**Introduction**

In the [previous part](https://jozef.io/r202-spark-r-dplyr-verbs/) of this series, we looked at writing R functions that can be executed directly by Spark without serialization overhead with a focus on writing functions as combinations of dplyr verbs and investigated how the SQL is generated and Spark plans created.

In this third part, we will look at how to write R functions that generate SQL queries 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. We also briefly present wrapping these approaches into functions that can be combined with other Spark operations.

**Contents**

1. [Preparation](https://jozef.io/r203-spark-r-sql/#preparation)
2. [R functions as Spark SQL generators](https://jozef.io/r203-spark-r-sql/#r-functions-as-spark-sql-generators)
3. [Executing the generated queries via Spark](https://jozef.io/r203-spark-r-sql/#executing-the-generated-queries-via-spark)
   1. [Using DBI as the interface](https://jozef.io/r203-spark-r-sql/#using-dbi-as-the-interface)
   2. [Invoking sql on a Spark session object](https://jozef.io/r203-spark-r-sql/#invoking-sql-on-a-spark-session-object)
   3. [Using tbl with dbplyr’s sql](https://jozef.io/r203-spark-r-sql/#using-tbl-with-dbplyrs-sql)
   4. [Wrapping the tbl approach into functions](https://jozef.io/r203-spark-r-sql/#wrapping-the-tbl-approach-into-functions)
4. [Combining multiple approaches and functions into lazy datasets](https://jozef.io/r203-spark-r-sql/#combining-multiple-approaches-and-functions-into-lazy-datasets)
5. [Where SQL can be better than dbplyr translation](https://jozef.io/r203-spark-r-sql/#where-sql-can-be-better-than-dbplyr-translation)
   1. [When translation is not there](https://jozef.io/r203-spark-r-sql/#when-translation-is-not-there)
   2. [When translation does not provide expected results](https://jozef.io/r203-spark-r-sql/#when-translation-does-not-provide-expected-results)
   3. [When portability is important](https://jozef.io/r203-spark-r-sql/#when-portability-is-important)
6. [References](https://jozef.io/r203-spark-r-sql/#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.

# 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 Spark SQL generators**

There are use cases where it is desirable to express the operations directly with SQL instead of combining dplyr verbs, for example when working within multi-language environments where re-usability is important. We can then send the SQL query directly to Spark to be executed. To create such queries, one option is to write R functions that work as query constructors.

Again using a very simple example, a naive implementation of column normalization could look as follows. Note that the use of SELECT \* is discouraged and only here for illustration purposes:

normalize\_sql <- function(df, colName, newColName) {

paste0(

"SELECT",

"\n ", df, ".\*", ",",

"\n (", colName, " - (SELECT avg(", colName, ") FROM ", df, "))",

" / ",

"(SELECT stddev\_samp(", colName,") FROM ", df, ") as ", newColName,

"\n", "FROM ", df

)

}

Using the weather dataset would then yield the following SQL query when normalizing the temp column:

normalize\_temp\_query <- normalize\_sql("weather", "temp", "normTemp")

cat(normalize\_temp\_query)

## SELECT

## weather.\*,

## (temp - (SELECT avg(temp) FROM weather)) / (SELECT stddev\_samp(temp) FROM weather) as normTemp

## FROM weather

Now that we have the query created, we can look at how to send it to Spark for execution.



Apache Spark and R logos

**Executing the generated queries via Spark**

**Using DBI as the interface**

The R package DBI provides an interface for communication between R and relational database management systems. We can simply use the dbGetQuery() function to execute our query, for instance:

res <- DBI::dbGetQuery(sc, statement = normalize\_temp\_query)

head(res)

