



Higher Order Functions

Herman van Hövell @westerflyer

2018-10-03, London Spark Summit EU 2018



About Me

- Software Engineer @Databricks
Amsterdam office
- Apache Spark Committer and
PMC member
- In a previous life Data Engineer &
Data Analyst.



Complex Data

Complex data types in Spark SQL

- Struct. For example: `struct(a: Int, b: String)`
- Array. For example: `array(a: Int)`
- Map. For example: `map(key: String, value: Int)`

This provides primitives to build tree-based data models

- High expressiveness. Often alleviates the need for 'flat-earth' multi-table designs.
- More natural, reality like data models

Complex Data - Tweet JSON

```
{
  "created_at": "Wed Oct 03 11:41:57 +0000 2018",
  "id_str": "994633657141813248",
  "text": "Looky nested data #spark #sseu",
  "display_text_range": [0, 140],
  "user": {
    "id_str": "12343453",
    "screen_name": "Westerflyer"
  },
  "extended_tweet": {
    "full_text": "Looky nested data #spark #sseu",
    "display_text_range": [0, 249],
    "entities": {
      "hashtags": [{
        "text": "spark",
        "indices": [211, 225]
      }, {
        "text": "sseu",
        "indices": [239, 249]
      }]
    }
  }
}
```

```
root
|-- created_at: string (nullable = true)
|-- id_str: string (nullable = true)
|-- text: string (nullable = true)
|-- user: struct (nullable = true)
|   |-- id_str: string (nullable = true)
|   |-- screen_name: string (nullable = true)
|-- display_text_range: array (nullable = true)
|   |-- element: long (containsNull = true)
|-- extended_tweet: struct (nullable = true)
|   |-- full_text: string (nullable = true)
|   |-- display_text_range: array (nullable = true)
|       |-- element: long (containsNull = true)
|   |-- entities: struct (nullable = true)
|       |-- hashtags: array (nullable = true)
|           |-- element: struct (containsNull = true)
|               |-- indices: array (nullable = true)
|                   |-- element: long (containsNull = true)
|                   |-- text: string (nullable = true)
```

Manipulating Complex Data

Structs are easy :)

Maps/Arrays not so much...

- Easy to read single values/retrieve keys
- Hard to transform or to summarize

Transforming an Array

Let's say we want to add 1 to every element of the `vals` field of every row in an input table.

Id	Vals
1	[1, 2, 3]
2	[4, 5, 6]



Id	Vals
1	[2, 3, 4]
2	[5, 6, 7]

How would we do this?

Transforming an Array

Option 1 - Explode and Collect

```
select    id,  
          collect_list(val + 1) as vals  
from      (select id,  
                explode(vals) as val  
            from    input_tbl) x  
group by id
```

Transforming an Array

Option 1 - Explode and Collect - Explode

```
select    id,  
          collect_list(val + 1) as vals  
from      (select id,  
                explode(vals) as val  ← 1. Explode  
            from    input_tbl) x  
group by id
```


Transforming an Array

Option 1 - Explode and Collect - Explode

Id	Vals
1	[1, 2, 3]
2	[4, 5, 6]



Id	Val
1	1
1	2
1	3
2	4
2	5
2	6

Transforming an Array

Option 1 - Explode and Collect - Collect

```
select    id,  
          collect_list(val + 1) as vals ← 2. Collect  
from      (select id,  
            explode(vals) as val ← 1. Explode  
            from    input_tbl) x  
group by id
```

Transforming an Array

Option 1 - Explode and Collect - Collect

Id	Val
1	1 + 1
1	2 + 1
1	3 + 1
2	4 + 1
2	5 + 1
2	6 + 1

]

]



Id	Vals
1	[2, 3, 4]
2	[5, 6, 7]


Transforming an Array

Option 1 - Explode and Collect - Complexity

```
== Physical Plan ==
ObjectHashAggregate(keys=[id], functions=[collect_list(val + 1)])
+- Exchange hashpartitioning(id, 200)
   +- ObjectHashAggregate(keys=[id], functions=[collect_list(val + 1)])
      +- Generate explode(vals), [id], false, [val]
         +- FileScan parquet default.input_tbl
```

Transforming an Array

Option 1 - Explode and Collect - Complexity

```
== Physical Plan ==
ObjectHashAggregate(keys=[id], functions=[collect_list(val + 1)])
+- Exchange hashpartitioning(id, 200) 
   +- ObjectHashAggregate(keys=[id], functions=[collect_list(val + 1)])
      +- Generate explode(vals), [id], false, [val]
         +- FileScan parquet default.input_tbl
```

