dplyr Backends

Jim Harner

1/6/2020

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

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

```
library(sparklyr)

# start the sparklyr session
master <- "local"
# master <- "spark://master:7077"
sc <- spark_connect(master, spark_home = Sys.getenv("SPARK_HOME"),
                    method = c("shell"), app_name = "sparklyr")
```

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.

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 3.29.0 [: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 SQL

`sparklyr` can import a wide range of data directly into Spark from an external data source, e.g., json. In addition, it is possible to query Spark DataFrames directly.

We will be using the `nycflights13` data again. The `flights` and `airlines` R data frames are copied into Spark.

```
library(nycflights13)
flights_sdf <- copy_to(sc, flights, "flights",  overwrite = TRUE)
airlines_sdf <- copy_to(sc, airlines, "airlines", overwrite = TRUE)
```

In Section 5.2.1 the `dplyr` verbs were used to manipulate a Spark DataFrame. However, we often have multiple related Spark tables which we need to combine prior to performing data manipulations.

A workflow was developed in Section 5.2.1 to find the flights with a departure delay greater than 1000 minutes. However, we did not have the carrier names since they were in a different table. Providing this information can be done with a `left_join`.

```
flights_sdf %>%
  left_join(airlines_sdf, by = "carrier") %>%
  select(carrier, name, flight, year:day, arr_delay, dep_delay) %>%
  filter(dep_delay > 1000) %>%
  arrange(desc(dep_delay))
```

```
## # Source:     spark<?> [?? x 8]
## # Ordered by: desc(dep_delay)
##   carrier name                   flight  year month   day arr_delay dep_delay
##   <chr>   <chr>                   <int> <int> <int> <int>     <dbl>     <dbl>
## 1 HA      Hawaiian Airlines Inc.     51  2013     1     9      1272      1301
## 2 MQ      Envoy Air                3535  2013     6    15      1127      1137
## 3 MQ      Envoy Air                3695  2013     1    10      1109      1126
## 4 AA      American Airlines Inc.    177  2013     9    20      1007      1014
## 5 MQ      Envoy Air                3075  2013     7    22       989      1005
```

Notice that three of the top five largest delays were associated with Envoy Air, which was not obvious based on the two-letter abbreviation.

`dplyr` has various verbs that combine two tables. If this is not adequate, then the joins, or other operations, must be done in the database prior to importing the data into Spark

It is also possible to use Spark DataFrames as tables in a “database” using the Spark SQL interface, which forms the basis of Spark DataFrames.

The `spark_connect` object implements a DBI interface for Spark, which allows you to use `dbGetQuery` to execute SQL commands. The returned result is an R data frame.

We now show that the above workflow can be done in R except that R data frames are used.

```
library(DBI)
flights_df <- dbGetQuery(sc, "SELECT * FROM flights")
airlines_df <- dbGetQuery(sc, "SELECT * FROM airlines")
```

```r
flights_df %>%
  left_join(airlines_df, by = "carrier") %>%
  select(carrier, name, flight, year:day, arr_delay, dep_delay) %>%
  filter(dep_delay > 1000) %>%
  arrange(desc(dep_delay))
```

```
##   carrier                   name flight year month day arr_delay dep_delay
## 1      HA Hawaiian Airlines Inc.     51 2013     1   9      1272      1301
## 2      MQ              Envoy Air   3535 2013     6  15      1127      1137
## 3      MQ              Envoy Air   3695 2013     1  10      1109      1126
## 4      AA American Airlines Inc.    177 2013     9  20      1007      1014
## 5      MQ              Envoy Air   3075 2013     7  22       989      1005
```

Of course, this assumes the Spark DataFrames can be imported into R, i.e., they must fit into local memory.

The `by` argument in the `left_join` is not needed if there is a single variable common to both tables. Alternately, we could use `by = c("carrier", "carrier")`, where the names could be different if they represent the same variable.