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

## 1 1 EWR 2013 1 1 1 39.02 26.06 59.37 270 10.35702

## 2 2 EWR 2013 1 1 2 39.02 26.96 61.63 250 8.05546

## 3 3 EWR 2013 1 1 3 39.02 28.04 64.43 240 11.50780

## 4 4 EWR 2013 1 1 4 39.92 28.04 62.21 250 12.65858

## 5 5 EWR 2013 1 1 5 39.02 28.04 64.43 260 12.65858

## 6 6 EWR 2013 1 1 6 37.94 28.04 67.21 240 11.50780

## wind\_gust precip pressure visib time\_hour normTemp

## 1 NaN 0 1012.0 10 2013-01-01 06:00:00 -0.9130047

## 2 NaN 0 1012.3 10 2013-01-01 07:00:00 -0.9130047

## 3 NaN 0 1012.5 10 2013-01-01 08:00:00 -0.9130047

## 4 NaN 0 1012.2 10 2013-01-01 09:00:00 -0.8624083

## 5 NaN 0 1011.9 10 2013-01-01 10:00:00 -0.9130047

## 6 NaN 0 1012.4 10 2013-01-01 11:00:00 -0.9737203

As we might have noticed thanks to the way the result is printed, a standard data frame is returned, as opposed to tibbles returned by most sparklyr operations.

It is important to note that using dbGetQuery() *automatically computes and collects* the results to the R session. This is in contrast with the dplyr approach which constructs the query and only collects the results to the R session when collect() is called, or computes them when compute() is called.

We will now examine 2 options to use the prepared query lazily and without collecting the results into the R session.

**Invoking sql on a Spark session object**

Without going into further details on the invoke() functionality of sparklyr which we will focus on in the fourth installment of the series, if the desire is to have a “lazy” SQL that does not get automatically computed and collected when called from R, we can invoke a [sql method](https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.SparkSession@sql(sqlText:String):org.apache.spark.sql.DataFrame) on a SparkSession class object.

The method takes a string SQL query as input and processes it using Spark, returning the result as a Spark DataFrame. This gives us the ability to only compute and collect the results when desired:

# Use the query "lazily" without execution:

normalized\_lazy\_ds <- sc %>%

spark\_session() %>%

invoke("sql", normalize\_temp\_query)

normalized\_lazy\_ds

##

## org.apache.spark.sql.Dataset

## [id: int, origin: string ... 15 more fields]

# Collect when needed:

normalized\_lazy\_ds %>% collect()

## # A tibble: 26,115 x 17

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

##

## 1 1 EWR 2013 1 1 1 39.0 26.1 59.4 270

## 2 2 EWR 2013 1 1 2 39.0 27.0 61.6 250

## 3 3 EWR 2013 1 1 3 39.0 28.0 64.4 240

## 4 4 EWR 2013 1 1 4 39.9 28.0 62.2 250

## 5 5 EWR 2013 1 1 5 39.0 28.0 64.4 260

## 6 6 EWR 2013 1 1 6 37.9 28.0 67.2 240

## 7 7 EWR 2013 1 1 7 39.0 28.0 64.4 240

## 8 8 EWR 2013 1 1 8 39.9 28.0 62.2 250

## 9 9 EWR 2013 1 1 9 39.9 28.0 62.2 260

## 10 10 EWR 2013 1 1 10 41 28.0 59.6 260

## # … with 26,105 more rows, and 7 more variables: wind\_speed ,

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

## # time\_hour , normTemp

**Using tbl with dbplyr’s sql**

The above method gives us a reference to a Java object as a result, which might be less intuitive to work with for R users. We can also opt to use dbplyr’s sql() function in combination with tbl() to get a more familiar result.

Note that when printing the below normalized\_lazy\_tbl, the query gets partially executed to provide the first few rows. Only when collect() is called the entire set is retrieved to the R session:

# Nothing is executed yet

normalized\_lazy\_tbl <- normalize\_temp\_query %>%

dbplyr::sql() %>%

tbl(sc, .)

# Print the first few rows

normalized\_lazy\_tbl

## # Source: spark

# Collect the entire result to the R session and print

normalized\_lazy\_tbl %>% collect()

