David Vrba
Jul 27, 2021 · 9 min read · Member-only · Listen

Higher-Order Functions with Spark 3.1

Processing Arrays in Spark SQL.

Photo by Donald Giannatti on Unsplash

Complex data structures, such as arrays, structs, and maps are very common in big data processing, especially in Spark. The situation occurs each time we want to represent in one column more 82 | 1 lue on each row, this can be a list of values in the case of array data type or a list of key-value pairs in the case of the map.

The support for processing these complex data types increased since Spark 2.4 by releasing higher-order functions (HOFs). In this article, we will take a look at what higher-order functions are, how they can be efficiently used and what related features were released in the last few Spark releases 3.0 and 3.1.1. For the code, we will use Python API.

After aggregations and window functions that we covered in the last article, HOFs are another group of more advanced transformations in Spark SQL.

Let's first see the difference between the three complex data types that Spark offers.

ArrayType

```
l = [(1, ['the', 'quick', 'braun', 'fox'])]

df = spark.createDataFrame(l, schema=['id', 'words'])

df.printSchema()

root
 |-- id: long (nullable = true)
 |-- words: array (nullable = true)
 |    |-- element: string (containsNull = true)
```

In the example above, we have a DataFrame with two columns, where the column *words* is of the array type, which means that on each row in the DataFrame we can represent a list of values and the list can have different size on each row. Also, the elements of the array have an order. The important property is that the arrays are homogeneous in terms of the element type, which means that all elements must have the same type. To access the elements of the arrays we can use indices as follows:

```
df.withColumn('first_element', col('words')[0])
```

StructType

StructType is used to group together some sub-fields that may have a different type (unlike arrays). Each sub-field has a type and also a name and this must be the same for all rows in the DataFrame. What might be unexpected is that the sub-fields inside a struct have an order, so comparing two structs *s1==s2* that have the same fields but in different order leads to *False*.

Notice the fundamental differences between array and struct:

- array: homogeneous in types, a different size on each row is allowed
- struct: heterogeneous in types, the same schema on each row is required

MapType

You can think of the map type as a mixture of the two previous types: array and struct. Imagine a situation where the schema for each row is not given, you need to support a different number of sub-fields on each row. In such a case you can not use struct. But using an array is not a good option for you either because each element has a name and a value (it is actually a key-value pair) or because the elements have a different type — that would be a good use-case for the map type. With a map type, you can store on each row a different number of key-value pairs, but each key must have the same type and also all values need to be of the same type (which can be different from the type of the keys). The order of the pairs matters.

Transforming Arrays

Before we start talking about transforming arrays, let's first see how we can create an array. The first way we have seen above, where we created the DataFrame from a local list of values. On the other hand, if we already have a DataFrame and we want to group some columns to an array we can use a function *array()* for this purpose. It allows you to create an array from other existing columns, so if you have columns *a*, *b*, *c* and you want to have the values inside an array instead of having it in individual columns you can do it as follows:

```
df.withColumn('my_arr', array('a', 'b', 'c'))
```

Apart from that, there are also some functions that produce an array as a result of a transformation. This is for example the function *split()* that will split a string into an array of words. Another example is *collect_list()* or *collect_set()* which are aggregation functions that will also produce an array.

And in practice, the most common way how to get an array to a DataFrame is by reading the data from a source that supports complex data structures such as Parquet. In this file format, some columns can be stored as arrays, so Spark will naturally read them also as arrays.

Now when we know how to create an array, let's see how arrays can be transformed.

Since Spark 2.4 there are plenty of functions for array transformation. For the complete list of them, check the PySpark documentation. For example, all the functions starting with *array_* can be used for array processing, you can find min-max values, deduplicate the arrays, sort them, join them, and so on. Next, there is also *concat()*, *flatten()*, *shuffle()*, *size()*, *slice()*, *sort_array()*. As you can see, the API is quite mature in this regard and there are lots of operations that you can do with arrays in Spark.

