`sparklyr` 1.3 is now available on CRAN, with the following major new features:

- Higher-order Functions to easily manipulate arrays and structs
- Support for Apache Avro, a row-oriented data serialization framework
- Custom Serialization using R functions to read and write any data format
- Other Improvements such as compatibility with EMR 6.0 & Spark 3.0, and initial support for Flint time series library

To install `sparklyr` 1.3 from CRAN, run

```
install.packages("sparklyr")
```

In this post, we shall highlight some major new features introduced in sparklyr 1.3, and showcase scenarios where such features come in handy. While a number of enhancements and bug fixes (especially those related to `spark_apply()`, Apache Arrow, and secondary Spark connections) were also an important part of this release, they will not be the topic of this post, and it will be an easy exercise for the reader to find out more about them from the sparklyr NEWS file.

Higher-order Functions

Higher-order functions are built-in Spark SQL constructs that allow user-defined lambda expressions to be applied efficiently to complex data types such as arrays and structs. As a quick demo to see why higher-order functions are useful, let's say one day Scrooge McDuck dove into his huge vault of money and found large quantities of pennies, nickels, dimes, and quarters. Having an impeccable taste in data structures, he decided to store the quantities and face values of everything into two Spark SQL array columns:

```
library(sparklyr)

sc <- spark_connect(master = "local", version = "2.4.5")
coins_tbl <- copy_to(
  sc,
  tibble::tibble(
    quantities = list(c(4000, 3000, 2000, 1000)),
    values = list(c(1, 5, 10, 25))
  )
)
```

Thus declaring his net worth of 4k pennies, 3k nickels, 2k dimes, and 1k quarters. To help Scrooge McDuck calculate the total value of each type of coin in sparklyr 1.3 or above, we can apply `hof_zip_with()`, the sparklyr equivalent of ZIP_WITH, to `quantities` column and `values` column, combining pairs of elements from arrays in both columns. As you might have guessed, we also need to specify how to combine those elements, and what better way to accomplish that than a concise one-sided formula `~ .x * .y` in R, which says we want (quantity * value) for each type of coin? So, we have the following:

```
result_tbl <- coins_tbl %>%
  hof_zip_with(~ .x * .y, dest_col = total_values) %>%
  dplyr::select(total_values)

result_tbl %>% dplyr::pull(total_values)
```

```
[1]  4000 15000 20000 25000
```

With the result `4000 15000 20000 25000` telling us there are in total $40 dollars worth of pennies, $150 dollars worth of nickels, $200 dollars worth of dimes, and $250 dollars worth of quarters, as expected.

Using another sparklyr function named `hof_aggregate()`, which performs an AGGREGATE operation in Spark, we can then compute the net worth of Scrooge McDuck based on `result_tbl`, storing the result in a new column named `total`. Notice for this aggregate operation to work, we need to ensure the starting value of aggregation has data type (namely, `BIGINT`) that is consistent with the data type of `total_values` (which is `ARRAY`), as shown below:

```
result_tbl %>%
  dplyr::mutate(zero = dplyr::sql("CAST (0 AS BIGINT)")) %>%
  hof_aggregate(start = zero, ~ .x + .y, expr = total_values, dest_col = total)
%>%
  dplyr::select(total) %>%
  dplyr::pull(total)
```

```
[1] 64000
```

So Scrooge McDuck's net worth is $640 dollars.

Other higher-order functions supported by Spark SQL so far include `transform`, `filter`, and `exists`, as documented in here, and similar to the example above, their counterparts (namely, `hof_transform()`, `hof_filter()`, and `hof_exists()`) all exist in sparklyr 1.3, so that they can be integrated with other `dplyr` verbs in an idiomatic manner in R.