- Shuffles the data around, which is very expensive
- collect_list does not respect pre-existing ordering

Transforming an Array

Option 1 - Explode and Collect - Pitfalls

Keys need to be unique

Id	Vals
1	[1, 2, 3]
1	[4, 5, 6]



Id	Vals
1	[5, 6, 7, 2, 3, 4]

Values need to have data

Id	Vals
1	null
2	[4, 5, 6]



Id	Vals
2	[5, 6, 7]

Transforming an Array

Option 2 - Scala UDF

```
def addOne (values: Seq[Int]): Seq[Int] = {  
  values.map(value => value + 1)  
}
```

Transforming an Array

Option 2 - Scala UDF

```
def addOne(values: Seq[Int]): Seq[Int] = {  
  values.map(value => value + 1)  
}  
val plusOneInt = spark.udf.register("plusOneInt", addOne(_:Seq[Int]):Seq[Int])
```


Transforming an Array

Option 2 - Scala UDF

```
def addOne(values: Seq[Int]): Seq[Int] = {  
  values.map(value => value + 1)  
}  
val plusOneInt = spark.udf.register("plusOneInt", addOne(_:Seq[Int]):Seq[Int])  
  
val newDf = spark.table("input_tbl").select($"id", plusOneInt($"vals"))
```

Transforming an Array

Option 2 - Scala UDF

Pros

- Is faster than Explode & Collect
- Does not suffer from correctness pitfalls

Cons

- Is relatively slow, we need to do a lot serialization
- You need to register UDFs per type
- Does not work for SQL
- Clunky

When are you going to talk
about
Higher Order Functions?

Higher Order Functions


Let's take another look at Option 2 - Scala UDF

```
def addOne(values: Seq[Int]): Seq[Int] = {  
  values.map(value => value + 1)  
}  
val plusOneInt = spark.udf.register("plusOneInt",  
  addOne(_:Seq[Int]):Seq[Int])  
  
val newDf = spark.table("input_tbl").select($"id", plusOneInt($"vals"))
```

Higher Order Functions



Let's take another look at Option 2 - Scala UDF

```
def addOne(values: Seq[Int]): Seq[Int] = {  
  values.map(value => value + 1)  
}  
val plusOneInt = spark.udf.register("plusOneInt",  
  addOne(_:Seq[Int]):Seq[Int])  
  
val newDf = spark.table("input_tbl").select($"id", plusOneInt($"vals"))
```

 Higher Order Function

Higher Order Functions

Let's take another look at Option 2 - Scala UDF

```
def addOne(values: Seq[Int]): Seq[Int] = {  
  values.map(value => value + 1)  Anonymous 'Lambda' Function  
}  
 Higher Order Function  
val plusOneInt = spark.udf.register("plusOneInt",  
  addOne(_:Seq[Int]):Seq[Int])  
  
val newDf = spark.table("input_tbl").select($"id", plusOneInt($"vals"))
```

Can we do the same for Spark SQL?

Higher Order Functions in Spark SQL

```
select  id, transform(vals, val -> val + 1) as vals
from    input_tbl
```

- Spark SQL native code: fast & no serialization needed
- Works for SQL

Higher Order Functions in Spark SQL

```
select id, transform(vals, val -> val + 1) as vals  
from   input_tbl
```




Higher Order Function

`transform` is the Higher Order Function. It takes an input array and an expression, it applies this expression to each element in the array

Higher Order Functions in Spark SQL

```
select id, transform(vals, val -> val + 1) as vals
from   input_tbl
```

 Anonymous 'Lambda' Function

`val -> val + 1` is the lambda function. It is the operation that is applied to each value in the array. This function is divided into two components separated by a `->` symbol:

