**Introduction**

In this 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.

**Preparation**

The full setup of Spark and sparklyr is not in the scope of this post, please check the below for some setup instructions and a ready-made Docker image.

**Setting up Spark with R and sparklyr**

The full instructions on setting up sparklyr are not in the scope of this article, below we only provide a quick set of instructions to get a local Spark instance working with sparklyr.



Apache Spark and R logos

**Using a ready-made Docker Image**

For the purpose of this series, a Docker image was built which you can use to experiment in the following ways by running one of the commands below within a terminal. If you are using RStudio 1.1 or newer, Terminal functionality is built into RStudio itself.

**Interactively with R and sparklyr**

Running the following should yield an interactive R session with all prerequisites to start working with the sparklyr package using a local Spark instance.

docker run --rm -it jozefhajnala/sparkly:test R

# Start using sparklyr

library(sparklyr)

sc <- spark\_connect("local")

**Interactively with the Spark shell**

Running the following should yield an interactive Scala REPL instance. A Spark context should be available as sc and a Spark session as spark.

docker run --rm -it jozefhajnala/sparkly:test /root/spark/spark-2.4.3-bin-hadoop2.7/bin/spark-shell

**Running an example R script**

Running the following should execute an example R script using sparklyr with output appearing in the terminal:

docker run --rm jozefhajnala/sparkly:test Rscript /root/.local/spark\_script.R

**Manual Installation**

The following are very basic instructions, for troubleshooting or more detailed step-by-step guides you can refer to RStudio’s spark website.

install.packages("sparklyr")

install.packages("nycflights13")

sparklyr::spark\_install(version = "2.4.3")

**Connecting and using a local Spark instance**

# Load packages

library(sparklyr)

library(dplyr)

library(nycflights13)

# Connect

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

# Copy the weather dataset to the instance

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

dest = sc,

df = nycflights13::weather,

name = "weather",

overwrite = TRUE

)

# Collect it back

tbl\_weather %>% collect()

**Sparklyr as a Spark interface provider**

The sparklyr package is an R *interface* to Apache Spark. The meaning of the word interface is very important in this context as the way we use this interface can significantly affect the performance benefits we get from using Spark.

To understand the meaning of the above a bit better, we will examine 3 very simple functions that are different in implementation but intend to provide the same results, and how they behave with regards to Spark. We will use datasets from the nycflights13 package for our examples.

**An R function translated to Spark SQL**

Using the following fun\_implemented() function will yield the expected results for both a local data frame nycflights13::weather and the remote Spark object referenced by tbl\_weather:

# An R function translated to Spark SQL

fun\_implemented <- function(df, col) {

df %>% mutate({{col}} := tolower({{col}}))

}

fun\_implemented(nycflights13::weather, origin)

fun\_implemented(tbl\_weather, origin)

This is because the R function tolower was translated by dbplyr to Spark SQL function LOWER and the resulting query was sent to Spark to be executed. We can see the actual translated SQL by running sql\_render() on the function call:

dbplyr::sql\_render(

fun\_implemented(tbl\_weather, origin)

)

<SQL> SELECT LOWER(`origin`) AS `origin`, `year`, `month`, `day`, `hour`,

`temp`, `dewp`, `humid`, `wind\_dir`, `wind\_speed`, `wind\_gust`, `precip`,

`pressure`, `visib`, `time\_hour`

FROM `weather`

**An R function not translated to Spark SQL**

Using the following fun\_r\_only() function will only yield the expected results for a local data frame nycflights13::weather. For the remote Spark object referenced by tbl\_weather we will get an error:

# An R function not translated to Spark SQL

fun\_r\_only <- function(df, col) {

df %>% mutate({{col}} := casefold({{col}}, upper = FALSE))

}

fun\_r\_only(nycflights13::weather, origin)

fun\_r\_only(tbl\_weather, origin)

Error: org.apache.spark.sql.catalyst.parser.ParseException:

mismatched input 'AS' expecting ')'(line 1, pos 32)

== SQL ==

SELECT casefold(`origin`, FALSE AS `upper`) AS `origin`,

`year`, `month`, `day`, `hour`,

`temp`, `dewp`, `humid`, `wind\_dir`, `wind\_speed`, `wind\_gust`,

`precip`, `pressure`, `visib`, `time\_hour`

--------------------------------^^^

FROM `weather`

This is because there simply is no translation provided by dbplyr for the casefold() function. The generated Spark SQL will therefore not be valid and throw an error once the Spark SQL parser tries to parse it.

