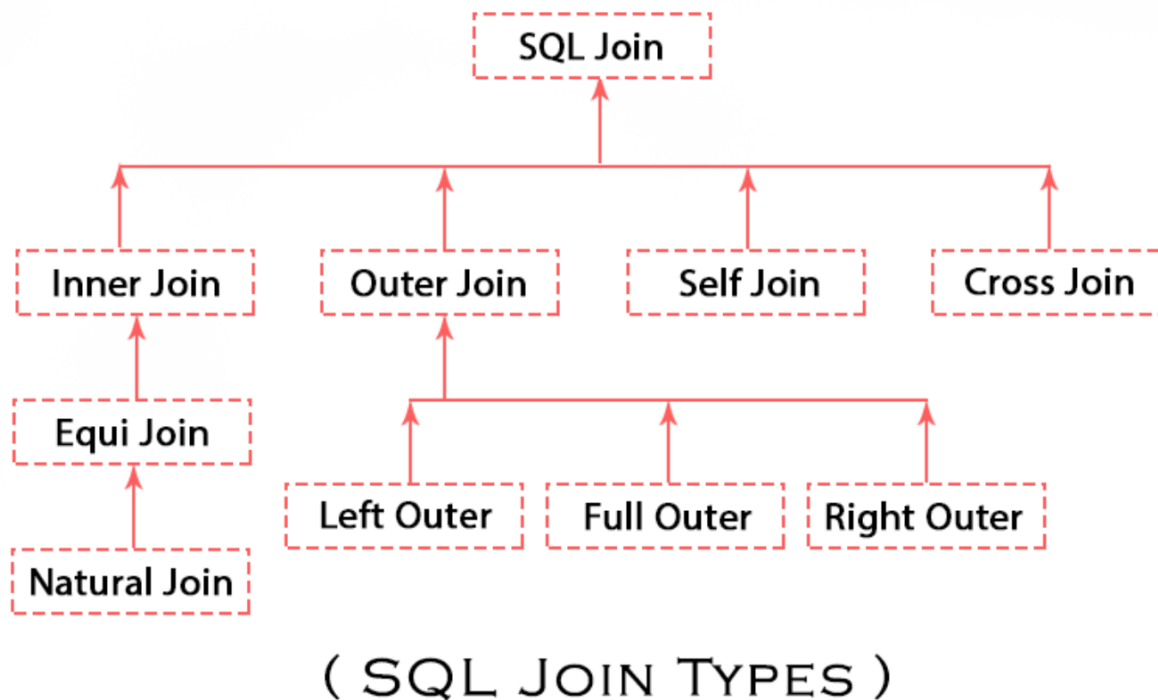


Figure 12.1: SQL_JOIN_TYPES

Full Outer Join – As our data scientist looks out at the three lines, she can clearly see the three distinct lines, her full outer join.

As the three judges start walking down the lines, she makes one more announcement.

4. There is a special prize for the man and woman who can guess the average age of the members of the opposite sex. To give everyone a chance to come up with an average age, she asks the men to stay in line and the women to move down the mens line in order circling back around until they get back to their starting point in line, (full outer join, every man seen by every woman and vice versa).

It is hard enough to tell someone's age when they don't have a mask, how do you get the average age when people have masks?

The hostess knows that there is usually some data anomalies. As she looks out she sees a small cluster of people who did not line up. Being the hostess with the mostest, she wants to get to know that small cluster better. Since they are far off and in costume, she cannot tell if they are men or women. More importantly, she does not know if they identify as a man or a woman, both – (kind of a stretch for a self join), neither, or something else. Ahh, the inquisitive mind wants to know.

12.3.2 Join Syntax

Join	dplyr	sql
inner	<code>inner_join(customer_tbl, rental_tbl, by = 'customer_id', suffix = c("c", "r"))</code> <code>customer_tbl %>% inner_join(rental_tbl, by = 'customer_id', suffix = c("c", "r"))</code>	<code>from customer c join rental r on c.customer_id = r.customer_id</code>
left	<code>left_join(customer_tbl, rental_tbl, by = 'customer_id', suffix = c("c", "r"))</code>	<code>from customer c left outer join rental r on c.customer_id = r.customer_id</code>

Join	dplyr	sql
right	customer_tbl %>% left_join(rental_tbl, by = 'customer_id', suffix = c("c", "r")) right_join(customer_tbl, rental_tbl, by = 'customer_id', suffix = c("c", "r"))	from customer c right outer join rental r on c.customer_id = r.customer_id
full	customer_tbl %>% right_join(rental_tbl, by = 'customer_id', suffix = c("c", "r")) full_join(customer_tbl, rental_tbl, by = 'customer_id', suffix = c("c", "r")) customer_tbl %>% full_join(rental_tbl, by = 'customer_id', suffix = c("c", "r"))	from customer c full outer join rental r on c.customer_id = r.customer_id

12.3.3 Join Tables

The dplyr join documentation describes two different types of joins, **mutating** and **filtering** joins. For those coming to R with a SQL background, the mutating documentation is misleading in one respect. Here is the `inner_join` documentation.

```
inner_join()
```

return all rows from x where there are matching values in y, and all columns from x and y. If there are multiple

The misleading part is that all the columns from *x* and *y*. If the join column is **KEY**, SQL will return *x*.KEY and *y*.KEY. Dplyr returns KEY. It appears that the KEY value comes from the key/driving table. This difference should become clear in the outer join examples.

In the next couple of examples, we will pull all the language and `smy_film` table data from the database into memory because the tables are small. In the `*_join` verbs, the **by** and **suffix** parameters are included because it helps document the actual join and the source of join columns.

12.4 Natural Join Time Bomb

The dplyr default join is a natural join, joining tables on common column names. One of many links why one should not use natural joins can be found [here](#).

12.5 Join Templates

In this section we look at two tables, `language` and `smy_film` and various joins using dplyr and SQL. Each dplyr code block has three purposes.

1. Show a working join example.
2. The code blocks can be used as templates for beginning more complex dplyr pipes.
3. The code blocks show the number of joins performed.

In these examples, the join condition, the **by** parameter,

```
by = c('language_id', 'language_id')
```

the two columns are the same. In multi-column joins, each `language_id` would be replaced with a vector of column names used in the join by position. Note the column names do not need to be identical by position.

The suffix parameter is a way to distinguish the same column name in the joined tables. The suffixes are usually a single letter to represent the name of the table.

```
language_table <- DBI::dbReadTable(con, "language")
film_table <- DBI::dbReadTable(con, "smy_film")
```

12.5.1 dplyr Inner Join Template

```
languages_ij <- language_table %>%
  inner_join(film_table, by = c("language_id", "language_id"), suffix(c(".l", ".f"))) %>%
  group_by(language_id, name) %>%
  summarize(inner_joins = n())
```

```
languages_ij
```

```
## # A tibble: 1 x 3
## # Groups:   language_id [?]
##   language_id name                inner_joins
##         <int> <chr>                  <int>
## 1           1 "English"                      1000
```

12.5.1.1 SQL Inner Join

```
rs <- dbGetQuery(
  con,
  "select l.language_id, l.name, count(*) n
   from language l join smy_film f on l.language_id = f.language_id
   group by l.language_id, l.name;"
)
```

```
rs
```

```
##   language_id      name      n
## 1           1 English    1000
```

The output tells us that there are 1000 inner joins between the `language_table` and the `film_table`.

12.5.2 dplyr Left Outer Join Template

```
languages_loj <- language_table %>%
  left_join(film_table, by = c("language_id", "language_id"), suffix(c(".l", ".f"))) %>%
  mutate(
    join_type = "loj"
    , film_lang_id = if_else(is.na(film_id), film_id, language_id)
  ) %>%
  group_by(join_type, language_id, name, film_lang_id) %>%
  summarize(lojs = n()) %>%
  select(join_type, language_id, film_lang_id, name, lojs)
print(languages_loj)
```

```
## # A tibble: 6 x 5
## # Groups:   join_type, language_id, name [6]
```

```
##   join_type language_id film_lang_id name      lojs
##   <chr>         <int>      <int> <chr>      <int>
## 1 loj           1          1 "English    " 1000
## 2 loj           2          NA "Italian    "   1
## 3 loj           3          NA "Japanese   "   1
## 4 loj           4          NA "Mandarin   "   1
## 5 loj           5          NA "French     "   1
## 6 loj           6          NA "German     "   1

# View(languages_loj)
# sp_print_df(languages_loj)
```

Compare the mutate verb in the above code block with `film_lang_id` in the equivalent SQL code block below.

12.5.2.1 SQL Left Outer Join

```
rs <- dbGetQuery(
  con,
  "select l.language_id
     ,f.language_id film_lang_id
     ,trim(l.name) as name
     ,count(*) lojs
  from language l left outer join smy_film f
    on l.language_id = f.language_id
 group by l.language_id,l.name,f.language_id
 order by l.language_id;"
)
# sp_print_df(rs)
rs
```

```
##   language_id film_lang_id      name lojs
## 1           1          1 English 1000
## 2           2          NA  Italian   1
## 3           3          NA Japanese   1
## 4           4          NA Mandarin   1
## 5           5          NA  French    1
## 6           6          NA   German    1
```

The `lojs` column returns the number of rows found on the keys from the left table, `language`, and the right table, the `film` table. For the “English” row, the `language_id` and `film_lang_id` match and a 1000 inner joins were performed. For all the other languages, there was only 1 join and they all came from the left outer table, the `language` table, `language_id`’s 2 - 6. The right table, the `film` table returned NA, because no match was found.

1. The left outer join always returns all rows from the left table, the driving/key table, if not reduced via a `filter()/where` clause.
2. All rows that inner join returns all the columns/derived columns specified in the select clause from both the left and right tables.
3. All rows from the left table, the outer table, without a matching row on the right returns all the columns/derived column values specified in the select clause from the left, but the values from right table have all values of NA.

12.5.2.2 dplyr Right Outer Join

```
languages_roj <- language_table %>%
  right_join(film_table, by = c("language_id", "language_id"), suffix(c(".l", ".f")), all = film_table)
  mutate(
    lang_id = if_else(is.na(name), 0L, language_id)
    , join_type = "rojs"
  ) %>%
  group_by(join_type, language_id, name, lang_id) %>%
  summarize(rojs = n()) %>%
  select(join_type, lang_id, language_id, name, rojs)

sp_print_df(languages_roj)
```

join_type	lang_id	language_id	name	rojs
rojs	1	1	English	1000
rojs	0	10	NA	1

```
languages_roj
```

```
## # A tibble: 2 x 5
## # Groups:   join_type, language_id, name [2]
##   join_type lang_id language_id name          rojs
##   <chr>      <int>      <int> <chr>      <int>
## 1 rojs          1          1 "English"    1000
## 2 rojs          0          10 <NA>         1
```

Review the mutate above with `l.language_id` below.

12.5.2.3 SQL Right Outer Join

```
rs <- dbGetQuery(
  con,
  "select 'roj' join_type, l.language_id, f.language_id language_id_f, l.name, count(*) rojs
   from language l right outer join smy_film f on l.language_id = f.language_id
   group by l.language_id, l.name, f.language_id
   order by l.language_id;"
)
sp_print_df(rs)
```

join_type	language_id	language_id_f	name	rojs
roj	1	1	English	1000
roj	NA	10	NA	1

```
rs
```

```
##   join_type language_id language_id_f      name rojs
## 1      roj           1           1 English    1000
## 2      roj          NA           10     <NA>     1
```

The `rojs` column returns the number of rows found on the keys from the right table, `film`, and the left table, the `language` table. For the “English” row, the `language_id` and `film_lang_id` match and a 1000 inner joins were performed. For `language_id = 10` from the right table, there was only 1 join to a non-existent row in the `language` table on the left.

1. The right outer join always returns all rows from the right table, the driving/key table, if not reduced via a `filter()`/where clause.

2. All rows that inner join returns all the columns/derived columns specified in the select clause from both the left and right tables.
3. All rows from the right table, the outer table, without a matching row on the left returns all the columns/derived column values specified in the select clause from the right, but the values from left table have all values of NA.

12.5.2.4 dplyr Full Outer Join

```
languages_foj <- language_table %>%
  full_join(film_table, by = c("language_id", "language_id"), suffix(c(".l", ".f"))) %>%
  mutate(film_lang = if_else(is.na(film_id), paste0("No ", name, " films."), if_else(is.na(name), "Alien", name)))
  group_by(language_id, name, film_lang) %>%
  summarize(n = n())

sp_print_df(languages_foj)
```

language_id	name	film_lang	n
1	English	English	1000
2	Italian	No Italian films.	1
3	Japanese	No Japanese films.	1
4	Mandarin	No Mandarin films.	1
5	French	No French films.	1
6	German	No German films.	1
10	NA	Alien	1

```
languages_foj
```

```
## # A tibble: 7 x 4
## # Groups:   language_id, name [?]
##   language_id name          film_lang          n
##   <int> <chr>          <chr>          <int>
## 1         1 "English"        "English"        1000
## 2         2 "Italian"        "No Italian      films.         1
## 3         3 "Japanese"       "No Japanese     films.         1
## 4         4 "Mandarin"       "No Mandarin     films.         1
## 5         5 "French"         "No French       films.         1
## 6         6 "German"         "No German       films.         1
## 7        10 <NA>         Alien            1
```

12.5.2.5 SQL full Outer Join

```
rs <- dbGetQuery(
  con,
  "select l.language_id,l.name,f.language_id language_id_f,count(*) fojs
   from language l full outer join smy_film f on l.language_id = f.language_id
  group by l.language_id,l.name,f.language_id
 order by l.language_id;"
)
sp_print_df(rs)
```

language_id	name	language_id_f	fojs
1	English	1	1000
2	Italian	NA	1
3	Japanese	NA	1
4	Mandarin	NA	1
5	French	NA	1
6	German	NA	1
NA	NA	10	1

rs

```
##   language_id      name language_id_f fojs
## 1           1 English              1 1000
## 2           2 Italian              NA   1
## 3           3 Japanese             NA   1
## 4           4 Mandarin             NA   1
## 5           5 French               NA   1
## 6           6 German               NA   1
## 7          NA              <NA>      10   1
```

Looking at the SQL output, the full outer join is the combination of the left and right outer joins.

1. Language_id = 1 is the inner join.
2. Language_id = 2 - 6 is the left outer join
3. Language_id = 10 is the right outer join.

One can also just look at the language_id on the left and language_id_f on the right for a non NA value to see which side is outer side/driving side of the join.

12.5.2.6 dplyr anti Join

The anti join is a left outer join without the inner joined rows. It only returns the rows from the left table that do not have a match from the right table.

```
languages_aj <- language_table %>%
  anti_join(film_table, by = c("language_id", "language_id"), suffix(c(".l", ".f"))) %>%
  mutate(type = "anti_join") %>%
  group_by(type, language_id, name) %>%
  summarize(anti_joins = n()) %>%
  select(type, language_id, name, anti_joins)
sp_print_df(languages_aj)
```

type	language_id	name	anti_joins
anti_join	2	Italian	1
anti_join	3	Japanese	1
anti_join	4	Mandarin	1
anti_join	5	French	1
anti_join	6	German	1

languages_aj

```
## # A tibble: 5 x 4
## # Groups:   type, language_id [5]
##   type      language_id name      anti_joins
##   <chr>          <int> <chr>      <int>
## 1 anti_join         2 "Italian"         1
## 2 anti_join         3 "Japanese"        1
## 3 anti_join         4 "Mandarin"        1
```

```
## 4 anti_join          5 "French          "          1
## 5 anti_join          6 "German          "          1
```

12.5.2.7 SQL anti Join 1, Left Outer Join where NULL on Right

SQL doesn't have an anti join key word. Here are three different ways to achieve the same result.

```
rs <- dbGetQuery(
  con,
  "select l.language_id,l.name,count(*) fojs
   from language l left outer join smy_film f on l.language_id = f.language_id
   where f.language_id is null
   group by l.language_id,l.name
   order by l.language_id;"
)
sp_print_df(rs)
```

language_id	name	fojs
2	Italian	1
3	Japanese	1
4	Mandarin	1
5	French	1
6	German	1

```
rs
```

```
##   language_id          name fojs
## 1           2 Italian          1
## 2           3 Japanese          1
## 3           4 Mandarin          1
## 4           5 French            1
## 5           6 German            1
```

12.5.2.8 SQL anti Join 2, ID in driving table and NOT IN lookup table

```
rs <- dbGetQuery(
  con,
  "select l.language_id,l.name,count(*) fojs
   from language l
   where l.language_id NOT IN (select language_id from film)
   group by l.language_id,l.name
   order by l.language_id;"
)
sp_print_df(rs)
```

language_id	name	fojs
2	Italian	1
3	Japanese	1
4	Mandarin	1
5	French	1
6	German	1

```
rs
```

```
##   language_id          name fojs
## 1           2 Italian          1
```

```
## 2          3 Japanese          1
## 3          4 Mandarin          1
## 4          5 French            1
## 5          6 German            1
```

12.5.2.9 SQL anti Join 3, NOT EXISTS and Correlated subquery

```
rs <- dbGetQuery(
  con,
  "select l.language_id,l.name,count(*) fojs
   from language l
   where not exists (select language_id from film f where f.language_id = l.language_id)
  group by l.language_id,l.name
"
)
sp_print_df(rs)
```

language_id	name	fojs
2	Italian	1
3	Japanese	1
4	Mandarin	1
5	French	1
6	German	1

```
rs
```

```
## language_id          name fojs
## 1          2 Italian          1
## 2          3 Japanese          1
## 3          4 Mandarin          1
## 4          5 French            1
## 5          6 German            1
```

12.6 SQL anti join Costs

```
sql_aj1 <- dbGetQuery(
  con,
  "explain analyze select l.language_id,l.name,count(*) fojs
   from language l left outer join smy_film f on l.language_id = f.language_id
   where f.language_id is null
  group by l.language_id,l.name
"
)

sql_aj2 <- dbGetQuery(
  con,
  "explain analyze select l.language_id,l.name,count(*) fojs
   from language l
   where l.language_id NOT IN (select language_id from film)
  group by l.language_id,l.name
"
)
```

```
sql_aj3 <- dbGetQuery(
  con,
  "explain analyze select l.language_id,l.name,count(*) fojs
   from language l
   where not exists (select language_id from film f where f.language_id = l.language_id)
   group by l.language_id,l.name
"
)
```

12.6.0.0.1 SQL Costs

```
print(glue("sql_aj1 loj-null costs=", sql_aj1[1, 1]))
```

```
## sql_aj1 loj-null costs=GroupAggregate (cost=68.56..68.61 rows=3 width=96) (actual time=19.525..19.645 r
```

```
print(glue("sql_aj2 not in costs=", sql_aj2[1, 1]))
```

```
## sql_aj2 not in costs=GroupAggregate (cost=67.60..67.65 rows=3 width=96) (actual time=18.669..18.807 row
```

```
print(glue("sql_aj3 not exist costs=", sql_aj3[1, 1]))
```

```
## sql_aj3 not exist costs=GroupAggregate (cost=24.24..24.30 rows=3 width=96) (actual time=0.527..0.660 ro
```