Apart from these aforementioned functions, there is also a group of functions that take as an argument another function that is then applied to each element of the array — these are called Higher-Order Functions (HOFs). The important thing to know about them is that in the Python API, they are supported since 3.1.1 and in Scala API they were released in 3.0. On the other hand, with SQL expressions, you can use them since 2.4.

To see some specific examples, consider the following simple DataFrame:

```
l = [(1, ['prague', 'london', 'tokyo', None, 'sydney'])]

df = spark.createDataFrame(l, ['id', 'cities'])

df.show(truncate=False)

+---+-------------------------------------+
|id |cities                               |
+---+-------------------------------------+
|1  |[prague, london, tokyo, null, sydney]|
+---+-------------------------------------+
```

Let’s say we want to do these five independent tasks:

1. Convert starting letter of each city to upper-case.
2. Get rid of null values in the array.
3. Check if there is an element that starts with the letter t.
4. Check if there is a null value in the array.
5. Sum the number of characters (the length) of each city in the array.

These are some typical examples of problems that can be solved with HOFs. So let's see them one by one:

TRANSFORM

For the first problem, we can use the *transform* HOF, which simply takes an anonymous function, applies it to each element of the original array, and returns another transformed array. The syntax is as follows:

```
df \
.withColumn('cities', transform('cities', lambda x: initcap(x))) \
.show(truncate=False)

+---+------------------------------------+
|id |cities                              |
+---+------------------------------------+
|1  |[Prague, London, Tokyo, null, Sydney]|
+---+------------------------------------+
```

As you can see, the *transform()* takes two arguments, the first one is the array that should be transformed and the second one is an anonymous function. Here, to achieve our transformation, we used *initcap()* inside the anonymous function and it was applied on each element of the array — this is exactly what the *transform* HOF allows us to do. With SQL expressions it can be used as follows:

```
df.selectExpr("id", "TRANSFORM(cities, x -> INITCAP(x)) AS cities")
```

Notice that the anonymous function in SQL is expressed with the arrow (->) notation.

FILTER

In the second problem, we want to filter out null values from the array. This (and any other filtering for that matter) can be handled using the *filter* HOF. It allows us to apply an anonymous function that returns boolean (*True/False*) on each element and it will return a new array that contains only elements for which the function returned *True*:

```
df \
.withColumn('cities', filter('cities', lambda x: x.isNotNull())) \
.show(truncate=False)

+---+-------------------------------+
|id |cities                         |
+---+-------------------------------+
|1  |[prague, london, tokyo, sydney]|
+---+-------------------------------+
```

Here, in the anonymous function we call PySpark function *isNotNull()*. The SQL syntax goes as follows:

```
df.selectExpr("id", "FILTER(cities, x -> x IS NOT NULL) AS cities")
```

EXISTS

In the next problem, we want to check if the array contains elements that satisfy some specific condition. Notice, that this is a more general example of a situation in which we want to check for the presence of some particular element. For example, if we want to check whether the array contains the city *prague*, we could just call the *array_contains* function:

```
df.withColumn('has_prague', array_contains('cities', 'prague'))
```

On the other hand, the *exists* HOF allows us to apply a more general condition to each element. The result is no longer an array as it was with the two previous HOFs, but it is just *True/False*:

```
df \
.withColumn('has_t_city',
  exists('cities', lambda x: x.startswith('t'))) \
.show(truncate=False)

+---+-------------------------------------+----------+
|id |cities                               |has_t_city|
+---+-------------------------------------+----------+
|1  |[prague, london, tokyo, null, sydney]|true      |
+---+-------------------------------------+----------+
```

Here in the anonymous function, we used the PySpark function *startswith()*.