**A Hive built-in function not existing in R**

On the other hand, using the below fun\_hive\_builtin() function will only yield the expected results for the remote Spark object referenced by tbl\_weather. For the local data frame nycflights13::weather we will get an error:

# A Hive built-in function not existing in R

fun\_hive\_builtin <- function(df, col) {

df %>% mutate({{col}} := lower({{col}}))

}

fun\_hive\_builtin(tbl\_weather, origin)

fun\_hive\_builtin(nycflights13::weather, origin)

Error: Evaluation error: could not find function "lower".

This is because the function lower does not exist in R itself. For a non-existing R function there obviously is no dbplyr translation either. In this case, dbplyr keeps it as-is when translating to SQL, and the SQL will be valid and executed without problems because lower is, in fact, a function built-in to Hive:

dbplyr::sql\_render(fun\_hive\_builtin(tbl\_weather, origin))

<SQL> SELECT lower(`origin`) AS `origin`,

`year`, `month`, `day`, `hour`,

`temp`, `dewp`, `humid`, `wind\_dir`, `wind\_speed`, `wind\_gust`,

`precip`, `pressure`, `visib`, `time\_hour`

FROM `weather`

**Using non-translated functions with sparklyr**

It can easily happen that one of the functions we want to use falls into the category where it is neither translated or a Hive built-in function. In this case, there is another interface provided by sparklyr that can allow us to do that - the spark\_apply() function. Here is an oversimplified example that will reach our goal with casefold():

fun\_r\_custom <- function(tbl, colName) {

tbl[[colName]] <- casefold(tbl[[colName]], upper = FALSE)

tbl

}

spark\_apply(tbl\_weather, fun\_r\_custom, context = {colName <- "origin"})

**What is so important about this distinction?**

We have now shown that we can also send code that was not translated by dbplyr to Spark and get it executed without issues using spark\_apply(). So what is the catch and where does the importance of the meaning of the word *interface* come in?

Let us quickly examine the performance of the operations:

mb = microbenchmark::microbenchmark(

times = 10,

hive\_builtin = fun\_hive\_builtin(tbl\_weather, origin) %>% collect(),

translated\_dplyr = fun\_implemented(tbl\_weather, origin) %>% collect(),

spark\_apply = spark\_apply(tbl\_weather, fun\_r\_custom, context = {colName <- "origin"}) %>% collect()

)

time (milliseconds)Simple column transformation on a small datasethive\_builtintranslated\_dplyrspark\_apply0100k200k300k400k500k

Note that the absolute values here will vary based on the setup, the important message is in the relative differences.

We can see that the operations executed via the SQL translation mechanism of dbplyr were executed in around *0.5 seconds* while those via spark\_apply took orders of magnitude longer - more than *6 minutes*.

**What happens when we use custom functions with spark\_apply**

We can now see that the operation with spark\_apply() is extremely slow compared to the other two. The key to understanding the difference is to examine how the custom transformations of data using R functions are performed within spark\_apply(). In simplified terms, this happens in a few steps:

1. the data is moved in row-format from Spark into the R process through a socket connection. This is inefficient as multiple data types need to be deserialized over each row
2. the data gets converted to columnar format since this is how R data frames are implemented
3. the R functions are applied to compute the results
4. the results are again converted to row-format, serialized row-by-row and sent back to Spark over the socket connection

**What happens when we use translated or Hive built-in functions**

When using functions that can be translated to Spark SQL the process is very different

* The call is translated to Spark SQL using the dbplyr backend
* The constructed query is sent to Spark for execution using DBI
* Only when collect() or compute() is called, the SQL is executed within Spark
* Only when collect() is called the results are also sent to the R session

This means that the transfer of data only happens once and only when collect() is called, which saves a vast amount of overhead.