12.7 dplyr Anti joins

In this next section we look at two methods to implemnt an anti join in dplyr.

```
customer_table <- tbl(con, "customer") # DBI::dbReadTable(con, "customer")
rental_table <- tbl(con, "rental") # DBI::dbReadTable(con, "rental")

# Method 1. dplyr anti_join
daj1 <-
  anti_join(customer_table, rental_table, by = "customer_id", suffix = c(".c", ".r")) %>%
  select(c("first_name", "last_name", "email")) %>%
  explain()

## <SQL>
## SELECT "first_name", "last_name", "email"
## FROM (SELECT * FROM "customer" AS "TBL_LEFT"
##
## WHERE NOT EXISTS (
##   SELECT 1 FROM "rental" AS "TBL_RIGHT"
##   WHERE ("TBL_LEFT"."customer_id" = "TBL_RIGHT"."customer_id")
## )) "lghnetfgzu"
##
## <PLAN>
## Hash Anti Join (cost=510.99..552.63 rows=300 width=334)
##   Hash Cond: ("TBL_LEFT".customer_id = "TBL_RIGHT".customer_id)
##   -> Seq Scan on customer "TBL_LEFT" (cost=0.00..14.99 rows=599 width=338)
##   -> Hash (cost=310.44..310.44 rows=16044 width=2)
##         -> Seq Scan on rental "TBL_RIGHT" (cost=0.00..310.44 rows=16044 width=2)
customer_table <- tbl(con, "customer") # DBI::dbReadTable(con, "customer")
rental_table <- tbl(con, "rental") # DBI::dbReadTable(con, "rental")
```

```
# Method 2. dplyr loj with NA
daj2 <-
  left_join(customer_table, rental_table, by = c("customer_id", "customer_id"), suffix = c(".c", ".r"))
  filter(is.na(rental_id)) %>%
  select(c("first_name", "last_name", "email")) %>%
  explain()

## <SQL>
## SELECT "first_name", "last_name", "email"
## FROM (SELECT "TBL_LEFT"."customer_id" AS "customer_id", "TBL_LEFT"."store_id" AS "store_id", "TBL_LEFT"
## FROM "customer" AS "TBL_LEFT"
## LEFT JOIN "rental" AS "TBL_RIGHT"
## ON ("TBL_LEFT"."customer_id" = "TBL_RIGHT"."customer_id")
## ) "bisndnmtew"
## WHERE (((("rental_id") IS NULL))

##

## <PLAN>
## Hash Right Join (cost=22.48..375.33 rows=80 width=334)
## Hash Cond: ("TBL_RIGHT".customer_id = "TBL_LEFT".customer_id)
## Filter: ("TBL_RIGHT".rental_id IS NULL)
## -> Seq Scan on rental "TBL_RIGHT" (cost=0.00..310.44 rows=16044 width=6)
## -> Hash (cost=14.99..14.99 rows=599 width=338)
## -> Seq Scan on customer "TBL_LEFT" (cost=0.00..14.99 rows=599 width=338)
```

12.7.1 dplyr Costs

```
<PLAN>
Hash Anti Join (cost=510.99..529.72 rows=1 width=45)
Hash Cond: ("TBL_LEFT".customer_id = "TBL_RIGHT".customer_id)
-> Seq Scan on customer "TBL_LEFT" (cost=0.00..14.99 rows=599 width=49)
-> Hash (cost=310.44..310.44 rows=16044 width=2)
-> Seq Scan on rental "TBL_RIGHT" (cost=0.00..310.44 rows=16044 width=2)

<PLAN>
Hash Right Join (cost=22.48..375.33 rows=1 width=45)
Hash Cond: ("TBL_RIGHT".customer_id = "TBL_LEFT".customer_id)
Filter: ("TBL_RIGHT".rental_id IS NULL)
-> Seq Scan on rental "TBL_RIGHT" (cost=0.00..310.44 rows=16044 width=6)
-> Hash (cost=14.99..14.99 rows=599 width=49)
-> Seq Scan on customer "TBL_LEFT" (cost=0.00..14.99 rows=599 width=49)
```

In this example, the dplyr anti_join verb is 1.4113447 to 22.7308719 times more expensive than the left outer join with a null condition.

```
sql_aj1 <- dbGetQuery(
  con,
  "explain analyze select c.customer_id,count(*) lojs
  from customer c left outer join rental r on c.customer_id = r.customer_id
  where r.customer_id is null
  group by c.customer_id
  order by c.customer_id;"
)
sp_print_df(sql_aj1)
```

```

QUERY PLAN
GroupAggregate (cost=564.97..570.22 rows=300 width=12) (actual time=268.422..268.436 rows=1 loops=1)
Group Key: c.customer_id
-> Sort (cost=564.97..565.72 rows=300 width=4) (actual time=268.389..268.403 rows=1 loops=1)
Sort Key: c.customer_id
Sort Method: quicksort Memory: 25kB
-> Hash Anti Join (cost=510.99..552.63 rows=300 width=4) (actual time=268.326..268.355 rows=1 loops=1)
Hash Cond: (c.customer_id = r.customer_id)
-> Seq Scan on customer c (cost=0.00..14.99 rows=599 width=4) (actual time=0.036..4.457 rows=600 loops=1)
-> Hash (cost=310.44..310.44 rows=16044 width=2) (actual time=259.140..259.147 rows=16044 loops=1)
Buckets: 16384 Batches: 1 Memory Usage: 661kB
-> Seq Scan on rental r (cost=0.00..310.44 rows=16044 width=2) (actual time=0.042..129.496 rows=16044 loops=1)
Planning time: 0.117 ms
Execution time: 268.621 ms
sql_aj1

```

```

## QUERY PLAN
## 1 GroupAggregate (cost=564.97..570.22 rows=300 width=12) (actual time=268.422..268.436 rows=1 loops=1)
## 2 Group Key: c.customer_id
## 3 -> Sort (cost=564.97..565.72 rows=300 width=4) (actual time=268.389..268.403 rows=1 loops=1)
## 4 Sort Key: c.customer_id
## 5 Sort Method: quicksort Memory: 25kB
## 6 -> Hash Anti Join (cost=510.99..552.63 rows=300 width=4) (actual time=268.326..268.355 rows=1 loops=1)
## 7 Hash Cond: (c.customer_id = r.customer_id)
## 8 -> Seq Scan on customer c (cost=0.00..14.99 rows=599 width=4) (actual time=0.036..4.457 rows=600 loops=1)
## 9 -> Hash (cost=310.44..310.44 rows=16044 width=2) (actual time=259.140..259.147 rows=16044 loops=1)
## 10 Buckets: 16384 Batches: 1 Memory Usage: 661kB
## 11 -> Seq Scan on rental r (cost=0.00..310.44 rows=16044 width=2) (actual time=0.042..129.496 rows=16044 loops=1)
## 12 Planning time: 0.117 ms
## 13 Execution time: 268.621 ms

```

```

sql_aj3 <- dbGetQuery(
  con,
  "explain analyze
select c.customer_id,count(*) lojs
  from customer c
  where not exists (select customer_id from rental r where c.customer_id = r.customer_id)
group by c.customer_id
"
)

print(glue("sql_aj1 loj-null costs=", sql_aj1[1, 1]))

```

```

## sql_aj1 loj-null costs=GroupAggregate (cost=564.97..570.22 rows=300 width=12) (actual time=268.422..268.436 rows=1 loops=1)
print(glue("sql_aj3 not exist costs=", sql_aj3[1, 1]))

```

```

## sql_aj3 not exist costs=HashAggregate (cost=554.13..557.13 rows=300 width=12) (actual time=246.510..246.524 rows=1 loops=1)

```

12.8 Exercises

12.8.1 Anti joins – Find customers who have never rented a movie, take 2.

This is a left outer join from customer to the rental table with an NA rental_id.

12.8.1.1 SQL Anti-Join

```
rs <- dbGetQuery(
  con,
  "select c.first_name
      ,c.last_name
      ,c.email
    from customer c
    left outer join rental r
      on c.customer_id = r.customer_id
    where r.rental_id is null;
  "
)
sp_print_df(head(rs))
```

first_name	last_name	email
Sophie	Yang	email@email.com

<- Add dplyr semi-join example ->

12.8.2 SQL Rows Per Table

In the examples above, we looked at how many rows were involved in each of the join examples and which side of the join they came from. It is often helpful to know how many rows are in each table as a sanity check on the joins.

Below is the SQL version to return all the row counts from each table in the DVD Rental System.

```
rs <- dbGetQuery(
  con,
  "select *
    from (
      select 'actor' tbl_name,count(*) from actor
    union select 'category' tbl_name,count(*) from category
    union select 'film' tbl_name,count(*) from film
    union select 'film_actor' tbl_name,count(*) from film_actor
    union select 'film_category' tbl_name,count(*) from film_category
    union select 'language' tbl_name,count(*) from language
    union select 'inventory' tbl_name,count(*) from inventory
    union select 'rental' tbl_name,count(*) from rental
    union select 'payment' tbl_name,count(*) from payment
    union select 'staff' tbl_name,count(*) from staff
    union select 'customer' tbl_name,count(*) from customer
    union select 'address' tbl_name,count(*) from address
    union select 'city' tbl_name,count(*) from city
    union select 'country' tbl_name,count(*) from country
    union select 'store' tbl_name,count(*) from store
    union select 'smy_film' tbl_name,count(*) from smy_film
    ) counts
    order by tbl_name
  ;
  "
)
sp_print_df(head(rs))
```


tbl_name	count
actor	200
address	603
category	16
city	600
country	109
customer	600

rs

```
##      tbl_name count
## 1      actor   200
## 2     address   603
## 3    category    16
## 4      city   600
## 5    country   109
## 6    customer   600
## 7       film  1000
## 8   film_actor  5462
## 9 film_category  1000
## 10   inventory  4581
## 11   language     6
## 12   payment 14596
## 13    rental 16044
## 14   smy_film  1001
## 15     staff     2
## 16     store     2
```

12.8.2.1 Exercise dplyr Rows Per Table

In the code block below

1. Get the row counts for a couple more tables
2. What is the structure of film_table object?

```
film_table <- tbl(con, "film") # DBI::dbReadTable(con, "customer")
language_table <- tbl(con, "language") # DBI::dbReadTable(con, "rental")

film_rows <- film_table %>% mutate(name = "film") %>% group_by(name) %>% summarize(rows = n())
language_rows <- language_table %>%
  mutate(name = "language") %>%
  group_by(name) %>%
  summarize(rows = n())
rows_per_table <- rbind(as.data.frame(film_rows), as.data.frame(language_rows))
rows_per_table

##      name rows
## 1    film 1000
## 2 language     6
```

12.8.2.2 SQL film distribution based on language

The SQL below is very similar to the SQL full Outer Join above. Instead of counting the joins, it counts the number films associated with each language.

```
rs <- dbGetQuery(
  con,
  "select l.language_id id
        ,l.name
        ,sum(case when f.language_id is not null then 1 else 0 end) total
  from language l
        full outer join film f
        on l.language_id = f.language_id
  group by l.language_id,l.name
  order by l.name;
  "
)
sp_print_df(head(rs))
```

id	name	total
1	English	1000
5	French	0
6	German	0
2	Italian	0
3	Japanese	0
4	Mandarin	0

```
rs
```

```
##   id          name total
## 1  1 English      1000
## 2  5 French         0
## 3  6 German         0
## 4  2 Italian         0
## 5  3 Japanese         0
## 6  4 Mandarin         0
```

12.8.2.3 Exercise dplyr film distribution based on language

Below is the code block from the `dplyr` Full Outer Join section above. Modify the code block to match the output from the SQL version.

```
rs <- dbGetQuery(
  con,
  "select l.language_id,l.name,f.language_id language_id_f,count(*) fojs
  from language l full outer join smy_film f on l.language_id = f.language_id
  group by l.language_id,l.name,f.language_id
  order by l.language_id;"
)
sp_print_df(rs)
```

language_id	name	language_id_f	fojs
1	English	1	1000
2	Italian	NA	1
3	Japanese	NA	1
4	Mandarin	NA	1
5	French	NA	1
6	German	NA	1
NA	NA	10	1

rs

```
##   language_id      name language_id_f fojs
## 1           1 English           1 1000
## 2           2 Italian           NA    1
## 3           3 Japanese          NA    1
## 4           4 Mandarin          NA    1
## 5           5 French            NA    1
## 6           6 German            NA    1
## 7           NA              <NA>    10    1
```

12.9 Store analysis

How are the stores performing.

12.9.1 SQL store revenue stream

How are the stores performing? The SQL code shows the payments made to each store in the business.

```
rs <- dbGetQuery(
  con,
  "select store_id,sum(p.amount) amt,count(*) cnt
    from payment p
    join staff s
    on p.staff_id = s.staff_id
   group by store_id order by 2 desc
  ;
"
)
sp_print_df(head(rs))
```

store_id	amt	cnt
2	31059.92	7304
1	30252.12	7292

12.9.1.1 Exercise dplyr store revenue stream

Complete the following code block to return the payments made to each store.

```
payment_table <- tbl(con, "payment") # DBI::dbReadTable(con, "payment")
staff_table <- tbl(con, "staff") # DBI::dbReadTable(con, "staff")

store_revenue <- payment_table %>%
  inner_join(staff_table, by = "staff_id", suffix = c(".p", ".s")) %>%
  head()

store_revenue

## # Source:   lazy query [?? x 16]
## # Database: postgres [postgres@localhost:5432/dvdrental]
##   payment_id customer_id staff_id rental_id amount payment_date
##   <int>      <int>      <int>      <int>  <dbl>  <dtm>
## 1      17503         341         2        1520    7.99 2007-02-15 22:25:46
```

```
## 2      17504      341      1      1778      1.99 2007-02-16 17:23:14
## 3      17505      341      1      1849      7.99 2007-02-16 22:41:45
## 4      17506      341      2      2829      2.99 2007-02-19 19:39:56
## 5      17507      341      2      3130      7.99 2007-02-20 17:31:48
## 6      17508      341      1      3382      5.99 2007-02-21 12:33:49
## # ... with 10 more variables: first_name <chr>, last_name <chr>,
## #   address_id <int>, email <chr>, store_id <int>, active <lgl>,
## #   username <chr>, password <chr>, last_update <dtm>, picture <blob>
```

12.9.2 SQL:Estimate Outstanding Balance

The following SQL code calculates for each store

1. the number of payments still open and closed from the DVD Rental Stores customer base.
2. the total amount that their customers have paid
3. the average price per/movie based off of the movies that have been paid.
4. the estimated outstanding balance based off the open unpaid rentals * the average price per paid movie.

```
rs <- dbGetQuery(
  con,
  "SELECT s.store_id store,sum(CASE WHEN payment_id IS NULL THEN 1 ELSE 0 END) open
    ,sum(CASE WHEN payment_id IS NOT NULL THEN 1 ELSE 0 END) paid
    ,sum(p.amount) paid_amt
    ,count(*) rentals
    ,round(sum(p.amount) / sum(CASE WHEN payment_id IS NOT NULL
                                THEN 1
                                ELSE 0
                                END), 2) avg_price
    ,round(round(sum(p.amount) / sum(CASE WHEN payment_id IS NOT NULL
                                THEN 1
                                ELSE 0
                                END), 2) * sum(CASE WHEN payment_id IS NULL
                                THEN 1
                                ELSE 0
                                END), 2) est_balance
  FROM rental r
  LEFT JOIN payment p
    ON r.rental_id = p.rental_id
  JOIN staff s
    ON r.staff_id = s.staff_id
  group by s.store_id;
"
)
sp_print_df(head(rs))
```

store	open	paid	paid_amt	rentals	avg_price	est_balance
1	713	7331	30498.71	8044	4.16	2966.08
2	739	7265	30813.33	8004	4.24	3133.36

rs

```
##   store open paid paid_amt rentals avg_price est_balance
## 1     1  713 7331 30498.71   8044     4.16    2966.08
## 2     2  739 7265 30813.33   8004     4.24    3133.36
```

12.9.2.1 Exercise Dplyr Modify the following dplyr code to match the SQL output from above.

```

payment_table <- tbl(con, "payment") # DBI::dbReadTable(con, "payment")
rental_table <- tbl(con, "rental") # DBI::dbReadTable(con, "rental")

est_bal <- rental_table %>%
  left_join(payment_table, by = c("rental_id", "rental_id"), suffix = c(".r", ".p")) %>%
  mutate(
    missing = ifelse(is.na(payment_id), 1, 0)
    , found = ifelse(!is.na(payment_id), 1, 0)
  ) %>%
  summarize(
    open = sum(missing, na.rm = TRUE)
    , paid = sum(found, na.rm = TRUE)
    , paid_amt = sum(amount, na.rm = TRUE)
    , rentals = n()
  ) %>%
  summarize(
    open = open
    , paid = paid
    , paid_amt = paid_amt
    , rentals = rentals
    , avg_price = paid_amt / paid
    , est_balance = paid_amt / paid * open
  )
est_bal

```

```

## # Source:   lazy query [?? x 6]
## # Database: postgres [postgres@localhost:5432/dvdrental]
##    open  paid paid_amt rentals      avg_price est_balance
##    <dbl> <dbl>    <dbl> <S3: integer64>    <dbl>        <dbl>
## 1  1452 14596    61312. 16048              4.20         6099.

```

12.9.3 SQL actual outstanding balance

In the previous exercise, we estimated the outstanding amount. After reviewing the rental table, the actual movie rental rate is in the table. We use that to calculate the outstanding balance below.

```

rs <- dbGetQuery(
  con,
  "SELECT sum(f.rental_rate) open_amt
    ,count(*) count
FROM rental r
LEFT JOIN payment p
  ON r.rental_id = p.rental_id
INNER JOIN inventory i
  ON r.inventory_id = i.inventory_id
INNER JOIN film f
  ON i.film_id = f.film_id
WHERE p.rental_id IS NULL
;"
)
sp_print_df(head(rs))

```

open_amt	count
4297.48	1452

rs

```
##   open_amt count
## 1  4297.48  1452
```

```
payment_table <- tbl(con, "payment") # DBI::dbReadTable(con, "payment")
rental_table <- tbl(con, "rental") # DBI::dbReadTable(con, "rental")
inventory_table <- tbl(con, "inventory") # DBI::dbReadTable(con, "inventory")
film_table <- tbl(con, "film") # DBI::dbReadTable(con, "film")

act_bal <- rental_table %>%
  left_join(payment_table, by = c("rental_id", "rental_id"), suffix = c(".r", ".p")) %>%
  inner_join(inventory_table, by = c("inventory_id", "inventory_id"), suffix = c(".r", ".i")) %>%
  inner_join(film_table, by = c("film_id", "film_id"), suffix = c(".i", ".f")) %>%
  head()

act_bal
```

```
## # Source:   lazy query [?? x 27]
## # Database: postgres [postgres@localhost:5432/dvdrental]
##   rental_id rental_date      inventory_id customer_id.r
##   <int>    <dtm>              <int>         <int>
## 1         1 2005-05-24 22:53:30         367           130
## 2         2 2005-05-24 22:54:33        1525           459
## 3         3 2005-05-24 23:03:39        1711           408
## 4         4 2005-05-24 23:04:41        2452           333
## 5         5 2005-05-24 23:05:21        2079           222
## 6         6 2005-05-24 23:08:07        2792           549
## # ... with 23 more variables: return_date <dtm>, staff_id.r <int>,
## #   last_update.r <dtm>, payment_id <int>, customer_id.p <int>,
## #   staff_id.p <int>, amount <dbl>, payment_date <dtm>, film_id <int>,
## #   store_id <int>, last_update.i <dtm>, title <chr>, description <chr>,
## #   release_year <int>, language_id <int>, rental_duration <int>,
## #   rental_rate <dbl>, length <int>, replacement_cost <dbl>, rating <S3:
## #   pq_mpa_rating>, last_update <dtm>, special_features <S3: pq_text>,
## #   fulltext <S3: pq_tsvector>
```

12.9.4 Rank customers with highest open amounts

```
rs <- dbGetQuery(
  con,
  "select c.customer_id,c.first_name,c.last_name,sum(f.rental_rate) open_amt,count(*) count
    from rental r
      left outer join payment p
        on r.rental_id = p.rental_id
      join inventory i
        on r.inventory_id = i.inventory_id
      join film f
        on i.film_id = f.film_id
      join customer c
        on r.customer_id = c.customer_id
  where p.rental_id is null
```

```

        group by c.customer_id,c.first_name,c.last_name
        order by open_amt desc
        limit 25
    ;"
)
sp_print_df(head(rs))

```

customer_id	first_name	last_name	open_amt	count
293	Mae	Fletcher	35.90	10
307	Joseph	Joy	31.90	10
316	Steven	Curley	31.90	10
299	James	Gannon	30.91	9
274	Naomi	Jennings	29.92	8
326	Jose	Andrew	28.93	7

```
rs
```

```

##      customer_id first_name  last_name open_amt count
## 1           293      Mae    Fletcher   35.90     10
## 2           307   Joseph      Joy    31.90     10
## 3           316   Steven    Curley    31.90     10
## 4           299    James    Gannon    30.91      9
## 5           274    Naomi   Jennings    29.92      8
## 6           326     Jose    Andrew    28.93      7
## 7           338   Dennis    Gilman    27.92      8
## 8           277     Olga   Jimenez    27.92      8
## 9           327    Larry   Thrasher    26.93      7
## 10          330    Scott   Shelley    26.93      7
## 11          322    Jason   Morrissey    26.91      9
## 12          340   Patrick   Newsom    25.92      8
## 13          336   Joshua     Mark    25.92      8
## 14          304    David     Royal    24.93      7
## 15          339   Walter   Perryman    23.94      6
## 16          239   Minnie    Romero    23.94      6
## 17          310   Daniel    Cabral    22.93      7
## 18          296   Ramona     Hale    22.93      7
## 19          313   Donald    Mahon    22.93      7
## 20          287    Becky     Miles    22.93      7
## 21          272     Kay    Caldwell    22.93      7
## 22          303   William Satterfield    22.93      7
## 23          329    Frank    Waggoner    22.91      9
## 24          311     Paul     Trout    21.92      8
## 25          109    Edna      West    20.93      7

