Databases in the tidyverse

ASA Webinar

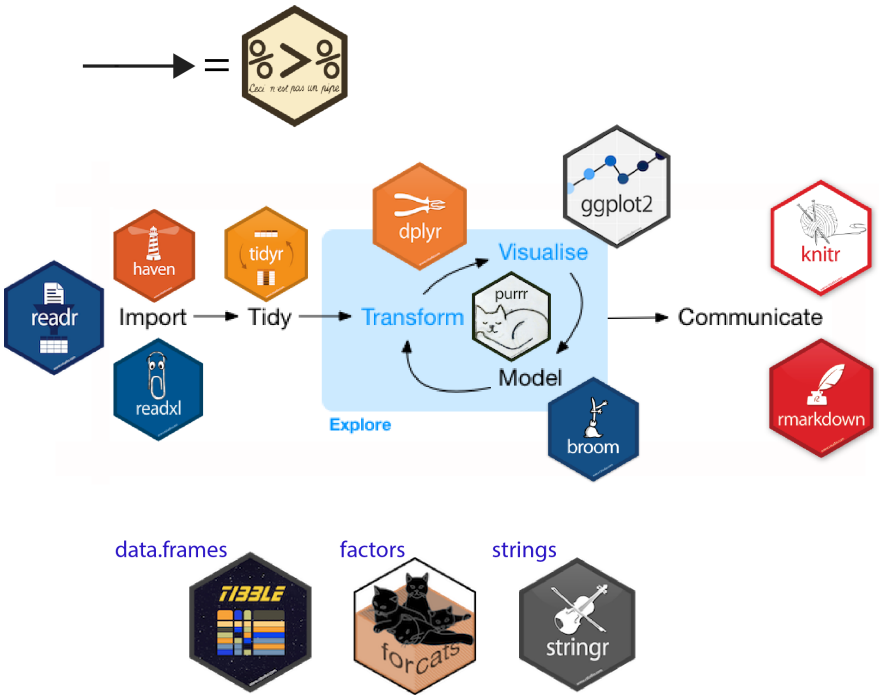
Ben Baumer & Nick Horton

November 15, 2017

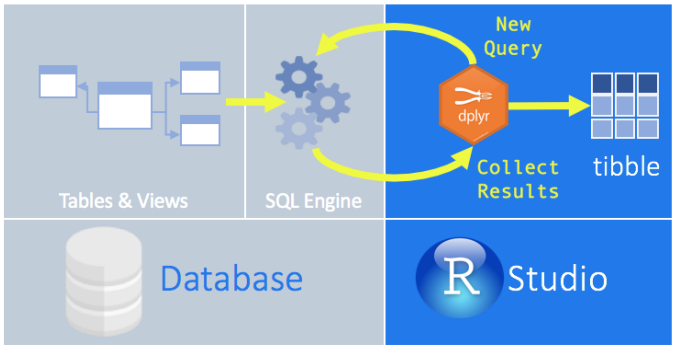
(https://github.com/beanumber/tidy-databases)

Map of this webinar

1. review key idioms for data wrangling with `dplyr`
2. introduce the backend interfaces for common database systems
3. provide examples of how the `dplyr` engine translates a data pipeline
4. discuss common misconceptions and performance issues
5. provide pointers on how to learn more



