**Introduction**

In this part, we will look at how to write R functions that interface with Spark via a lower-level invocation API that lets us use all the functionality that is exposed by the Scala Spark APIs. We will also show how such R calls relate to Scala code.

**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.

**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.

# 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

)

**The invoke() API of sparklyr**

So far when interfacing with Spark from R, we have used the sparklyr package in three ways:

* Writing combinations of dplyr verbs that would be translated to Spark SQL via the dbplyr package and the SQL executed by Spark when requested
* Generating Spark SQL code directly and sending it for execution in multiple ways
* Combinations of the above two methods

What these methods have in common is that they translate operations written in R to Spark SQL and that SQL code is then sent for execution by our Spark instance.

There is however another approach that we can use with sparklyr, which will be more familiar to users or developers who have worked with packages like rJava or rscala before. Even though arguably less convenient than the APIs provided by the 2 aforementioned packages, sparklyr provides an invocation API that exposes 3 functions:

1. invoke(jobj, method, ...) to execute a method on a Java object reference
2. invoke\_static(sc, class, method, ...) to execute a static method associated with a Java class
3. invoke\_new(sc, class, ...) to invoke a constructor associated with a Java class



Apache Spark and R logos

Let us have a look at how we can use those functions in practice to efficiently work with Spark from R.

**Getting started with the invoke API**

We can start with a few very simple examples of invoke() usage, for instance getting the number of rows of the tbl\_flights:

# Get the count of rows

tbl\_flights %>% spark\_dataframe() %>%

invoke("count")

## [1] 336776

We see one extra operation before invoking the count: spark\_dataframe(). This is because the invoke() interface works with Java object references and not tbl objects in remote sources such as tbl\_flights. We, therefore, need to convert tbl\_flights to a Java object reference, for which we use the spark\_dataframe() function.

Now, for something more exciting, let us compute a summary of the variables in tbl\_flights using the describe method:

tbl\_flights\_summary <- tbl\_flights %>% spark\_dataframe() %>%

invoke("describe", as.list(colnames(tbl\_flights))) %>%

sdf\_register()

tbl\_flights\_summary

## # Source: spark [?? x 19]

## summary year month day dep\_time sched\_dep\_time dep\_delay arr\_time

##

## 1 count 3367… 3367… 3367… 328521 336776 328521 328063

## 2 mean 2013… 6.54… 15.7… 1349.10… 1344.25484001… 12.63907… 1502.05…

## 3 stddev 0.0 3.41… 8.76… 488.281… 467.335755734… 40.21006… 533.264…

## 4 min 2013 1 1 1 106 -43.0 1

## 5 max 2013 12 31 2400 2359 1301.0 2400

## # … with 11 more variables: sched\_arr\_time , arr\_delay ,

## # carrier , flight , tailnum , origin , dest ,

## # air\_time , distance , hour , minute

We also one see extra operation after invoking the describe method: sdf\_register(). This is because the invoke() interface also *returns* Java object references and we may like to see a more user-friendly tbl object instead. This is where sdf\_register() comes in to register a Spark DataFrame and return a tbl\_spark object back to us.

And indeed, we can see that the wrapper sdf\_describe() provided by the sparklyr package itself works in a very similar fashion:

sparklyr::sdf\_describe

## function(x, cols = colnames(x)) {

## in\_df <- cols %in% colnames(x)

## if (any(!in\_df)) {

## msg <- paste0("The following columns are not in the data frame: ",

## paste0(cols[which(!in\_df)], collapse = ", "))

## stop(msg)

## }

## cols <- cast\_character\_list(cols)

##

## x %>%

## spark\_dataframe() %>%

## invoke("describe", cols) %>%

## sdf\_register()

## }

##

If we so wish, for DataFrame related object references, we can also call collect() to retrieve the results directly, without using sdf\_register() first, for instance retrieving the full content of the origin column:

tbl\_flights %>% spark\_dataframe() %>%

invoke("select", "origin", list()) %>%

collect()