```

12.9.5 what film has been rented the most

```

rs <- dbGetQuery(
  con,
  "SELECT i.film_id
     ,f.title
     ,rental_rate
     ,sum(rental_rate) revenue
     ,count(*) count --16044
FROM rental r

```

```

INNER JOIN inventory i
  ON r.inventory_id = i.inventory_id
INNER JOIN film f
  ON i.film_id = f.film_id
GROUP BY i.film_id
        ,f.title
        ,rental_rate
ORDER BY count DESC
LIMIT 25
;"
)
sp_print_df(head(rs))

```

film_id	title	rental_rate	revenue	count
103	Bucket Brotherhood	4.99	169.66	34
738	Rocketeer Mother	0.99	32.67	33
489	Juggler Hardly	0.99	31.68	32
730	Ridgemont Submarine	0.99	31.68	32
767	Scalawag Duck	4.99	159.68	32
331	Forward Temple	2.99	95.68	32

rs

```

##      film_id      title rental_rate revenue count
## 1      103  Bucket Brotherhood      4.99  169.66    34
## 2      738  Rocketeer Mother      0.99   32.67    33
## 3      489    Juggler Hardly      0.99   31.68    32
## 4      730  Ridgemont Submarine      0.99   31.68    32
## 5      767    Scalawag Duck      4.99  159.68    32
## 6      331    Forward Temple      2.99   95.68    32
## 7      382    Grit Clockwork      0.99   31.68    32
## 8      735    Robbers Joon      2.99   92.69    31
## 9      973      Wife Turn      4.99  154.69    31
## 10     621    Network Peak      2.99   92.69    31
## 11    1000      Zorro Ark      4.99  154.69    31
## 12      31    Apache Divine      4.99  154.69    31
## 13     369  Goodfellas Salute      4.99  154.69    31
## 14     753    Rush Goodfellas      0.99   30.69    31
## 15     891    Timberland Sky      0.99   30.69    31
## 16     418    Hobbit Alien      0.99   30.69    31
## 17     127    Cat Coneheads      4.99  149.70    30
## 18     559    Married Go      2.99   89.70    30
## 19     374    Graffiti Love      0.99   29.70    30
## 20     748  Rugrats Shakespeare      0.99   29.70    30
## 21     239    Dogma Family      4.99  149.70    30
## 22     285    English Bulworth      0.99   29.70    30
## 23     109  Butterfly Chocolat      0.99   29.70    30
## 24     450    Idols Snatchers      2.99   89.70    30
## 25     609    Muscle Bright      2.99   89.70    30

```


12.9.6 what film has been generated the most revenue assuming all amounts are collected

```
rs <- dbGetQuery(
  con,
  "select i.film_id,f.title,rental_rate
      ,sum(rental_rate) revenue,count(*) count  --16044
    from rental r
      join inventory i
        on r.inventory_id = i.inventory_id
      join film f
        on i.film_id = f.film_id
    group by i.film_id,f.title,rental_rate
    order by revenue desc
  ;"
)
sp_print_df(head(rs))
```

film_id	title	rental_rate	revenue	count
103	Bucket Brotherhood	4.99	169.66	34
767	Scalawag Duck	4.99	159.68	32
973	Wife Turn	4.99	154.69	31
31	Apache Divine	4.99	154.69	31
369	Goodfellas Salute	4.99	154.69	31
1000	Zorro Ark	4.99	154.69	31

12.9.7 which films are in one store but not the other.

```
rs <- dbGetQuery(
  con,
  "select coalesce(i1.film_id,i2.film_id) film_id
      ,f.title,f.rental_rate,i1.store_id,i1.count,i2.store_id,i2.count
    from      (select film_id,store_id,count(*) count
                from inventory where store_id = 1
                group by film_id,store_id) as i1
    full outer join
      (select film_id,store_id,count(*) count
        from inventory where store_id = 2
        group by film_id,store_id
      ) as i2
      on i1.film_id = i2.film_id
    join film f
      on coalesce(i1.film_id,i2.film_id) = f.film_id
    where i1.film_id is null or i2.film_id is null
    order by f.title ;
  "
)
sp_print_df(head(rs))
```

film_id	title	rental_rate	store_id	count	store_id..6	count..7
2	Ace Goldfinger	4.99	NA	NA	2	3
3	Adaptation Holes	2.99	NA	NA	2	4
5	African Egg	2.99	NA	NA	2	3
8	Airport Pollock	4.99	NA	NA	2	4
13	Ali Forever	4.99	NA	NA	2	4
20	Amelie Hellfighters	4.99	1	3	NA	NA

12.9.8 Compute the outstanding balance.

```
rs <- dbGetQuery(
  con,
  "select sum(f.rental_rate) open_amt,count(*) count
    from rental r
    left outer join payment p
      on r.rental_id = p.rental_id
    join inventory i
      on r.inventory_id = i.inventory_id
    join film f
      on i.film_id = f.film_id
  where p.rental_id is null
  ;"
)
sp_print_df(head(rs))
```

open_amt	count
4297.48	1452

12.10 Different strategies for interacting with the database

select examples dbGetQuery returns the entire result set as a data frame.

For large returned datasets, complex or inefficient SQL statements, this may take a long time.

dbSendQuery: parses, compiles, creates the optimized execution plan.

dbFetch: Execute optimized execution plan and return the dataset.

dbClearResult: remove pending query results from the database to your R environment

12.10.1 Use dbGetQuery

How many customers are there in the DVD Rental System

```
rs1 <- dbGetQuery(con, "select * from customer;")
sp_print_df(head(rs1))
```

customer_id	store_id	first_name	last_name	email	address_id	activebool	cre
524	1	Jared	Ely	jared.ely@sakilacustomer.org	530	TRUE	200
1	1	Mary	Smith	mary.smith@sakilacustomer.org	5	TRUE	200
2	1	Patricia	Johnson	patricia.johnson@sakilacustomer.org	6	TRUE	200
3	1	Linda	Williams	linda.williams@sakilacustomer.org	7	TRUE	200
4	2	Barbara	Jones	barbara.jones@sakilacustomer.org	8	TRUE	200
5	1	Elizabeth	Brown	elizabeth.brown@sakilacustomer.org	9	TRUE	200

```
pco <- dbSendQuery(con, "select * from customer;")
rs2 <- dbFetch(pco)
dbClearResult(pco)
sp_print_df(head(rs2))
```

customer_id	store_id	first_name	last_name	email	address_id	activebool	cre
524	1	Jared	Ely	jared.ely@sakilacustomer.org	530	TRUE	200
1	1	Mary	Smith	mary.smith@sakilacustomer.org	5	TRUE	200
2	1	Patricia	Johnson	patricia.johnson@sakilacustomer.org	6	TRUE	200
3	1	Linda	Williams	linda.williams@sakilacustomer.org	7	TRUE	200
4	2	Barbara	Jones	barbara.jones@sakilacustomer.org	8	TRUE	200
5	1	Elizabeth	Brown	elizabeth.brown@sakilacustomer.org	9	TRUE	200

12.10.2 Use dbExecute

12.10.3 Anti join – Find Sophie who has never rented a movie.

```
customer_table <- DBI::dbReadTable(con, "customer")
rental_table <- DBI::dbReadTable(con, "rental")

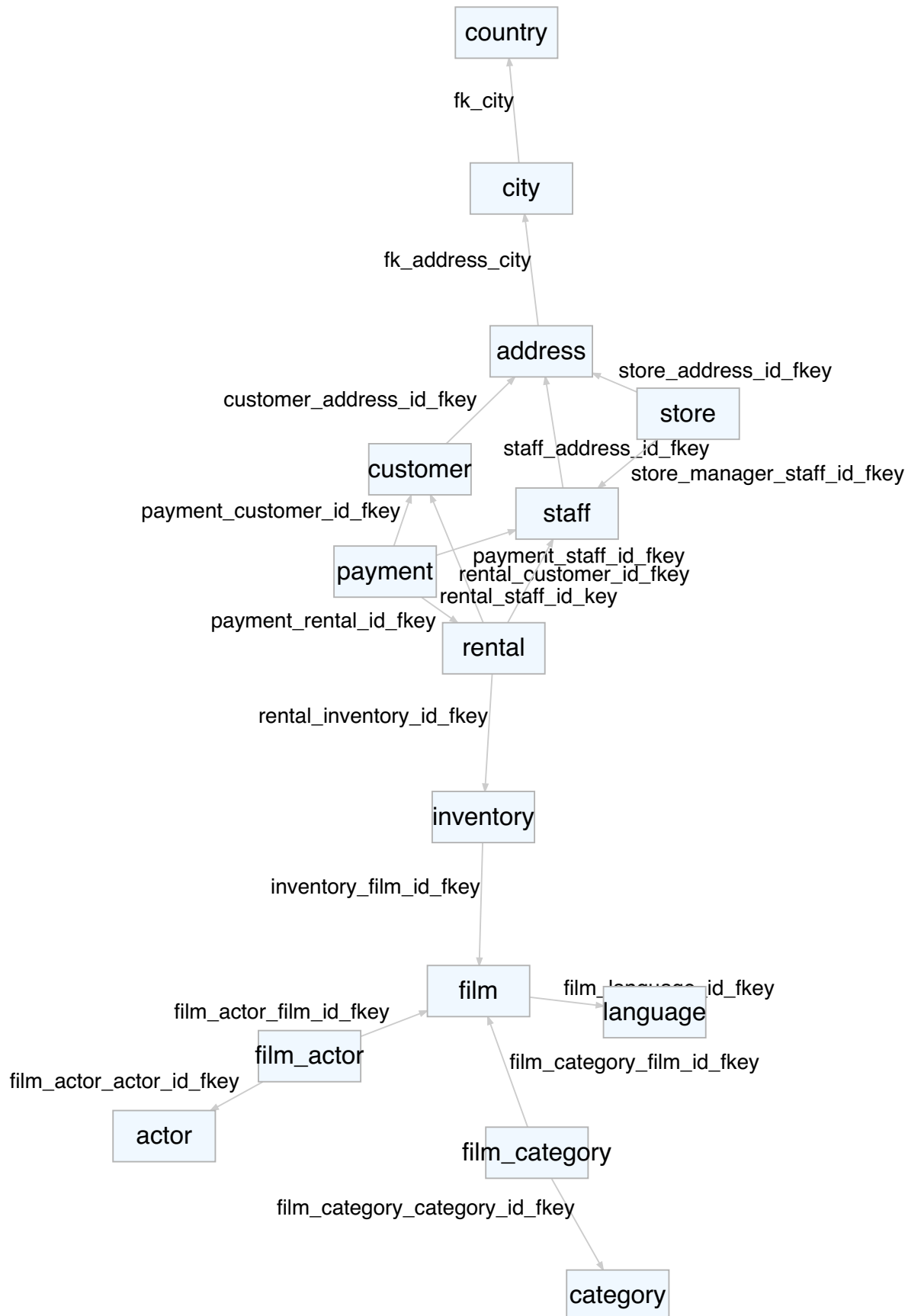
customer_tbl <- dplyr::tbl(con, "customer")
rental_tbl <- dplyr::tbl(con, "rental")

dplyr_tbl_loj <-
  left_join(customer_tbl, rental_tbl, by = "customer_id", suffix = c(".c", ".r")) %>%
  filter(is.na(rental_id)) %>%
  select(c("first_name", "last_name", "email"))

rs <- dbGetQuery(
  con,
  "select c.first_name
      ,c.last_name
      ,c.email
  from customer c
      left outer join rental r
      on c.customer_id = r.customer_id
  where r.rental_id is null;
  "
)
sp_print_df(head(rs))
```

first_name	last_name	email
Sophie	Yang	email@email.com

```
View(dplyr_tbl_loj)
```



```
# diconnect from the db
dbDisconnect(con)

sp_docker_stop("sql-pet")

knitr::knit_exit()
```


Chapter 13

SQL Quick start - simple retrieval (15)

This chapter demonstrates:

- Several elementary SQL statements
- SQL databases and 3rd normal form

13.1 Intro

- Coverage in this book. There are many SQL tutorials that are available. For example, we are drawing some materials from a tutorial we recommend. In particular, we will not replicate the lessons there, which you might want to complete. Instead, we are showing strategies that are recommended for R users. That will include some translations of queries that are discussed there.
- <https://datacarpentry.org/R-ecology-lesson/05-r-and-databases.html> Very good intro. How is ours different?

Start up the `docker-pet` container

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

Now connect to the `dvdrental` database with R

```
con <- sp_get_postgres_connection(  
  user = Sys.getenv("DEFAULT_POSTGRES_USER_NAME"),  
  password = Sys.getenv("DEFAULT_POSTGRES_PASSWORD"),  
  dbname = "dvdrental",  
  seconds_to_test = 10)  
con  
  
## <PqConnection> dvdrental@localhost:5432  
  
colFmt <- function(x,color)  
{  
  # x string  
  # color  
  outputFormat = knitr::opts_knit$get("rmarkdown.pandoc.to")  
  if(outputFormat == 'latex')  
    paste("\\textcolor{" ,color,"}-{",x,"}",sep="")  
  else if(outputFormat == 'html')  
    paste("<span style='color:",color,">{",x,"}</span>",sep="")  
}
```

```

paste("<font color='",color,"'>",x,"</font>",sep="")
else
  x
}

# sample call
# * `r colFmt('Cover inline tables in future section','red')`

```

Moved this from 11-elementary-queries

```

dplyr_summary_df <-
  read.delim(
    '11-dplyr_sql_summary_table.tsv',
    header = TRUE,
    sep = '\t',
    as.is = TRUE
  )

head(dplyr_summary_df)

```

```

##   In           Dplyr_Function
## 1 Y              arrange()
## 2 Y?             distinct()
## 3 Y      select() rename()
## 4 N              pull()
## 5 Y    mutate() transmute()
## 6 Y summarise() summarize()
##
##                                     description
## 1                               Arrange rows by variables
## 2                Return rows with matching conditions
## 3              Select/rename variables by name
## 4                Pull out a single variable
## 5                  Add new variables
## 6 Reduces multiple values down to a single value
##
##               SQL-Clause Notes              Category
## 1                ORDER BY          NA Basic single-table verbs
## 2          SELECT distinct *          NA Basic single-table verbs
## 3    SELECT column_name alias_name      NA Basic single-table verbs
## 4          SELECT column_name;          NA Basic single-table verbs
## 5 SELECT computed_value computed_name    NA Basic single-table verbs
## 6 SELECT aggregate_functions GROUP BY    NA Basic single-table verbs

```

13.2 Databases and Third Normal Form - 3NF

Most relational database applications are designed to be third normal form “like”, 3NF. The key benefits of 3NF are

1. speedy on-line transactional processing, OLTP.
2. improved referential integrity, reduce modification anomalies that can occur during an insert, update, or delete operation.
3. reduced storage, elimination of redundant data.

3NF is great for database application input performance, but not so great for getting the data back out for the data analyst or report writer. As a data analyst, you might get the ubiquitous Excel spreadsheet

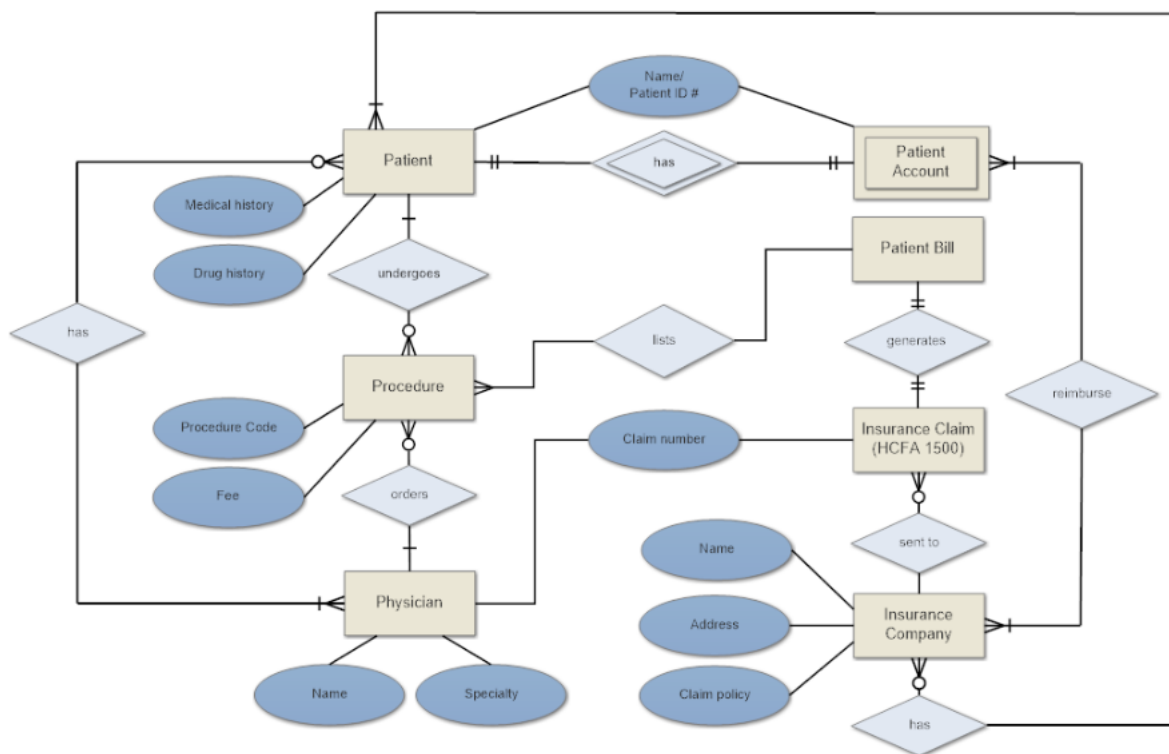


Figure 13.1: hospital-billing-erd

with all the information needed to start an Exploratory Data Analysis, EDA. The spreadsheet may have provider, patient, diagnosis, procedure, and insurance information all “neatly” arranged on a single row. At least “neatly” when compared to the same information stored in the database, in at least 5 tables.

For this tutorial, the most important thing to know about 3NF is that the data you are looking for gets spread across many many tables. Working in a relational database requires you to

1. find the many many different tables that contains your data.
2. Understand the relationships that tie the tables together correctly to ensure that data is not dropped or duplicated. Data that is dropped or duplicated can either over or understate your aggregated numeric values.

<https://www.smartdraw.com/entity-relationship-diagram/examples/hospital-billing-entity-relationship-diagram/>

Real life applications have 100’s or even 1000’s of tables supporting the application. The goal is to transform the application data model into a useful data analysis model using the DDL and DML SQL statements.

13.3 SQL Commands

SQL commands fall into four categories.

SQL Category	Definition
DDL:Data Definition Language	DBA’s execute these commands to define objects in the database.

SQL Category	Definition
DML:Data Manipulation Language	Users and developers execute these commands to investigate data.
DCL:Data Control Language	DBA's execute these commands to grant/revoke access to
TCL:Transaction Control Language	Developers execute these commands when developing applications.

Data analysts use the SELECT DML command to learn interesting things about the data stored in the database. Applications are used to control the insert, update, and deletion of data in the database. Data users can update the database objects via the application which enforces referential integrity in the database. Data users should never directly update data application database objects. Leave this task to the developers and DBA's.

DBA's can setup a sandbox within the database for a data analyst. The application(s) do not maintain the data in the sandbox.

The `sql-pet` database is tiny, but for the purposes of these exercises, we assume that data so large that it will not easily fit into the memory of your laptop.

This tutorial focuses on the most frequently used SQL statement, the SQL SELECT statement.

A SQL SELECT statement consists of 1 to 6 clauses.