Use dplyr to interact with the database



dplyr

`dplyr` highlights

The Five Verbs

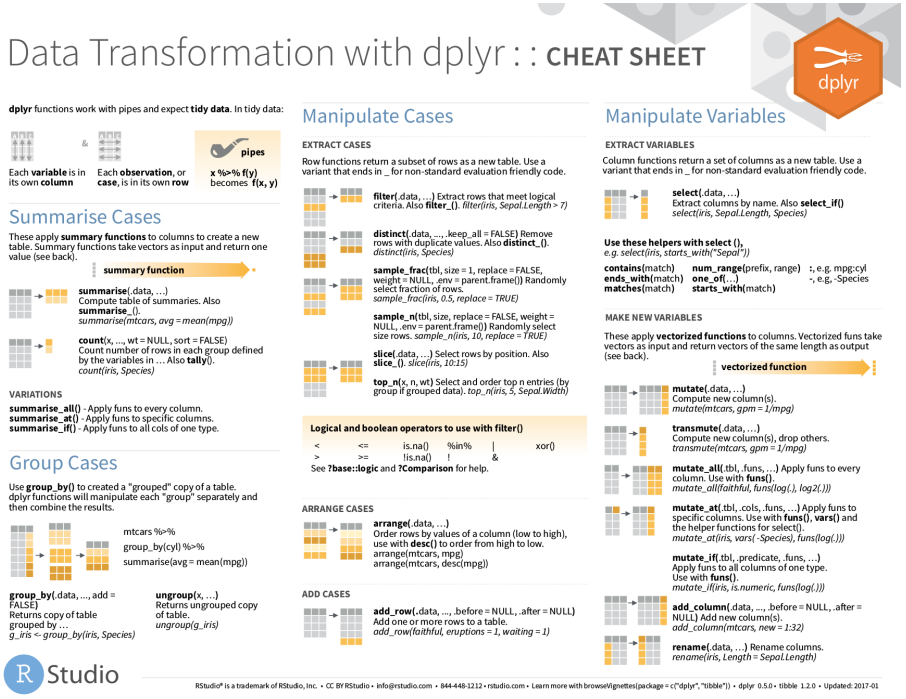
- `select()`
- `filter()`
- `mutate()`
- `arrange()`
- `summarize()`

Plus:

- `group_by()`
- `rename()`
- `inner_join()`, `left_join()`, etc.
- `do()`

Philosophy

- Each *verb* takes a data frame and returns a data frame
  - actually a `tbl_df` (more on that later)
  - allows chaining with `%>%` (more on that later)
- Idea:
  - master a few simple commands
  - use your creativity to combine them
- Cheat Sheet:
  - (https://www.rstudio.com/resources/cheatsheets/)



What is a tibble?

- object of class `tbl`
- a re-imagining of a `data.frame`
- it looks and acts like a `data.frame`
- but it's even better...
- `tidyverse` works with tibbles

`select()` : take a subset of the columns

Column functions return a set of columns as a new table. Use a variant that ends in _ for non-standard evaluation friendly code.

select(.data, ...**)**
Extract columns by name. Also **select_if()**
select(iris, Sepal.Length, Species)

Use these helpers with select (),
e.g. select(iris, starts_with("Sepal"))

contains(match**)**	**num_range(**prefix, range**)**	**:**, e.g. mpg:cyl
ends_with(match**)**	**one_of(**...**)**	**-**, e.g, -Species
matches(match**)**	**starts_with(**match**)**	

`filter()` : take a subset of the rows

EXTRACT CASES

Row functions return a subset of rows as a new table. Use a variant that ends in _ for non-standard evaluation friendly code.

filter(.data, ...**)** Extract rows that meet logical criteria. Also **filter_()**. *filter(iris, Sepal.Length > 7)*

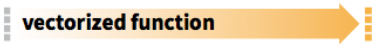
distinct(.data, ..., .keep_all = FALSE**)** Remove rows with duplicate values. Also **distinct_()**.
distinct(iris, Species)

sample_frac(tbl, size = 1, replace = FALSE, weight = NULL, .env = parent.frame()**)** Randomly select fraction of rows.
sample_frac(iris, 0.5, replace = TRUE)

`mutate()` : add or modify a column

MAKE NEW VARIABLES

These apply **vectorized functions** to columns. Vectorized funs take vectors as input and return vectors of the same length as output (see back).



mutate(.data, ...**)**
Compute new column(s).
mutate(mtcars, gpm = 1/mpg)

transmute(.data, ...**)**
Compute new column(s), drop others.
transmute(mtcars, gpm = 1/mpg)

13

`arrange()` : sort the rows

ARRANGE CASES

arrange(.data, …**)**
Order rows by values of a column (low to high), use with **desc()** to order from high to low.
arrange(mtcars, mpg)
arrange(mtcars, desc(mpg))

14

`summarize()` : collapse to a single row

Summarise Cases

These apply **summary functions** to columns to create a new table. Summary functions take vectors as input and return one value (see back).