We can sample random rows of a Spark DataFrame using:

- `sample_n` for a fixed number;
- `sample_frac` for a fixed fraction.

```r
sample_n(flights_sdf, 10)
```

```
## # Source: spark<?> [?? 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
## # ... with 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>
```

```r
sample_frac(flights_sdf, 0.01)
```

```
## # Source: spark<?> [?? 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      645            647        -2      815            810
##  2  2013     1     1      840            845        -5     1053           1102
##  3  2013     1     1      857            905        -8     1107           1120
##  4  2013     1     1     1044           1045        -1     1231           1212
##  5  2013     1     1     1101           1043        18     1345           1332
##  6  2013     1     1     1217           1220        -3     1414           1350
##  7  2013     1     1     1512           1518        -6     1805           1823
##  8  2013     1     1     1726           1729        -3     2042           2100
```

```
##  9  2013     1     1     1757           1703        54     1904           1813
## 10  2013     1     1     1909           1910        -1     2212           2224
## # ... with more rows, and 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>
```

Sampling is often done during the development and testing cycle to limit the size of the data.

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

```r
# 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
```

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

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

```
## [1] "airlines" "faithful" "flights"
```

```r
# 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.

`dplyr` is an R package for performing operations on structured data. The data is always a table-like structure, i.e., an R `data.frame` (or tibble), a SQL data table, or a Spark DataFrame among others. Ideally, the structure should be in tidy form, i.e., each row is an observation and each column is a variable. Tidy data matches its semantics with how it is stored.

Besides providing functions for manipulating data frames in R, `dplyr` forms an interface for manipulating DataFrames directly in Spark using R. The user can performs operations on Spark DataFrames such as:

- selecting, filtering, and aggregating;
- sampling (by window functions);
- performing joins;

As we will see in Sections 5.2.1 and 5.2.2 below, `dplyr` can be used to:

- convert R data frames to Spark DataFrames using the `copy_to` function, or
- convert Spark DataFames to R data frames using the `collect` function.

Perhaps the most powerful feature of `dplyr` is its support for building data-science workflows in both R and Spark using the forward-pipe operator (`%>%`) from the `magrittr` package.

`dplyr` verbs manipulate structured data in the form of tables. When the tables are Spark DataFrames, `dplyr` translates the commands to Spark SQL statements. The `dplyr`'s five verbs and their SQL equivalents are:

- `select` (SELECT);
- `filter` (WHERE);
- `arrange` (ORDER);
- `summarise` (aggregators such as sum, min, etc.);
- `mutate` (operators such as +, *, log, etc.).

We use the `flights` data from the `nycflights13` packages to illustrate some of the `dplyr` verbs. First, we copy the `flights` and the airlines data frames to Spark.

```
library(nycflights13)
flights_sdf <- copy_to(sc, flights, "flights_sdf", overwrite = TRUE)
airlines_sdf <- copy_to(sc, airlines, "airlines_sdf", overwrite = TRUE)
src_tbls(sc)
```

```
## [1] "airlines"     "airlines_sdf" "faithful"     "flights"      "flights_sdf"
```

By default these Spark DataFrames are cached into memory, but they are not partitioned across nodes. Note that we have used `sdf` as a suffix for Spark DataFrames to distinguish them from R data frames, which either have no suffix or use `df`.

Suppose we want to find the flights with a departure delay greater than 1000 minutes with supporting information about the flight.

```
select(flights_sdf,  carrier, flight, year:day, arr_delay, dep_delay) %>%
  filter(dep_delay > 1000) %>%
  arrange(desc(dep_delay))
```

```
## # Source:     spark<?> [?? x 7]
## # Ordered by: desc(dep_delay)
##   carrier flight  year month   day arr_delay dep_delay
##   <chr>    <int> <int> <int> <int>     <dbl>     <dbl>
```

```
## 1 HA          51  2013     1     9       1272       1301
## 2 MQ        3535  2013     6    15       1127       1137
## 3 MQ        3695  2013     1    10       1109       1126
## 4 AA         177  2013     9    20       1007       1014
## 5 MQ        3075  2013     7    22        989       1005
```

Here we are building a Spark workflow using `magrittr` pipes, which is a strong feature of R for building data science workflows. If the full name of the carrier is wanted, we need to join `flights_sdf` with `airlines_sdf`. This will be done in the next section.

The average delay for all flights is computed with the `summarise` verb:

