In this section, we will showcase three new dplyr functionalities that were shipped with

sparklyr 1.5.

## Stratified sampling

Stratified sampling on an R dataframe can be accomplished with a combination of dplyr::group\_by() followed by dplyr::sample\_n() or dplyr::sample\_frac(), where the grouping variables specified in the dplyr::group\_by() step are the ones that define each stratum. For instance, the following query will group mtcars by number of cylinders and return a weighted random sample of size two from each group, without replacement, and weighted by the mpg column:

mtcars %> dplyr::group\_by(cyl) %>%

dplyr::sample\_n(size = 2, weight = mpg, replace = FALSE) %>% print()

|  |  |  |
| --- | --- | --- |
| ## | # | A tibble: 6 x 11 |
| ## | # | Groups: cyl [3] |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ##  ## | mpg | cyl | disp | hp | drat | wt | qsec | vs | am | gear | carb |
| ## | 1 33.9 | 4 | 71.1 | 65 | 4.22 | 1.84 | 19.9 | 1 | 1 | 4 | 1 |
| ## | 2 22.8 | 4 | 108 | 93 | 3.85 | 2.32 | 18.6 | 1 | 1 | 4 | 1 |
| ## | 3 21.4 | 6 | 258 | 110 | 3.08 | 3.22 | 19.4 | 1 | 0 | 3 | 1 |
| ## | 4 21 | 6 | 160 | 110 | 3.9 | 2.62 | 16.5 | 0 | 1 | 4 | 4 |
| ## | 5 15.5 | 8 | 318 | 150 | 2.76 | 3.52 | 16.9 | 0 | 0 | 3 | 2 |
| ## | 6 19.2 | 8 | 400 | 175 | 3.08 | 3.84 | 17.0 | 0 | 0 | 3 | 2 |

Starting from sparklyr 1.5, the same can also be done for Spark dataframes with Spark 3.0 or above, e.g.,:

library(sparklyr)

sc <- spark\_connect(master = "local", version = "3.0.0") mtcars\_sdf <- copy\_to(sc, mtcars, replace = TRUE, repartition = 3

mtcars\_sdf %>% dplyr::group\_by(cyl) %>%

dplyr::sample\_n(size = 2, weight = mpg, replace = FALSE) %>% print()

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| #  # | Source:  Groups: | spark [??  cyl | | x | 11] | | | | | | | |
|  | mpg | cyl | disp | hp | | drat | wt | qsec | vs | am | gear | carb |
| 1 | 21 | 6 | 160 | 110 | | 3.9 | 2.62 | 16.5 | 0 | 1 | 4 | 4 |
| 2 | 21.4 | 6 | 258 | 110 | | 3.08 | 3.22 | 19.4 | 1 | 0 | 3 | 1 |
| 3 | 27.3 | 4 | 79 | 66 | | 4.08 | 1.94 | 18.9 | 1 | 1 | 4 | 1 |
| 4 | 32.4 | 4 | 78.7 | 66 | | 4.08 | 2.2 | 19.5 | 1 | 1 | 4 | 1 |
| 5 | 16.4 | 8 | 276. | 180 | | 3.07 | 4.07 | 17.4 | 0 | 0 | 3 | 3 |
| 6 | 18.7 | 8 | 360 | 175 | | 3.15 | 3.44 | 17.0 | 0 | 0 | 3 | 2 |
| or |  |  |  |  | |  |  |  |  |  |  |  |

mtcars\_sdf %>% dplyr::group\_by(cyl) %>%

dplyr::sample\_frac(size = 0.2, weight = mpg, replace = FALSE) %>% print()

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ##  ## | #  # | Source:  Groups: | spark [??  cyl | | x | 11] | | | | | | | |
| ## |  | mpg | cyl | disp | hp | | drat | wt | qsec | vs | am | gear | carb |
| ## |  |  |  |  |  | |  |  |  |  |  |  |  |
| ## | 1 | 21 | 6 | 160 | 110 | | 3.9 | 2.62 | 16.5 | 0 | 1 | 4 | 4 |
| ## | 2 | 21.4 | 6 | 258 | 110 | | 3.08 | 3.22 | 19.4 | 1 | 0 | 3 | 1 |
| ## | 3 | 22.8 | 4 | 141. | 95 | | 3.92 | 3.15 | 22.9 | 1 | 0 | 4 | 2 |
| ## | 4 | 33.9 | 4 | 71.1 | 65 | | 4.22 | 1.84 | 19.9 | 1 | 1 | 4 | 1 |
| ## | 5 | 30.4 | 4 | 95.1 | 113 | | 3.77 | 1.51 | 16.9 | 1 | 1 | 5 | 2 |
| ## | 6 | 15.5 | 8 | 318 | 150 | | 2.76 | 3.52 | 16.9 | 0 | 0 | 3 | 2 |
| ## | 7 | 18.7 | 8 | 360 | 175 | | 3.15 | 3.44 | 17.0 | 0 | 0 | 3 | 2 |
| ## | 8 | 16.4 | 8 | 276. | 180 | | 3.07 | 4.07 | 17.4 | 0 | 0 | 3 | 3 |

