The Problem

Real life data is often riddled with missing values – or `NA`s – where no data value are stored for the variable in observation. Missing data such as this can have a significant effect on the conclusions which can be drawn from the data. For example individuals dropping out of a study or subjects not properly reporting responses. A common solution to this problem is to fill those `NA` values with the most recent non-`NA` value prior to it – this is called the Last Observation Carried Forward (LOCF) method.

This blog post will look at how to implement this particular solution using a combination of {dplyr}, {dbplyr} and {sparklyr}.

Solving the Problem with {dbplyr}

For this problem we will be using the following temperature data.

```
# # Source: spark [?? x 3]
#     year month  temp
#
#  1  2016     1   6.9
#  2  2016     2  NA
#  3  2016     3  10.1
#  4  2016     4  11.5
#  5  2016     5  16.3
#  6  2016     6  19.1
#  7  2016     7  NA
#  8  2016     8  NA
#  9  2016     9  17.6
# 10  2016    10  13.6
# # … with more rows
```

It might be reasonable to assume that we can fill the `NA` values of the `temp` column with the previous month's data.

The strategy to forward fill in Spark is to use what's known as a window function. A window function performs a calculation across a set of table rows that are somehow related to the current row. This is comparable to the type of calculation that can be done with an aggregate function. But unlike regular aggregate functions, use of a window function does not cause rows to become grouped into a single output row — the rows retain their separate identities. Behind the scenes, the window function is able to access more than just the current row of the query result.

We can perform window functions in Spark using `dbplyr::win_over()`. In this function we define the window `frame` which – for the LOCF method – includes all rows from the beginning of time up to the current row, i.e. `-Inf` to `0`. We need to specify that our data follow a particular `order` which is the `month` in ascending order. We also need to partition the data by the `year` which can be thought of as acting like `dplyr::group_by()`. Finally we define the expression to perform on the data which, since we need the last non-`NA` value, can be obtained using the Spark SQL function `last()`.

> `last(expr[, isIgnoreNull])` – Returns the last value of expr for a group of

rows. If `isIgnoreNull` is `true`, returns only non-null values.

Putting this all together, we have the following operation.

```
temperature <- dplyr::mutate(
  .data = temperature,
  tempLOCF = dbplyr::win_over(
    expr = dplyr::sql("last(temp, true)"),
    partition = "year",
    order = "month",
    frame = c(-Inf, 0),
    con = sc
  )
)
temperature
# # Source: spark [?? x 4]
#      year month  temp tempLOCF
#
#  1  2016     1   6.9      6.9
#  2  2016     2  NA        6.9
#  3  2016     3  10.1     10.1
#  4  2016     4  11.5     11.5
#  5  2016     5  16.3     16.3
#  6  2016     6  19.1     19.1
#  7  2016     7  NA       19.1
#  8  2016     8  NA       19.1
#  9  2016     9  17.6     17.6
# 10  2016    10  13.6     13.6
# # … with more rows
```

So now we can see that we have filled the values as expected. The way this works is that {dbplyr} takes our regular {dplyr} code and translates it to SQL code. We can see the SQL that {dbplyr} generates for us with `dbplyr::sql_render()`.

```
dbplyr::sql_render(query = temperature)
#  SELECT `year`, `month`, `temp`, last(temp, true) OVER (PARTITION BY
`year` ORDER BY `month` ROWS UNBOUNDED PRECEDING) AS `tempLOCF`
# FROM `temperature`
```

Note that the `con = sc` here is required such that {dbplyr} can set up the SQL for the particular database engine you are working with – in this case, Spark.

Bonus – Next Observation Carried Backwards

The Next Observation Carried Backwards method is very similar to LOCF except, you guessed it, we carry the next non-`NA` value backwards. This can be achieved with the exact same code expect the `frame` changes to be from the current row to the end of the partition (or group) and we take the `first()` non-`NA` value from the window frame.

```
dplyr::mutate(
  .data = temperature,
  tempNOCB = dbplyr::win_over(
    expr = dplyr::sql("first(temp, true)"),
    partition = "year",
    order = "month",
    frame = c(0, Inf),
    con = sc
  )
)
# # Source: spark [?? x 5]
#     year month  temp tempLOCF tempNOCB
#
#  1 2016     1   6.9      6.9      6.9
#  2 2016     2  NA        6.9     10.1
#  3 2016     3  10.1     10.1     10.1
#  4 2016     4  11.5     11.5     11.5
#  5 2016     5  16.3     16.3     16.3
#  6 2016     6  19.1     19.1     19.1
#  7 2016     7  NA       19.1     17.6
#  8 2016     8  NA       19.1     17.6
#  9 2016     9  17.6     17.6     17.6
# 10 2016    10  13.6     13.6     13.6
# # … with more rows
```

For what it's worth, this functionality very recently became available via the `tidyr::fill()`, for which there is now a `tbl_lazy` method in {dbplyr}.