Databases with Dbplyr

Byteflow Dynamics

10/21/2017

Contents

1 Introduction to dbplyr

- When to use dbplyr:

1. your data is already in a database

2. your data does not fit in memory, external storage engine is needed.

- Getting started: install dbplyr

```
# install.packages("dbplyr")
```

Commonnly used backends to connect with dbplyr:

- RMySQL connects to MySQL and MariaDB
- RPostgreSQL connects to Postgres and Redshift.
- RSQLite embeds a SQLite database.
- odbc connects to many commercial databases via the open database connectivity protocol.
- bigrquery connects to Google's BigQuery.

2 RSQLite

Connecting to RSQLite

```
library(dbplyr)
library(dplyr)
library(RSQLite)
con <- DBI::dbConnect(RSQLite::SQLite(), path = ":memory:")
```

Arguments for connecting with different databases:

- for RSQLite it's RSQLite::SQLite()
- for RMySQL, it's RMySQL::MySQL

- RPostgreSQL, it's RPostgreSQL::PostgreSQL()
- BigQuery, it's bigrquery::bigquery()

- odbc, odbc::odbc()

To create a temporary in-memory database, we use: ":memory:"

3 Connecting and adding data to the database

Add data to our newly created database

```
library(nycflights13)

copy_to(con, nycflights13::flights, "flights",
  temporary = FALSE,
  indexes = list(
    c("year", "month", "day"),
    "carrier",
    "tailnum",
    "dest"
  )
)
```

To call the data

```
flights_db <- tbl(con, "flights")
flights_db
```

```
## # Source:   table<flights> [?? x 19]
## # Database: sqlite 3.19.3 []
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     1      517            515         2      830
##  2  2013     1     1      533            529         4      850
##  3  2013     1     1      542            540         2      923
##  4  2013     1     1      544            545        -1     1004
##  5  2013     1     1      554            600        -6      812
##  6  2013     1     1      554            558        -4      740
##  7  2013     1     1      555            600        -5      913
##  8  2013     1     1      557            600        -3      709
##  9  2013     1     1      557            600        -3      838
## 10  2013     1     1      558            600        -2      753
## # ... with more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dbl>
```

4 Making sql queries

Task: Calculate the mean delayed time by destination.

```
avg_del <- flights_db %>%
  group_by(dest) %>%
  summarise(delay = mean(dep_time))
avg_del
```



```
## # Source:   lazy query [?? x 2]
## # Database: sqlite 3.19.3 []
##     dest    delay
##    <chr>    <dbl>
##  1   ABQ 2005.732
##  2   ACK 1032.664
##  3   ALB 1627.189
##  4   ANC 1635.375
##  5   ATL 1293.291
##  6   AUS 1521.476
##  7   AVL 1174.764
##  8   BDL 1490.311
##  9   BGR 1689.717
## 10   BHM 1943.640
## # ... with more rows
```

4.0.1 working with ordinary dataframes vs. remote database

When working with remote databases, dplyer * never pulls data into R unless you explicitly ask for it * sends the request in one step (collects together everything you want to do first)

In the following code

```
tailnum_delay_db <- flights_db %>%
  group_by(tailnum) %>%
  summarise(delay = mean(arr_delay),
    n = n()) %>%
  arrange(desc(delay)) %>%
  filter(n > 100)
```

- Dplyr never touches the database.
- It waits until you ask for the data: printing tailnum_delay_db.
- even then, it only pulls a few rows.

```
#tailnum_delay_db
```

- you cannot check what the last few rows

```
#tail(tailnum_delay_db)
```

- Use collect() to pull the full dataset

```
collect(tailnum_delay_db)
```

```
## # A tibble: 1,201 x 3
##    tailnum    delay     n
##      <chr>    <dbl> <int>
##  1  N11119 30.30657   148
##  2  N16919 29.88745   251
##  3  N14998 27.92202   230
##  4  N15910 27.61132   280
##  5  N13123 25.97345   121
##  6  N11192 25.85235   154
##  7  N14950 25.28780   219
##  8  N21130 24.96610   126
##  9  N24128 24.91803   129
## 10  N22971 24.74766   230
```



```
## # ... with 1,191 more rows
```

To see how dplyr translates R code into SQL use show_query()

```
tailnum_delay_db %>% show_query()
```

```
## <SQL>
## SELECT *
## FROM (SELECT *
## FROM (SELECT `tailnum`, AVG(`arr_delay`) AS `delay`, COUNT() AS `n`
## FROM `flights`
## GROUP BY `tailnum`)
## ORDER BY `delay` DESC)
## WHERE (`n` > 100.0)
```

Normally, you will be iterating serveral times to figure out what data you need to pull out. Once you figure it out, you can save it locally.

```
tailnum_delay <- tailnum_delay_db %>% collect()
```

When done working with the database, you can disconnect using: dbDisconnect()

```
#dbDisconnect(con)
```

