**Introduction**

In the [first part](https://jozef.io/r201-spark-r-1/) of this series, we looked at how the sparklyr interface communicates with the Spark instance and what this means for performance with regards to arbitrarily defined R functions. We also examined how Apache Arrow can increase the performance of data transfers between the R session and the Spark instance.

In this second part, we will look at how to write R functions that can be executed directly by Spark without serialization overhead that we have shown in the previous installment. We will focus on writing functions as combinations of dplyr verbs that can be translated using dbplyr and investigate how the SQL is generated and Spark plans created.

**Contents**

1. [Preparation](https://jozef.io/r202-spark-r-dplyr-verbs/#preparation)
2. [R functions as combinations of dplyr verbs and Spark](https://jozef.io/r202-spark-r-dplyr-verbs/#r-functions-as-combinations-of-dplyr-verbs-and-spark)
   1. [Trying it with base R functions](https://jozef.io/r202-spark-r-dplyr-verbs/#trying-it-with-base-r-functions)
   2. [Using a combination of supported dplyr verbs and operations](https://jozef.io/r202-spark-r-dplyr-verbs/#using-a-combination-of-supported-dplyr-verbs-and-operations)
   3. [Investigating the SQL translation and its Spark plan](https://jozef.io/r202-spark-r-dplyr-verbs/#investigating-the-sql-translation-and-its-spark-plan)
3. [A more complex use case – Joins, group bys, and aggregations](https://jozef.io/r202-spark-r-dplyr-verbs/#a-more-complex-use-case---joins-group-bys-and-aggregations)
4. [Using the functions with local versus remote datasets](https://jozef.io/r202-spark-r-dplyr-verbs/#using-the-functions-with-local-versus-remote-datasets)
5. [The take-home message](https://jozef.io/r202-spark-r-dplyr-verbs/#the-take-home-message)
6. [References](https://jozef.io/r202-spark-r-dplyr-verbs/#references)

**Preparation**

The full setup of Spark and sparklyr is not in the scope of this post, please check the [previous one](https://jozef.io/r201-spark-r-1/#setting-up-spark-with-r-and-sparklyr) for some setup instructions and a ready-made Docker image.

If you have docker available, running

docker run -d -p 8787:8787 -e PASSWORD=pass --name rstudio jozefhajnala/sparkly:add-rstudio

Should make RStudio available by navigating to [http://localhost:8787](http://localhost:8787/) in your browser. You can then use the user name rstudio and password pass to login and continue experimenting with the code in this post.



Apache Spark and R logos

First, we will attach the needed packages and copy some test data from the nycflights13 package into our local Spark instance:

# Load packages

suppressPackageStartupMessages({

library(sparklyr)

library(dplyr)

library(nycflights13)

})

# Prepare the data

weather <- nycflights13::weather %>%

mutate(id = 1L:nrow(nycflights13::weather)) %>%

select(id, everything())

# Connect

sc <- sparklyr::spark\_connect(master = "local")

# Copy the weather dataset to the instance

tbl\_weather <- dplyr::copy\_to(

dest = sc,

df = weather,

name = "weather",

overwrite = TRUE

)

# Copy the flights dataset to the instance

tbl\_flights <- dplyr::copy\_to(

dest = sc,

df = nycflights13::flights,

name = "flights",

overwrite = TRUE

)

**R functions as combinations of dplyr verbs and Spark**

One of the approaches to retain the performance of Spark with arbitrary R functionality is to carefully design our functions such that in its entirety when using it with sparklyr, the function call can be translated directly to Spark SQL using dbplyr.

This allows us to write, package, test, and document the functions as we normally would, while still getting the performance benefits of Apache Spark.

Let’s look at an example where we would like to do simple transformations of data stored in a column of a data frame, such as normalization of one of the columns. For illustration purposes, we will normalize the values of a column by first subtracting the mean value and then dividing the values by the standard deviation.