SQL Clause	DPLYR Verb	SQL Description
SELECT	SELECT()	Contains a list of column names from an object or a derived value.
FROM	mutate()	Contains a list of related tables from which the SELECT list of columns is derived.
WHERE	filter()	Provides the filter conditions the objects in the FROM clause must meet.
GROUP BY	group_by()	Contains a list rollup aggregation columns.
HAVING		Provides the filter condition on the the GROUP BY clause.
ORDER BY	arrange()	Contains a list of column names indicating the order of the column value. Each column can be either ASCending or DESCending.

The foundation of the SQL language is based set theory and the result of a SQL SELECT statement is referred to as a result set. A SQL SELECT statement is "guaranteed" to return the same set of data, but not necessarily in the same order. However, in practice, the result set is usually in the same order.

SQL SELECT statements can be broken up into two categories, SELECT detail statements and SELECT aggregate statements.

SELECT DETAIL	SELECT AGGREGATE
select det_coll...det_coln	select det_agg1..., agg1,...,aggN
from same	from same
where same	where same
	group by det_agg1
	having
order by same	order by same

Table 13.4: select all columns

store_id	manager_staff_id	address_id	last_update
1	1	1	2006-02-15 09:57:12
2	2	2	2006-02-15 09:57:12

SELECT DETAIL	SELECT AGGREGATE
---------------	------------------

The difference between the two statements is the AGGREGATE has

1. select clause has one or more detail columns, det_agg1..., on which values get aggregated against/rolled up to.
2. select clause zero or more aggregated values, agg1, ..., aggn
3. group by clause is required and matches the one or more detail columns, det_agg1.
4. having clause is optional and adds a filter condition on one or more agg1 ... aggn values.

13.4 SQL SELECT Quick Start

This section focuses on getting new SQL users familiar with the six SQL query clauses and a single table. SQL queries from multiple tables are discussed in the JOIN section of this tutorial. The JOIN section resolves the issue introduced with 3NF, the splitting of data into many many tables, back into a denormalized format similar to the Excel spreadsheet.

The DBI::dbGetQuery function is used to submit SQL SELECT statements to the Postgres database. At a minimum it requires two parameters, a connection object and a SQL SELECT statement.

In the following section we only look at SELECT DETAIL statements.

13.4.1 SELECT Clause: Column Selection – Vertical Partitioning of Data

13.4.1.1 1. Simplest SQL query: All rows and all columns from a single table.

```
rs <-
  DBI::dbGetQuery(
    con,
    "
    select * from store;
    ")
kable(rs,caption = 'select all columns')
```

13.4.1.2 2. Same Query as 1, but only show first two columns;

```
rs <-
  DBI::dbGetQuery(
    con,
    "
    select STORE_ID, manager_staff_id from store;
    ")
kable(rs,caption = 'select first two columns only')
```

Table 13.5: select first two columns only

store_id	manager_staff_id
1	1
2	2

Table 13.6: reverse the column order

manager_staff_id	store_id
1	1
2	2

13.4.1.3 3. Same Query as 2, but reverse the column order

dvdrental=# select manager_staff_id,store_id from store;

```
rs <-
  DBI::dbGetQuery(
    con,
    "
    select manager_staff_id,store_id from store;
    ")
kable(rs,caption = 'reverse the column order')
```

13.4.1.4 4. Rename Columns – SQL column alias in the result set

```
rs <-
  DBI::dbGetQuery(
    con,
    "
    select manager_staff_id mgr_sid,store_id st_id from store;
    ")
kable(rs,caption = 'Rename Columns')
```

The manager_staff_id has changed to mgr_sid.
store_id has changed to st_id.

Note that the column names have changed in the result set only, not in the actual database table. The DBA's will not allow a space or other special characters in a database table column name.

Some motivations for aliasing the result set column names are

1. Some database table column names are not user friendly.
2. When multiple tables are joined, the column names may be the same in one or more tables and one needs to d

Table 13.7: Rename Columns

mgr_sid	st_id
1	1
2	2

Table 13.8: Adding Meta Data Columns

showing	store_id	manager_staff_id	address_id	last_update	db	user	dtts
derived column	1	1	1	2006-02-15 09:57:12	dvdrental	postgres	2018/12/06 2
derived column	2	2	2	2006-02-15 09:57:12	dvdrental	postgres	2018/12/06 2

13.4.1.5 5. Adding Meta Data Columns to the Result Set

```
rs <-
  DBI::dbGetQuery(
    con,
    "
    select 'derived column' showing
      ,*
      ,current_database() db
      ,user
      ,to_char(now(),'YYYY/MM/DD HH24:MI:SS') dtts
    from store;
    ")
kable(rs,caption = 'Adding Meta Data Columns')
```

All the previous examples easily fit on a single line. This one is longer. Each column is entered on its own

1. The showing column is a hard coded string surrounded by single quotes. Note that single quotes are for ha
2. The db and dtts, date timestamp, are new columns generated from Postgres System Information Functions.
3. Note that `user` is not a function call, no parenthesis.

13.4.2 SQL Comments

SQL supports both a single line comment, precede the line with two dashes, --, and a C like block comment, `* ... */`.

13.4.2.1 6. Single line comment –

```
rs <-
  DBI::dbGetQuery(
    con,
    "
    select 'single line comment, dtts' showing
      ,*
      ,current_database() db
      ,user
      -- ,to_char(now(),'YYYY/MM/DD HH24:MI:SS') dtts
    from store;
    ")
kable(rs,caption = 'Sincle line comment')
```

The dtts line is commented out with the two dashes and is dropped from the end of the result set columns.

Table 13.9: Single line comment

showing	store_id	manager_staff_id	address_id	last_update	db	user
single line comment, dtts	1	1	1	2006-02-15 09:57:12	dvdrental	postgres
single line comment, dtts	2	2	2	2006-02-15 09:57:12	dvdrental	postgres

Table 13.10: Multi-line comment

showing	store_id	manager_staff_id	address_id	last_update
block comment drop db, user, and dtts	1	1	1	2006-02-15 09:57:12
block comment drop db, user, and dtts	2	2	2	2006-02-15 09:57:12

13.4.2.2 7. Multi-line comment `/*...*/`

```
rs <-
  DBI::dbGetQuery(
    con,
    "
      select 'block comment drop db, user, and dtts' showing
        ,*
        /*
        ,current_database() db
        ,user
        ,to_char(now(),'YYYY/MM/DD HH24:MI:SS') dtts
        */
      from store;
    ")
kable(rs,caption = 'Multi-line comment')
```

The three columns db, user, and dtts, between the `/*` and `*/` have been commented and no longer appear as the

13.4.3 FROM Clause

The **FROM** clause contains one or more datasets, usually database tables/views, from which the **SELECT** columns are derived. For now, in the examples, we are only using a single table. If the database reflects a relational model, your data is likely spread out over several tables. The key take away when beginning your analysis is to pick the table that has most of the data that you need for your analysis. This table becomes your main or driving table to build your SQL query statement around. After identifying your driving table, potentially save yourself a lot of time and heart ache, review any view that is built on your driving table. If one or more exist, especially, if vendor built, may already have the additional information needed for your analysis.

[Insert SQL here or link to Views dependent on what](#)

In this tutorial, there is only a single user hitting the database and row/table locking is not necessary and considered out of scope.

13.4.3.1 Table Uses

- A table can be used more than once in a FROM clause. These are self-referencing tables. An example is an EMPLOYEE table which contains a foreign key to her manager. Her manager also has a foreign

key to her manager, etc up the corporate ladder.

- In the example above, the EMPLOYEE table plays two roles, employee and manager. The next line shows the FROM clause showing the same table used twice.

FROM EMPLOYEE EE, EMPLOYEE MGR

- The EE and MGR are aliases for the EMPLOYEE table and represent the different roles the EMPLOYEE table plays.
- Since all the column names are exactly the same for the EE and MGR role, the column names need to be prefixed with their role alias, e.g., SELECT MGR.EE_NAME, EE.EE_NAME ... shows the manager name and her employee name(s) who work for her.
- It is a good habit to always alias your tables and prefix your column names with the table alias to eliminate any ambiguity as to where the column came from. This is critical where there is inconsistent table column naming convention. It also helps when debugging larger SQL queries.
- **Cover inline tables in future section**

Side Note: Do not create an unintended Cartesian join. If one has more than one table in the FROM clause, make

13.4.4 WHERE Clause: Row Selection – Horizontal Partitioning of Data

In the previous SELECT clause section, the SELECT statement either partitioned data vertically across the table columns or derived vertical column values. This section provides examples that partitions the table data across rows in the table.

The WHERE clause defines all the conditions the data must meet to be included or excluded in the final result set. If all the conditions are met data is returned or it is rejected. This is commonly referred to as the data set filter condition.

Side Note: For performance optimization reasons, the WHERE clause should reduce the dataset down to the smallest

The WHERE condition(s) can be simple or complex, but in the end are the application of the logic rules shown in the table below.

p	q	p and q	p or q
T	T	T	T
T	F	F	T
T	N	N	T
F	F	F	F
F	N	F	T
N	N	N	N

When the filter logic is complex, it is sometimes easier to represent the where clause symbolically and apply a version of DeMorgan's law which is shown below.

1. $(A \text{ and } B)' = A' \text{ or } B'$
2. $(A \text{ or } B)' = A' \text{ and } B'$

13.4.4.1 Examples Continued

We begin with 1, our simplest SQL query.

Table 13.12: select all columns

store_id	manager_staff_id	address_id	last_update
1	1	1	2006-02-15 09:57:12
2	2	2	2006-02-15 09:57:12

Table 13.13: WHERE always FALSE

store_id	manager_staff_id	address_id	last_update
----------	------------------	------------	-------------

```
rs <-
  DBI::dbGetQuery(
    con,
    "
    select * from store;
    ")
kable(rs,caption = 'select all columns')
```

13.4.4.2 8 WHERE condition logically never TRUE.

```
rs <-
  DBI::dbGetQuery(
    con,
    "
    select * from store where 1 = 0;
    ")
kable(rs,caption = 'WHERE always FALSE')
```

Since $1 = 0$ is always false, no rows are ever returned. Initially this construct seems useless, but actually

13.4.4.3 9 WHERE condition logically always TRUE.

```
rs <-
  DBI::dbGetQuery(
    con,
    "
    select * from store where 1 = 1;
    ")
kable(rs,caption = 'WHERE always TRUE')
```

Since $1 = 1$ is always true, all rows are always returned. Initially this construct seems useless, but actually

Table 13.14: WHERE always TRUE

store_id	manager_staff_id	address_id	last_update
1	1	1	2006-02-15 09:57:12
2	2	2	2006-02-15 09:57:12

Table 13.15: WHERE EQUAL

store_id	manager_staff_id	address_id	last_update
2	2	2	2006-02-15 09:57:12

Table 13.16: WHERE NOT EQUAL

store_id	manager_staff_id	address_id	last_update
1	1	1	2006-02-15 09:57:12

13.4.4.4 10 WHERE equality condition

```
rs <-
  DBI::dbGetQuery(
    con,
    "
    select * from store where store_id = 2;
    ")
kable(rs,caption = 'WHERE EQUAL')
```

The only row where the store_id = 2 is row 2 and it is the only row returned.

13.4.4.5 11 WHERE NOT equal conditions

```
rs <-
  DBI::dbGetQuery(
    con,
    "
    select * from store where store_id <> 2;
    ")
kable(rs,caption = 'WHERE NOT EQUAL')
```

<> is syntactically the same as !=

The only row where the store_id <> 2 is row 1 and only row 1 is returned.

13.4.4.6 12 WHERE OR condition

```
rs <-
  DBI::dbGetQuery(
    con,
    "
    select * from store where manager_staff_id = 1 or store_id < 3;
    ")
kable(rs,caption = 'WHERE OR condition')
```

The first condition manager_staff_id = 1 returns a single row and the second condition store_id < 3 returns t

Following table is modified from <http://www.tutorialspoint.com/sql/sql-operators>

SQL Comparison Operators

Table 13.17: WHERE OR condition

store_id	manager_staff_id	address_id	last_update
1	1	1	2006-02-15 09:57:12
2	2	2	2006-02-15 09:57:12

Operator	Description	example
=	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(a = b) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(a != b) is true.
<>	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(a <> b) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(a > b) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(a < b) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(a >= b) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(a <= b) is true.
!<	Checks if the value of left operand is not less than the value of right operand, if yes then condition becomes true.	(a !< b) is false.
!>	Checks if the value of left operand is not greater than the value of right operand, if yes then condition becomes true.	(a !> b) is true.

Operator	Description
ALL	The ALL operator is used to compare a value to all values in another value set.
AND	The AND operator allows the existence of multiple conditions in an SQL statement's WHERE clause.
ANY	The ANY operator is used to compare a value to any applicable value in the list as per the condition.
BETWEEN	The BETWEEN operator is used to search for values that are within a set of values, given the minimum value and the maximum value.
EXISTS	The EXISTS operator is used to search for the presence of a row in a specified table that meets a certain criterion.
IN	The IN operator is used to compare a value to a list of literal values that have been specified.
LIKE	The LIKE operator is used to compare a value to similar values using wildcard operators.
NOT	The NOT operator reverses the meaning of the logical operator with which it is used. Eg: NOT EXISTS, NOT BETWEEN, NOT IN, etc. This is a negate operator.
OR	The OR operator is used to combine multiple conditions in an SQL statement's WHERE clause.
IS	The NULL operator is used to compare a value with a NULL value.
NULL	
UNIQUE	The UNIQUE operator searches every row of a specified table for uniqueness (no duplicates).

<https://pgexercisess.com/questions/basic>

TO-DO's

1. inline tables
2. correlated subqueries

13.5 Paradigm Shift from R-Dplyr to SQL

Paraphrasing what some have said with an R dplyr background and no SQL experience, "It is like working

1. The SQL SELECT statement begins at the end, the SELECT clause, and drills backwards, loosely speaking, to derive the desired result set.
2. SQL SELECT statements are an all or nothing proposition. One gets nothing if there is any kind of syntax error.
3. SQL SELECT result sets can be quite opaque. The WHERE clause can be very dense and difficult to trace through. It is rarely ever linear in nature.
4. Validating all the permutations in the where clause can be tough and tedious.

13.5.1 Big bang versus piped incremental steps.

1. Dplyr starts with one or more sources joined together in a conceptually similar way that SQL joins sources.
2. The pipe and filter() function breaks down the filter conditions into small manageable logical steps. This makes it much easier to understand what is happening in the derivation of the final tibble. Adding tees through out the pipe line gives one full trace back of all the data transformations at every pipe.

Helpful tidyverse functions that output tibbles: tbl_module function in <https://github.com/nhemerson/tibbleColumns> package;

Mental picture: SQL approach: Imagine a data lake named Niagara Falls and drinking from it without drowning. R-Dplyr approach: Imagine a restaurant at the bottom of the Niagara Falls data lake and having a refreshing drink out of the water faucet.

13.5.2 SQL Execution Order

The table below is derived from this site. <https://www.periscopedata.com/blog/sql-query-order-of-operations>
It shows what goes on under the hood SQL SELECT hood.

SEQ	SQL	Function	Dplyr
1	WITH	Common Table expression, CTE, one or more datasets/tables used FROM clause.	.data parameter in dplyr functions
2	FROM	Choose and join tables to get base data	.data parameter in dplyr functions
3	ON	Choose and join tables to get base data	dplyr join family of functions
4	JOIN	Choose and join tables to get base data	dplyr join family of functions
5	WHERE	filters the base data	dplyr filter()
6	GROUP BY	aggregates the base data	dplyr group_by family of functions
7	WITH CUBE/ROLLUP	aggregates the base data	is this part of the dplyr grammar
8	HAVING	filters aggregated data	dplyr filter()
9	SELECT	Returns final data set	dplyr select()
10	DISTINCT	Dedupe the final data set	dplyr distinct()
11	ORDER BY	Sorts the final data set	arrange()
12	TOP/LIMIT	Limits the number of rows in data set	
13	OFFSET/FETCH	Limits the number of rows in data set	

The SEQ column shows the standard order of SQL execution. One take away for this tutorial is that the

SELECT clause actually executes late in the process, even though it is the first clause in the entire SELECT statement. A second take away is that SQL execution order, or tweaked order, plays a critical role in SQL query tuning.

6. SQL for View table dependencies.
7. Add cartesian join exercise.

Chapter 14

Getting metadata about and from the database (21)

This chapter demonstrates:

- What kind of data about the database is contained in a dbms
- Several methods for obtaining metadata from the dbms

The following packages are used in this chapter:

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

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

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

Connect to the database:

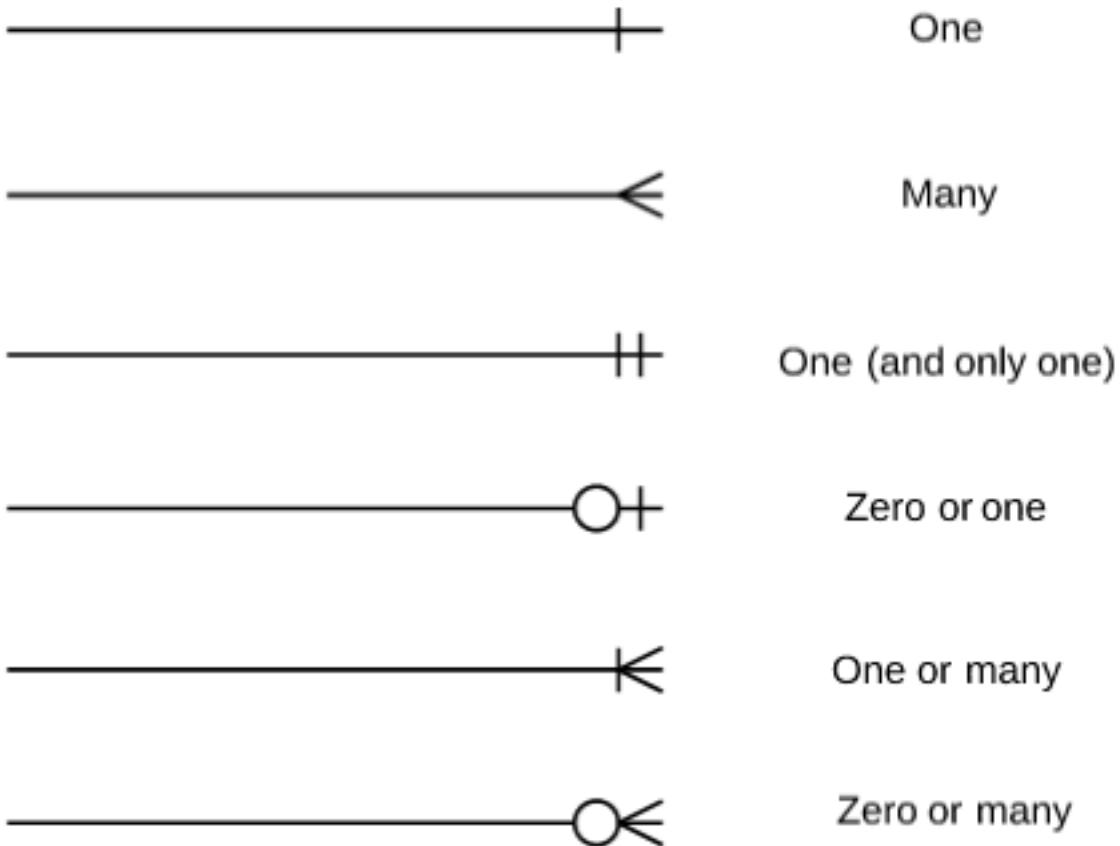
```
con <- sp_get_postgres_connection(
  user = Sys.getenv("DEFAULT_POSTGRES_USER_NAME"),
  password = Sys.getenv("DEFAULT_POSTGRES_PASSWORD"),
  dbname = "dvdrental",
  seconds_to_test = 10
)
```

14.1 Database contents and structure

After just looking at the data you seek, it might be worthwhile stepping back and looking at the big picture.

14.1.1 Database structure

For large or complex databases you need to use both the available documentation for your database (e.g., the dvdrental database) and the other empirical tools that are available. For example it's worth learning to interpret the symbols in an Entity Relationship Diagram:



The `information_schema` is a trove of information *about* the database. Its format is more or less consistent across the different SQL implementations that are available. Here we explore some of what's available using several different methods. Postgres stores a lot of metadata.

14.1.2 Contents of the `information_schema`

For this chapter R needs the `dbplyr` package to access alternate schemas. A schema is an object that contains one or more tables. Most often there will be a default schema, but to access the metadata, you need to explicitly specify which schema contains the data you want.

