Spark dplyr

Jim Harner

5/30/2020

Load `sparklyr` and establish the Spark connection.

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

# start the sparklyr session
master <- "local"
# master <- "spark://master:7077"
sc <- spark_connect(master = master)
```

`sparklyr` has a `dplyr` compatible back-end to Spark.

5.2 Spark dplyr

`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.

5.2.1 `dplyr` Verbs

`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_sdf" "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:

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

```
## 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: 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:

```
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.

5.2.2 Laziness

`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`.

5.2.3 Grouping and Shuffling

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.

```
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.

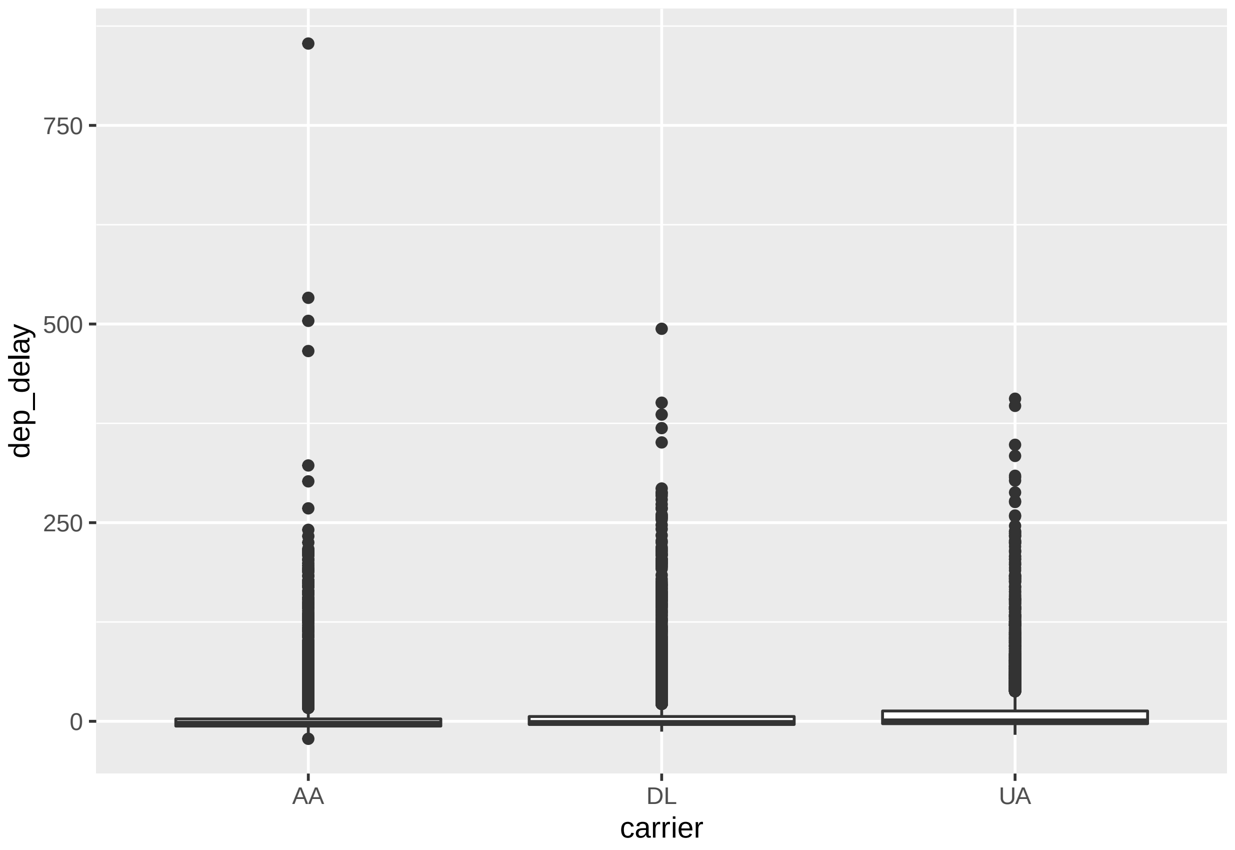
5.2.4 Analyses in R

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).
```



5.2.5 Window Functions

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)
```