**Which R functionality is currently translated and built-in to Hive**

An important question to answer with regards to performance then is what amount of functionality is available using the fast dbplyr backend. As seen above, these features can be categorized into two groups:

1. R functions translatable to Spark SQL via dbplyr. The full list of such functions is available on RStudio’s sparklyr website
2. Hive built-in functions that get translated as they are and can be evaluated by Spark. The full list is available on the Hive Operators and User-Defined Functions website.

**Making serialization faster with Apache Arrow**

**What is Apache Arrow and how it improves performance**

Our benchmarks have shown that using spark\_apply() does not scale well and the penalty of the bottleneck in performance caused by serialization, deserialization, and transfer is too high.

To partially mitigate this we can take advantage of Apache Arrow, a cross-language development platform for in-memory data that specifies a standardized language-independent columnar memory format for flat and hierarchical data.

By adding support for Arrow in sparklyr, it makes Spark perform the row-format to column-format conversion in parallel in Spark, data is then transferred through the socket but no custom serialization takes place and all the R process needs to do is copy this data from the socket into its heap, transform it and copy it back to the socket connection.

This makes the process significantly faster:

mb = microbenchmark::microbenchmark(

times = 10,

setup = library(arrow),

hive\_builtin = fun\_hive\_builtin(tbl\_weather, origin) %>% collect(),

translated\_dplyr = fun\_implemented(tbl\_weather, origin) %>% collect(),

spark\_apply\_arrow = spark\_apply(tbl\_weather, fun\_r\_custom, context = {colName <- "origin"}) %>% collect()

)

We can see that the timing on spark\_apply() decreased from more than 6 minutes to around 4.5 seconds, which is a very signigicant performance boost. Compared to the other methods we however still experience an order of magnitude difference.

time (milliseconds)Simple column transformation on a small datasethive\_builtintranslated\_dplyrspark\_apply\_arrow01k2k3k4k5k

**Notes on the setup of Apache Arrow**

It is worth noting that the implementation of Apache Arrow into R arrived on CRAN early August 2019, which means at the time of writing of this article it is on CRAN about 3 weeks. The functionality also depends on the Arrow C++ library, so installation is a bit more difficult than with some other R packages.

Care should also be taken with regards to the capability of the C++ library, the arrow R package version and the version of sparklyr. We had good results with using the R package arrow version 0.14.1, sparklyr 1.0.2 and the 0.14.1 version of the C++ libraries.

The aforementioned Docker image has both the C++ libraries and the R arrow package available for use.

**The take-home message**

Adding Arrow to the mix certainly significantly improved the performance of our example code, but is still quite slow compared to the native approach. Based on the above, we could conclude that

Performance benefits are present mainly when all the computation is performed within Spark and R serves merely as a “messaging agent”, sending commands to Spark to be executed. If there are object serialization and transfer of larger objects present, performance is strongly impacted.

The take-home message from this exercise is that we should strive to only use R code that can be executed within the Spark instance. If we need some data retrieved, it is advisable that this is data that was previously heavily aggregated within Spark and only a small amount is transferred to the R session.

**But we still need arbitrary R function to run fast on Spark**

In the next installments of this series, we will investigate a few options that allow us to retain the performance of Spark while still being able to write arbitrary R functions (i.e. using methods already implemented and available in the Spark API from R by implementing R functions not directly provided by the sparklyr interface) by:

1. Rewriting the functions as collections of dplyr verbs that all support translation to Spark SQL
2. Rewriting the functions as series of Scala method invocations
3. Rewriting the functions into Spark SQL and using DBI to execute directly

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.

**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.