Ch 15 from Modern Data Science with R

Database querying using SQL

In this chapter, we will explore approaches for working with data sets that are larger than what we've been exploring so far; for example, they will fit on a personal computer's hard disk, but not necessarily in its memory. Thankfully, a venerable solution for retrieving what we'll call medium-sized data from a database has been around since the 1970s: SQL (structured query language). Database management systems implementing SQL provide a ubiquitous architecture for storing and querying data that is relational in nature. The wide deployment of SQL makes it a "must-know" tool for data scientists.

Section 15.1: From dplyr to SQL

Recall the `flights` data from the nycflights13 package that we encountered in Data Science 1. Even though that data seemed large (336776 x 19), since it only involves the year 2013 and 3 origin airports in the NYC area, it's really somewhat small by database standards, even if we consider the related data sets from the nycflights13 package (`airports`, `carriers`, etc.).

However, if we consider additional years and origin airports, the `flights` data frame can become very large very quickly. Going back to 1987, there are more than 169 million individual flights – each comprising a different row in this table. These data occupy nearly 20 gigabytes as CSVs and thus are problematic to store in a personal computer's memory. Instead, one strategy is to write these data to disk and use a querying language to access only those rows that interest us. The authors of MDSR created the `dbConnect_scidb()` function to provide a connection to the airlines database that lives on a remote MySQL server that they set up ahead of time; this connection is stored as the object `db`. The tbl() function from `dplyr` maps the `flights` table in that `airlines` database to an object in R, in this case also called `flights`. The same is done for the `carriers` table.

```
library(nycflights13)
old_flights <- flights
class(old_flights)
```

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

```
old_flights
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     1      517            515         2      830            819
##  2  2013     1     1      533            529         4      850            830
##  3  2013     1     1      542            540         2      923            850
##  4  2013     1     1      544            545        -1     1004           1022
##  5  2013     1     1      554            600        -6      812            837
##  6  2013     1     1      554            558        -4      740            728
##  7  2013     1     1      555            600        -5      913            854
```

```
##  8  2013     1     1      557            600        -3      709            723
##  9  2013     1     1      557            600        -3      838            846
## 10  2013     1     1      558            600        -2      753            745
## # i 336,766 more rows
## # i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
## #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
## #   hour <dbl>, minute <dbl>, time_hour <dttm>
```

```
table(old_flights$year)
```

```
##
##   2013
## 336776
```

```
table(old_flights$origin)
```

```
##
##    EWR    JFK    LGA
## 120835 111279 104662
```

```
db <- dbConnect_scidb("airlines")
flights <- tbl(db, "flights")
carriers <- tbl(db, "carriers")

class(flights)
```

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

```
dim(flights)
```

```
## [1] NA 21
```

Note that while we can use the `flights` and `carriers` objects as if they were data frames, they are not, in fact, data.frames. Rather, they have class `tbl_MariaSQLConnection`, and more generally, `tbl_sql`. A `tbl` is a special kind of object created by `dplyr` that behaves similarly to a data.frame. This allows `dplyr` to interact with these tbls as if they were data.frames in our R session – a powerful and convenient illusion! Therefore, we can use our usual `dplyr` verbs to, for example, retrieve the top on-time carriers with at least 100 flights arriving at JFK in September 2016:

```
q <- flights |>
  filter(
    year == 2016 & month == 9,
    dest == "JFK"
  ) |>
  inner_join(carriers, by = c("carrier" = "carrier")) |>
  group_by(name) |>
  summarize(
    N = n(),
    pct_ontime = sum(arr_delay <= 15) / n()
```

```
  ) |>
  filter(N >= 100) |>
  arrange(desc(pct_ontime))
head(q, 4)
```

```
## Warning: Missing values are always removed in SQL aggregation functions.
## Use `na.rm = TRUE` to silence this warning
## This warning is displayed once every 8 hours.
```

```
## # Source:     SQL [4 x 3]
## # Database:   mysql  [mdsr_public@mdsr.cdc7tgkkqd0n.us-east-1.rds.amazonaws.com:NA/airlines]
## # Ordered by: desc(pct_ontime)
##   name                        N pct_ontime
##   <chr>                 <int64>      <dbl>
## 1 Delta Air Lines Inc.     2396      0.869
## 2 Virgin America            347      0.833
## 3 JetBlue Airways          3463      0.817
## 4 American Airlines Inc.   1397      0.782
```