summary function

summarise(.data, …)
Compute table of summaries. Also **summarise_**().
summarise(mtcars, avg = mean(mpg))

count(x, ..., wt = NULL, sort = FALSE)
Count number of rows in each group defined by the variables in … Also **tally**().
count(iris, Species)

15

`group_by()` : apply to groups

Group Cases

Use **group_by()** to created a "grouped" copy of a table. dplyr functions will manipulate each "group" separately and then combine the results.

```
mtcars %>%
group_by(cyl) %>%
summarise(avg = mean(mpg))
```

16

The pipe

The pipe operator

- Inspired by pipe (|) in UNIX
- Provided by `magrittr` package

- The Treachery of Images
- Rene Magritte, 1929

How does the pipe work?

dplyr functions work with pipes and expect **tidy data**. In tidy data:

Each **variable** is in its own **column**

&

Each **observation**, or **case**, is in its own **row**

pipes

x %>% f(y) becomes **f(x, y)**

Using the pipe

The expression

```
mydata %>%
  verb(arguments)
```

is the same as:

```
verb(mydata, arguments)
```

In effect, `function(x, args) = x %>% function(args)`.

Why the pipe?

Instead of having to read/write:

```
select(filter(mutate(data, args1), args2), args3)
```

You can do:

```
data %>%
  mutate(args1) %>%
  filter(args2) %>%
  select(args3)
```

Coding Little Bunny Foo Foo

- Nested form:

```
bop(scoop(hop(foo_foo, through = forest), up = field_mice), on = head
```

- With pipes:

```
foo_foo %>%
  hop(through = forest) %>%
  scoop(up = field_mouse) %>%
  bop(on = head)
```

(https://github.com/hadley/r4ds/blob/master/pipes.Rmd)

Example: fuel economy

```
mtcars %>%
  filter(am == 1) %>%
  group_by(cyl) %>%
  summarize(num_models = n(),
            mean_mpg = mean(mpg)) %>%
  arrange(desc(mean_mpg))
```

```
## # A tibble: 3 x 3
##     cyl num_models mean_mpg
##   <dbl>      <int>    <dbl>
## 1     4          8 28.07500
## 2     6          3 20.56667
## 3     8          2 15.40000
```


Example: the 300-300 club

```
library(Lahman)
Batting %>%
  group_by(playerID) %>%
  summarize(span = paste(min(yearID), max(yearID), sep = "-"),
            career_HR = sum(HR), career_SB = sum(SB)) %>%
  filter(career_HR >= 300, career_SB >= 300) %>%
  left_join(Master, by = "playerID") %>%
  mutate(player_name = paste(nameLast, nameFirst, sep = ", ")) %>%
  select(player_name, span, career_HR, career_SB) %>%
  arrange(desc(career_HR))
```

```
## # A tibble: 8 x 4
##       player_name      span career_HR career_SB
##             <chr>     <chr>     <int>     <int>
## 1    Bonds, Barry 1986-2007       762       514
## 2 Rodriguez, Alex 1994-2016       696       329
## 3    Mays, Willie 1951-1973       660       338
## 4   Dawson, Andre 1976-1996       438       314
## 5 Beltran, Carlos 1998-2016       421       312
## 6    Bonds, Bobby 1968-1981       332       461
## 7 Sanders, Reggie 1991-2007       305       304
## 8   Finley, Steve 1989-2007       304       320
```


dbplyr

A brief primer on SQL

- SQL is not just one thing
  - MySQL, PostgreSQL, SQLite
  - Oracle, Big Query, Vertica
- Theory developed in 1970s (E.F. Codd)
- Ingres implemented mid-1970s
- robust, time-tested, well understood

`dplyr` <-> SQL

`dplyr`

```
table %>%
  filter(field == "value") %>%
  left_join(lkup,
    by = c("lkup_id" = "id") %>%
  group_by(year) %>%
  summarize(N = sum(1)) %>%
  filter(N > 100) %>%
  arrange(desc(N)) %>%
  head(10)
```

MySQL

```
SELECT
  year, sum(1) as N
FROM table t
LEFT JOIN lkup l
  ON t.lkup_id = l.id
WHERE field = "value"
GROUP BY year
HAVING N > 100
ORDER BY N desc
LIMIT 0, 10;
```


README.md

dbplyr

build passing CRAN 0.5.0

coverage 70%

dbplyr is the database backend for dbplyr.

