SQL & R

Data Wrangling and Husbandry

03/22/2020

SQL

- SQL stands for Structured Query Language.
- SQL is used to communicate with a database
- Most commercial relational database management systems rely on SQL for their query language
- Databases can be very large and are optimized for certain data manipulation tools, so it can be advantageous to use database tools for what they do best and R tools for what they do best

An aside

One philosophy is to think of R as a flexible user interface. Some tools are built in (e.g., linear models), but other tools interface with other components of the computational world: shell commands, SQL, Spark, and others. Depending on their implementation, they can cut down on the cognitive burden of task switching.

Why work with a database

- Your data is already in a database
  - Relying on, say, csv written out from the database will lead to aging problems
- You have so much data it won't fit in memory
- (Besides using a database, there are custom R approaches to the latter situation—the CRAN task view on high performance computing has some details in the "Large memory and out-of-memory data" section)

Concept mapping

Database	R
Table	Data frame
Field, column	Variable, column
Record, row	Observation, row
to select all cols	blank to select all cols
NULL	NA
single quotes	single or double quotes
not case sensitive	case sensitive

- There are nice introductions to SQL at https://www.sqlteaching.com and http://2016.padjo.org/tutorials/hello-sqlite-and-sqlite-clients/ , but our goal is not to learn SQL. Instead, our goal is to have R generate and execute SQL commands.

The `dplyr` package interfaces with SQL!

In the past, there were database specific functions to connect to databases: * `src_mysql` connects to a mysql database * `src_postgres` connects to a postgresql database. * `src_sqlite` connects to a sqlite database

The current approach is to use the function `DBI::dbConnect()`. The `DBI` package can be used for database work beyond what `dplyr` can do.

`DBI::dbConnect()`

The arguments to DBI::dbConnect() vary from database to database, but the first argument is always the database backend.

- `RSQLite::SQLite()` for RSQLite
- `RMySQL::MySQL()` for RMySQL
- `RPostgreSQL::PostgreSQL()` for RPostgreSQL
- `odbc::odbc()` for odbc
- `bigrquery::bigquery()` for BigQuery

For a remote database, this will look more like

```
my_db <- DBI::dbConnect(RMySQL::MySQL(),
  host = "example.database.com",
  user = "example",
  password = rstudioapi::askForPassword("Database password'
)
```

You'll have to get help from your database administrator.

An extended example

Rather than try to tie into an existing database, we will make our own local sqlite database. This is not the way it would work in practice, since if your data fits in R you probably don't need an external database. We will use

```
my_db <- DBI::dbConnect(RSQLite::SQLite(), path = "demo.sql
```

```
library(gapminder)
copy_to(my_db, gapminder, temporary = FALSE, indexes = list
  "country", "continent", "year"))
```

```
gapminder_sqlite <- tbl(my_db, "gapminder")
```

dplyr commands for databases

Once you've set up the connection to the database with `src_*`, you can almost treat the database as a collection of data frames.

```
## notice the top matter
select(gapminder_sqlite, country, continent)
```

```
## # Source:   lazy query [?? x 2]
## # Database: sqlite 3.30.1 []
##    country     continent
##    <chr>       <chr>
##  1 Afghanistan Asia
##  2 Afghanistan Asia
##  3 Afghanistan Asia
##  4 Afghanistan Asia
##  5 Afghanistan Asia
##  6 Afghanistan Asia
##  7 Afghanistan Asia
##  8 Afghanistan Asia
##  9 Afghanistan Asia
```

```r
gapminder_sqlite %>% filter(country == "Peru")
```

```
## # Source:   lazy query [?? x 6]
## # Database: sqlite 3.30.1 []
##    country continent  year lifeExp      pop gdpPercap
##    <chr>   <chr>     <int>   <dbl>    <int>     <dbl>
##  1 Peru    Americas   1952    43.9  8025700     3759.
##  2 Peru    Americas   1957    46.3  9146100     4245.
##  3 Peru    Americas   1962    49.1 10516500     4957.
##  4 Peru    Americas   1967    51.4 12132200     5788.
##  5 Peru    Americas   1972    55.4 13954700     5938.
##  6 Peru    Americas   1977    58.4 15990099     6281.
##  7 Peru    Americas   1982    61.4 18125129     6435.
##  8 Peru    Americas   1987    64.1 20195924     6361.
##  9 Peru    Americas   1992    66.5 22430449     4446.
## 10 Peru    Americas   1997    68.4 24748122     5838.
## # ... with more rows
```