## # A tibble: 26,115 x 17

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

##

## 1 1 EWR 2013 1 1 1 39.0 26.1 59.4 270

## 2 2 EWR 2013 1 1 2 39.0 27.0 61.6 250

## 3 3 EWR 2013 1 1 3 39.0 28.0 64.4 240

## 4 4 EWR 2013 1 1 4 39.9 28.0 62.2 250

## 5 5 EWR 2013 1 1 5 39.0 28.0 64.4 260

## 6 6 EWR 2013 1 1 6 37.9 28.0 67.2 240

## 7 7 EWR 2013 1 1 7 39.0 28.0 64.4 240

## 8 8 EWR 2013 1 1 8 39.9 28.0 62.2 250

## 9 9 EWR 2013 1 1 9 39.9 28.0 62.2 260

## 10 10 EWR 2013 1 1 10 41 28.0 59.6 260

## # … with 26,105 more rows, and 7 more variables: wind\_speed ,

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

## # time\_hour , normTemp

**Wrapping the tbl approach into functions**

In the approach above we provided sc in the call to tbl(). When wrapping such processes into a function, it might however be useful to take the specific DataFrame reference as an input instead of the generic Spark connection reference.

In that case, we can use the fact that the connection reference is also stored in the DataFrame reference, in the con sub-element of the src element. For instance, looking at our tbl\_weather:

class(tbl\_weather[["src"]][["con"]])

## [1] "spark\_connection" "spark\_shell\_connection"

## [3] "DBIConnection"

Putting this together, we can create a simple wrapper function that lazily sends a SQL query to be processed on a particular Spark DataFrame reference:

lazy\_spark\_query <- function(tbl, qry) {

qry %>%

dbplyr::sql() %>%

dplyr::tbl(tbl[["src"]][["con"]], .)

}

And use it to do the same as we did above with a single function call:

lazy\_spark\_query(tbl\_weather, normalize\_temp\_query) %>%

collect()

## # A tibble: 26,115 x 17

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

##

## 1 1 EWR 2013 1 1 1 39.0 26.1 59.4 270

## 2 2 EWR 2013 1 1 2 39.0 27.0 61.6 250

## 3 3 EWR 2013 1 1 3 39.0 28.0 64.4 240

## 4 4 EWR 2013 1 1 4 39.9 28.0 62.2 250

## 5 5 EWR 2013 1 1 5 39.0 28.0 64.4 260

## 6 6 EWR 2013 1 1 6 37.9 28.0 67.2 240

## 7 7 EWR 2013 1 1 7 39.0 28.0 64.4 240

## 8 8 EWR 2013 1 1 8 39.9 28.0 62.2 250

## 9 9 EWR 2013 1 1 9 39.9 28.0 62.2 260

## 10 10 EWR 2013 1 1 10 41 28.0 59.6 260

## # … with 26,105 more rows, and 7 more variables: wind\_speed ,

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

## # time\_hour , normTemp

**Combining multiple approaches and functions into lazy datasets**

The power of Spark partly comes from the lazy execution and we can take advantage of this in ways that are not immediately obvious. Consider the following function we have shown previously:

lazy\_spark\_query

## function(tbl, qry) {

## qry %>%

## dbplyr::sql() %>%

## dplyr::tbl(tbl[["src"]][["con"]], .)

## }

Since the output of this function without collection is actually only a translated SQL statement, we can take that output and keep combinining it with other operations, for instance:

qry <- normalize\_sql("flights", "dep\_delay", "dep\_delay\_norm")

lazy\_spark\_query(tbl\_flights, qry) %>%

group\_by(origin) %>%

summarise(mean(dep\_delay\_norm)) %>%

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: 3 x 2

## origin `mean(dep\_delay\_norm)`

##

## 1 EWR 0.0614

## 2 JFK -0.0131

## 3 LGA -0.0570

The crucial advantage is that even though the lazy\_spark\_query would return the entire updated weather dataset when collected stand-alone, in combination with other operations Spark first figures out how to execute all the operations together efficiently and only then physically executes them and returns only the grouped and aggregated data to the R session.