View all of README.md

`dbplyr` = `dplyr` + SQL connection

- `dplyr` can access a SQL database directly
- Instead of `tbl_df`, you have a `tbl_sql`
- Data is stored and processed in SQL
  - Tiny memory footprint in R
  - **Lazy evaluation**
  - server-side processing
- `dplyr` to SQL translation via `show_query()`

Example: connect to remote IMDB mirror

```
db <- src_mysql(db = "imdb", host = "scidb.smith.edu",
                user = "mth292", password = "RememberPi")
title <- tbl(db, "title")
title
```

```
## # Source:   table<title> [?? x 12]
## # Database: mysql 5.5.57-0ubuntu0.14.04.1 [mth292@scidb.smith.edu:/imdb]
##       id                                          title imdb_index
##    <int>                                          <chr>      <chr>
##  1 78460 Adults Recat to the Simpsons (30th Anniversary)       <NA>
##  2 70273                                    (2016-05-18)       <NA>
##  3 60105                                    (2014-04-11)       <NA>
##  4 32120                                    (2008-05-01)       <NA>
##  5 97554                                Schmölders Traum       <NA>
##  6 57966                                         (#1.1)       <NA>
##  7 76391                                     Anniversary       <NA>
##  8 11952              Angus Black/Lester Barrie/DC Curry       <NA>
##  9  1554                                     New Orleans       <NA>
## 10 58442                                    Kiss Me Kate       <NA>
## # ... with more rows, and 9 more variables: kind_id <int>,
## #   production_year <int>, imdb_id <int>, phonetic_code <chr>,
## #   episode_of_id <int>, season_nr <int>, episode_nr <int>,
## #   series_years <chr>, md5sum <chr>
```


`title` contains 4.6 million rows, but...

- ...it takes up almost no space

```
print(object.size(title), units = "Kb")
```

```
## 3.8 Kb
```

- `title` looks like a `data.frame` but...

```
class(title)
```

```
## [1] "tbl_dbi"  "tbl_sql"  "tbl_lazy" "tbl"
```

- ...it's not actually a `data.frame`

Example: `show_query()`

```
star_wars <- title %>%
  filter(title == "Star Wars", kind_id == 1) %>%
  select(production_year, title)
star_wars
```

```
## # Source:   lazy query [?? x 2]
## # Database: mysql 5.5.57-0ubuntu0.14.04.1 [mth292@scidb.smith.edu:/imdb]
##   production_year     title
##             <int>     <chr>
## 1            1977 Star Wars
```

```
show_query(star_wars)
```

```
## <SQL>
## SELECT `production_year` AS `production_year`, `title` AS `title`
## FROM `title`
## WHERE ((`title` = 'Star Wars') AND (`kind_id` = 1.0))
```


Translation of basic functions

```
library(dbplyr)
translate_sql(ceiling(mpg))
```

```
## <SQL> CEIL("mpg")
```

```
translate_sql(mean(mpg))
```

```
## <SQL> avg("mpg") OVER ()
```

```
translate_sql(cyl == 4)
```

```
## <SQL> "cyl" = 4.0
```

```
translate_sql(cyl %in% c(4, 6, 8))
```

```
## <SQL> "cyl" IN (4.0, 6.0, 8.0)
```

Code pass-thru of other functions

```
# no PASTE() in SQL
translate_sql(paste("hp", "wt", "vs"))
```

```
## <SQL> PASTE('hp', 'wt', 'vs')
```

```
# works, but no CONCAT() in R
translate_sql(CONCAT("hp", "wt", "vs"))
```

```
## <SQL> CONCAT('hp', 'wt', 'vs')
```

```
# nonsense
translate_sql(CRAZY_FUNCTION(mpg))
```

```
## <SQL> CRAZY_FUNCTION("mpg")
```