## # A tibble: 336,776 x 1

## origin

##

## 1 EWR

## 2 LGA

## 3 JFK

## 4 JFK

## 5 LGA

## 6 EWR

## 7 EWR

## 8 LGA

## 9 JFK

## 10 LGA

## # … with 336,766 more rows

It can also be helpful to investigate the schema of our DataFrame:

tbl\_flights %>% spark\_dataframe() %>%

invoke("schema")

##

## org.apache.spark.sql.types.StructType

## StructType(StructField(year,IntegerType,true), StructField(month,IntegerType,true), StructField(day,IntegerType,true), StructField(dep\_time,IntegerType,true), StructField(sched\_dep\_time,IntegerType,true), StructField(dep\_delay,DoubleType,true), StructField(arr\_time,IntegerType,true), StructField(sched\_arr\_time,IntegerType,true), StructField(arr\_delay,DoubleType,true), StructField(carrier,StringType,true), StructField(flight,IntegerType,true), StructField(tailnum,StringType,true), StructField(origin,StringType,true), StructField(dest,StringType,true), StructField(air\_time,DoubleType,true), StructField(distance,DoubleType,true), StructField(hour,DoubleType,true), StructField(minute,DoubleType,true), StructField(time\_hour,TimestampType,true))

We can also use the invoke interface on other objects, for instance the SparkContext. Let’s for instance retrieve the uiWebUrl of our context:

sc %>% spark\_context() %>%

invoke("uiWebUrl") %>%

invoke("toString")

## [1] "Some(<http://localhost:4040>)"

**Grouping and aggregation with invoke chains**

Imagine we would like to do simple aggregations of a Spark DataFrame, such as an average of a column grouped by another column. For reference, we can do this very simply using the dplyr approach. Let’s compute the average departure delay by origin of the flight:

tbl\_flights %>%

group\_by(origin) %>%

summarise(avg(dep\_delay))

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

## origin `avg(dep\_delay)`

##

## 1 EWR 15.1

## 2 JFK 12.1

## 3 LGA 10.3

Now we will show how to do the same aggregation via the lower level API. Using the Spark shell we would simply do:

flights.

groupBy("origin").

agg(avg("dep\_delay"))

Translating that into the lower level invoke() API provided by sparklyr looks something like this:

tbl\_flights %>%

spark\_dataframe() %>%

invoke("groupBy", "origin", list()) %>%

invoke("agg", invoke\_static(sc, "org.apache.spark.sql.functions", "expr", "avg(dep\_delay)"), list()) %>%

sdf\_register()

**What is all that extra code?**

Now, compared to the very simple 2 operations in the Scala version, we have some gotchas to examine:

* one of the invoke() calls is quite long. Instead of just avg("dep\_delay") like in the Scala example, we use invoke\_static(sc, "org.apache.spark.sql.functions", "expr", "avg(dep\_delay)"). This is because the avg("dep\_delay") expression is somewhat of a syntactic sugar provided by Scala, but when calling from R we need to provide the object reference hidden behind that sugar.
* the empty list() at the end of the "groupBy" and "agg" invokes. This is needed as a workaround some Scala methods take String, String\* as arguments and sparklyr currently does not support variable parameters. We can pass list() to represent an empty String[] in Scala as the needed second argument.

**Wrapping the invocations into R functions**

Seeing the above example, we can quickly write a useful wrapper to ease the pain a little. First, we can create a small function that will generate the aggregation expression we can use with invoke("agg", ...):

agg\_expr <- function(tbl, exprs) {

sparklyr::invoke\_static(

tbl[["src"]][["con"]],

"org.apache.spark.sql.functions",

"expr",

exprs

)

}

Next, we can wrap around the entire process to make a more generic aggregation function, using the fact that a remote tibble has the details on sc within its tbl[["src"]][["con"]] element:

grpagg\_invoke <- function(tbl, colName, groupColName, aggOperation) {

avgColumn <- tbl %>% agg\_expr(paste0(aggOperation, "(", colName, ")"))

tbl %>% spark\_dataframe() %>%

invoke("groupBy", groupColName, list()) %>%

invoke("agg", avgColumn, list()) %>%

sdf\_register()

