**Introduction**

In the previous parts of this series, we have shown how to write functions as both [combinations of dplyr verbs](https://jozef.io/r202-spark-r-dplyr-verbs/) and [SQL query generators](https://jozef.io/r203-spark-r-sql/) that can be executed by Spark, how to execute them with DBI and how to achieve lazy SQL statements that only get executed when needed.

In this fourth 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.

**Contents**

1. [Introduction](https://jozef.io/r204-spark-r-invoke-scala/#introduction)
2. [Contents](https://jozef.io/r204-spark-r-invoke-scala/#contents)
3. [Preparation](https://jozef.io/r204-spark-r-invoke-scala/#preparation)
4. [The invoke() API of sparklyr](https://jozef.io/r204-spark-r-invoke-scala/#the-invoke-api-of-sparklyr)
5. [Getting started with the invoke API](https://jozef.io/r204-spark-r-invoke-scala/#getting-started-with-the-invoke-api)
6. [Grouping and aggregation with invoke chains](https://jozef.io/r204-spark-r-invoke-scala/#grouping-and-aggregation-with-invoke-chains)
7. [Wrapping the invocations into R functions](https://jozef.io/r204-spark-r-invoke-scala/#wrapping-the-invocations-into-r-functions)
8. [Reconstructing variable normalization](https://jozef.io/r204-spark-r-invoke-scala/#reconstructing-variable-normalization)
9. [Where invoke can be better than dplyr translation or SQL](https://jozef.io/r204-spark-r-invoke-scala/#where-invoke-can-be-better-than-dplyr-translation-or-sql)
10. [Conclusion](https://jozef.io/r204-spark-r-invoke-scala/#conclusion)
11. [References](https://jozef.io/r204-spark-r-invoke-scala/#references)

**Preparation**

The full setup of Spark and sparklyr is not in the scope of this post, please check the [first 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.

# 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](https://jozef.io/r901-primer-java-from-r-1/) 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\*](https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Dataset@groupBy(col1:String,cols:String*):org.apache.spark.sql.RelationalGroupedDataset) 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.