Fine-looking R code

```
title %>%
  filter(title %like% '%Star Wars%',
         kind_id == 1,
         !is.na(production_year)) %>%
  select(title, production_year) %>%
  arrange(production_year)
```

```
## # Source:     lazy query [?? x 2]
## # Database:   mysql 5.5.57-0ubuntu0.14.04.1 [mth292@scidb.smith.edu:/imdb]
## # Ordered by: production_year
##                                                 title production_year
##                                                 <chr>           <int>
##  1                                          Star Wars            1977
##  2    Star Wars: Episode V - The Empire Strikes Back             1980
##  3                                  Star Wars Underoos           1980
##  4        Star Wars: Episode VI - Return of the Jedi             1983
##  5                             Tezukuri no Star Wars             1990
##  6         Star Wars: Episode I - The Phantom Menace             1999
##  7                               Star Wars Gangsta Rap           2000
##  8                                   Star Wars Returns           2001
##  9 Star Wars: Attack of the Clones - A Jigsaw Puzzle             2002
## 10      Star Wars Episode V 1/2: The Han Solo Affair             2002
## # ... with more rows
```


Weird-looking hybrid code

```
title %>%
  filter(title %like% '%Star Wars%',
         kind_id == 1,
         !is.na(production_year)) %>%
  mutate(before_dash = SUBSTRING_INDEX(title, '-', 1)) %>%
  select(before_dash, production_year) %>%
  arrange(production_year)
```

```
## # Source:     lazy query [?? x 2]
## # Database:   mysql 5.5.57-0ubuntu0.14.04.1 [mth292@scidb.smith.edu:/imdb]
## # Ordered by: production_year
##                                    before_dash production_year
##                                          <chr>           <int>
##  1                                   Star Wars            1977
##  2                        Star Wars: Episode V             1980
##  3                           Star Wars Underoos            1980
##  4                       Star Wars: Episode VI             1983
##  5                        Tezukuri no Star Wars            1990
##  6                        Star Wars: Episode I             1999
##  7                        Star Wars Gangsta Rap            2000
##  8                            Star Wars Returns            2001
##  9             Star Wars: Attack of the Clones             2002
```


Why `dplyr` vs. SQL?

R + `dplyr` good at:

- fitting models
- plotting
- wrangling data of all kinds
- working with small data
- being an *interface to SQL*

SQL good at:

- storage and retrieval
- medium-to-big data
- multi-user, asynchronous access
- serving institutional needs
- web/mobile apps

Data size for a single user



“Size”	size	hardware	software
small	< several GB	RAM	R
medium	several GB – a few TB	hard disk	SQL
big	many TB or more	cluster	Spark?

DBI

DBI

DBI

build unknown codecov 42% CRAN 0.7

The DBI package defines a common interface between the R and database management systems (DBMS). The interface defines a small set of classes and methods similar in spirit to Perl's DBI, Java's JDBC, Python's DB-API, and Microsoft's ODBC. It defines a set of classes and methods defines what operations are possible and how they are performed:

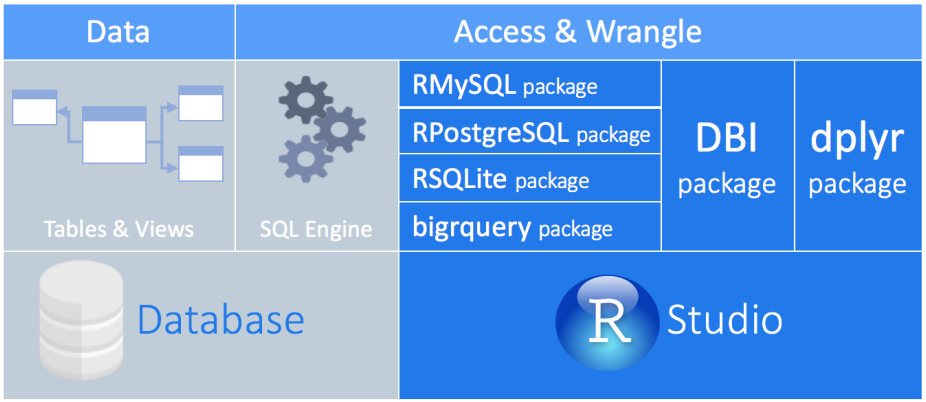
- connect/disconnect to the DBMS
- create and execute statements in the DBMS
- extract results/output from statements
- error/exception handling
- information (meta-data) from database objects
- transaction management (optional)

DBI separates the connectivity to the DBMS into a "front-end" and a "back-end". Applications use only the exposed "front-end" API. The facilities that communicate with specific DBMSs (SQLite, MySQL, PostgreSQL, MonetDB, etc.) are provided by "drivers" (other packages) that get invoked automatically through S4 methods.