14.1.3 What tables are in the database?

The simplest way to get a list of tables is with

```
table_list <- DBI::dbListTables(con)
kable(table_list)
```

x
actor_info
customer_list
film_list
nicer_but_slower_film_list
sales_by_film_category
staff
sales_by_store
staff_list
category
film_category
country
actor
language
inventory
payment
rental
city
store
film
address
film_actor
customer
smy_film

14.1.4 Digging into the information_schema

We usually need more detail than just a list of tables. Most SQL databases have an `information_schema` that has a standard structure to describe and control the database.

The `information_schema` is in a different schema from the default, so to connect to the `tables` table in the `information_schema` we connect to the database in a different way:

```
table_info_schema_table <- tbl(con, dbplyr::in_schema("information_schema", "tables"))
```

The `information_schema` is large and complex and contains 211 tables. So it's easy to get lost in it.

This query retrieves a list of the tables in the database that includes additional detail, not just the name of the table.

```
table_info <- table_info_schema_table %>%
  filter(table_schema == "public") %>%
  select(table_catalog, table_schema, table_name, table_type) %>%
  arrange(table_type, table_name) %>%
  collect()

kable(table_info)
```

table_catalog	table_schema	table_name	table_type
dvdrental	public	actor	BASE TABLE
dvdrental	public	address	BASE TABLE
dvdrental	public	category	BASE TABLE
dvdrental	public	city	BASE TABLE
dvdrental	public	country	BASE TABLE
dvdrental	public	customer	BASE TABLE
dvdrental	public	film	BASE TABLE
dvdrental	public	film_actor	BASE TABLE
dvdrental	public	film_category	BASE TABLE
dvdrental	public	inventory	BASE TABLE
dvdrental	public	language	BASE TABLE
dvdrental	public	payment	BASE TABLE
dvdrental	public	rental	BASE TABLE
dvdrental	public	smy_film	BASE TABLE
dvdrental	public	staff	BASE TABLE
dvdrental	public	store	BASE TABLE
dvdrental	public	actor_info	VIEW
dvdrental	public	customer_list	VIEW
dvdrental	public	film_list	VIEW
dvdrental	public	nicer_but_slower_film_list	VIEW
dvdrental	public	sales_by_film_category	VIEW
dvdrental	public	sales_by_store	VIEW
dvdrental	public	staff_list	VIEW

In this context `table_catalog` is synonymous with `database`.

Notice that *VIEWS* are composites made up of one or more *BASE TABLES*.

The SQL world has its own terminology. For example `rs` is shorthand for `result set`. That's equivalent to using `df` for a `data frame`. The following SQL query returns the same information as the previous one.

```
rs <- dbGetQuery(
  con,
  "select table_catalog, table_schema, table_name, table_type
  from information_schema.tables
  where table_schema not in ('pg_catalog','information_schema')
  order by table_type, table_name
  ;"
)
kable(rs)
```


table_catalog	table_schema	table_name	table_type
dvdrental	public	actor	BASE TABLE
dvdrental	public	address	BASE TABLE
dvdrental	public	category	BASE TABLE
dvdrental	public	city	BASE TABLE
dvdrental	public	country	BASE TABLE
dvdrental	public	customer	BASE TABLE
dvdrental	public	film	BASE TABLE
dvdrental	public	film_actor	BASE TABLE
dvdrental	public	film_category	BASE TABLE
dvdrental	public	inventory	BASE TABLE
dvdrental	public	language	BASE TABLE
dvdrental	public	payment	BASE TABLE
dvdrental	public	rental	BASE TABLE
dvdrental	public	smv_film	BASE TABLE
dvdrental	public	staff	BASE TABLE
dvdrental	public	store	BASE TABLE
dvdrental	public	actor_info	VIEW
dvdrental	public	customer_list	VIEW
dvdrental	public	film_list	VIEW
dvdrental	public	nicer_but_slower_film_list	VIEW
dvdrental	public	sales_by_film_category	VIEW
dvdrental	public	sales_by_store	VIEW
dvdrental	public	staff_list	VIEW

14.2 What columns do those tables contain?

Of course, the DBI package has a `dbListFields` function that provides the simplest way to get the minimum, a list of column names:

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

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

But the `information_schema` has a lot more useful information that we can use.

```
columns_info_schema_table <- tbl(con, dbplyr::in_schema("information_schema", "columns"))
```

Since the `information_schema` contains 1868 columns, we are narrowing our focus to just one table. This query retrieves more information about the `rental` table:

```
columns_info_schema_info <- columns_info_schema_table %>%
  filter(table_schema == "public") %>%
  select(
    table_catalog, table_schema, table_name, column_name, data_type, ordinal_position,
    character_maximum_length, column_default, numeric_precision, numeric_precision_radix
  ) %>%
  collect(n = Inf) %>%
  mutate(data_type = case_when(
    data_type == "character_varying" ~ paste0(data_type, " (", character_maximum_length, ")"),
    data_type == "real" ~ paste0(data_type, " (", numeric_precision, ",", numeric_precision_radix, ")"),
    TRUE ~ data_type
  )) %>%
  filter(table_name == "rental") %>%
```

```

select(-table_schema, -numeric_precision, -numeric_precision_radix)

glimpse(columns_info_schema_info)

## Observations: 7
## Variables: 7
## $ table_catalog      <chr> "dvdrental", "dvdrental", "dvdrental"...
## $ table_name         <chr> "rental", "rental", "rental", "rental..."
## $ column_name        <chr> "rental_id", "rental_date", "inventor..."
## $ data_type          <chr> "integer", "timestamp without time zo..."
## $ ordinal_position   <int> 1, 2, 3, 4, 5, 6, 7
## $ character_maximum_length <int> NA, NA, NA, NA, NA, NA, NA
## $ column_default     <chr> "nextval('rental_rental_id_seq'::regc..."

kable(columns_info_schema_info)

```

table_catalog	table_name	column_name	data_type	ordinal_position	character_maximum_length
dvdrental	rental	rental_id	integer	1	
dvdrental	rental	rental_date	timestamp without time zone	2	
dvdrental	rental	inventory_id	integer	3	
dvdrental	rental	customer_id	smallint	4	
dvdrental	rental	return_date	timestamp without time zone	5	
dvdrental	rental	staff_id	smallint	6	
dvdrental	rental	last_update	timestamp without time zone	7	

14.2.1 What is the difference between a VIEW and a BASE TABLE?

The BASE TABLE has the underlying data in the database

```

table_info_schema_table %>%
  filter(table_schema == "public" & table_type == "BASE TABLE") %>%
  select(table_name, table_type) %>%
  left_join(columns_info_schema_table, by = c("table_name" = "table_name")) %>%
  select(
    table_type, table_name, column_name, data_type, ordinal_position,
    column_default
  ) %>%
  collect(n = Inf) %>%
  filter(str_detect(table_name, "cust")) %>%
  kable()

```

table_type	table_name	column_name	data_type	ordinal_position	column_default
BASE TABLE	customer	store_id	smallint	2	NA
BASE TABLE	customer	first_name	character varying	3	NA
BASE TABLE	customer	last_name	character varying	4	NA
BASE TABLE	customer	email	character varying	5	NA
BASE TABLE	customer	address_id	smallint	6	NA
BASE TABLE	customer	active	integer	10	NA
BASE TABLE	customer	customer_id	integer	1	nextval('customer_cu
BASE TABLE	customer	activebool	boolean	7	true
BASE TABLE	customer	create_date	date	8	('now'::text)::date
BASE TABLE	customer	last_update	timestamp without time zone	9	now()

Probably should explore how the VIEW is made up of data from BASE TABLES.

```
table_info_schema_table %>%
  filter(table_schema == "public" & table_type == "VIEW") %>%
  select(table_name, table_type) %>%
  left_join(columns_info_schema_table, by = c("table_name" = "table_name")) %>%
  select(
    table_type, table_name, column_name, data_type, ordinal_position,
    column_default
  ) %>%
  collect(n = Inf) %>%
  filter(str_detect(table_name, "cust")) %>%
  kable()
```

table_type	table_name	column_name	data_type	ordinal_position	column_default
VIEW	customer_list	id	integer	1	NA
VIEW	customer_list	name	text	2	NA
VIEW	customer_list	address	character varying	3	NA
VIEW	customer_list	zip code	character varying	4	NA
VIEW	customer_list	phone	character varying	5	NA
VIEW	customer_list	city	character varying	6	NA
VIEW	customer_list	country	character varying	7	NA
VIEW	customer_list	notes	text	8	NA
VIEW	customer_list	sid	smallint	9	NA

14.2.2 What data types are found in the database?

```
columns_info_schema_info %>% count(data_type)
```

```
## # A tibble: 3 x 2
##   data_type      n
##   <chr>      <int>
## 1 integer        2
## 2 smallint       2
## 3 timestamp without time zone  3
```

14.3 Characterizing how things are named

Names are the handle for accessing the data. Tables and columns may or may not be named consistently or in a way that makes sense to you. You should look at these names *as data*.

14.3.1 Counting columns and name reuse

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

```
public_tables <- columns_info_schema_table %>%
  filter(table_schema == "public") %>%
  collect()

public_tables %>%
  count(table_name, sort = TRUE) %>% head(n = 15) %>%
  kable()
```

table_name	n
film	13
smy_film	13
staff	11
customer	10
customer_list	9
address	8
film_list	8
nicer_but_slower_film_list	8
staff_list	8
rental	7
payment	6
actor	4
actor_info	4
city	4
inventory	4

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

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

```
## # A tibble: 40 x 2
##   column_name      n
##   <chr>         <int>
## 1 last_update     15
## 2 film_id         5
## 3 address_id      4
## 4 description     4
## 5 first_name      4
## 6 last_name       4
## 7 length         4
## 8 name            4
## 9 rating          4
## 10 store_id       4
## # ... with 30 more rows
```

How many column names are unique?

```
public_tables %>% count(column_name) %>% filter(n == 1) %>% count()
```

```
## # A tibble: 1 x 1
##       nn
##   <int>
## 1     18
```

14.4 Database keys

14.4.1 Direct SQL

How do we use this output? Could it be generated by dplyr?

```
rs <- dbGetQuery(
  con,
  "
--SELECT conrelid::regclass as table_from
```

```

select table_catalog||'.'||table_schema||'.'||table_name table_name
, conname, pg_catalog.pg_get_constraintdef(r.oid, true) as condef
FROM information_schema.columns c,pg_catalog.pg_constraint r
WHERE 1 = 1 --r.conrelid = '16485'
      AND r.contype in ('f','p') ORDER BY 1
;"
)
glimpse(rs)

```

```

## Observations: 61,644
## Variables: 3
## $ table_name <chr> "dvdrental.information_schema.administrable_role_au...
## $ conname <chr> "actor_pkey", "actor_pkey", "actor_pkey", "country_...
## $ condef <chr> "PRIMARY KEY (actor_id)", "PRIMARY KEY (actor_id)",...

```

```
kable(head(rs))
```

table_name	conname	condef
dvdrental.information_schema.administrable_role_authorizations	actor_pkey	PRIMARY KEY (actor_id)
dvdrental.information_schema.administrable_role_authorizations	actor_pkey	PRIMARY KEY (actor_id)
dvdrental.information_schema.administrable_role_authorizations	actor_pkey	PRIMARY KEY (actor_id)
dvdrental.information_schema.administrable_role_authorizations	country_pkey	PRIMARY KEY (country_id)
dvdrental.information_schema.administrable_role_authorizations	country_pkey	PRIMARY KEY (country_id)
dvdrental.information_schema.administrable_role_authorizations	country_pkey	PRIMARY KEY (country_id)

The following is more compact and looks more useful. What is the difference between the two?

```

rs <- dbGetQuery(
  con,
  "select conrelid::regclass as table_from
    ,c.conname
    ,pg_get_constraintdef(c.oid)
  from pg_constraint c
  join pg_namespace n on n.oid = c.connamespace
 where c.contype in ('f','p')
       and n.nspname = 'public'
 order by conrelid::regclass::text, contype DESC;
"
)
glimpse(rs)

```

```

## Observations: 33
## Variables: 3
## $ table_from <chr> "actor", "address", "address", "category"...
## $ conname <chr> "actor_pkey", "address_pkey", "fk_address..."
## $ pg_get_constraintdef <chr> "PRIMARY KEY (actor_id)", "PRIMARY KEY (a..."

```

```
kable(head(rs))
```

table_from	conname	pg_get_constraintdef
actor	actor_pkey	PRIMARY KEY (actor_id)
address	address_pkey	PRIMARY KEY (address_id)
address	fk_address_city	FOREIGN KEY (city_id) REFERENCES city(city_id)
category	category_pkey	PRIMARY KEY (category_id)
city	city_pkey	PRIMARY KEY (city_id)
city	fk_city	FOREIGN KEY (country_id) REFERENCES country(country_id)

```
dim(rs)[1]
```

```
## [1] 33
```

14.4.2 Database keys with dplyr

This query shows the primary and foreign keys in the database.

```
tables <- tbl(con, dbplyr::in_schema("information_schema", "tables"))
table_constraints <- tbl(con, dbplyr::in_schema("information_schema", "table_constraints"))
key_column_usage <- tbl(con, dbplyr::in_schema("information_schema", "key_column_usage"))
referential_constraints <- tbl(con, dbplyr::in_schema("information_schema", "referential_constraints"))
constraint_column_usage <- tbl(con, dbplyr::in_schema("information_schema", "constraint_column_usage"))

keys <- tables %>%
  left_join(table_constraints, by = c(
    "table_catalog" = "table_catalog",
    "table_schema" = "table_schema",
    "table_name" = "table_name"
  )) %>%
  # table_constraints %>%
  filter(constraint_type %in% c("FOREIGN KEY", "PRIMARY KEY")) %>%
  left_join(key_column_usage,
    by = c(
      "table_catalog" = "table_catalog",
      "constraint_catalog" = "constraint_catalog",
      "constraint_schema" = "constraint_schema",
      "table_name" = "table_name",
      "table_schema" = "table_schema",
      "constraint_name" = "constraint_name"
    )
  ) %>%
  # left_join(constraint_column_usage) %>% # does this table add anything useful?
  select(table_name, table_type, constraint_name, constraint_type, column_name, ordinal_position) %>%
  arrange(table_name) %>%
  collect()
glimpse(keys)
```

```
## Observations: 35
## Variables: 6
## $ table_name      <chr> "actor", "address", "address", "category", "c...
## $ table_type      <chr> "BASE TABLE", "BASE TABLE", "BASE TABLE", "BA...
## $ constraint_name <chr> "actor_pkey", "address_pkey", "fk_address_cit...
## $ constraint_type <chr> "PRIMARY KEY", "PRIMARY KEY", "FOREIGN KEY", ...
## $ column_name     <chr> "actor_id", "address_id", "city_id", "categor...
## $ ordinal_position <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, ...
```

```
kable(keys)
```

table_name	table_type	constraint_name	constraint_type	column_name	ordinal_pos
actor	BASE TABLE	actor_pkey	PRIMARY KEY	actor_id	
address	BASE TABLE	address_pkey	PRIMARY KEY	address_id	
address	BASE TABLE	fk_address_city	FOREIGN KEY	city_id	
category	BASE TABLE	category_pkey	PRIMARY KEY	category_id	
city	BASE TABLE	city_pkey	PRIMARY KEY	city_id	
city	BASE TABLE	fk_city	FOREIGN KEY	country_id	
country	BASE TABLE	country_pkey	PRIMARY KEY	country_id	
customer	BASE TABLE	customer_address_id_fkey	FOREIGN KEY	address_id	
customer	BASE TABLE	customer_pkey	PRIMARY KEY	customer_id	
film	BASE TABLE	film_language_id_fkey	FOREIGN KEY	language_id	
film	BASE TABLE	film_pkey	PRIMARY KEY	film_id	
film_actor	BASE TABLE	film_actor_actor_id_fkey	FOREIGN KEY	actor_id	
film_actor	BASE TABLE	film_actor_film_id_fkey	FOREIGN KEY	film_id	
film_actor	BASE TABLE	film_actor_pkey	PRIMARY KEY	actor_id	
film_actor	BASE TABLE	film_actor_pkey	PRIMARY KEY	film_id	
film_category	BASE TABLE	film_category_category_id_fkey	FOREIGN KEY	category_id	
film_category	BASE TABLE	film_category_film_id_fkey	FOREIGN KEY	film_id	
film_category	BASE TABLE	film_category_pkey	PRIMARY KEY	film_id	
film_category	BASE TABLE	film_category_pkey	PRIMARY KEY	category_id	
inventory	BASE TABLE	inventory_film_id_fkey	FOREIGN KEY	film_id	
inventory	BASE TABLE	inventory_pkey	PRIMARY KEY	inventory_id	
language	BASE TABLE	language_pkey	PRIMARY KEY	language_id	
payment	BASE TABLE	payment_customer_id_fkey	FOREIGN KEY	customer_id	
payment	BASE TABLE	payment_pkey	PRIMARY KEY	payment_id	
payment	BASE TABLE	payment_rental_id_fkey	FOREIGN KEY	rental_id	
payment	BASE TABLE	payment_staff_id_fkey	FOREIGN KEY	staff_id	
rental	BASE TABLE	rental_customer_id_fkey	FOREIGN KEY	customer_id	
rental	BASE TABLE	rental_inventory_id_fkey	FOREIGN KEY	inventory_id	
rental	BASE TABLE	rental_pkey	PRIMARY KEY	rental_id	
rental	BASE TABLE	rental_staff_id_key	FOREIGN KEY	staff_id	
staff	BASE TABLE	staff_address_id_fkey	FOREIGN KEY	address_id	
staff	BASE TABLE	staff_pkey	PRIMARY KEY	staff_id	
store	BASE TABLE	store_address_id_fkey	FOREIGN KEY	address_id	
store	BASE TABLE	store_manager_staff_id_fkey	FOREIGN KEY	manager_staff_id	
store	BASE TABLE	store_pkey	PRIMARY KEY	store_id	

What do we learn from the following query? How is it useful?

```
rs <- dbGetQuery(
  con,
  "SELECT r.*,
  pg_catalog.pg_get_constraintdef(r.oid, true) as condef
  FROM pg_catalog.pg_constraint r
  WHERE 1=1 --r.conrelid = '16485' AND r.contype = 'f' ORDER BY 1;
  "
)

head(rs)
```

```
##               conname connamespace contype condeferrable
## 1 cardinal_number_domain_check      12703      c      FALSE
## 2               yes_or_no_check      12703      c      FALSE
## 3                  year_check        2200      c      FALSE
```

```

## 4          actor_pkey          2200      p      FALSE
## 5          address_pkey        2200      p      FALSE
## 6          category_pkey       2200      p      FALSE
##   condeferred convalidated conrelid contypid conindid confrelid
## 1          FALSE          TRUE      0    12716      0      0
## 2          FALSE          TRUE      0    12724      0      0
## 3          FALSE          TRUE      0    16397      0      0
## 4          FALSE          TRUE    16420      0    16555      0
## 5          FALSE          TRUE    16461      0    16557      0
## 6          FALSE          TRUE    16427      0    16559      0
##   confupdtype confdeltype confmatchtype conislocal coninhcount
## 1                                TRUE      0
## 2                                TRUE      0
## 3                                TRUE      0
## 4                                TRUE      0
## 5                                TRUE      0
## 6                                TRUE      0
##   connoinherit conkey  confkey  confreqop  conppeqop  confreqop  conexclp
## 1          FALSE  <NA>   <NA>    <NA>    <NA>    <NA>    <NA>
## 2          FALSE  <NA>   <NA>    <NA>    <NA>    <NA>    <NA>
## 3          FALSE  <NA>   <NA>    <NA>    <NA>    <NA>    <NA>
## 4           TRUE   {1}   <NA>    <NA>    <NA>    <NA>    <NA>
## 5           TRUE   {1}   <NA>    <NA>    <NA>    <NA>    <NA>
## 6           TRUE   {1}   <NA>    <NA>    <NA>    <NA>    <NA>
##
## 1
## 2 {SCALARARRAYOPEXPR :opno 98 :opfuncid 67 :useOr true :inputcollid 100 :args ({RELABELTYPE :arg {COERCET
## 3                                     {BOOLEXPRESS :boolop and :args
## 4
## 5
## 6
##
##                                     consrc
## 1                                     (VALUE >= 0)
## 2 ((VALUE)::text = ANY ((ARRAY['YES']::character varying, 'NO']::character varying))::text[]))
## 3                                     ((VALUE >= 1901) AND (VALUE <= 2155))
## 4                                     <NA>
## 5                                     <NA>
## 6                                     <NA>
##
##                                     condef
## 1                                     CHECK (VALUE >= 0)
## 2 CHECK (VALUE::text = ANY (ARRAY['YES']::character varying, 'NO']::character varying)::text[]))
## 3                                     CHECK (VALUE >= 1901 AND VALUE <= 2155)
## 4                                     PRIMARY KEY (actor_id)
## 5                                     PRIMARY KEY (address_id)
## 6                                     PRIMARY KEY (category_id)