**Trying it with base R functions**

The first attempt could be quite simple, we could attempt to take advantage of R’s base function scale() to do the work for us:

normalize\_dplyr\_scale <- function(df, col, newColName) {

df %>% mutate(!!newColName := scale({{col}}))

}

This function would work fine with a local data frame such as weather:

weather %>%

normalize\_dplyr\_scale(temp, "normTemp") %>%

select(id, temp, normTemp)

## # A tibble: 26,115 x 3

## id temp normTemp[,1]

##

## 1 1 39.0 -0.913

## 2 2 39.0 -0.913

## 3 3 39.0 -0.913

## 4 4 39.9 -0.862

## 5 5 39.0 -0.913

## 6 6 37.9 -0.974

## 7 7 39.0 -0.913

## 8 8 39.9 -0.862

## 9 9 39.9 -0.862

## 10 10 41 -0.802

## # … with 26,105 more rows

However for a Spark DataFrame this would throw an error. This is because the base R function scale() is not translated by dbplyr at the moment and it is not a Hive built-in function either:

tbl\_weather %>%

normalize\_dplyr\_scale(temp, "normTemp") %>%

select(id, temp, normTemp)

Error: org.apache.spark.sql.AnalysisException: Undefined function: 'scale'.

**Using a combination of supported dplyr verbs and operations**

To run the function successfully, we will need to rewrite it as a combination of functions and operations that are supported by the dbplyr translation to Spark SQL. One example implementation is as follows:

normalize\_dplyr <- function(df, col, newColName) {

df %>% mutate(

!!newColName := ({{col}} - mean({{col}}, na.rm = TRUE)) /

sd({{col}}, na.rm = TRUE)

)

}

Using this function yields the desired results for both local and Spark data frames:

# Local data frame

weather %>%

normalize\_dplyr(temp, "normTemp") %>%

select(id, temp, normTemp)

## # A tibble: 26,115 x 3

## id temp normTemp

##

## 1 1 39.0 -0.913

## 2 2 39.0 -0.913

## 3 3 39.0 -0.913

## 4 4 39.9 -0.862

## 5 5 39.0 -0.913

## 6 6 37.9 -0.974

## 7 7 39.0 -0.913

## 8 8 39.9 -0.862

## 9 9 39.9 -0.862

## 10 10 41 -0.802

## # … with 26,105 more rows

# Spark DataFrame

tbl\_weather %>%

normalize\_dplyr(temp, "normTemp") %>%

select(id, temp, normTemp) %>%

collect()

## # A tibble: 26,115 x 3

## id temp normTemp

##

## 1 1 39.0 -0.913

## 2 2 39.0 -0.913

## 3 3 39.0 -0.913

## 4 4 39.9 -0.862

## 5 5 39.0 -0.913

## 6 6 37.9 -0.974

## 7 7 39.0 -0.913

## 8 8 39.9 -0.862

## 9 9 39.9 -0.862

## 10 10 41 -0.802

## # … with 26,105 more rows

**Investigating the SQL translation and its Spark plan**

Another advantage of this approach is that we can investigate the plan by which the actions will be executed by Spark using the explain() function from the dplyr package. This will print both the SQL query constructed by dbplyr and the plan generated by Spark, which can help us investigate performance issues:

tbl\_weather %>%

normalize\_dplyr(temp, "normTemp") %>%

dplyr::explain()

##

## SELECT `id`, `origin`, `year`, `month`, `day`, `hour`, `temp`, `dewp`, `humid`, `wind\_dir`, `wind\_speed`, `wind\_gust`, `precip`, `pressure`, `visib`, `time\_hour`, (`temp` - AVG(`temp`) OVER ()) / stddev\_samp(`temp`) OVER () AS `normTemp`

## FROM `weather`

##

##

## == Physical Plan ==

## \*(1) Project [id#24, origin#25, year#26, month#27, day#28, hour#29, temp#30, dewp#31, humid#32, wind\_dir#33, wind\_speed#34, wind\_gust#35, precip#36, pressure#37, visib#38, time\_hour#39, ((temp#30 - \_we0#948) / \_we1#949) AS normTemp#934]

## +- Window [avg(temp#30) windowspecdefinition(specifiedwindowframe(RowFrame, unboundedpreceding$(), unboundedfollowing$())) AS \_we0#948, stddev\_samp(temp#30) windowspecdefinition(specifiedwindowframe(RowFrame, unboundedpreceding$(), unboundedfollowing$())) AS \_we1#949]

## +- Exchange SinglePartition

## +- InMemoryTableScan [id#24, origin#25, year#26, month#27, day#28, hour#29, temp#30, dewp#31, humid#32, wind\_dir#33, wind\_speed#34, wind\_gust#35, precip#36, pressure#37, visib#38, time\_hour#39]

## +- InMemoryRelation [id#24, origin#25, year#26, month#27, day#28, hour#29, temp#30, dewp#31, humid#32, wind\_dir#33, wind\_speed#34, wind\_gust#35, precip#36, pressure#37, visib#38, time\_hour#39], StorageLevel(disk, memory, deserialized, 1 replicas)