FORALL

In the fourth question, we want to verify if all elements in the array satisfy some condition, in our example, we want to check if they are all not null:

```
df \
.withColumn('nulls_free',forall('cities', lambda x: x.isNotNull()))\
.show(truncate=False)

+---+-------------------------------------+----------+
|id |cities                               |nulls_free|
+---+-------------------------------------+----------+
|1  |[prague, london, tokyo, null, sydney]|false     |
+---+-------------------------------------+----------+
```

As you can see, *forall* is quite similar to *exists*, but now we are checking if the condition holds for all elements, before we were looking for at least one.

AGGREGATE

In the last problem, we want to sum the lengths of each word in the array. It is an example of a situation in which we want to reduce the entire array to a single value and for this kind of problem we can use the HOF *aggregate*.

```
df \
.withColumn('cities', filter('cities', lambda x: x.isNotNull())) \
.withColumn('cities_len',
  aggregate('cities', lit(0), lambda x, y: x + length(y))) \
.show(truncate=False)

+---+------------------------------+----------+
|id |cities                        |cities_len|
+---+------------------------------+----------+
|1  |[prague, london, tokyo, sydney]|23        |
+---+------------------------------+----------+
```

And with SQL:

```
df \
.withColumn("cities", filter("cities", lambda x: x.isNotNull())) \
.selectExpr(
    "cities",
    "AGGREGATE(cities, 0,(x, y) -> x + length(y)) AS cities_len"
)
```

As you can see the syntax is slightly more complex as compared to the previous HOFs. The *aggregate* takes more arguments, the first one is still the array that we want to transform, the second argument is the initial value that we want to start with. In our case, the initial value is zero (*lit(0)*) and we will be adding to it the length of each city. The third argument is the anonymous function and now this function itself takes two arguments — the first one (in our example *x*) is the running buffer to which we are adding the length of the next element that is represented with the second argument (in our example *y*).

Optionally, there can be a fourth argument provided which is another anonymous function that transforms the final result. This is useful if we want to do a more complex aggregation, for example, if we want to compute the average length, we need to keep around two values, the *sum* and also the *count* and we would divide them in the last transformation as follows:

```
(
    df
    .withColumn('cities', filter('cities', lambda x: x.isNotNull()))
    .withColumn('cities_avg_len',
        aggregate(
            'cities',
            struct(lit(0).alias('sum'), lit(0).alias('count')),
            lambda x, y: struct(
                (x.sum + length(y)).alias('sum'),
                (x.count + 1).alias('count')
            ),
            lambda x: x.sum / x.count
        )
    )
).show(truncate=False)

+---+------------------------------+--------------+
|id |cities                        |cities_avg_len|
+---+------------------------------+--------------+
|1  |[prague, london, tokyo, sydney]|5.75          |
+---+------------------------------+--------------+
```

As you can see, this is a more advanced example in which we need to keep around two values during the aggregation and we represent them using *struct()* that has two subfields *sum* and *count.* Using the first anonymous function we compute the final sum of all lengths and also the count which is the number of summed elements. In the second anonymous function, we just divide these two values to get the final average. Also notice, that before using *aggregate* we first filtered out null values because if we keep the null value in the array, the sum (and also the average) will become null.

To see another example of the aggregate HOF where it is used with SQL expressions, check this Stack Overflow question.

Apart from these five aforementioned HOFs, there is also *zip_with* that can be used to merge two arrays into a single one. Besides that, there are also other HOFs such as *map_filter*, *map_zip_with*, *transform_keys*, and *transform_values* that are used with maps and we will take a look at them in a future article.

Conclusion

In this article, we covered higher-order functions (HOFs) which is a feature that was released in Spark 2.4. First, it was supported only with SQL expressions, but since 3.1.1 it is supported also in the Python API. We have seen examples of five HOFs, that allow us to transform, filter, check for existence, and aggregate elements in the Spark arrays. Before HOFs were released, most of these problems had to be solved using User-Defined functions. The HOF approach is however more efficient in terms of performance and to see some performance benchmarks, see my other recent article that shows some concrete numbers.