```r
summarise(flights_sdf, mean(dep_delay))
```

```
## # Source: spark<?> [?? x 1]
##   `mean(dep_delay)`
##               <dbl>
## 1              12.6
```

Thus, the average delay for all flights is 12.64 minutes.

We can use `mutate` together with `summarise` to compute the average speed:

```r
mutate(flights_sdf, speed = distance / air_time * 60) %>%
  summarise(mean(speed))
```

```
## # Source: spark<?> [?? x 1]
##   `mean(speed)`
##           <dbl>
## 1          394.
```

The average speed is 394.27 miles/hour.

`dplyr` evaluates lazily, i.e., it:

- does not pull data into R until you ask for it;
- delays doing work until required.

We pull data into R using the `collect` function.

The average delay computed above keeps the computation in Spark whether or not we explicitly assign the result to a Spark DataFrame. Consider:

```r
mean_dep_delay_sdf <- summarise(flights_sdf, mean(dep_delay))
mean_dep_delay_sdf # this statement causes Spark to evaluate the above expression
```

```
## # Source: spark<?> [?? x 1]
##   `mean(dep_delay)`
##               <dbl>
## 1              12.6
```

```r
class(mean_dep_delay_sdf)
```

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

The result is identical to the computation above, but here we can explore the structure of `mean_dep_delay_sdf`. Notice its inheritance path. `mean_dep_delay_sdf` is the tibble version of a Spark DataFrame, which is a type of SQL tibble, which is a lazy tibble, i.e., not evaluated from the first statement in the chunk.

Next we collect `mean_dep_delay_sdf` into R and get an R data frame.

```r
mean_dep_delay <- collect(mean_dep_delay_sdf)
mean_dep_delay
```

```
## # A tibble: 1 x 1
##   `mean(dep_delay)`
##               <dbl>
## 1              12.6
```

```r
class(mean_dep_delay)
```

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

Here, the tibble data frame inherits from tibble, which in turn is a type of `data.frame`.

The `group_by` function allows us to perform calculations for the groups (or levels) of a variable.

Suppose we want to compare the departure delays for `AA` (American Airlines), `DL` (Delta Air Lines), and `UA` (United Air Lines) for the month of May.

```r
carrier_dep_delay_sdf <- flights_sdf %>%
  filter(month == 5, carrier %in% c('AA', 'DL', 'UA')) %>%
  select(carrier, dep_delay) %>%
  arrange(carrier)
carrier_dep_delay_sdf
```

```
## # Source:     spark<?> [?? x 2]
## # Ordered by: carrier
##    carrier dep_delay
##    <chr>       <dbl>
##  1 AA             -5
##  2 AA             -7
##  3 AA              0
##  4 AA              0
##  5 AA             -4
##  6 AA             -6
##  7 AA             -3
##  8 AA             -5
##  9 AA             -2
## 10 AA             -7
## # ... with more rows
```

The `arrange` statement in the above workflow is not advised since it causes Spark shuffling, but is given here to illustrate the verb. At this point we have only subsetted the Spark DataFrame by filtering rows and selecting columns.

Next we group-by carrier and summarise the results.

```r
carrier_dep_delay_sdf %>%
  group_by(carrier) %>%
  summarise(count =n(), mean_dep_delay = mean(dep_delay))
```