```

14.5 Creating your own data dictionary

If you are going to work with a database for an extended period it can be useful to create your own data dictionary. This can take the form of keeping detailed notes as well as extracting metadata from the dbms. Here is an illustration of the idea.


```

some_tables <- c("rental", "city", "store")

all_meta <- map_df(some_tables, sp_get_dbms_data_dictionary, con = con)

all_meta

## # A tibble: 15 x 11
##   table_name var_name var_type num_rows num_blank num_unique min   q_25
##   <chr>      <chr>   <chr>    <int>    <int>    <int> <chr> <chr>
## 1 rental    rental_~ integer  16044      0    16044 1     4013
## 2 rental    rental_~ double  16044      0    15815 2005~ 2005~
## 3 rental    invento~ integer  16044      0    4580 1     1154
## 4 rental    custome~ integer  16044      0    599 1     148
## 5 rental    return_~ double  16044    183    15836 2005~ 2005~
## 6 rental    staff_id integer  16044      0      2 1     1
## 7 rental    last_up~ double  16044      0      3 2006~ 2006~
## 8 city      city_id  integer   600      0    600 1     150
## 9 city      city     charact~  600      0    599 A Co~ Dzer~
## 10 city     country~ integer   600      0    109 1     28
## 11 city     last_up~ double   600      0      1 2006~ 2006~
## 12 store    store_id integer    2      0      2 1     1
## 13 store    manager~ integer    2      0      2 1     1
## 14 store    address~ integer    2      0      2 1     1
## 15 store    last_up~ double    2      0      1 2006~ 2006~
## # ... with 3 more variables: q_50 <chr>, q_75 <chr>, max <chr>

glimpse(all_meta)

```

```

## Observations: 15
## Variables: 11
## $ table_name <chr> "rental", "rental", "rental", "rental", "rental", "...
## $ var_name <chr> "rental_id", "rental_date", "inventory_id", "custom...
## $ var_type <chr> "integer", "double", "integer", "integer", "double"...
## $ num_rows <int> 16044, 16044, 16044, 16044, 16044, 16044, 16044, 60...
## $ num_blank <int> 0, 0, 0, 0, 183, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
## $ num_unique <int> 16044, 15815, 4580, 599, 15836, 2, 3, 600, 599, 109...
## $ min <chr> "1", "2005-05-24 22:53:30", "1", "1", "2005-05-25 2...
## $ q_25 <chr> "4013", "2005-07-07 00:58:00", "1154", "148", "2005...
## $ q_50 <chr> "8025", "2005-07-28 16:03:27", "2291", "296", "2005...
## $ q_75 <chr> "12037", "2005-08-17 21:13:35", "3433", "446", "200...
## $ max <chr> "16049", "2006-02-14 15:16:03", "4581", "599", "200...

kable(head(all_meta))

```

table_name	var_name	var_type	num_rows	num_blank	num_unique	min	q_25
rental	rental_id	integer	16044	0	16044	1	4013
rental	rental_date	double	16044	0	15815	2005-05-24 22:53:30	2005-07-07 00:
rental	inventory_id	integer	16044	0	4580	1	1154
rental	customer_id	integer	16044	0	599	1	148
rental	return_date	double	16044	183	15836	2005-05-25 23:55:21	2005-07-10 15:
rental	staff_id	integer	16044	0	2	1	1

14.6 Save your work!

The work you do to understand the structure and contents of a database can be useful for others (including future-you). So at the end of a session, you might look at all the data frames you want to save. Consider saving them in a form where you can add notes at the appropriate level (as in a Google Doc representing table or columns that you annotate over time).

```
ls()
```

```
## [1] "all_meta"                "columns_info_schema_info"
## [3] "columns_info_schema_table" "con"
## [5] "constraint_column_usage"  "cranex"
## [7] "key_column_usage"         "keys"
## [9] "public_tables"            "referential_constraints"
## [11] "rs"                       "some_tables"
## [13] "table_constraints"         "table_info"
## [15] "table_info_schema_table"  "table_list"
## [17] "tables"
```

Chapter 15

Drilling into your DBMS environment (22)

This chapter investigates:

- Elements of the database environment
- Differences between a database, a schema, and other objects
- Exercises

The following packages are used in this chapter:

```
# These packages are called in almost every chapter of the book:
library(tidyverse)
library(DBI)
library(RPostgres)
require(knitr)
library(dbplyr)
library(sqlpetr)

display_rows <- 15 # as a default, show 15 rows
```

Start up the docker-pet container

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

Now connect to the dvdrental database with R

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

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

15.1 Which database?

Your DBA will create your user accounts and privileges for the database(s) that you can access.

One of the challenges when working with a database(s) is finding where your data actually resides. Your best resources will be one or more subject matter experts, SME, and your DBA. Your data may actually reside in multiple databases, e.g., a detail and summary databases. In our tutorial, we focus on the one database, `dvdrental`. Database names usually reflect something about the data that they contain.

Your laptop is a server for the Docker Postgres databases. A database is a collection of files that Postgres manages in the background.

15.2 How many databases reside in the Docker Container?

```
rs <-
  DBI::dbGetQuery(
    con,
    "SELECT 'DB Names in Docker' showing
      ,datname DB
    FROM pg_database
    WHERE datistemplate = false;
  "
  )
kable(rs)
```

showing	db
DB Names in Docker	postgres
DB Names in Docker	dvdrental

Which databases are available?

Modify the connection call to connect to the `postgres` database.

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

con
```

```
## [1] "There is no connection"
```

```
if (con != "There is no connection") {
  dbDisconnect(con)
}
```

```
# Answer: con <PgConnection> postgres@localhost:5432
```

```
# Reconnect to dvdrental
```

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

con
```

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

Note that the two `Sys.getenv` function calls work in this tutorial because both the user and password are available in both databases. This is a common practice in organizations that have implemented single sign on across their organization.

Gotcha:

If one has data in multiple databases or multiple environments, Development, Integration, and Production, it is very easy to connect to the wrong database in the wrong environment. Always double check your connection information when logging in and before performing any inserts, updates, or deletes against the database.

The following code block should be used to reduce propagating the above gotcha. `Current_database()`, `CURRENT_DATE` or `CURRENT_TIMESTAMP`, and 'result set' are the most useful and last three not so much. Instead of the host IP address having the actual hostname would be a nice addition.

```
rs1 <-
  DBI::dbGetQuery(
    con,
    "SELECT current_database() DB
      ,CURRENT_DATE
      ,CURRENT_TIMESTAMP
      ,'result set description' showing
      ,session_user
      ,inet_server_addr() host
      ,inet_server_port() port
    "
  )
kable(rs1)
```

db	current_date	current_timestamp	showing	session_user	host	port
dvdrental	2018-12-06	2018-12-06 15:16:34	result set description	postgres	172.17.0.2	5432

Since we will only be working in the `dvdrental` database in this tutorial and reduce the number of output columns shown, only the 'result set description' will be used.

15.3 Which Schema?

In the code block below, we look at the `information_schema.table` which contains information about all the schemas and table/views within our `dvdrental` database. Databases can have one or more schemas, containers that hold tables or views. Schemas partition the database into big logical blocks of related data. Schema names usually reflect an application or logically related datasets. Occasionally a DBA will set up a new schema and use a users name.

What schemas are in the `dvdrental` database? How many entries are in each schema?

```
## Database Schemas
#
rs1 <-
  DBI::dbGetQuery(
    con,
    "SELECT 'DB Schemas' showing,t.table_catalog DB,t.table_schema,COUNT(*) tbl_vws
      FROM information_schema.tables t
      GROUP BY t.table_catalog,t.table_schema
    "
  )
```


[illegible]

```
# Use the results from above to pull interesting columns from just the information_schema
rs4 <- DBI::dbGetQuery(con, "SELECT '4. information_schema.tables' showing
                             , 'your code goes here'
                             FROM information_schema.tables t
                             where 'your code goes here' = 'your code goes here'
                             ")
head(rs4, display rows)
```

```
##                                showing                                ?column?
## 1  4. information_schema.tables your code goes here
## 2  4. information_schema.tables your code goes here
## 3  4. information_schema.tables your code goes here
## 4  4. information_schema.tables your code goes here
## 5  4. information_schema.tables your code goes here
## 6  4. information_schema.tables your code goes here
## 7  4. information_schema.tables your code goes here
## 8  4. information_schema.tables your code goes here
## 9  4. information_schema.tables your code goes here
## 10 4. information_schema.tables your code goes here
## 11 4. information_schema.tables your code goes here
## 12 4. information_schema.tables your code goes here
## 13 4. information_schema.tables your code goes here
## 14 4. information_schema.tables your code goes here
## 15 4. information_schema.tables your code goes here
```

[illegible]

showing	db_info	table_name	table_type
7. information_schema.tables	dvdrental.information_schema	collations	VIEW
7. information_schema.tables	dvdrental.information_schema	collation_character_set_applicability	VIEW
7. information_schema.tables	dvdrental.information_schema	column_domain_usage	VIEW
7. information_schema.tables	dvdrental.information_schema	column_privileges	VIEW
7. information_schema.tables	dvdrental.information_schema	column_udt_usage	VIEW
7. information_schema.tables	dvdrental.information_schema	columns	VIEW
7. information_schema.tables	dvdrental.information_schema	constraint_column_usage	VIEW
7. information_schema.tables	dvdrental.information_schema	key_column_usage	VIEW
7. information_schema.tables	dvdrental.information_schema	role_column_grants	VIEW
7. information_schema.tables	dvdrental.information_schema	table_constraints	VIEW
7. information_schema.tables	dvdrental.information_schema	table_privileges	VIEW
7. information_schema.tables	dvdrental.information_schema	tables	VIEW
7. information_schema.tables	dvdrental.information_schema	triggered_update_columns	VIEW
7. information_schema.tables	dvdrental.information_schema	view_column_usage	VIEW
7. information_schema.tables	dvdrental.information_schema	_pg_foreign_table_columns	VIEW

```
rs8 <- DBI::dbGetQuery(con, "SELECT '8. information_schema.tables' showing
                             ,table_catalog||'.'||table_schema db_info, table_name, table_type
                             FROM information_schema.tables t
                             where table_schema = 'information_schema'
                             and table_type = 'VIEW'
                             and table_name like 'table%' OR table_name like '%col%'
                             ")
kable(head(rs8, display_rows))
```

showing	db_info	table_name	table_type
8. information_schema.tables	dvdrental.information_schema	column_options	VIEW
8. information_schema.tables	dvdrental.information_schema	_pg_foreign_table_columns	VIEW
8. information_schema.tables	dvdrental.information_schema	view_column_usage	VIEW
8. information_schema.tables	dvdrental.information_schema	triggered_update_columns	VIEW
8. information_schema.tables	dvdrental.information_schema	tables	VIEW
8. information_schema.tables	dvdrental.information_schema	table_privileges	VIEW
8. information_schema.tables	dvdrental.information_schema	table_constraints	VIEW
8. information_schema.tables	dvdrental.information_schema	role_column_grants	VIEW
8. information_schema.tables	dvdrental.information_schema	key_column_usage	VIEW
8. information_schema.tables	dvdrental.information_schema	constraint_column_usage	VIEW
8. information_schema.tables	dvdrental.information_schema	columns	VIEW
8. information_schema.tables	dvdrental.information_schema	column_udt_usage	VIEW
8. information_schema.tables	dvdrental.information_schema	column_privileges	VIEW
8. information_schema.tables	dvdrental.information_schema	column_domain_usage	VIEW
8. information_schema.tables	dvdrental.information_schema	collation_character_set_applicability	VIEW

Operator/Element	Associativity	Description
.	left	table/column name separator
::	left	PostgreSQL-style typecast
[]	left	array element selection
-	right	unary minus
^	left	exponentiation
/ %	left	multiplication, division, modulo
+ -	left	addition, subtraction
IS		IS TRUE, IS FALSE, IS UNKNOWN, IS NULL
ISNULL		test for null
NOTNULL		test for not null

Operator/Element	Associativity	Description
(any other)	left	all other native and user-defined operators
IN		set membership
BETWEEN		range containment
OVERLAPS		time interval overlap
LIKE ILIKE SIMILAR		string pattern matching
< >		less than, greater than
=	right	equality, assignment
NOT	right	logical negation
AND	left	logical conjunction
OR	left	logical disjunction

```
rs1 <- DBI::dbGetQuery(con, "SELECT t.table_catalog DB ,t.table_schema
                             ,t.table_name,t.table_type
                             FROM information_schema.tables t")

rs2 <- DBI::dbGetQuery(con, "SELECT t.table_catalog DB ,t.table_schema
                             ,t.table_type,COUNT(*) tbls
                             FROM information_schema.tables t
                             group by t.table_catalog ,t.table_schema
                             ,t.table_type
                             ")

rs3 <- DBI::dbGetQuery(con, "SELECT distinct t.table_catalog DB ,t.table_schema
                             ,t.table_type tbls
                             FROM information_schema.tables t
                             ")

# kable(head(rs1 %>% arrange (table_name)))
# View(rs1)
# View(rs2)
# View(rs3)
kable(head(rs1))
```

db	table_schema	table_name	table_type
dvdrental	public	actor_info	VIEW
dvdrental	public	customer_list	VIEW
dvdrental	public	film_list	VIEW
dvdrental	public	nicer_but_slower_film_list	VIEW
dvdrental	public	sales_by_film_category	VIEW
dvdrental	public	staff	BASE TABLE

```
kable(head(rs2))
```

db	table_schema	table_type	tbls
dvdrental	information_schema	BASE TABLE	7
dvdrental	information_schema	VIEW	60
dvdrental	pg_catalog	BASE TABLE	62
dvdrental	public	BASE TABLE	16
dvdrental	public	VIEW	7
dvdrental	pg_catalog	VIEW	59

```
kable(head(rs3))
```

db	table_schema	tbls
dvdrental	information_schema	BASE TABLE
dvdrental	information_schema	VIEW
dvdrental	pg_catalog	BASE TABLE
dvdrental	public	BASE TABLE
dvdrental	public	VIEW
dvdrental	pg_catalog	VIEW

www.dataquest.io/blog/postgres-internals

Comment on the practice of putting a comma at the beginning of a line in SQL code.

```
## Explain a `dplyr::join`
```

```
tbl_pk_fk_df <- DBI::dbGetQuery(
  con,
  "
SELECT --t.table_catalog,t.table_schema,
       c.table_name
       ,kcu.column_name
       ,c.constraint_name
       ,c.constraint_type
       ,coalesce(c2.table_name, '') ref_table
       ,coalesce(kcu2.column_name, '') ref_table_col
FROM   information_schema.tables t
LEFT JOIN information_schema.table_constraints c
      ON t.table_catalog = c.table_catalog
      AND t.table_schema = c.table_schema
      AND t.table_name = c.table_name
LEFT JOIN information_schema.key_column_usage kcu
      ON c.constraint_schema = kcu.constraint_schema
      AND c.constraint_name = kcu.constraint_name
LEFT JOIN information_schema.referential_constraints rc
      ON c.constraint_schema = rc.constraint_schema
      AND c.constraint_name = rc.constraint_name
LEFT JOIN information_schema.table_constraints c2
      ON rc.unique_constraint_schema = c2.constraint_schema
      AND rc.unique_constraint_name = c2.constraint_name
LEFT JOIN information_schema.key_column_usage kcu2
      ON c2.constraint_schema = kcu2.constraint_schema
      AND c2.constraint_name = kcu2.constraint_name
      AND kcu.ordinal_position = kcu2.ordinal_position
WHERE  c.constraint_type IN ('PRIMARY KEY', 'FOREIGN KEY')
      AND c.table_catalog = 'dvdrental'
      AND c.table_schema = 'public'
ORDER BY c.table_name;
"
)

# View(tbl_pk_fk_df)

tables_df <- tbl_pk_fk_df %>% distinct(table_name)
# View(tables_df)
```

```

library(DiagrammeR)

table_nodes_ndf <- create_node_df(
  n <- nrow(tables_df)
  , type <- "table"
  , label <- tables_df$table_name
  ,
  shape = "rectangle"
  , width = 1
  , height = .5
  , fontsize = 18
)

tbl_pk_fk_ids_df <- inner_join(tbl_pk_fk_df, table_nodes_ndf
  ,
  by = c("table_name" = "label")
  , suffix(c("st", "s"))
) %>%
  rename("src_tbl_id" = id) %>%
  left_join(table_nodes_ndf
    ,
    by = c("ref_table" = "label")
    , suffix(c("st", "t"))
  ) %>%
  rename("fk_tbl_id" = id)

tbl_fk_df <- tbl_pk_fk_ids_df %>% filter(constraint_type == "FOREIGN KEY")
tbl_pk_df <- tbl_pk_fk_ids_df %>% filter(constraint_type == "PRIMARY KEY")
# View(tbl_pk_fk_ids_df)
# View(tbl_fk_df)
# View(tbl_pk_df)
kable(head(tbl_fk_df))

```

table_name	column_name	constraint_name	constraint_type	ref_table	ref_table_col	src_tbl_id
address	city_id	fk_address_city	FOREIGN KEY	city	city_id	2
city	country_id	fk_city	FOREIGN KEY	country	country_id	4
customer	address_id	customer_address_id_fkey	FOREIGN KEY	address	address_id	6
film	language_id	film_language_id_fkey	FOREIGN KEY	language	language_id	7
film_actor	actor_id	film_actor_actor_id_fkey	FOREIGN KEY	actor	actor_id	8
film_actor	film_id	film_actor_film_id_fkey	FOREIGN KEY	film	film_id	8

```
kable(head(tbl_pk_df))
```

table_name	column_name	constraint_name	constraint_type	ref_table	ref_table_col	src_tbl_id	type.x
actor	actor_id	actor_pkey	PRIMARY KEY			1	table
address	address_id	address_pkey	PRIMARY KEY			2	table
category	category_id	category_pkey	PRIMARY KEY			3	table
city	city_id	city_pkey	PRIMARY KEY			4	table
country	country_id	country_pkey	PRIMARY KEY			5	table
customer	customer_id	customer_pkey	PRIMARY KEY			6	table

