dplyr Backends

Jim Harner

1/12/2021

The `dplyr` provides a grammar of data manipulation using a set of verbs for transforming tibbles (or data frames) in R or across various backend data sources. For example, `dplyr` provides an interface to `sparklyr`, which is RStudio's R interface to Spark.

```
library(dplyr, warn.conflicts = FALSE)
library(RPostgreSQL)
```

```
## Loading required package: DBI
```

```
library(sparklyr)
sc <- spark_connect(master = "local")
```

This section illustrates `dplyr` often using the NYC flight departures data as a context.

```
library(nycflights13)
```

3.2 Data manipulation with `dplyr`

A powerful feature of `dplyr` is its ability to operate on various backends, including databases and Spark among others.

3.2.1 Databases

`dplyr` allows you to use the same verbs in a remote database as you would in R. It takes care of generating SQL for you so that you can avoid learning it.

The material for this subsection is taken from Hadley Wickham's dplyr Database Vignette.

The reason you'd want to use `dplyr` with a database is because:

- your data is already in a database, or
- you have so much data that it does not fit in memory, or
- you want to speed up computations.

Currently `dplyr` supports the three most popular open source databases (`sqlite`, `mysql` and `postgresql`), and Google's `bigquery`.

If you have a lot of data in a database, you can't just dump it into R due to memory limitations. Instead, you'll have to work with subsets or aggregates. `dplyr` generally make this task easy.

The goal of `dplyr` is not to replace every SQL function with an R function; that would be difficult and error prone. Instead, `dplyr` only generates `SELECT` statements, the SQL you write most often as an analyst for data extraction.

Initially, we work with the built-in SQLite database.

```
con <- DBI::dbConnect(RSQLite::SQLite(), dbname = ":memory")
# contruct the database
copy_to(con, nycflights13::flights, "flights", overwrite = TRUE)
flights_db <- tbl(con, "flights")
```

`tbl` allows us to reference the database.

We now calculate the average arrival delay by tail number.

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

```
## 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 3]
## # Database:   sqlite 3.30.1 [:memory]
## # Ordered by: desc(delay)
##    tailnum delay     n
##    <chr>   <dbl> <int>
##  1 N11119   30.3   148
##  2 N16919   29.9   251
##  3 N14998   27.9   230
##  4 N15910   27.6   280
##  5 N13123   26.0   121
##  6 N11192   25.9   154
##  7 N14950   25.3   219
##  8 N21130   25.0   126
##  9 N24128   24.9   129
## 10 N22971   24.7   230
## # ... with more rows
```

The calculations are not actually performed until `tailnum_delay_db` is requested.

We will focus on `PostgreSQL` since it provides much stronger support for `dplyr`. This code will become operational once the `airlines` database is built.

```
# my_dbh is a handle to the airlines database
# the airlines database is not yet built
my_dbh <- src_postgres("airlines")

# The following statement was run initially to put flights in the database
# flights_pg <- copy_to(my_dbh, flights, temporary=FALSE)

# tbl creates a table from a data source
flights_pg <- tbl(my_dbh, "flights")
flights_pg
```

You can use SQL:

```
flights_out <- tbl(my_dbh, sql("SELECT * FROM flights"))
```

You use the five verbs:

```
select(flights_pg, year:day, dep_delay, arr_delay)
filter(flights_pg, dep_delay > 240)
# The comments below are only used to shorten the output.
# arrange(flights_pg, year, month, day)
# mutate(flights_pg, speed = air_time / distance)
# summarise(flights_pg, delay = mean(dep_time))
```

The expressions in `select()`, `filter()`, `arrange()`, `mutate()`, and `summarise()` are translated into SQL so they can be run on the database.

Workflows can be constructed by the `%>%` operator:

```
output <-
  filter(flights_pg, year == 2013, month == 1, day == 1) %>%
  select( year, month, day, carrier, dep_delay, air_time, distance) %>%
  mutate(speed = distance / air_time * 60) %>%
  arrange(year, month, day, carrier)
collect(output)
```

This sequence of operations never actually touches the database. It's not until you ask for the data that `dplyr` generates the SQL and requests the results from the database. `collect()` pulls down all the results and returns a `tbl_df`.

How the database execute the query is given by `explain()`:

```
explain(output)
```

There are three ways to force the computation of a query:

- `collect()` executes the query and returns the results to R.
- `compute()` executes the query and stores the results in a temporary table in the database.
- `collapse()` turns the query into a table expression.

`dplyr` uses the `translate_sql()` function to convert R expressions into SQL.

PostgreSQL is much more powerful database than SQLite. It has:

- a much wider range of built-in functions
- support for window functions, which allow grouped subsets and mutates to work.

We can perform grouped `filter` and `mutate` operations with PostgreSQL. Because you can't filter on *window functions* directly, the SQL generated from the grouped filter is quite complex; so they instead have to go in a subquery.

```
daily <- group_by(flights_pg, year, month, day)

# Find the most and least delayed flight each day
bestworst <- daily %>%
  select(flight, arr_delay) %>%
  filter(arr_delay == min(arr_delay) || arr_delay == max(arr_delay))
collect(bestworst)
explain(bestworst)

# Rank each flight within a daily
```

```
ranked <- daily %>%
  select(arr_delay) %>%
  mutate(rank = rank(desc(arr_delay)))
collect(ranked)
explain(ranked)
```

3.2.2 Spark

Spark can be used as a data source using `dplyr`.

```
# Copy the R data.frame to a Spark DataFrame
copy_to(sc, faithful, "faithful")
```

```
## # Source: spark<faithful> [?? x 2]
##    eruptions waiting
##        <dbl>   <dbl>
##  1      3.6       79
##  2      1.8       54
##  3      3.33      74
##  4      2.28      62
##  5      4.53      85
##  6      2.88      55
##  7      4.7       88
##  8      3.6       85
##  9      1.95      51
## 10      4.35      85
## # ... with more rows
```

```
faithful_tbl <- tbl(sc, "faithful")

# List the available tables
src_tbls(sc)
```

```
## [1] "faithful"
```

```
# filter the Spark DataFrame and use collect to return an R data.frame
faithful_df <- faithful_tbl %>%
  filter(waiting < 50) %>%
  collect()
head(faithful_df)
```

```
## # A tibble: 6 x 2
##   eruptions waiting
##       <dbl>   <dbl>
## 1      1.75      47
## 2      1.75      47
## 3      1.87      48
## 4      1.75      48
## 5      2.17      48
## 6      2.1       49
```

This is a demonstration of getting the `faithful` data into Spark and the use of simple data manipulations on the data.

The `sparklyr` package is the basis for data manipulation and machine learning based on a data frame workflow. This approach has limitations, but it covers most use cases.

```
spark_disconnect(sc)
```