```
## # Source: spark<?> [?? x 3]
##   carrier count mean_dep_delay
##   <chr>   <dbl>          <dbl>
## 1 AA       2803           9.66
## 2 DL       4082           9.74
## 3 UA       4960          12.3
```

The `group_by` function seems innocent enough, but it may not be so. It has some of the same problems as Hadoop. Hadoop is terrible for complex workflows since data is constantly read from and written to HDFS and each cycle of MapReduce involves the dreaded shuffle.

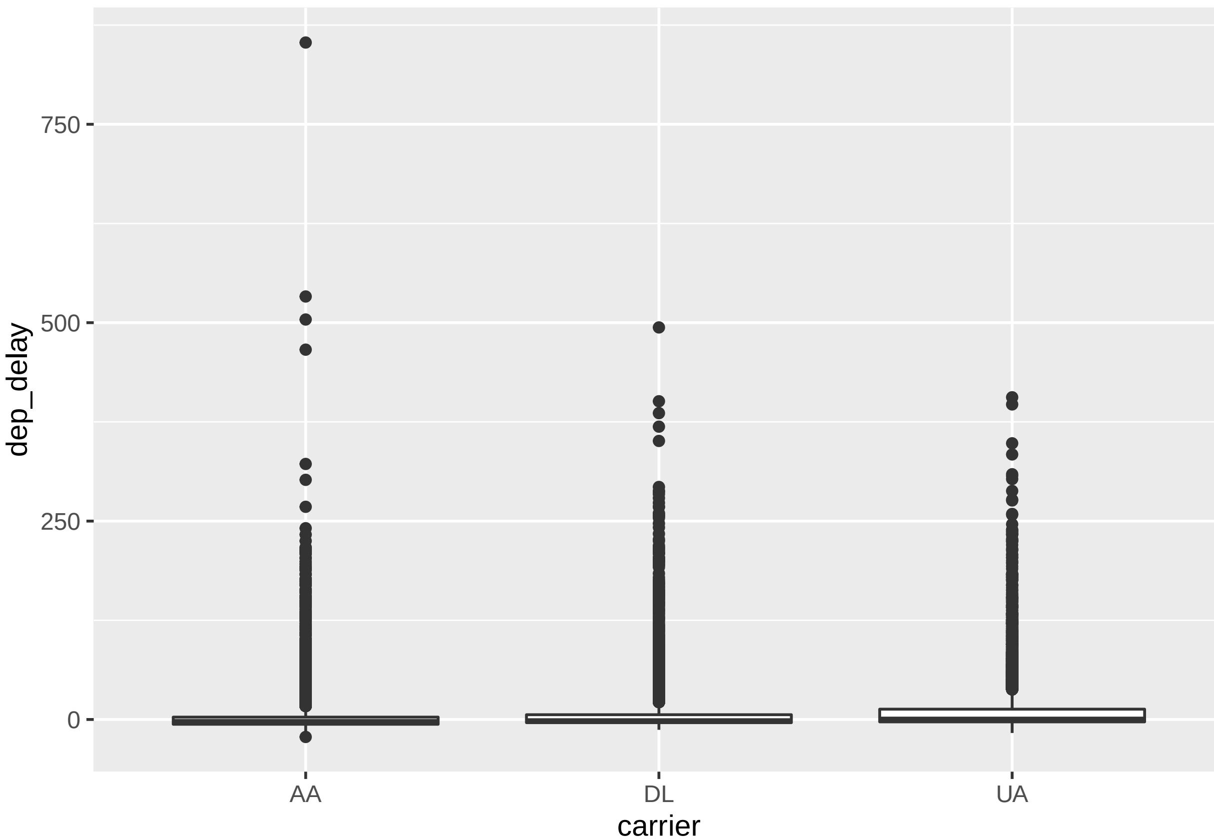
Unless data is spread among the nodes of a cluster by group, which is not likely, then the data will need to be moved for analysis by shuffling it. This can be time consuming and should be avoided if possible, e.g., by partitioning according to groups in the first place.

R has powerful statistical functions through a huge number of R packages. We can take advantage of R by converting a Spark DataFrame into an R data frame and then do modeling and plotting using the `dplyr`'s `collect` function.

```
carrier_dep_delay_df <- collect(carrier_dep_delay_sdf)
```

```
library(ggplot2)
carrier_dep_delay_df %>%
  ggplot(aes(carrier, dep_delay)) + geom_boxplot()
```

```
## Warning: Removed 89 rows containing non-finite values (stat_boxplot).
```



An aggregation function, such as `mean()`, takes n inputs and return a single value, whereas a window function returns n values. The output of a window function depends on all its input values, so window functions don't include functions that work element-wise, like `+` or `round()`. Window functions in R include variations on aggregate functions, like `cummean()`, functions for ranking and ordering, like `rank()`, and functions for taking offsets, like `lead()` and `lag()`.

Similarly, Spark supports certain window functions.

```
spark_disconnect(sc)
```