## Row sums

The rowSums() functionality offered by dplyr is handy when one needs to sum up a large number of columns within an R dataframe that are impractical to be enumerated individually. For example, here we have a six-column dataframe of random real numbers, where the partial\_sum column in the result contains the sum of columns b through d within each row:

ncols <- 6

nums <- seq(ncols) %>% lapply(function(x) runif(5)) names(nums) <- letters[1:ncols]

tbl <- tibble::as\_tibble(nums)

tbl %>%

dplyr::mutate(partial\_sum = rowSums(.[2:5])) %>% print()

## # A tibble: 5 x 7

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ##  ## | a | b | c | d | e | f | partial\_sum |
| ## | 1 0.781 | 0.801 | 0.157 | 0.0293 | 0.169 | 0.0978 | 1.16 |
| ## | 2 0.696 | 0.412 | 0.221 | 0.941 | 0.697 | 0.675 | 2.27 |
| ## | 3 0.802 | 0.410 | 0.516 | 0.923 | 0.190 | 0.904 | 2.04 |
| ## | 4 0.200 | 0.590 | 0.755 | 0.494 | 0.273 | 0.807 | 2.11 |
| ## | 5 0.00149 | 0.711 | 0.286 | 0.297 | 0.107 | 0.425 | 1.40 |

Beginning with sparklyr 1.5, the same operation can be performed with Spark dataframes:

library(sparklyr)

sc <- spark\_connect(master = "local")

sdf <- copy\_to(sc, tbl, overwrite = TRUE)

sdf %>%

dplyr::mutate(partial\_sum = rowSums(.[2:5])) %>% print()

## # Source: spark [?? x 7]

## a b c d e f partial\_sum

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ## |  | | | | | | | |
| ## | 1 | 0.781 | 0.801 | 0.157 | 0.0293 | 0.169 | 0.0978 | 1.16 |
| ## | 2 | 0.696 | 0.412 | 0.221 | 0.941 | 0.697 | 0.675 | 2.27 |
| ## | 3 | 0.802 | 0.410 | 0.516 | 0.923 | 0.190 | 0.904 | 2.04 |
| ## | 4 | 0.200 | 0.590 | 0.755 | 0.494 | 0.273 | 0.807 | 2.11 |
| ## | 5 | 0.00149 | 0.711 | 0.286 | 0.297 | 0.107 | 0.425 | 1.40 |

As a bonus from implementing the rowSums feature for Spark dataframes, sparklyr 1.5 now also offers limited support for the column-subsetting operator on Spark dataframes. For example, all code snippets below will return some subset of columns from the dataframe named sdf:

# select columns `b` through `e` sdf[2:5]

# select columns `b` and `c` sdf[c("b", "c")]

# drop the first and third columns and return the rest sdf[c(-1, -3)]

## Weighted-mean summarizer

Similar to the two dplyr functions mentioned above, the weighted.mean() summarizer is another useful function that has become part of the dplyr interface for Spark dataframes in sparklyr 1.5. One can see it in action by, for example, comparing the output from the following

library(sparklyr)

sc <- spark\_connect(master = "local")

mtcars\_sdf <- copy\_to(sc, mtcars, replace = TRUE mtcars\_sdf %>%

dplyr::group\_by(cyl) %>%

dplyr::summarize(mpg\_wm = weighted.mean(mpg, wt)) %>% print()

with output from the equivalent operation on mtcars in R:

mtcars %> dplyr::group\_by(cyl) %>%

dplyr::summarize(mpg\_wm = weighted.mean(mpg, wt)) %>% print()

both of them should evaluate to the following:

|  |  |  |  |
| --- | --- | --- | --- |
| ##  ## |  | cyl | mpg\_wm |
| ## | 1 | 4 | 25.9 |
| ## | 2 | 6 | 19.6 |
| ## | 3 | 8 | 14.8 |

# New additions to the sdf\_\* family of functions

sparklyr provides a large number of convenience functions for working with Spark dataframes, and all of them have names starting with the sdf\_ prefix.

