**Introduction**

In the below 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.

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]

## <int> <dbl> <dbl>

## 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

## <int> <dbl> <dbl>

## 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

## <int> <dbl> <dbl>

## 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()

## <SQL>

## 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`

##

## <PLAN>

## == 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)`

## <dbl> <dbl>

## 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()

## <SQL>

## 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`

##

## <PLAN>

## == 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

## <dbl>

## 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

## <int>

## 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(temp)) %>% count()

## # A tibble: 1 x 1

## n

## <int>

## 1 1573

joined\_spark %>% filter(is.na(temp)) %>% count()

## # A tibble: 1 x 1

## n

## <int>

## 1 1697

# Look at counts of NaN values

joined\_local %>% filter(is.nan(temp)) %>% count()

## # A tibble: 1 x 1

## n

## <int>

## 1 0

joined\_spark %>% filter(is.nan(temp)) %>% count()

## # A tibble: 1 x 1

## n

## <int>

## 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

## <int> <dttm>

## 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

## <int> <dttm>

## 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

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.

**Preparation**

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