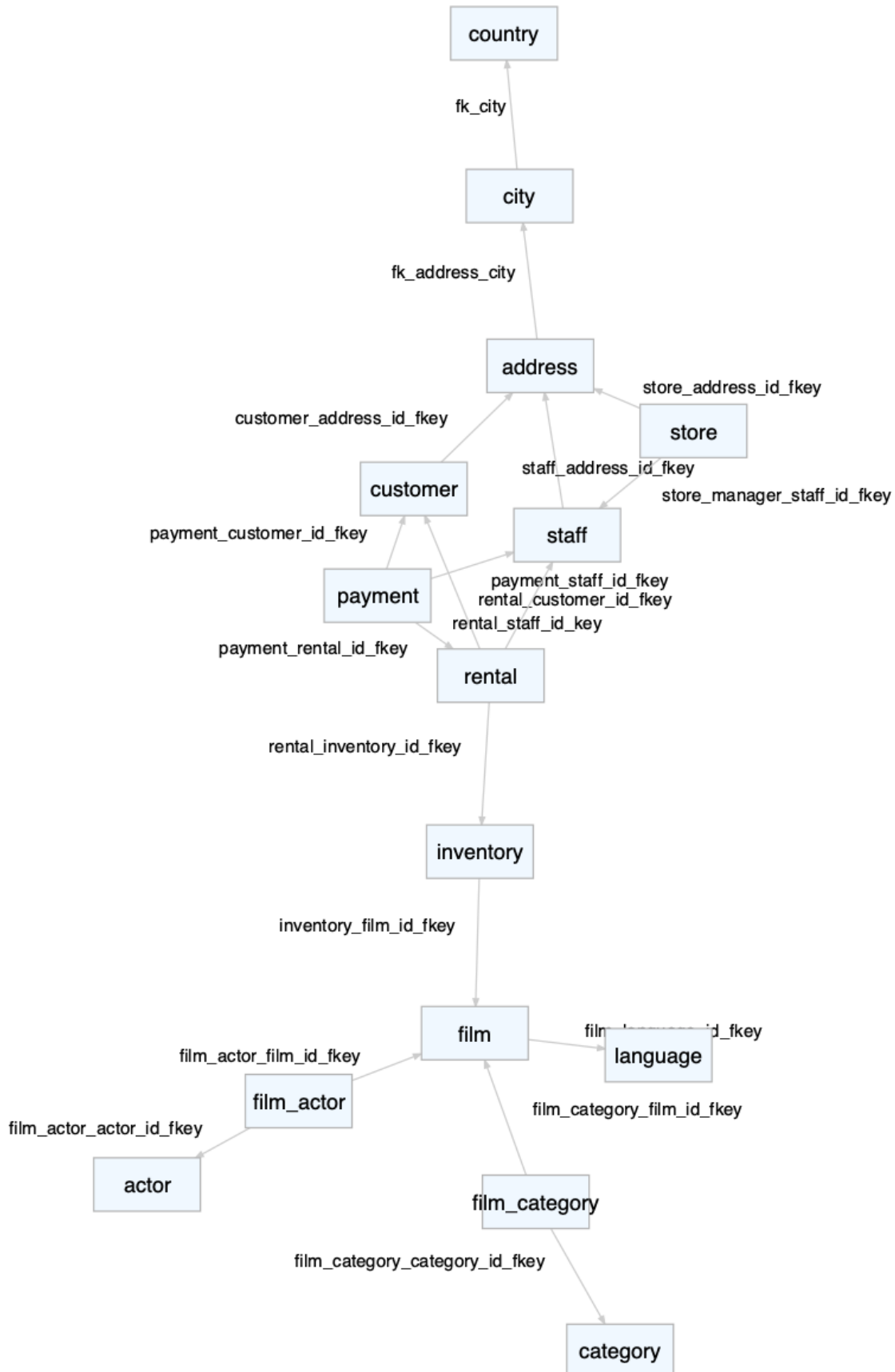
```
# Create an edge data frame, edf
```

```

fk_edf <-
  create_edge_df(

```

```
    from = tbl_fk_df$src_tbl_id,  
    to = tbl_fk_df$fk_tbl_id,  
    rel = "fk",  
    label = tbl_fk_df$constraint_name,  
    fontsize = 15  
  )  
  # View(fk_edf)  
  
fkgraph_widget <-  
  create_graph(  
    nodes_df = table_nodes_ndf,  
    edges_df = fk_edf,  
    graph_name = "Simple FK Graph"  
  ) %>%  
  render_graph()  
  
# export to image files  
fkgraph_file <- sqlpetr::sp_make_image_files(  
  fkgraph_widget,  
  "diagrams",  
  "fkgraph"  
)
```



```
dbDisconnect(con)  
# system2('docker', 'stop sql-pet')
```


Chapter 16

Explain queries (71)

This chapter demonstrates:

- How to investigate SQL query performance

These packages are called in almost every chapter of the book:

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

- examining dplyr queries (dplyr::show_query on the R side v EXPLAIN on the PostgreSQL side)

Start up the docker-pet container

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

now connect to the 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)
```

16.1 Performance considerations

Explain a `dplyr::join`

Explain the equivalent SQL join

```
rs1 <- DBI::dbGetQuery(con
  , "SELECT c.*
     FROM pg_catalog.pg_class c
     JOIN pg_catalog.pg_namespace n ON n.oid = c.relnamespace
     WHERE  n.nspname = 'public'
           AND  c.relname = 'cust_movies'
           AND  c.relkind = 'r'
```

```

        ;
    "
)

```

```
head(rs1)
```

```
## [1] relname          relnamespace      reltype
## [4] reloftype        reloverner        relam
## [7] relfilenode      reltablespace     relpages
## [10] reltuples        relallvisible     reltoastrelid
## [13] relhasindex      relisshared       relpersistence
## [16] relkind          relnatts          relchecks
## [19] relhasoids       relhaspkey        relhasrules
## [22] relhastriggers   relhassubclass    relrowsecurity
## [25] relforcerowsecurity relispopulated    relreplident
## [28] relispartition   relfrozenxid      relminmxid
## [31] relacl           reloptions        relpartbound
## <0 rows> (or 0-length row.names)
```

This came from 14-sql_pet-examples-part-b.Rmd

```
rs1 <- DBI::dbGetQuery(con,
    "explain select r.*
      from rental r
    ;"
)

```

```
head(rs1)
```

```
##                                     QUERY PLAN
## 1 Seq Scan on rental r (cost=0.00..310.44 rows=16044 width=36)
```

```
rs2 <- DBI::dbGetQuery(con,
    "explain select count(*) count
      from rental r
      left outer join payment p
        on r.rental_id = p.rental_id
      where p.rental_id is null
    ;"
)

```

```
head(rs2)
```

```
##                                     QUERY PLAN
## 1                                     Aggregate (cost=2086.78..2086.80 rows=1 width=8)
## 2                                     -> Merge Anti Join (cost=0.57..2066.73 rows=8022 width=0)
## 3                                     Merge Cond: (r.rental_id = p.rental_id)
## 4                                     -> Index Only Scan using rental_pkey on rental r (cost=0.29..1024.95 rows=16044 width=4)
## 5                                     -> Index Only Scan using idx_fk_rental_id on payment p (cost=0.29..819.23 rows=14596 width=4)
```

```
rs3 <- DBI::dbGetQuery(con,
    "explain select sum(f.rental_rate) open_amt, count(*) count
      from rental r
      left outer join payment p
        on r.rental_id = p.rental_id
      join inventory i
        on r.inventory_id = i.inventory_id
      join film f
        on i.film_id = f.film_id
      where p.rental_id is null
    ;"
)

```

```
head(rs3)
```

```
##                                QUERY PLAN
## 1          Aggregate  (cost=2353.64..2353.65 rows=1 width=40)
## 2      -> Hash Join   (cost=205.14..2313.53 rows=8022 width=12)
## 3                      Hash Cond: (i.film_id = f.film_id)
## 4      -> Hash Join   (cost=128.64..2215.88 rows=8022 width=2)
## 5                      Hash Cond: (r.inventory_id = i.inventory_id)
## 6      -> Merge Anti Join (cost=0.57..2066.73 rows=8022 width=4)
```

```
rs4 <- DBI::dbGetQuery(con,
  "explain select c.customer_id,c.first_name,c.last_name,sum(f.rental_rate) open_amt,count(*)
    from rental r
      left outer join payment p
        on r.rental_id = p.rental_id
      join inventory i
        on r.inventory_id = i.inventory_id
      join film f
        on i.film_id = f.film_id
      join customer c
        on r.customer_id = c.customer_id
  where p.rental_id is null
  group by c.customer_id,c.first_name,c.last_name
  order by open_amt desc
  ;"
)
```

```
head(rs4)
```

```
##                                QUERY PLAN
## 1          Sort      (cost=2452.49..2453.99 rows=599 width=260)
## 2              Sort Key: (sum(f.rental_rate)) DESC
## 3      -> HashAggregate (cost=2417.37..2424.86 rows=599 width=260)
## 4              Group Key: c.customer_id
## 5      -> Hash Join   (cost=227.62..2357.21 rows=8022 width=232)
## 6              Hash Cond: (r.customer_id = c.customer_id)
```

16.2 Clean up

```
# dbRemoveTable(con, "cars")
# dbRemoveTable(con, "mtcars")
# dbRemoveTable(con, "cust_movies")

# diconnect from the db
dbDisconnect(con)

sp_docker_stop("sql-pet")
```


Chapter 17

SQL queries behind the scenes (72)

This chapter explains:

- Some details about how SQL queries work behind the scenes
- SQL queries are executed behind the scenes
- You can pass values to SQL queries

```
# These packages are called in almost every chapter of the book:
library(tidyverse)
library(DBI)
library(RPostgres)
library(glue)
library(here)
require(knitr)
library(dbplyr)
library(sqlpetr)
```

Start up the docker-pet container

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

now connect to the 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)
```

17.1 SQL Execution Steps

- Parse the incoming SQL query
- Compile the SQL query
- Plan/optimize the data acquisition path
- Execute the optimized query / acquire and return data

```
dbWriteTable(con, "mtcars", mtcars, overwrite = TRUE)
rs <- dbSendQuery(con, "SELECT * FROM mtcars WHERE cyl = 4")
dbFetch(rs)
```

```
##      mpg  cyl  disp  hp drat    wt  qsec vs am gear carb
## 1  22.8    4 108.0  93 3.85 2.320 18.61  1  1    4    1
```

```
## 2 24.4 4 146.7 62 3.69 3.190 20.00 1 0 4 2
## 3 22.8 4 140.8 95 3.92 3.150 22.90 1 0 4 2
## 4 32.4 4 78.7 66 4.08 2.200 19.47 1 1 4 1
## 5 30.4 4 75.7 52 4.93 1.615 18.52 1 1 4 2
## 6 33.9 4 71.1 65 4.22 1.835 19.90 1 1 4 1
## 7 21.5 4 120.1 97 3.70 2.465 20.01 1 0 3 1
## 8 27.3 4 79.0 66 4.08 1.935 18.90 1 1 4 1
## 9 26.0 4 120.3 91 4.43 2.140 16.70 0 1 5 2
## 10 30.4 4 95.1 113 3.77 1.513 16.90 1 1 5 2
## 11 21.4 4 121.0 109 4.11 2.780 18.60 1 1 4 2
```

```
dbClearResult(rs)
```

17.2 Passing values to SQL statements

```
#Pass one set of values with the param argument:
rs <- dbSendQuery(con,"SELECT * FROM mtcars WHERE cyl = 4")
dbFetch(rs)
```

```
##      mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## 1  22.8   4 108.0  93 3.85 2.320 18.61 1 1  4   1
## 2  24.4   4 146.7  62 3.69 3.190 20.00 1 0  4   2
## 3  22.8   4 140.8  95 3.92 3.150 22.90 1 0  4   2
## 4  32.4   4  78.7  66 4.08 2.200 19.47 1 1  4   1
## 5  30.4   4  75.7  52 4.93 1.615 18.52 1 1  4   2
## 6  33.9   4  71.1  65 4.22 1.835 19.90 1 1  4   1
## 7  21.5   4 120.1  97 3.70 2.465 20.01 1 0  3   1
## 8  27.3   4  79.0  66 4.08 1.935 18.90 1 1  4   1
## 9  26.0   4 120.3  91 4.43 2.140 16.70 0 1  5   2
## 10 30.4   4  95.1 113 3.77 1.513 16.90 1 1  5   2
## 11 21.4   4 121.0 109 4.11 2.780 18.60 1 1  4   2
```

```
dbClearResult(rs)
```

17.3 Pass multiple sets of values with dbBind():

```
rs <- dbSendQuery(con, "SELECT * FROM mtcars WHERE cyl = $1")
dbBind(rs, list(6L)) # cyl = 6
dbFetch(rs)
```

```
##      mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## 1  21.0   6 160.0 110 3.90 2.620 16.46 0 1  4   4
## 2  21.0   6 160.0 110 3.90 2.875 17.02 0 1  4   4
## 3  21.4   6 258.0 110 3.08 3.215 19.44 1 0  3   1
## 4  18.1   6 225.0 105 2.76 3.460 20.22 1 0  3   1
## 5  19.2   6 167.6 123 3.92 3.440 18.30 1 0  4   4
## 6  17.8   6 167.6 123 3.92 3.440 18.90 1 0  4   4
## 7  19.7   6 145.0 175 3.62 2.770 15.50 0 1  5   6
```

```
dbBind(rs, list(8L)) # cyl = 8
dbFetch(rs)
```

```
##      mpg cyl  disp  hp drat    wt  qsec vs am gear carb
```

```
## 1  18.7   8 360.0 175 3.15 3.440 17.02 0 0   3   2
## 2  14.3   8 360.0 245 3.21 3.570 15.84 0 0   3   4
## 3  16.4   8 275.8 180 3.07 4.070 17.40 0 0   3   3
## 4  17.3   8 275.8 180 3.07 3.730 17.60 0 0   3   3
## 5  15.2   8 275.8 180 3.07 3.780 18.00 0 0   3   3
## 6  10.4   8 472.0 205 2.93 5.250 17.98 0 0   3   4
## 7  10.4   8 460.0 215 3.00 5.424 17.82 0 0   3   4
## 8  14.7   8 440.0 230 3.23 5.345 17.42 0 0   3   4
## 9  15.5   8 318.0 150 2.76 3.520 16.87 0 0   3   2
## 10 15.2   8 304.0 150 3.15 3.435 17.30 0 0   3   2
## 11 13.3   8 350.0 245 3.73 3.840 15.41 0 0   3   4
## 12 19.2   8 400.0 175 3.08 3.845 17.05 0 0   3   2
## 13 15.8   8 351.0 264 4.22 3.170 14.50 0 1   5   4
## 14 15.0   8 301.0 335 3.54 3.570 14.60 0 1   5   8
```

```
dbClearResult(rs)
```

17.4 Clean up

```
# dbRemoveTable(con, "cars")
dbRemoveTable(con, "mtcars")
# dbRemoveTable(con, "cust_movies")

# diconnect from the db
dbDisconnect(con)

sp_docker_stop("sql-pet")
```


Chapter 18

Writing to the DBMS (73)

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

- Write queries in R using docker container.
- Start and connect to the database with R.
- Create, Modify, and remove the table.

Start up the `docker-pet` container:

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

Now connect to the database with R using your login info:

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

18.1 Create a new table

This is an example from the DBI help file.

```
dbWriteTable(con, "cars", head(cars, 3)) # "cars" is a built-in dataset, not to be confused with mtcars

dbReadTable(con, "cars") # there are 3 rows
```

```
##   speed dist
## 1     4    2
## 2     4   10
## 3     7    4
```

18.2 Modify an existing table

To add additional rows or instances to the “cars” table, we will use INSERT command with their values.

There are two different ways of adding values: list them or pass values using the `param` argument.

```
dbExecute(
  con,
```

```
"INSERT INTO cars (speed, dist) VALUES (1, 1), (2, 2), (3, 3)"
)
```

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

```
dbReadTable(con, "cars")  # there are now 6 rows
```

```
##    speed dist
## 1      4    2
## 2      4   10
## 3      7    4
## 4      1    1
## 5      2    2
## 6      3    3
```

```
# Pass values using the param argument:
```

```
dbExecute(
  con,
  "INSERT INTO cars (speed, dist) VALUES ($1, $2)",
  param = list(4:7, 5:8)
)
```

```
## [1] 4
```

```
dbReadTable(con, "cars")  # there are now 10 rows
```

```
##    speed dist
## 1      4    2
## 2      4   10
## 3      7    4
## 4      1    1
## 5      2    2
## 6      3    3
## 7      4    5
## 8      5    6
## 9      6    7
## 10     7    8
```

18.3 Remove table and Clean up

Here you will remove the table “cars”, disconnect from the database and exit docker.

```
dbRemoveTable(con, "cars")

# diconnect from the db
dbDisconnect(con)

sp_docker_stop("sql-pet")
```

Appendix A

Other resources (89)

A.1 Editing this book

- Here are instructions for editing this tutorial

A.2 Docker alternatives

- Choosing between Docker and Vagrant

A.3 Docker and R

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

A.4 Documentation for Docker and Postgres

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

A.5 SQL and dplyr

- Why SQL is not for analysis but dplyr is
- Data Manipulation with dplyr (With 50 Examples)

A.6 More Resources

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

Appendix B

Mapping your local environment (92)

B.1 Environment Tools Used in this Chapter

Note that `tidyverse`, `DBI`, `RPostgres`, `glue`, and `knitr` are loaded. Also, we've sourced the `[db-login-batch-code.R]` (`'r-database-docker/book-src/db-login-batch-code.R'`) file which is used to log in to PostgreSQL.

```
library(rstudioapi)
```

The following code block defines `Tool` and versions for the graph that follows. The information order corresponds to the order shown in the graph.

```
library(DiagrammeR)

## OS information
os_lbl <- .Platform$OS.type
os_ver <- 0
if (os_lbl == 'windows') {
  os_ver <- system2('cmd', stdout = TRUE) %>%
    grep(x = ., pattern = 'Microsoft Windows \\[', value = TRUE) %>%
    gsub(x = ., pattern = "^Microsoft.+Version |\\]", replace = '')
}

if (os_lbl == 'unix' || os_lbl == 'Linux' || os_lbl == 'Mac') {
  os_ver <- system2('uname', '-r', stdout = TRUE)
}

## Command line interface into Docker Apps
## CLI/system2
cli <- array(dim = 3)
cli[1] <- "docker [OPTIONS] COMMAND ARGUMENTS\n\nsystem2(docker,[OPTIONS,]\n, COMMAND,ARGUMENTS)"
cli[2] <- 'docker exec -it sql-pet bash\n\nsystem2(docker,exec -it sql-pet bash)'
cli[3] <- 'docker exec -ti sql-pet psql -a \n-p 5432 -d dvdrental -U postgres\n\nsystem2(docker,exec -t'

# R Information
r_lbl <- names(R.Version())[1:7]
r_ver <- R.Version()[1:7]

# RStudio Information
rstudio_lbl <- c('RStudio version','Current program mode')
```

```

rstudio_ver <- c(as.character(rstudioapi::versionInfo()$version),rstudioapi::versionInfo()$mode)

# Docker Information
docker_lbl <- c('client version','server version')
docker_ver <- system2("docker", "version", stdout = TRUE) %>%
  grep(x = ., pattern = 'Version',value = TRUE) %>%
  gsub(x = ., pattern = ' +Version: +', replacement = '')

# Linux Information
linux_lbl <- 'Linux Version'
linux_ver <- system2('docker', 'exec -i sql-pet /bin/uname -r', stdout = TRUE)

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

postgres_ver <- dbGetQuery(con,"select version()") %>%
  gsub(x = ., pattern = '\\(.*$', replacement = '')

```

The following code block uses the data generated from the previous code block as input to the subgraphs, the ones outlined in red. The application nodes are the parents of the subgraphs and are not outlined in red. The **Environment** application node represents the machine you are running the tutorial on and hosts the sub-applications.

Note that the '@@' variables are populated at the end of the **Environment** definition following the ## @@1 - @@5 source data comment.

```

grViz("
digraph Envgraph {

  # graph, node, and edge definitions
  graph [compound = true, nodesep = .5, ranksep = .25,
    color = red]

  node [fontname = Helvetica, fontcolor = darkslategray,
    shape = rectangle, fixedsize = true, width = 1,
    color = darkslategray]

  edge [color = grey, arrowhead = none, arrowtail = none]

  # subgraph for Environment information
  subgraph cluster1 {
    node [fixedsize = true, width = 3]
    '@@1-1'
  }

  # subgraph for R information
  subgraph cluster2 {
    node [fixedsize = true, width = 3]
    '@@2-1' -> '@@2-2' -> '@@2-3' -> '@@2-4'
    '@@2-4' -> '@@2-5' -> '@@2-6' -> '@@2-7'
  }
}

```

```

# subgraph for RStudio information
subgraph cluster3 {
  node [fixedsize = true, width = 3]
    '@@3-1' -> '@@3-2'
}

# subgraph for Docker information
subgraph cluster4 {
  node [fixedsize = true, width = 3]
    '@@4-1' -> '@@4-2'
}

# subgraph for Docker-Linux information
subgraph cluster5 {
  node [fixedsize = true, width = 3]
    '@@5-1'
}

# subgraph for Docker-Postgres information
subgraph cluster6 {
  node [fixedsize = true, width = 3]
    '@@6-1'
}

# subgraph for Docker-Postgres information
subgraph cluster7 {
  node [fixedsize = true, height = 1.25, width = 4.0]
    '@@7-1' -> '@@7-2' -> '@@7-3'
}

CLI [label='CLI\nRStudio system2',height = .75,width=3.0, color = 'blue' ]
Environment [label = 'Linux,Mac,Windows',width = 2.5]
Environment -> R
Environment -> RStudio
Environment -> Docker

Environment -> '@@1' [lhead = cluster1] # Environment Information
R -> '@@2-1' [lhead = cluster2] # R Information
RStudio -> '@@3' [lhead = cluster3] # RStudio Information
Docker -> '@@4' [lhead = cluster4] # Docker Information
Docker -> '@@5' [lhead = cluster5] # Docker-Linux Information
Docker -> '@@6' [lhead = cluster6] # Docker-Postgres Information

'@@1' -> CLI
CLI -> '@@7' [lhead = cluster7] # CLI
'@@7-2' -> '@@5'
'@@7-3' -> '@@6'
}

[1]: paste0(os_lbl, '\n', os_ver)
[2]: paste0(r_lbl, '\n', r_ver)
[3]: paste0(rstudio_lbl, '\n', rstudio_ver)
[4]: paste0(docker_lbl, '\n', docker_ver)
[5]: paste0(linux_lbl, '\n', linux_ver)

```

```
[6]: paste0('PostgreSQL:\\n', postgres_ver)
[7]: cli
")
```

One sub-application not shown above is your local console/terminal/CLI application. In the tutorial, fully constructed docker commands are printed out and then executed. If for some reason the executed docker command fails, one can copy and paste it into your local terminal window to see additional error information. Failures seem more prevalent in the Windows environment.