In this section we will briefly mention four new additions and show some example scenarios in which those functions are useful.

## sdf\_expand\_grid()

As the name suggests, sdf\_expand\_grid() is simply the Spark equivalent of expand.grid(). Rather than running expand.grid() in R and importing the resulting R dataframe to Spark, one can now run sdf\_expand\_grid(), which accepts both R vectors and Spark dataframes and supports hints for broadcast hash joins. The example below shows sdf\_expand\_grid() creating a 100-by-100-by-10-by-10 grid in Spark over 1000 Spark partitions, with broadcast hash join hints on variables with small cardinalities:

library(sparklyr)

sc <- spark\_connect(master = "local") grid\_sdf <- sdf\_expand\_grid(

sc,

var1 = seq(100), var2 = seq(100), var3 = seq(10), var4 = seq(10),

broadcast\_vars = c(var3, var4), repartition = 1000

)

grid\_sdf %>% sdf\_nrow() %>% print() ## [1] 1e+06

## sdf\_partition\_sizes()

As sparklyr user @sbottelli, one thing that would be great to have in sparklyr is an efficient way to query partition sizes of a Spark dataframe. In sparklyr 1.5, sdf\_partition\_sizes() does exactly that:

library(sparklyr)

sc <- spark\_connect(master = "local") sdf\_len(sc, 1000, repartition = 5) %>%

sdf\_partition\_sizes() %>%

|  |  |  |
| --- | --- | --- |
| ## | print(row.names =  partition\_index | FALSE)  partition\_size |
| ## | 0 | 200 |
| ## | 1 | 200 |
| ## | 2 | 200 |
| ## | 3 | 200 |
| ## | 4 | 200 |

## sdf\_unnest\_longer() and sdf\_unnest\_wider()

sdf\_unnest\_longer() and sdf\_unnest\_wider() are the equivalents of

tidyr::unnest\_longer() and tidyr::unnest\_wider() for Spark dataframes. sdf\_unnest\_longer() expands all elements in a struct column into multiple rows, and sdf\_unnest\_wider() expands them into multiple columns. As illustrated with an example dataframe below,

library(sparklyr)

sc <- spark\_connect(master = "local") sdf <- copy\_to(

sc, tibble::tibble(

id = seq(3), attribute = list(

list(name = "Alice", grade = "A"), list(name = "Bob", grade = "B"), list(name = "Carol", grade = "C")

)

)

)

sdf %>%

sdf\_unnest\_longer(col = record, indices\_to = "key", values\_to = "value") %>%

print()

evaluates to

## # Source: spark [?? x 3]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ##  ## |  | id | value | key |
| ## | 1 | 1 | A | grade |
| ## | 2 | 1 | Alice | name |
| ## | 3 | 2 | B | grade |
| ## | 4 | 2 | Bob | name |
| ## | 5 | 3 | C | grade |
| ## | 6 | 3 | Carol | name |

whereas

sdf %>%

sdf\_unnest\_wider(col = record) %>% print()

evaluates to

## # Source: spark [?? x 3]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ##  ## |  | id | grade | name |
| ## | 1 | 1 | A | Alice |
| ## | 2 | 2 | B | Bob |
| ## | 3 | 3 | C | Carol |

# RDS-based serialization routines

Some readers must be wondering why a brand new serialization format would need to be

implemented in sparklyr at all. Long story short, the reason is that RDS serialization is a strictly better replacement for its CSV predecessor. It possesses all desirable attributes the CSV format has, while avoiding a number of disadvantages that are common among text-based data formats.

In this section, we will briefly outline why sparklyr should support at least one serialization format other than arrow, deep-dive into issues with CSV-based serialization, and then show how the new RDS-based serialization is free from those issues.

## Why arrow is not for everyone?

To transfer data between Spark and R correctly and efficiently, sparklyr must rely on some data serialization format that is well-supported by both Spark and R. Unfortunately, not many serialization formats satisfy this requirement, and among the ones that do are text-based formats such as CSV and JSON, and binary formats such as Apache Arrow, Protobuf, and as of recent, a small subset of RDS version 2. Further complicating the matter is the additional consideration that sparklyr should support at least one serialization format whose implementation can be fully self-contained within the sparklyr code base, i.e., such serialization should not depend on any external R package or system library, so that it can accommodate users who want to use sparklyr but who do not necessarily have the required C++ compiler tool chain and other system dependencies for setting up R packages such as arrow or protolite. Prior to sparklyr 1.5, CSV-based serialization was the default alternative to fallback to when users do not have the arrow package installed or when the type of data being transported from R to Spark is unsupported by the version of arrow available.