1. The Argument list.
2. The expression used to calculate the new value.

Higher Order Functions in Spark SQL

Nesting

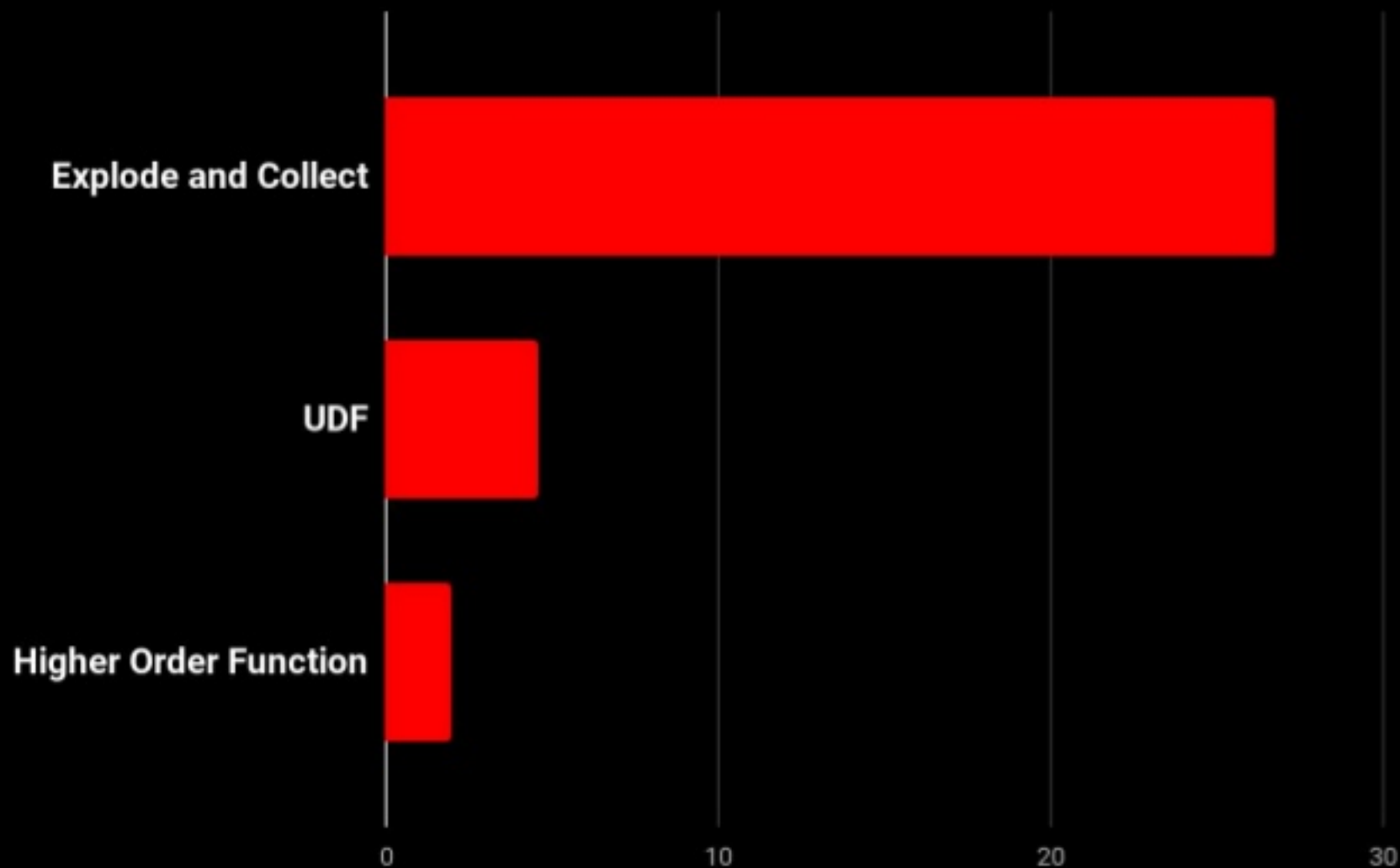
```
select  id,  
        transform(vals, val ->  
                  transform(val, e -> e + 1)) as vals  
from    nested_input_tbl
```

Capture

```
select  id,  
        ref_value,  
        transform(vals, val -> ref_value + val) as vals  
from    nested_input_tbl
```

Didn't you say these were
faster?

Performance



Higher Order Functions in Spark SQL

Spark 2.4 will ship with following higher order functions:

Array

- transform
- filter
- exists
- aggregate/reduce
- zip_with

Map

- transform_keys
- transform_values
- map_filter
- map_zip_with

A lot of new collection based expression were also added...

Future work

Arrays and Maps have received a lot of love. However working with wide structs fields is still non-trivial (a lot of typing). We can do better here:

- The following dataset functions should work for nested fields:
 - withColumn()
 - withColumnRenamed()
- The following functions should be added for struct fields:
 - select()
 - withColumn()
 - withColumnRenamed()

Disclaimer: All of this is speculative and has not been discussed on the Dev list!



Questions?