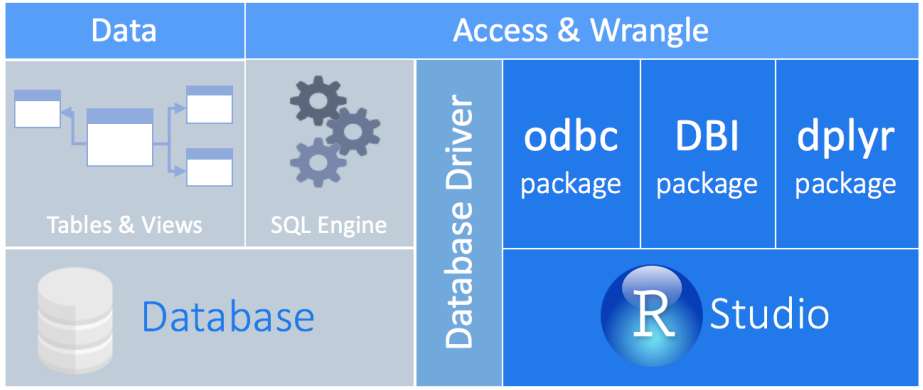
The following example illustrates some of the DBI capabilities:

```
library(DBI)
```


Open Source Databases



Commercial Databases



Supported databases

`r-dbi`:

- `RSQLite` CRAN 2.0
- `RMySQL` CRAN 0.10.13
- `odbc` CRAN 1.1.3
- `bigrquery` CRAN 0.4.1
- `RPostgres` CRAN not published

Others:

- `RPostgreSQL` CRAN 0.6-2
- `MonetDBLite` CRAN 0.4.1

`DBI` underneath `dbplyr`

```
class(db)
```

```
## [1] "src_dbi" "src_sql" "src"
```

```
str(db)
```

```
## List of 2
##  $ con  :Formal class 'MySQLConnection' [package "RMySQL"] with 1 slot
##   .. ..@ Id: int [1:2] 0 1
##  $ disco:<environment: 0x9d51b50>
##  - attr(*, "class")= chr [1:3] "src_dbi" "src_sql" "src"
```

```
class(db$con)
```

```
## [1] "MySQLConnection"
## attr(,"package")
## [1] "RMySQL"
```


Common interface

```
dbListTables(db$con)
```

```
##  [1] "aka_name"         "aka_title"        "cast_info"
##  [4] "char_name"        "comp_cast_type"   "company_name"
##  [7] "company_type"     "complete_cast"    "info_type"
## [10] "keyword"          "kind_type"        "link_type"
## [13] "movie_companies"  "movie_info"       "movie_info_idx"
## [16] "movie_keyword"    "movie_link"       "name"
## [19] "person_info"      "role_type"        "title"
```

```
dbListFields(db$con, "title")
```

```
##  [1] "id"              "title"           "imdb_index"
##  [4] "kind_id"         "production_year" "imdb_id"
##  [7] "phonetic_code"   "episode_of_id"   "season_nr"
## [10] "episode_nr"      "series_years"    "md5sum"
```

3 ways to bring SQL data into R

1. `dbplyr`
   - via a `tbl_sql` (see previous examples)
2. `DBI`
   - via `dbGetQuery()`
3. `rmarkdown`
   - via an SQL chunk

Using `dbGetQuery()`

```
query <- "SELECT production_year, title
          FROM title
          WHERE title = 'Star Wars' AND kind_id = 1;"
dbGetQuery(db$con, query)
```

```
##   production_year     title
## 1            1977 Star Wars
```