In fact, the usual basic commands of `select`, `filter`, `arrange`, `mutate`, and `summarize` work exactly as expected. However, they are translated to SQL and then run on the the database rather than within R

```
(example_query <- gapminder_sqlite %>%
   filter(year == "2007") %>%
   group_by(Continent) %>%
   summarize(lifeExp = mean(lifeExp)))
```

```
## Warning: Missing values are always removed in SQL.
## Use `mean(x, na.rm = TRUE)` to silence this warning
## This warning is displayed only once per session.
```

```
## # Source:   lazy query [?? x 2]
## # Database: sqlite 3.30.1 []
##   continent lifeExp
##   <chr>       <dbl>
## 1 Africa       54.8
## 2 Americas     73.6
## 3 Asia         70.7
```

```r
show_query(example_query)
```

```
## <SQL>
## SELECT `Continent`, AVG(`lifeExp`) AS `lifeExp`
## FROM `gapminder`
## WHERE (`year` = '2007')
## GROUP BY `Continent`
```

If you know SQL, you can insert your own code

```
tbl(my_db, sql("SELECT country, continent FROM gapminder"))
```

```
## # Source:   SQL [?? x 2]
## # Database: sqlite 3.30.1 []
##    country     continent
##    <chr>       <chr>
##  1 Afghanistan Asia
##  2 Afghanistan Asia
##  3 Afghanistan Asia
##  4 Afghanistan Asia
##  5 Afghanistan Asia
##  6 Afghanistan Asia
##  7 Afghanistan Asia
##  8 Afghanistan Asia
##  9 Afghanistan Asia
## 10 Afghanistan Asia
## # ... with more rows
```

Laziness

- dplyr is *lazy* when it comes to SQL commands
  - Never brings data into R unless explictly asked
  - Collects commands and sends it to the database in one step at the last possible moment
- In the example below, the SQL code is created but not run

```
lazy_ex <- gapminder_sqlite %>% filter(continent == "Americ
  select(country, year, pop, gdpPercap) %>%
  mutate(gdp = pop * gdpPercap) %>% arrange(desc(gdp))
```

If you print the object, dplyr generates the QSL and sends it to the database, but only requests the first 10 rows.

```
lazy_ex
```

```
## # Source:     lazy query [?? x 5]
## # Database:   sqlite 3.30.1 []
## # Ordered by: desc(gdp)
##    country        year       pop gdpPercap     gdp
##    <chr>         <int>     <int>     <dbl>   <dbl>
##  1 United States  2007 301139947    42952. 1.29e13
##  2 United States  2002 287675526    39097. 1.12e13
##  3 United States  1997 272911760    35767. 9.76e12
##  4 United States  1992 256894189    32004. 8.22e12
##  5 United States  1987 242803533    29884. 7.26e12
##  6 United States  1982 232187835    25010. 5.81e12
##  7 United States  1977 220239000    24073. 5.30e12
##  8 United States  1972 209896000    21806. 4.58e12
##  9 United States  1967 198712000    19530. 3.88e12
## 10 United States  1962 186538000    16173. 3.02e12
## # ... with more rows
```

To get the full results, use `collect()`

```r
collect(lazy_ex)
```

```
## # A tibble: 300 x 5
##    country        year       pop gdpPercap      gdp
##    <chr>         <int>     <int>     <dbl>    <dbl>
##  1 United States  2007 301139947    42952. 1.29e13
##  2 United States  2002 287675526    39097. 1.12e13
##  3 United States  1997 272911760    35767. 9.76e12
##  4 United States  1992 256894189    32004. 8.22e12
##  5 United States  1987 242803533    29884. 7.26e12
##  6 United States  1982 232187835    25010. 5.81e12
##  7 United States  1977 220239000    24073. 5.30e12
##  8 United States  1972 209896000    21806. 4.58e12
##  9 United States  1967 198712000    19530. 3.88e12
## 10 United States  1962 186538000    16173. 3.02e12
## # ... with 290 more rows
```

Grouping

Compare the following

```
gapminder %>% filter(year == 2007) %>%
  select(country, continent, lifeExp) %>%
  group_by(continent) %>% filter(lifeExp == min(lifeExp))
```

```
## # A tibble: 5 x 3
## # Groups:   continent [5]
##   country     continent lifeExp
##   <fct>       <fct>       <dbl>
## 1 Afghanistan Asia         43.8
## 2 Haiti       Americas     60.9
## 3 New Zealand Oceania      80.2
## 4 Swaziland   Africa       39.6
## 5 Turkey      Europe       71.8
```

```
gapminder_sqlite %>% filter(year == 2007) %>%
  select(country, continent, lifeExp)
  %>% group_by(continent) %>% filter(lifeExp == min(lifeExp
```

```
## Error: <text>:3:3: unexpected SPECIAL
## 2:   select(country, continent, lifeExp)
## 3:   %>%
##      ^
```

SQLlite has very limited “window” functions—basically, you can summarize but not filter or mutate. PostgreSQL does allow these, and is generally more powerful.

For our example, we can get around the problem by bringing the data to R just before the last filtering step

```r
group_filter_ex <- gapminder_sqlite %>%
  filter(year == 2007) %>%
  select(country, continent, lifeExp) %>%
  group_by(continent) %>% collect()

group_filter_ex %>% filter(lifeExp == min(lifeExp))
```

```
## # A tibble: 5 x 3
## # Groups:   continent [5]
##   country     continent lifeExp
##   <chr>       <chr>       <dbl>
## 1 Afghanistan Asia         43.8
## 2 Haiti       Americas     60.9
## 3 New Zealand Oceania      80.2
## 4 Swaziland   Africa       39.6
## 5 Turkey      Europe       71.8
```

Joins

Joins generally require the data to be in the same location.