## Why is the CSV format not ideal?

There are at least three reasons to believe CSV format is not the best choice when it comes to exporting data from R to Spark.

One reason is efficiency. For example, a double-precision floating point number such as

.Machine$double.eps needs to be expressed as "2.22044604925031e-16" in CSV format in order to not incur any loss of precision, thus taking up 20 bytes rather than 8 bytes.

But more important than efficiency are correctness concerns. In a R dataframe, one can store both NA\_real\_ and NaN in a column of floating point numbers. NA\_real\_ should ideally translate to null within a Spark dataframe, whereas NaN should continue to be NaN when transported from R to Spark. Unfortunately, NA\_real\_ in R becomes indistinguishable from NaN once serialized in CSV format, as evident from a quick demo shown below:

original\_df <- data.frame(x = c(NA\_real\_, NaN))

original\_df %>% dplyr::mutate(is\_nan = is.nan(x)) %>% print() ## x is\_nan

## 1 NA FALSE ## 2 NaN TRUE

csv\_file <- "/tmp/data.csv"

write.csv(original\_df, file = csv\_file, row.names = FALSE) deserialized\_df <- read.csv(csv\_file)

deserialized\_df %>% dplyr::mutate(is\_nan = is.nan(x)) %>% print() ## x is\_nan

## 1 NA FALSE ## 2 NA FALSE

Another correctness issue very much similar to the one above was the fact that "NA" and NA within a string column of an R dataframe become indistinguishable once serialized in CSV format

## RDS to the rescue!

RDS format is one of the most widely used binary formats for serializing R objects. Among advantages of the RDS format are efficiency and accuracy: it has a reasonably efficient implementation in base R, and supports all R data types.

Also worth noticing is the fact that when an R dataframe containing only data types with sensible equivalents in Apache Spark (e.g., RAWSXP, LGLSXP, CHARSXP, REALSXP, etc) is saved using RDS version 2, (e.g., serialize(mtcars, connection = NULL, version = 2L, xdr

= TRUE)), only a tiny subset of the RDS format will be involved in the serialization process, and implementing deserialization routines in Scala capable of decoding such a restricted subset of RDS constructs is in fact a reasonably simple and straightforward task.

Last but not least, because RDS is a binary format, it allows NA\_character\_, "NA", NA\_real\_, and NaN to all be encoded in an unambiguous manner, hence allowing sparklyr

1.5 to avoid all correctness issues detailed above in non-arrow serialization use cases.

## Other benefits of RDS serialization

In addition to correctness guarantees, RDS format also offers quite a few other advantages.

One advantage is of course performance: for example, importing a non-trivially-sized dataset such as nycflights13::flights from R to Spark using the RDS format in sparklyr 1.5 is roughly 40%-50% faster compared to CSV-based serialization in sparklyr 1.4. The current RDS- based implementation is still nowhere as fast as arrow-based serialization though (arrow is about 3-4x faster), so for performance-sensitive tasks involving heavy serialization, arrow should still be the top choice.

Another advantage is that with RDS serialization, sparklyr can import R dataframes containing raw columns directly into binary columns in Spark. Thus, use cases such as the one below will work in sparklyr 1.5

library(sparklyr)

sc <- spark\_connect(master = "local") tbl <- tibble::tibble(

x = list(serialize("sparklyr", NULL), serialize(c(123456, 789), NULL))

)

sdf <- copy\_to(sc, tbl)

While most sparklyr users probably won’t find this capability of importing binary columns to Spark immediately useful in their typical sparklyr::copy\_to() or sparklyr::collect() usages, it does play a crucial role in reducing serialization overheads in the Spark-based foreach parallel backend that was first introduced in sparklyr 1.2. This is because Spark workers can directly fetch the serialized R closures to be computed from a binary Spark column instead of extracting those serialized bytes from intermediate representations such as base64-

encoded strings. Similarly, the R results from executing worker closures will be directly available in RDS format which can be efficiently deserialized in R, rather than being delivered in other less efficient formats.