What is actually happening is that `dplyr` translates our pipeline into SQL. We can see the translation by passing the pipeline through the `show_query()` function. `dplyr` automatically translate our R code into SQL anytime we're using an object of class `tbl_sql`.

```
show_query(q)
```

```
## <SQL>
## SELECT
##   `name`,
##   COUNT(*) AS `N`,
##   SUM(`arr_delay` <= 15.0) / COUNT(*) AS `pct_ontime`
## FROM (
##   SELECT `LHS`.*, `name`
##   FROM (
##     SELECT *
##     FROM `flights`
##     WHERE (`year` = 2016.0 AND `month` = 9.0) AND (`dest` = 'JFK')
##   ) `LHS`
##   INNER JOIN `carriers`
##     ON (`LHS`.`carrier` = `carriers`.`carrier`)
## ) `q01`
## GROUP BY `name`
## HAVING (COUNT(*) >= 100.0)
## ORDER BY `pct_ontime` DESC
```

Here is what that SQL query would look like if we wrote it from scratch in a more readable format:

```
SELECT
  c.name,
  SUM(1) AS N,
  SUM(arr_delay <= 15) / SUM(1) AS pct_ontime
FROM flights AS f
```

```
JOIN carriers AS c ON f.carrier = c.carrier
WHERE year = 2016 AND month = 9
  AND dest = 'JFK'
GROUP BY name
HAVING N >= 100
ORDER BY pct_ontime DESC
LIMIT 0,4;
```

PAUSE List all of the parallels you see between our R code and our SQL query. If you need a hint, you can use the `translate_sql` function in the `dbplyr` package.

```
library(dbplyr)
```

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

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

```
translate_sql(
  mean(arr_delay, na.rm = TRUE),
  con = db
)
```

```
## <SQL> AVG(`arr_delay`) OVER ()
```

When using `dplyr` with a `tbl_sql` backend, one must be careful to use expressions that SQL can understand. See the example below involving `paste0`, which is not recoginized by SQL. This is just one more reason why it is important to know SQL on its own and not rely entirely on the dplyr front-end (as wonderful as it is).

```
# Store the paste0 function as my_paste.  This function pastes several
#   strings together into a single string.
my_paste <- paste0

# Note my_paste is not recognized and translated - it passes straight through
translate_sql(
  my_paste("this", "is", "a", "string"),
  con = db
)

# This throws an error because dplyr code is translated to SQL because
#   carriers is a tbl_sql object, but there is no translation for my_paste
carriers |>
  mutate(name_code = my_paste(name, "(", carrier, ")"))

# Solution 1: replace my_paste with its SQL equivalent
carriers |>
  mutate(name_code = CONCAT(name, "(", carrier, ")"))

# Solution 2: first pull the carrier data into R using collect(), which
```

```
#   breaks the connection to the MySQL server and returns a data.frame
carriers |>
  collect() |>
  mutate(name_code = my_paste(name, "(", carrier, ")"))
```

Section 15.2: Flat-file databases

Be able to describe the difference between flat-file databases and relational databases, along with limitations of flat-files databases.

Section 15.3: The SQL universe

Become familiar with major implementations of SQL. We will focus on MySQL.

Section 15.4: The SQL data manipulation language

```
library(DBI)

dbGetQuery(db,'
  SHOW TABLES;
')
```

```
##   Tables_in_airlines
## 1           airports
## 2           carriers
## 3            flights
## 4             planes
```

```
dbGetQuery(db,'
  DESCRIBE airports;
')
```

```
##     Field          Type Null Key Default Extra
## 1     faa    varchar(3)   NO PRI
## 2    name  varchar(255)  YES        <NA>
## 3     lat decimal(10,7)  YES        <NA>
## 4     lon decimal(10,7)  YES        <NA>
## 5     alt       int(11)  YES        <NA>
## 6      tz   smallint(4)  YES        <NA>
## 7     dst       char(1)  YES        <NA>
## 8    city  varchar(255)  YES        <NA>
## 9 country  varchar(255)  YES        <NA>
```

```
  DESCRIBE airports;
```

Table 1: 9 records

Field	Type	Null	Key	Default	Extra
faa	varchar(3)	NO	PRI		
name	varchar(255)	YES		NA	
lat	decimal(10,7)	YES		NA	
lon	decimal(10,7)	YES		NA	
alt	int(11)	YES		NA	
tz	smallint(4)	YES		NA	
dst	char(1)	YES		NA	
city	varchar(255)	YES		NA	
country	varchar(255)	YES		NA	

```
SELECT
  c.name,
  SUM(1) AS N,
  SUM(arr_delay <= 15) / SUM(1) AS pct_ontime
FROM flights AS f
JOIN carriers AS c ON f.carrier = c.carrier
WHERE year = 2016 AND month = 9
  AND dest = 'JFK'
GROUP BY name
HAVING N >= 100
ORDER BY pct_ontime DESC
LIMIT 0,4;
```

```
mydataframe
```

```
##                     name    N pct_ontime
## 1   Delta Air Lines Inc. 2396     0.8689
## 2         Virgin America  347     0.8329
## 3        JetBlue Airways 3463     0.8169
## 4 American Airlines Inc. 1397     0.7817
```