## +- Scan ExistingRDD[id#24,origin#25,year#26,month#27,day#28,hour#29,temp#30,dewp#31,humid#32,wind\_dir#33,wind\_speed#34,wind\_gust#35,precip#36,pressure#37,visib#38,time\_hour#39]

If we are only interested in the SQL itself as a character string, we can use dbplyr’s sql\_render():

tbl\_weather %>%

normalize\_dplyr(temp, "normTemp") %>%

dbplyr::sql\_render() %>%

unclass()

## [1] "SELECT `id`, `origin`, `year`, `month`, `day`, `hour`, `temp`, `dewp`, `humid`, `wind\_dir`, `wind\_speed`, `wind\_gust`, `precip`, `pressure`, `visib`, `time\_hour`, (`temp` - AVG(`temp`) OVER ()) / stddev\_samp(`temp`) OVER () AS `normTemp`\nFROM `weather`"

**A more complex use case – Joins, group bys, and aggregations**

The dplyr syntax makes it very easy to construct more complex aggregations across multiple Spark DataFrames. An example of a function that joins 2 Spark DataFrames and computes a mean of a selected column, grouped by another column can look as follows:

joingrpagg\_dplyr <- function(

df1, df2,

joinColNames = intersect(colnames(df1), colnames(df2)),

col, groupCol

) {

df1 %>%

right\_join(df2, by = joinColNames) %>%

group\_by({{groupCol}}) %>%

summarise(mean({{col}})) %>%

arrange({{groupCol}})

}

We can then use this function for instance to look at the mean arrival delay of flights grouped by visibility. Note that we are only collecting heavily aggregated data – 20 rows in total. The overhead of data transfer from the Spark instance to the R session is therefore small. Also, just assigning the function call to delay\_by\_visib does not actually execute or collect anything, execution really starts only when collect() is called:

delay\_by\_visib <- joingrpagg\_dplyr(

tbl\_flights, tbl\_weather,

col = arr\_delay, groupCol = visib

)

delay\_by\_visib %>% collect()

## 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.

## # A tibble: 20 x 2

## visib `mean(arr\_delay)`

##

## 1 0 24.9

## 2 0.06 28.5

## 3 0.12 45.4

## 4 0.25 20.8

## 5 0.5 39.8

## 6 0.75 41.4

## 7 1 37.6

## 8 1.25 65.1

## 9 1.5 34.7

## 10 1.75 45.6

## 11 2 26.3

## 12 2.5 21.7

## 13 3 21.7

## 14 4 17.7

## 15 5 18.9

## 16 6 17.3

## 17 7 16.4

## 18 8 16.1

## 19 9 15.6

## 20 10 4.32

We can look at the plan and the generated SQL query as well:

delay\_by\_visib %>% dplyr::explain()

##

## SELECT `visib`, AVG(`arr\_delay`) AS `mean(arr\_delay)`

## FROM (SELECT `RHS`.`year` AS `year`, `RHS`.`month` AS `month`, `RHS`.`day` AS `day`, `LHS`.`dep\_time` AS `dep\_time`, `LHS`.`sched\_dep\_time` AS `sched\_dep\_time`, `LHS`.`dep\_delay` AS `dep\_delay`, `LHS`.`arr\_time` AS `arr\_time`, `LHS`.`sched\_arr\_time` AS `sched\_arr\_time`, `LHS`.`arr\_delay` AS `arr\_delay`, `LHS`.`carrier` AS `carrier`, `LHS`.`flight` AS `flight`, `LHS`.`tailnum` AS `tailnum`, `RHS`.`origin` AS `origin`, `LHS`.`dest` AS `dest`, `LHS`.`air\_time` AS `air\_time`, `LHS`.`distance` AS `distance`, `RHS`.`hour` AS `hour`, `LHS`.`minute` AS `minute`, `RHS`.`time\_hour` AS `time\_hour`, `RHS`.`id` AS `id`, `RHS`.`temp` AS `temp`, `RHS`.`dewp` AS `dewp`, `RHS`.`humid` AS `humid`, `RHS`.`wind\_dir` AS `wind\_dir`, `RHS`.`wind\_speed` AS `wind\_speed`, `RHS`.`wind\_gust` AS `wind\_gust`, `RHS`.`precip` AS `precip`, `RHS`.`pressure` AS `pressure`, `RHS`.`visib` AS `visib`