- No `dplyr` pipeline
- Write SQL as a character string and pass it

Using `rmarkdown`

```
# ```{sql, connection=db$con, output.var = "mydataframe"}
# SELECT production_year, title
# FROM title
# WHERE title = 'Star Wars' AND kind_id = 1;
# ```
```

- Instead of R chunk, SQL chunk
- `connection` talks to database
- `output.var` stores the result

```
head(mydataframe)
```

```
##   production_year     title
## 1            1977 Star Wars
```


Examples

49

Performance

50

Recall that `tbl_sql`'s are tiny

```
title <- tbl(db, "title")
class(title)
```

```
## [1] "tbl_dbi"  "tbl_sql"  "tbl_lazy" "tbl"
```

```
print(object.size(title), units = "Kb")
```

```
## 3.8 Kb
```

51

Pipelines are evaluated lazily

```
old_movies <- title %>%
  filter(production_year < 1950,
         kind_id == 1)
class(old_movies)
```

```
## [1] "tbl_dbi"  "tbl_sql"  "tbl_lazy" "tbl"
```

```
dim(old_movies)
```

```
## [1] NA 12
```

```
print(object.size(old_movies), units = "Kb")
```

```
## 6.8 Kb
```

52

Use `collect()` to bring into R

```
old_movies_local <- old_movies %>%
  collect()
class(old_movies_local)
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

```
dim(old_movies_local)
```

```
## [1] 184837     12
```

```
print(object.size(old_movies_local), units = "Mb")
```

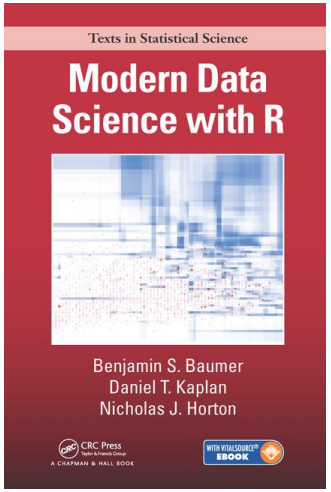
```
## 39.2 Mb
```

Lazy evaluation

- Pipelines that don't ask for data
  - evaluated fast
- Processing tiggered when data is needed
  - `print()`, `head()`, `glimpse()`, etc.
  - plotting
  - `collect()`

What next?

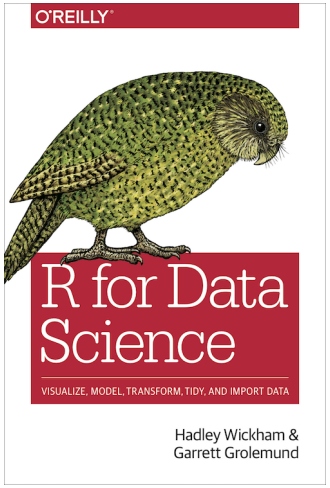
Modern Data Science with R



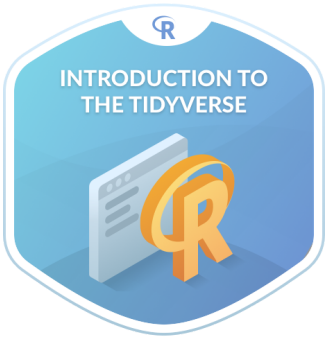
- CRC Press
- Amazon
- (http://mdsr-book.github.io)
  - sample chapters
  - R Markdown examples
  - exercise solutions

R for Data Science

- (http://r4ds.had.co.nz/)
- free, online
- open source, written in `bookdown`
- Garrett Grolemund and Hadley Wickham



DataCamp



- Intro to the Tidyverse
  - David Robinson
- Intro to SQL for Data Science
  - Nick Carchedi
- Data Manipulation in R with `dplyr`
  - Garrett Grolemund

db.rstudio.com

Resources

- Github repo: https://github.com/beanumber/tidy-databases
- these slides
- sql-example1 (Rmd | HTML)
- sql-example2 (Rmd | HTML)

- Setting the stage for data science: integration of data management and databases in statistics courses (CHANCE 2015), also https://arxiv.org/abs/1401.3269

Thank you!