We can therefore effectively combine multiple approaches to interfacing with Spark while still keeping the benefit of retrieving only very small, aggregated amounts of data to the R session. The effect is quite significant even with a dataset as small as flights (336,776 rows of 19 columns) and with a local Spark instance. The chart below compares executing a query lazily, aggregating within Spark and only retrieving the aggregated data, versus retrieving first and aggregating locally. The third boxplot shows the cost of pure collection on the query itself:

bench <- microbenchmark::microbenchmark(

times = 20,

collect\_late = lazy\_spark\_query(tbl\_flights, qry) %>%

group\_by(origin) %>%

summarise(mean(dep\_delay\_norm)) %>%

collect(),

collect\_first = lazy\_spark\_query(tbl\_flights, qry) %>%

collect() %>%

group\_by(origin) %>%

summarise(mean(dep\_delay\_norm)),

collect\_only = lazy\_spark\_query(tbl\_flights, qry) %>%

collect()

)

**Where SQL can be better than dbplyr translation**

**When a translation is not there**

We have discussed in the [first part](https://jozef.io/r201-spark-r-1/#an-r-function-not-translated-to-spark-sql) that the set of operations translated to Spark SQL via dbplyr may not cover all possible use cases. In such a case, the option to write SQL directly is very useful.

**When translation does not provide expected results**

In some instances using dbplyr to translate R operations to Spark SQL can lead to unexpected results. As one example, consider the following integer division on a column of a local data frame.

# id\_div\_5 is as expected

weather %>%

mutate(id\_div\_5 = id %/% 5L) %>%

select(id, id\_div\_5)

## # A tibble: 26,115 x 2

## id id\_div\_5

##

## 1 1 0

## 2 2 0

## 3 3 0

## 4 4 0

## 5 5 1

## 6 6 1

## 7 7 1

## 8 8 1

## 9 9 1

## 10 10 2

## # … with 26,105 more rows

As expected, we get the result of integer division in the id\_div\_5 column. However, applying the very same operation on a Spark DataFrame yields unexpected results:

# id\_div\_5 is normal division, not integer division

tbl\_weather %>%

mutate(id\_div\_5 = id %/% 5L) %>%

select(id, id\_div\_5)

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

## id id\_div\_5

##

## 1 1 0.2

## 2 2 0.4

## 3 3 0.6

## 4 4 0.8

## 5 5 1

## 6 6 1.2

## 7 7 1.4

## 8 8 1.6

## 9 9 1.8

## 10 10 2

## # … with more rows

This is due to the fact that translation to integer division is quite difficult to implement: <https://github.com/tidyverse/dbplyr/issues/108>. We could certainly figure our a way to fix this particular issue, but the workarounds may prove inefficient:

tbl\_weather %>%

mutate(id\_div\_5 = as.integer(id %/% 5L)) %>%

select(id, id\_div\_5)

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

## id id\_div\_5

##

## 1 1 0

## 2 2 0

## 3 3 0

## 4 4 0

## 5 5 1

## 6 6 1

## 7 7 1

## 8 8 1

## 9 9 1

## 10 10 2

## # … with more rows

# Not too efficient:

tbl\_weather %>%

mutate(id\_div\_5 = as.integer(id %/% 5L)) %>%

select(id, id\_div\_5) %>%

explain()

##

## SELECT `id`, CAST(`id` / 5 AS INT) AS `id\_div\_5`

## FROM `weather`

##

##

## == Physical Plan ==

## \*(1) Project [id#24, cast((cast(id#24 as double) / 5.0) as int) AS id\_div\_5#4273]

## +- InMemoryTableScan [id#24]

## +- 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 SQL and the knowledge that Hive does provide a built-in [DIV arithmetic operator](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+UDF#LanguageManualUDF-ArithmeticOperators), we can get the desired results very simply and efficiently with writing SQL:

"SELECT `id`, `id` DIV 5 `id\_div\_5` FROM `weather`" %>%

dbplyr::sql() %>%

tbl(sc, .)

## # Source: spark

Even though the numeric value of the results is correct here, we may still notice that the class of the returned id\_div\_5 column is actually numeric instead of integer. Such is the life of developers using data processing interfaces.

**When portability is important**

Since the languages that provide interfaces to Spark are not limited to R and multi-language setups are quite common, another reason to use SQL statements directly is the portability of such solutions. A SQL statement can be executed by interfaces provided for all languages – Scala, Java, and Python, without the need to rely on R-specific packages such as dbplyr.

**References**

* The [first part](https://jozef.io/r201-spark-r-1/) of this series
* The [second part](https://jozef.io/r202-spark-r-dplyr-verbs/) 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).
* The [DBI package](https://cran.r-project.org/package=DBI) on CRAN