**Introduction**

Apache Spark is a popular open-source analytics engine for big data processing and thanks to the sparklyr and SparkR packages, the power of Spark is also available to R users.

This series of articles will attempt to provide practical insights into using the sparklyr interface to gain the benefits of Apache Spark while still retaining the ability to use R code organized in custom-built functions and packages.

In this first part, we will examine how the sparklyr interface communicates with the Spark instance and what this means for performance with regards to arbitrarily defined R functions. We will also look at how Apache Arrow can improve the performance of object serialization.

**Contents**

1. [Setting up Spark with R and sparklyr](https://jozef.io/r201-spark-r-1/#setting-up-spark-with-r-and-sparklyr)
   1. [Using a ready-made Docker Image](https://jozef.io/r201-spark-r-1/#using-a-ready-made-docker-image)
   2. [Manual Installation](https://jozef.io/r201-spark-r-1/#manual-installation)
   3. [Connecting and using a local Spark instance](https://jozef.io/r201-spark-r-1/#connecting-and-using-a-local-spark-instance)
2. [Sparklyr as a Spark interface provider](https://jozef.io/r201-spark-r-1/#sparklyr-as-a-spark-interface-provider)
   1. [An R function translated to Spark SQL](https://jozef.io/r201-spark-r-1/#an-r-function-translated-to-spark-sql)
   2. [An R function not translated to Spark SQL](https://jozef.io/r201-spark-r-1/#an-r-function-not-translated-to-spark-sql)
   3. [A Hive built-in function not existing in R](https://jozef.io/r201-spark-r-1/#a-hive-built-in-function-not-existing-in-r)
3. [Using non-translated functions with sparklyr](https://jozef.io/r201-spark-r-1/#using-non-translated-functions-with-sparklyr)
   1. [What is so important about this distinction?](https://jozef.io/r201-spark-r-1/#what-is-so-important-about-this-distinction)
   2. [What happens when we use custom functions with spark\_apply](https://jozef.io/r201-spark-r-1/#what-happens-when-we-use-custom-functions-with-spark_apply)
   3. [What happens when we use translated or Hive built-in functions](https://jozef.io/r201-spark-r-1/#what-happens-when-we-use-translated-or-hive-built-in-functions)
   4. [Which R functionality is currently translated and built-in to Hive](https://jozef.io/r201-spark-r-1/#which-r-functionality-is-currently-translated-and-built-in-to-hive)
4. [Making serialization faster with Apache Arrow](https://jozef.io/r201-spark-r-1/#making-serialization-faster-with-apache-arrow)
   1. [What is Apache Arrow and how it improves performance](https://jozef.io/r201-spark-r-1/#what-is-apache-arrow-and-how-it-improves-performance)
   2. [Notes on the setup of Apache Arrow](https://jozef.io/r201-spark-r-1/#notes-on-the-setup-of-apache-arrow)
5. [The take-home message](https://jozef.io/r201-spark-r-1/#the-take-home-message)
6. [But we still need arbitrary R function to run fast on Spark](https://jozef.io/r201-spark-r-1/#but-we-still-need-arbitrary-r-function-to-run-fast-on-spark)
7. [References](https://jozef.io/r201-spark-r-1/#references)

**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](https://hub.docker.com/r/jozefhajnala/sparkly) 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](https://jozef.io/r905-rstudio-terminal/) 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/spark: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](https://spark.rstudio.com/#installation).

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](https://cran.r-project.org/package=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)

)

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))

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()

)

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](https://spark.rstudio.com/dplyr/#sql-translation)
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](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+UDF) 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](https://arrow.apache.org/), 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.

**Notes on the setup of Apache Arrow**

It is worth noting that the implementation of Apache Arrow [into R](https://github.com/apache/arrow/tree/master/r) arrived on [CRAN](https://cran.r-project.org/package=arrow) 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](https://arrow.apache.org/install/), 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](https://hub.docker.com/r/jozefhajnala/sparkly) 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