Avro

Another highlight of the sparklyr 1.3 release is its built-in support for Avro data sources. Apache Avro is a widely used data serialization protocol that combines the efficiency of a binary data format with the flexibility of JSON schema definitions. To make working with Avro data sources simpler, in sparklyr 1.3, as soon as a Spark connection is instantiated with `spark_connect(..., package = "avro")`, sparklyr will automatically figure out which version of `spark-avro` package to use with that connection, saving a lot of potential headaches for sparklyr users trying to determine the correct version of `spark-avro` by themselves. Similar to how `spark_read_csv()` and `spark_write_csv()` are in place to work with CSV data, `spark_read_avro()` and `spark_write_avro()` methods were implemented in sparklyr 1.3 to facilitate reading and writing Avro files through an Avro-capable Spark connection, as illustrated in the example below:

```
library(sparklyr)

# The `package = "avro"` option is only supported in Spark 2.4 or higher
sc <- spark_connect(master = "local", version = "2.4.5", package = "avro")

sdf <- sdf_copy_to(
  sc,
  tibble::tibble(
    a = c(1, NaN, 3, 4, NaN),
    b = c(-2L, 0L, 1L, 3L, 2L),
    c = c("a", "b", "c", "", "d")
  )
)

# This example Avro schema is a JSON string that essentially says all columns
# ("a", "b", "c") of `sdf` are nullable.
avro_schema <- jsonlite::toJSON(list(
  type = "record",
  name = "topLevelRecord",
  fields = list(
    list(name = "a", type = list("double", "null")),
    list(name = "b", type = list("int", "null")),
    list(name = "c", type = list("string", "null"))
  )
), auto_unbox = TRUE)
```

```
# persist the Spark data frame from above in Avro format
spark_write_avro(sdf, "/tmp/data.avro", as.character(avro_schema))

# and then read the same data frame back
spark_read_avro(sc, "/tmp/data.avro")
```

```
# Source: spark [?? x 3]
      a     b c

  1     1    -2 "a"
  2   NaN     0 "b"
  3     3     1 "c"
  4     4     3 ""
  5   NaN     2 "d"
```

Custom Serialization

In addition to commonly used data serialization formats such as CSV, JSON, Parquet, and Avro, starting from sparklyr 1.3, customized data frame serialization and deserialization procedures implemented in R can also be run on Spark workers via the newly implemented `spark_read()` and `spark_write()` methods. We can see both of them in action through a quick example below, where `saveRDS()` is called from a user-defined writer function to save all rows within a Spark data frame into 2 RDS files on disk, and `readRDS()` is called from a user-defined reader function to read the data from the RDS files back to Spark:

```
library(sparklyr)

sc <- spark_connect(master = "local")
sdf <- sdf_len(sc, 7)
paths <- c("/tmp/file1.RDS", "/tmp/file2.RDS")

spark_write(sdf, writer = function(df, path) saveRDS(df, path), paths = paths)
spark_read(sc, paths, reader = function(path) readRDS(path), columns = c(id =
"integer"))
```

```
# Source: spark [?? x 1]
     id

1      1
2      2
3      3
4      4
5      5
6      6
7      7
```

Other Improvements

Sparklyr.flint

Sparklyr.flint is a sparklyr extension that aims to make functionalities from the Flint time-series library easily accessible from R. It is currently under active development. One piece of good news is that, while the original Flint library was designed to work with Spark 2.x, a slightly modified fork of it will work well with Spark 3.0, and within the existing sparklyr extension framework. `sparklyr.flint` can automatically determine which version of the Flint library to load based on the version of Spark it's connected to. Another bit of good news is, as previously mentioned, `sparklyr.flint` doesn't know too much about its own destiny yet. Maybe you can play an active part in shaping its future!

EMR 6.0

This release also features a small but important change that allows sparklyr to correctly connect to the version of Spark 2.4 that is included in Amazon EMR 6.0.

Previously, sparklyr automatically assumed any Spark 2.x it was connecting to was built with Scala 2.11 and attempted to load any required Scala artifacts built with Scala 2.11 as well. This became problematic when connecting to Spark 2.4 from Amazon EMR 6.0, which is built with Scala 2.12. Starting from sparklyr 1.3, such problem can be fixed by simply specifying `scala_version = "2.12"` when calling `spark_connect()` (e.g., `spark_connect(master = "yarn-client", scala_version = "2.12")`).

Spark 3.0

Last but not least, it is worthwhile to mention sparklyr 1.3.0 is known to be fully compatible with the recently released Spark 3.0. We highly recommend upgrading your copy of sparklyr to 1.3.0 if you plan to have Spark 3.0 as part of your data workflow in future.