}

And finally use our wrapper to get the same results in a more user-friendly way:

tbl\_flights %>%

grpagg\_invoke("arr\_delay", groupColName = "origin", aggOperation = "avg")

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

## origin `avg(arr\_delay)`

##

## 1 EWR 9.11

## 2 JFK 5.55

## 3 LGA 5.78

**Reconstructing variable normalization**

Now we will attempt to construct the variable normalization that we have shown in the previous parts with dplyr verbs and SQL generation – we will normalize the values of a column by first subtracting the mean value and then dividing the values by the standard deviation:

normalize\_invoke <- function(tbl, colName) {

sdf <- tbl %>% spark\_dataframe()

stdCol <- agg\_expr(tbl, paste0("stddev\_samp(", colName, ")"))

avgCol <- agg\_expr(tbl, paste0("avg(", colName, ")"))

avgTemp <- sdf %>% invoke("agg", avgCol, list()) %>% invoke("first")

stdTemp <- sdf %>% invoke("agg", stdCol, list()) %>% invoke("first")

newCol <- sdf %>%

invoke("col", colName) %>%

invoke("minus", as.numeric(avgTemp)) %>%

invoke("divide", as.numeric(stdTemp))

sdf %>%

invoke("withColumn", colName, newCol) %>%

sdf\_register()

}

tbl\_weather %>% normalize\_invoke("temp")

## # Source: spark [?? x 16]

## id origin year month day hour temp dewp humid wind\_dir

##

## 1 1 EWR 2013 1 1 1 -0.913 26.1 59.4 270

## 2 2 EWR 2013 1 1 2 -0.913 27.0 61.6 250

## 3 3 EWR 2013 1 1 3 -0.913 28.0 64.4 240

## 4 4 EWR 2013 1 1 4 -0.862 28.0 62.2 250

## 5 5 EWR 2013 1 1 5 -0.913 28.0 64.4 260

## 6 6 EWR 2013 1 1 6 -0.974 28.0 67.2 240

## 7 7 EWR 2013 1 1 7 -0.913 28.0 64.4 240

## 8 8 EWR 2013 1 1 8 -0.862 28.0 62.2 250

## 9 9 EWR 2013 1 1 9 -0.862 28.0 62.2 260

## 10 10 EWR 2013 1 1 10 -0.802 28.0 59.6 260

## # … with more rows, and 6 more variables: wind\_speed ,

## # wind\_gust , precip , pressure , visib ,

## # time\_hour

The above implementation is just an example and far from optimal, but it also has a few interesting points about it:

* Using invoke("first") will actually compute and collect the value into the R session
* Those collected values are then sent back during the invoke("minus", as.numeric(avgTemp)) and invoke("divide", as.numeric(stdTemp))

This means that there is unnecessary overhead when sending those values from the Spark instance into R and back, which will have slight performance penalties.

**Where invoke can be better than dplyr translation or SQL**

As we have seen in the above examples, working with the invoke() API can prove more difficult than using the intuitive syntax of dplyr or SQL queries. In some use cases, the trade-off may still be worth it. In our practice, these are some examples of such situations:

* When Scala’s Spark API is more flexible, powerful or suitable for a particular task and the translation is not as good
* When performance is crucial and we can produce more optimal solutions using the invocations
* When we know the Scala API well and not want to invest time to learn the dplyr syntax, but it is easier to translate the Scala calls into a series of invoke() calls
* When we need to interact and manipulate other Java objects apart from the standard Spark DataFrames

**Conclusion**

In this part of the series, we have looked at how to use the lower-level invoke interface provided by sparklyr to manipulate Spark objects and other Java object references. In the following part, we will dig a bit deeper and look into using Java’s reflection API to make the invoke interface more accessible from R, getting detail invocation logs and more.