B.2 Communicating with Docker Applications

In this tutorial, the two main ways to interface with the applications in the Docker container are through the CLI or the RStudio `system2` command. The blue box in the diagram above represents these two interfaces.

Appendix C

Creating the sql-pet Docker container a step at a time

Step-by-step Docker container setup with dvdrental database installed This needs to run *outside a project* to compile correctly because of the complexities of how knitr sets working directories (or because we don't really understand how it works!) The purpose of this code is to

- Replicate the docker container generated in Chapter 5 of the book, but in a step-by-step fashion
- Show that the `dvdrental` database persists when stopped and started up again.

C.1 Overview

Doing all of this in a step-by-step way that might be useful to understand how each of the steps involved in setting up a persistent PostgreSQL database works. If you are satisfied with the method shown in Chapter 5, skip this and only come back if you're interested in picking apart the steps.

```
library(tidyverse)

## -- Attaching packages ----- tidyverse 1.2.1 --

## v ggplot2 3.1.0      v purrr   0.2.5
## v tibble  1.4.2      v dplyr   0.7.8
## v tidyr   0.8.2      v stringr 1.3.1
## v readr   1.1.1      v forcats 0.3.0

## -- Conflicts ----- tidyverse_conflicts()
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()

library(DBI)
library(RPostgres)
library(glue)

##
## Attaching package: 'glue'

## The following object is masked from 'package:dplyr':
##
## collapse
```

```
require(knitr)

## Loading required package: knitr
library(dbplyr)

##
## Attaching package: 'dbplyr'
## The following objects are masked from 'package:dplyr':
##
##      ident, sql
library(sqlpetr)
library(here)

## here() starts at /Users/jds/Documents/Library/R/r-system/sql-pet
```

C.2 Download the dvdrental backup file

The first step is to get a local copy of the dvdrental PostgreSQL **restore file**. It comes in a zip format and needs to be un-zipped.

```
opts_knit$set(root.dir = normalizePath('../'))
if (!require(downloader)) install.packages("downloader")

## Loading required package: downloader
library(downloader)

download("http://www.postgresqltutorial.com/wp-content/uploads/2017/10/dvdrental.zip", destfile = glue(
unzip("dvdrental.zip", exdir = here()) # creates a tar archive named "dvdrental.tar"
```

Check on where we are and what we have in this directory:

```
dir(path = here(), pattern = "^dvdrental\\.tar|.zip")
```

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

```
sp_show_all_docker_containers()
```

```
## CONTAINER ID      IMAGE             COMMAND              CREATED      STATUS      PORTS
## 6f649719ecb2      postgres-dvdrental "docker-entrypoint.s..." About a minute ago Exited (0) 6 seconds ago
```

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

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

C.3 Build the Docker Container

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

```
wd <- here()
wd

## [1] "/Users/jds/Documents/Library/R/r-system/sql-pet"

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

docker_cmd

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

## [1] "2797b7463b9da1cf3549d18469bf0927b250d2c74538ce3fa7de27268be395f3"

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

# local file system:
dir(path = here(), pattern = "^dvdrental.tar")

## [1] "dvdrental.tar"

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

## [1] "dvdrental.tar"

Sys.sleep(3)
```

C.4 Create the database and restore from the backup

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

```
sp_show_all_docker_containers()

## CONTAINER ID      IMAGE          COMMAND                                CREATED          STATUS          PORTS          I
## 2797b7463b9d      postgres:10    "docker-entrypoint.s..."  4 seconds ago    Up 3 seconds    0.0.0.0:5

inside Docker, execute the postgres SQL command-line program to create the dvdrental database:

system2('docker', 'exec sql-pet psql -U postgres -c "CREATE DATABASE dvdrental;"',
  stdout = TRUE, stderr = TRUE)

## [1] "CREATE DATABASE"

Sys.sleep(3)
```

The `psql` program repeats back to us what it has done, e.g., to create a database named `dvdrental`. Next we execute a different program in the Docker container, `pg_restore`, and tell it where the restore file is

located. If successful, the `pg_restore` just responds with a very laconic `character(0)`. restore the database from the `.tar` file

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

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

```
Sys.sleep(3)
```

C.5 Connect to the database with R

If you are interested take a look inside the `sp_get_postgres_connection` function to see how the DBI package is being used.

```
con <- sp_get_postgres_connection(user = Sys.getenv("DEFAULT_POSTGRES_USER_NAME"),
                                password = Sys.getenv("DEFAULT_POSTGRES_PASSWORD"),
                                dbname = "dvdrental",
                                seconds_to_test = 20)
```

```
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"
```

```
dbDisconnect(con)
```

```
# 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")
```

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

```
dbListTables(con)
```

```
## [1] "actor_info"          "customer_list"
## [3] "film_list"           "nicer_but_slower_film_list"
## [5] "sales_by_film_category" "staff"
## [7] "sales_by_store"      "staff_list"
## [9] "category"            "film_category"
## [11] "country"             "actor"
```

```
## [13] "language"      "inventory"
## [15] "payment"       "rental"
## [17] "city"         "store"
## [19] "film"         "address"
## [21] "film_actor"   "customer"
```

C.6 Cleaning up

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

```
dbDisconnect(con)
```

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

```
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
## 2797b7463b9d	postgres:10	"docker-entrypoint.s..."	15 seconds ago	Exited (0)	Less than a second ago

We are leaving the `sql-pet` container intact so it can be used in running the rest of the examples and book.

Clean up by removing the local files used in creating the database:

```
file.remove(here("dvdrental.zip"))
```

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

```
file.remove(here("dvdrental.tar"))
```

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


Appendix D

APPENDIX D - Quick Guide to SQL (94)

SQL stands for Structured Query Language. It is a database language where we can perform certain operations on the existing database and we can use it to create a new database. There are four main categories where the SQL commands fall into: DDL, DML, DCL, and TCL.

##Data Definition Language (DDL)

It consists of the SQL commands that can be used to define database schema. The DDL commands include:

1. CREATE
2. ALTER
3. TRUNCATE
4. COMMENT
5. RENAME
6. DROP

##Data Manipulation Language (DML)

These four SQL commands deal with the manipulation of data in the database.

1. SELECT
2. INSERT
3. UPDATE
4. DELETE

##Data Control Language (DCL)

The DCL commands deal with user's rights, permissions and other controls in database management system.

1. GRANT
2. REVOKE

##Transaction Control Language (TCL)

These commands deal with the control over transaction within the database. Transaction combines a set of tasks into single execution.

1. SET TRANSACTION
2. SAVEPOINT
3. ROLLBACK
4. COMMIT

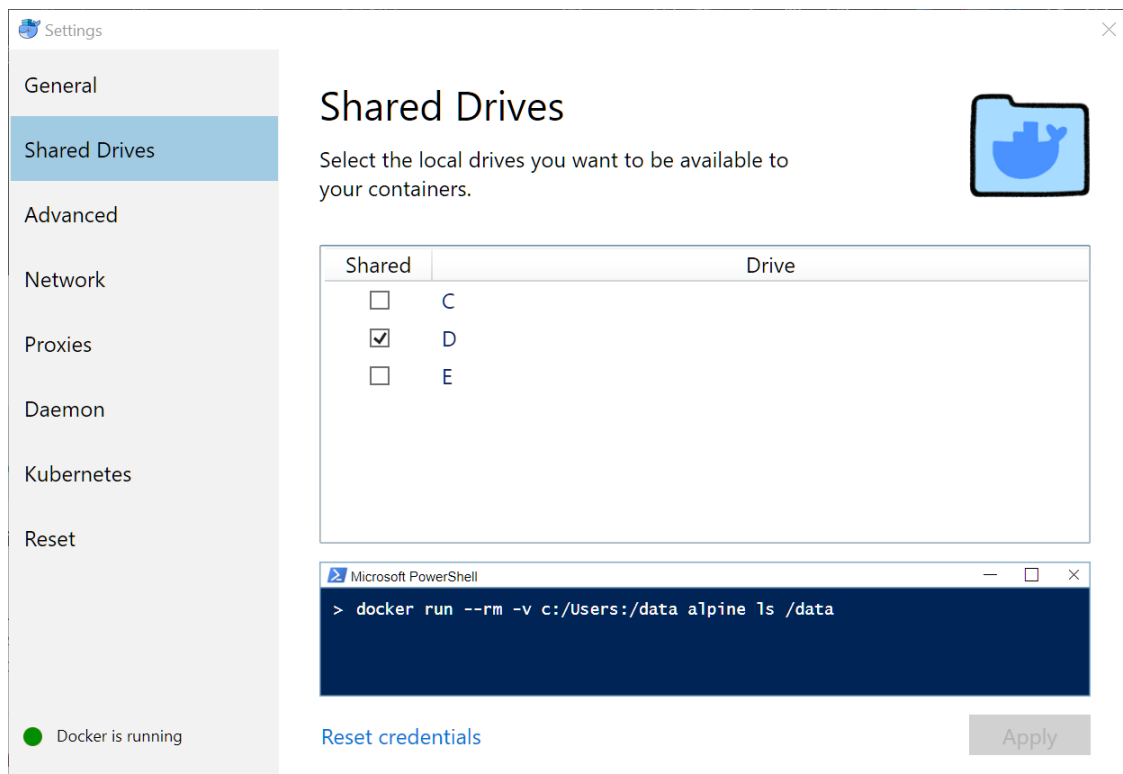
Appendix E

Additional technical details for Windows users (95)

E.1 Docker for Windows settings

E.1.1 Shared drives

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

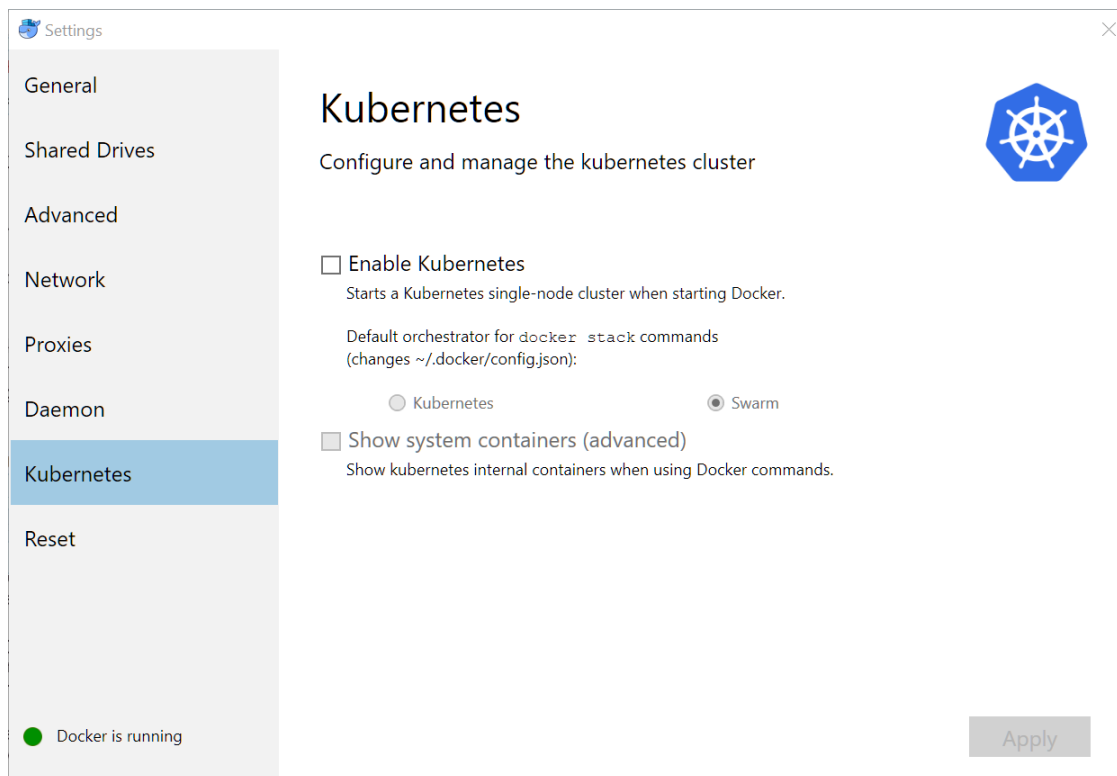


E.1.2 Kubernetes

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

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

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



E.2 Git, GitHub and line endings

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

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

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

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

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

In particular, executable `sh` or `bash` scripts will fail in a Docker container if they have Windows line endings. You may see an error message with `\r` in it, which means the shell saw the carriage return (`\r`) and gave up.

But often you'll see no hint at all what the problem was.

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

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

This repo includes a sample: `.gitattributes`

Appendix F

Dplyr functions and SQL cross-walk (96)

Where are these covered and should they be included?

Dplyr Function	description	SQL Clause	Where Category
all_equal() all.equal()	Flexible equality comparison for data frames		Two-table verbs
all_vars() any_vars()	Apply predicate to all variables		scoped-Operate on a selection of variables
arrange()	Arrange rows by variables	ORDER BY	13.1.4 Basic (21) single-table verbs
arrange_all() arrange_at() arrange_if() auto_copy()	Arrange rows by a selection of variables Copy tables to same source, if necessary	ORDER BY	scoped-Operate on a selection of variables Remote tables
between()	Do values in a numeric vector fall in specified range?		Vector functions
bind_rows() bind_cols() combine()	Efficiently bind multiple data frames by row and column		Two-table verbs
case_when() coalesce()	A general vectorised if Find first non-missing element		Vector functions Vector functions
compute() collect() collapse()	Force computation of a database query		Remote tables
copy_to()	Copy a local data frame to a remote src		Remote tables
cumall() cumany() cummean()	Cumulativate versions of any, all, and mean		Vector functions
desc()	Descending order		Vector functions

Dplyr Function	description	SQL Clause	When Category
distinct()	Return rows with matching conditions	SELECT distinct *	Basic single-table verbs
distinct()	Select distinct/unique rows	SELECT distinct {colname1,...colnamen}	Basic single-table verbs
do()	Do anything	NA	Basic single-table verbs
explain() show_query()	Explain details of a tbl		Remote tables
filter_all() filter_if()	Filter within a selection of variables		scoped-Operate on a selection of variables
filter_at()			scoped-Operate on a selection of variables
funs()	Create a list of functions calls.		scoped-Operate on a selection of variables
group_by() ungroup()	Objects exported from other packages	GROUP BY no ungroup	Basic single-table verbs
group_by_all()	Group by a selection of variables		scoped-Operate on a selection of variables
group_by_at()			Metadata
group_by_if()			Metadata
groups() group_vars()	Return grouping variables		Metadata
ident()	Flag a character vector as SQL identifiers		Remote tables
if_else()	Vectorised if		Vector functions
inner_join() left_join()	Join two tbls together		Two-table verbs
right_join() full_join()			
semi_join() anti_join()			
inner_join() left_join()	Join data frame tbls		Two-table verbs
right_join() full_join()			
semi_join() anti_join()			
intersect() union()	Set operations		Two-table verbs
union_all() setdiff()			
setequal()			
lead() lag()	Lead and lag.		Vector functions
mutate() transmute()	Add new variables	SELECT computed_value computed_name	11.5. Basic (13) single-table verbs
n()	The number of observations in the current group.		Vector functions
n_distinct()	Efficiently count the number of unique values in a set of vector		Vector functions
na_if()	Convert values to NA		Vector functions
near()	Compare two numeric vectors		Vector functions
nth() first() last()	Extract the first, last or nth value from a vector		Vector functions

Dplyr Function	description	SQL Clause	When Category
order_by()	A helper function for ordering window function output		Vector functions
pull()	Pull out a single variable	SELECT column_name;	Basic single-table verbs
recode() recode_factor()	Recode values		Vector functions
row_number() ntile()	Windowed rank functions.		Vector functions
min_rank()			
dense_rank()			
percent_rank()			
cume_dist()			
rowwise()	Group input by rows		Other backends
sample_n()	Sample n rows from a table	ORDER BY RANDOM() LIMIT 10	Basic
sample_frac()			single-table verbs
select() rename()	Select/rename variables by name	SELECT column_name alias_name	9.1.8 Basic (11) single-table verbs
select_all()	Select and rename a selection of variables		scoped-Operate on a selection of variables
rename_all() select_if()			
rename_if() select_at()			
rename_at()			
slice()	Select rows by position	SELECT row_number() over (partition by expression(s) order_by exp)	Basic single-table verbs
sql()	SQL escaping.		Remote tables
src_mysql()	Source for database		Remote tables
src_postgres()	backends		
src_sqlite()			
summarise_all()	Summarise and mutate multiple columns.		scoped-Operate on a selection of variables
summarise_if()			
summarise_at()			
summarize_all()			
summarize_if()			
summarize_at()			
mutate_all()			
mutate_if()			
mutate_at()			
transmute_all()			
transmute_if()			
transmute_at()			
summarize()	Reduces multiple values down to a single value	SELECT aggregate_functions GROUP BY	11.5. Basic (13) single-table verbs
tally()	Count/tally	GROUP BY	9.1.6 Single-table
count() add_tally()	observations by group		(11) helpers
add_count()			
tbl() is.tbl() as.tbl()	Create a table from a data source		Remote tables

Dplyr Function	description	SQL Clause	Where Category
top_n()	Select top (or bottom) n rows (by value)	ORDER BY VALUE {DESC} LIMIT 10	Single-table helpers
vars()	Select variables		scoped-Operate on a selection of variables

Appendix G

DBI package functions - coverage (96b)

Where are these covered and should the by included?

DBI	1st time	Call Example/Notes
DBIConnct	6.3.2 (04)	in sp_get_postgres_connection
dbAppendTable		
dbCreateTable		
dbDisconnect	6.4n (04)	dbDisconnect(con)
dbExecute	10.4.2 (13)	Executes a statement and returns the number of rows affected. dbExecute() comes with a default implementation (which should work with most backends) that calls dbSendStatement(), then dbGetRowsAffected(), ensuring that the result is always free-d by dbClearResult().
dbExistsTable		dbExistsTable(con,'actor')
dbFetch	17.1 (72)	dbFetch(rs)
dbGetException		
dbGetInfo		dbGetInfo(con)
dbGetQuery	10.4.1 (13)	dbGetQuery(con,'select * from store;')
dbIsReadOnly		dbIsReadOnly(con)
dbIsValid		dbIsValid(con)
dbListFields	6.3.3 (04)	DBI::dbListFields(con, "mtcars")
dbListObjects		dbListObjects(con)
dbListTables	6.3.2 (04)	DBI::dbListTables(con, con)
dbReadTable	8.1.2	DBI::dbReadTable(con, "rental")
dbRemoveTable		
dbSendQuery	17.1 (72)	rs <- dbSendQuery(con, "SELECT * FROM mtcars WHERE cyl = 4")
dbSendStatement		The dbSendStatement() method only submits and synchronously executes the SQL data manipulation statement (e.g., UPDATE, DELETE, INSERT INTO, DROP TABLE, ...) to the database engine.
dbWriteTable	6.3.3 (04)	dbWriteTable(con, "mtcars", mtcars, overwrite = TRUE)

Bibliography

(2018). A grammar of data manipulation • dplyr. .