## FROM `flights` AS `LHS`

## RIGHT JOIN `weather` AS `RHS`

## ON (`LHS`.`year` = `RHS`.`year` AND `LHS`.`month` = `RHS`.`month` AND `LHS`.`day` = `RHS`.`day` AND `LHS`.`origin` = `RHS`.`origin` AND `LHS`.`hour` = `RHS`.`hour` AND `LHS`.`time\_hour` = `RHS`.`time\_hour`)

## ) `dbplyr\_003`

## GROUP BY `visib`

## ORDER BY `visib`

##

##

## == Physical Plan ==

## \*(6) Sort [visib#38 ASC NULLS FIRST], true, 0

## +- Exchange rangepartitioning(visib#38 ASC NULLS FIRST, 2)

## +- \*(5) HashAggregate(keys=[visib#38], functions=[avg(arr\_delay#409)])

## +- Exchange hashpartitioning(visib#38, 2)

## +- \*(4) HashAggregate(keys=[visib#38], functions=[partial\_avg(arr\_delay#409)])

## +- \*(4) Project [arr\_delay#409, visib#38]

## +- SortMergeJoin [cast(year#401 as double), cast(month#402 as double), day#403, origin#413, hour#417, time\_hour#419], [year#26, month#27, day#28, origin#25, cast(hour#29 as double), time\_hour#39], RightOuter

## :- \*(2) Sort [cast(year#401 as double) ASC NULLS FIRST, cast(month#402 as double) ASC NULLS FIRST, day#403 ASC NULLS FIRST, origin#413 ASC NULLS FIRST, hour#417 ASC NULLS FIRST, time\_hour#419 ASC NULLS FIRST], false, 0

## : +- Exchange hashpartitioning(cast(year#401 as double), cast(month#402 as double), day#403, origin#413, hour#417, time\_hour#419, 2)

## : +- \*(1) Filter (((((isnotnull(month#402) && isnotnull(day#403)) && isnotnull(origin#413)) && isnotnull(year#401)) && isnotnull(time\_hour#419)) && isnotnull(hour#417))

## : +- InMemoryTableScan [year#401, month#402, day#403, arr\_delay#409, origin#413, hour#417, time\_hour#419], [isnotnull(month#402), isnotnull(day#403), isnotnull(origin#413), isnotnull(year#401), isnotnull(time\_hour#419), isnotnull(hour#417)]

## : +- InMemoryRelation [year#401, month#402, day#403, dep\_time#404, sched\_dep\_time#405, dep\_delay#406, arr\_time#407, sched\_arr\_time#408, arr\_delay#409, carrier#410, flight#411, tailnum#412, origin#413, dest#414, air\_time#415, distance#416, hour#417, minute#418, time\_hour#419], StorageLevel(disk, memory, deserialized, 1 replicas)

## : +- Scan ExistingRDD[year#401,month#402,day#403,dep\_time#404,sched\_dep\_time#405,dep\_delay#406,arr\_time#407,sched\_arr\_time#408,arr\_delay#409,carrier#410,flight#411,tailnum#412,origin#413,dest#414,air\_time#415,distance#416,hour#417,minute#418,time\_hour#419]

## +- \*(3) Sort [year#26 ASC NULLS FIRST, month#27 ASC NULLS FIRST, day#28 ASC NULLS FIRST, origin#25 ASC NULLS FIRST, cast(hour#29 as double) ASC NULLS FIRST, time\_hour#39 ASC NULLS FIRST], false, 0

## +- Exchange hashpartitioning(year#26, month#27, day#28, origin#25, cast(hour#29 as double), time\_hour#39, 2)

## +- InMemoryTableScan [origin#25, year#26, month#27, day#28, hour#29, visib#38, time\_hour#39]

## +- InMemoryRelation [id#24, origin#25, year#26, month#27, day#28, hour#29, temp#30, dewp#31, humid#32, wind\_dir#33, wind\_speed#34, wind\_gust#35, precip#36, pressure#37, visib#38, time\_hour#39], StorageLevel(disk, memory, deserialized, 1 replicas)

## +- Scan ExistingRDD[id#24,origin#25,year#26,month#27,day#28,hour#29,temp#30,dewp#31,humid#32,wind\_dir#33,wind\_speed#34,wind\_gust#35,precip#36,pressure#37,visib#38,time\_hour#39]

**Using the functions with local versus remote datasets**

Some of the appeal of the dplyr syntax comes from the fact that we can use the same functions to conveniently manipulate local data frames in memory and, with the very same code, data from remote sources such as relational databases, data.tables and even data within Spark.