You can use `*_join(x, y, copy = TRUE)` to copy `y` into the same source as `x`. Keep in mind that it might be time-consuming. With large data, when `copy = TRUE` it can speed things up to set `auto_index = TRUE`

We'll run some examples using the built-in `src` in the `dbplyr` package.

```r
library(dbplyr)
```

```
##
## Attaching package: 'dbplyr'

## The following objects are masked from 'package:dplyr':
##
##     ident, sql
```

```r
flights_db <- nycflights13_sqlite()
```

```r
library(nycflights13)
flights2 <- flights %>%
  select(year:day, hour, tailnum, carrier)
right_join(airlines, flights2, by = "carrier")
```

```
## # A tibble: 336,776 x 7
##    carrier name                        year month   day  h
##    <chr>   <chr>                      <int> <int> <int> <c
##  1 UA      United Air Lines Inc.       2013     1     1
##  2 UA      United Air Lines Inc.       2013     1     1
##  3 AA      American Airlines Inc.      2013     1     1
##  4 B6      JetBlue Airways             2013     1     1
##  5 DL      Delta Air Lines Inc.        2013     1     1
##  6 UA      United Air Lines Inc.       2013     1     1
##  7 B6      JetBlue Airways             2013     1     1
##  8 EV      ExpressJet Airlines Inc.    2013     1     1
##  9 B6      JetBlue Airways             2013     1     1
## 10 AA      American Airlines Inc.      2013     1     1
## # ... with 336,766 more rows
```

```r
flights2db <- tbl(flights_db, "flights") %>%
  select(year:day, hour, tailnum, carrier)
right_join(airlines, flights2db, by = "carrier", copy = TRU
```

```
## # A tibble: 336,776 x 7
##    carrier name                        year month   day  h
##    <chr>   <chr>                      <int> <int> <int> <d
##  1 UA      United Air Lines Inc.       2013     1     1
##  2 UA      United Air Lines Inc.       2013     1     1
##  3 AA      American Airlines Inc.      2013     1     1
##  4 B6      JetBlue Airways             2013     1     1
##  5 DL      Delta Air Lines Inc.        2013     1     1
##  6 UA      United Air Lines Inc.       2013     1     1
##  7 B6      JetBlue Airways             2013     1     1
##  8 EV      ExpressJet Airlines Inc.    2013     1     1
##  9 B6      JetBlue Airways             2013     1     1
## 10 AA      American Airlines Inc.      2013     1     1
## # ... with 336,766 more rows
```

```r
flights2db_r <- collect(tbl(flights_db, "flights")  %>%
    select(year:day, hour, tailnum, carrier), n = Inf)
right_join(airlines, flights2db_r, by = "carrier")
```

```
## # A tibble: 336,776 x 7
##    carrier name                        year month   day  h
##    <chr>   <chr>                      <int> <int> <int> <c
##  1 UA      United Air Lines Inc.       2013     1     1
##  2 UA      United Air Lines Inc.       2013     1     1
##  3 AA      American Airlines Inc.      2013     1     1
##  4 B6      JetBlue Airways             2013     1     1
##  5 DL      Delta Air Lines Inc.        2013     1     1
##  6 UA      United Air Lines Inc.       2013     1     1
##  7 B6      JetBlue Airways             2013     1     1
##  8 EV      ExpressJet Airlines Inc.    2013     1     1
##  9 B6      JetBlue Airways             2013     1     1
## 10 AA      American Airlines Inc.      2013     1     1
## # ... with 336,766 more rows
```

`dplyr` takes a lazy approach to take advantage of the database's optimized functions. It also has a few protective features:

- `nrow()` returns NA
- printing a `tbl` gives just the first 10 rows
- `tail()` gives an error

If you know SQL, there are a few other potentially helpful commands

- `compute()` and `collapse()` do not bring results back to R, and so can be used to optimize performance
- `translate_sql()` translates R expressions to SQL

DBI

R has a nice database interface package that enables developers to, well, interface nicely with databases. As a result, there are multiple approaches (e.g., an R-Oracle interface). `dplyr`'s approach is just one; we've looked at it because it fits nicely in the framework of the course.

In class exercise

0. Type

```r
library(dbplyr)
example_sql <- lahman_sqlite()
```

to set up a local SQLite database of the Lahman data

1. Type `src_tbls(example_sql)` for a list of the tables in the database
2. Use a `*_join()` function to find all individuals in the `Managers` table who are also in the `HallOfFame` table. Don't bring it into R.
3. Pipe the result of the previous question to `select()` to select just the `category` variable.
4. Use what you've done to provide a table (in the R, not SQL, sense) of the category variable for individuals in the Hall of Fame who worked as managers