This unified front-end, however, comes with some important differences that we must be aware of when applying and porting code from using it to manipulate and compute on local data versus on remote sources. The same holds for remote Spark DataFrames that we are manipulating when using dplyr functions.

An example of a different behavior is joining. The very simplest example – trying to inner join two tables can lead to a different amount of rows for the remote Spark DataFrames and the local R data frames:

bycols <- c("year", "month", "day", "origin", "hour", "time\_hour")

# Look at count of rows of Inner join of the Spark data frames

tbl\_flights %>% inner\_join(tbl\_weather, by = bycols) %>% count()

## # Source: spark [?? x 1]

## n

##

## 1 335096

# Look at count of rows of Inner join of the local data frames

flights %>% inner\_join(weather, by = bycols) %>% count()

## # A tibble: 1 x 1

## n

##

## 1 335220

Another example of differences can arise from handling NA and NaN values:

# Create (lazy) left joins

joined\_spark <- tbl\_flights %>% left\_join(tbl\_weather, by = bycols) %>% collect()

joined\_local <- flights %>% left\_join(weather, by = bycols)

# Look at counts of NA values

joined\_local %>% filter([is.na](http://is.na)(temp)) %>% count()

## # A tibble: 1 x 1

## n

##

## 1 1573

joined\_spark %>% filter([is.na](http://is.na)(temp)) %>% count()

## # A tibble: 1 x 1

## n

##

## 1 1697

# Look at counts of NaN values

joined\_local %>% filter(is.nan(temp)) %>% count()

## # A tibble: 1 x 1

## n

##

## 1 0

joined\_spark %>% filter(is.nan(temp)) %>% count()

## # A tibble: 1 x 1

## n

##

## 1 1697

Special care must also be taken when dealing with date/time values and their time zones:

# Note the time\_hour values are different

weather %>% select(id, time\_hour)

## # A tibble: 26,115 x 2

## id time\_hour

##

## 1 1 2013-01-01 01:00:00

## 2 2 2013-01-01 02:00:00

## 3 3 2013-01-01 03:00:00

## 4 4 2013-01-01 04:00:00

## 5 5 2013-01-01 05:00:00

## 6 6 2013-01-01 06:00:00

## 7 7 2013-01-01 07:00:00

## 8 8 2013-01-01 08:00:00

## 9 9 2013-01-01 09:00:00

## 10 10 2013-01-01 10:00:00

## # … with 26,105 more rows

tbl\_weather %>% select(id, time\_hour)

## # Source: spark [?? x 2]

## id time\_hour

##

## 1 1 2013-01-01 06:00:00

## 2 2 2013-01-01 07:00:00

## 3 3 2013-01-01 08:00:00

## 4 4 2013-01-01 09:00:00

## 5 5 2013-01-01 10:00:00

## 6 6 2013-01-01 11:00:00

## 7 7 2013-01-01 12:00:00

## 8 8 2013-01-01 13:00:00

## 9 9 2013-01-01 14:00:00

## 10 10 2013-01-01 15:00:00

## # … with more rows

And, rather obviously, when using Hive built-in functions in the dplyr-based function, we will most likely not be able to execute it on the local data frames, as we have [seen previously](https://jozef.io/r201-spark-r-1/#a-hive-built-in-function-not-existing-in-r).

**The take-home message**

In this part of the series, we have shown that we can take advantage of the performance of Spark while still writing arbitrary R functions by using dplyr syntax, which supports translation to Spark SQL using the dbplyr backend. We have also looked at some important differences when applying the same dplyr transformations to local and remote data sets.

With this approach, we can use R development best practices, testing, and documentation methods in a standard way when writing our R packages, getting the best of both worlds – Apache Spark for performance and R for convenient development of data science applications.

In the next installment, we will look at writing R functions that will be using SQL directly, instead of relying on dbplyr for the translation, and how we can efficiently send them to the Spark instance for execution and optionally retrieve the results to our R session.

**References**

* The [first part](https://jozef.io/r201-spark-r-1/) of this series
* Documentation on [Hive Operators and User-Defined Functions](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+UDF) website.
* A [Docker image](https://hub.docker.com/r/jozefhajnala/sparkly) with R, Spark, sparklyr and Arrow available and [its Dockerfile](https://gitlab.com/jozefhajnala/dockerfiles/blob/master/sparkly/Dockerfile).
* Overview of the [dplyr syntax](https://dplyr.tidyverse.org/)