databricks6.1 Higher-order Functions

Higher-order functions

Higher order functions in Spark SQL allow you to work directly with complex data types. When working with hierarchical data, as we were in the previous lesson and lab, records are frequently stored as array or map type objects. This lesson will demonstrate how to use higher-order functions to transform, filter, and flag array data while preserving the original structure. In this notebook, we work strictly with arrays of strings; in the subsquent notebook, you will work with more functions and numerical data. Skilled application of these functions can make your work with this kind of data faster, more powerful, and more reliable.

In this notebook, you will:

- Apply higher-order functions (`TRANSFORM` , `FILTER` , `EXISTS`) to arrays of strings

Run the following queries to learn about how to work with higher-order functions.

Getting Started

Run the cell below to set up your classroom environment

```
%run ../Includes/Classroom-Setup
```

Mounting course-specific datasets to **/mnt/training**...
Datasets are already mounted to **/mnt/training** from **s3a://databricks-corp-training/common**

```
res1: Boolean = false
```

```
res2: Boolean = false
```

Working with Text Data

We can use higher-order functions to easily work with arrays of text data. The exercises in this section are meant to demonstrate the `TRANSFORM`, `FILTER`, and `EXISTS` functions to manipulate data and create flags for when a value does or does not exist.

These examples use data collected about Databricks blog posts. Run the cell below to create the table. Then, run the next cell to view the schema.

In this data set, the `authors` and `categories` columns are both ArrayType; we'll be using these columns with higher-order functions.

```
CREATE TABLE IF NOT EXISTS DatabricksBlog
  USING json
  OPTIONS (
    path "dbfs:/mnt/training/databricks-blog.json",
    inferSchema "true"
  )
```

OK

```
DESCRIBE DatabricksBlog
```

	col_name	data_type	comment
1	authors	array<string>	null
2	categories	array<string>	null
3	content	string	null
4	creator	string	null
5	dates	struct<createdOn:string,publishedOn:string,tz:string>	null
6	description	string	null
7	id	bigint	null

Showing all 11 rows.

Filter

Filter (https://spark.apache.org/docs/latest/api/sql/#filter) allows us to create a new column based on whether or not values in an array meet a certain condition. Let's say we want to remove the category `"Engineering Blog"` from all records in our `categories` column. I can use the `FILTER` function to create a new column that excludes that value from the each array.

Let's dissect this line of code to better understand the function:

```
FILTER (categories, category -> category <> "Engineering Blog") woEngineer
```

`FILTER` : the name of the higher-order function
`categories` : the name of our input array
`category` : the name of the iterator variable. You choose this name and then use it in the lambda function. It iterates over the array, cycling each value into the function one at a time.
`->` : Indicates the start of a function
`category <> "Engineering Blog"` : This is the function. Each value is checked to see if it **is different** than the value `"Engineering Blog"`. If it is, it gets filtered into the new column, `woEnginieering`

```
SELECT
  categories,
  FILTER (categories, category -> category <> "Engineering Blog") woEngineering
FROM DatabricksBlog
```

	categories	woEngineering
1	["Company Blog", "Partners"]	["Company Blog"
2	["Apache Spark", "Engineering Blog", "Machine Learning"]	["Apache Spark",
3	["Company Blog", "Partners"]	["Company Blog"
4	["Apache Spark", "Engineering Blog"]	["Apache Spark"]
5	["Apache Spark", "Engineering Blog"]	["Apache Spark"]
6	["Apache Spark", "Ecosystem", "Engineering Blog"]	["Apache Spark",
7	["Company Blog", "Customers"]	["Company Blog"

Showing all 365 rows.

Filter, subqueries, and `WHERE`

You may write a filter that produces a lot of empty arrays in the created column. When that happens, it can be useful to use a `WHERE` clause to show only non-empty array values in the returned column.

In this example, we accomplish that by using a subquery. A **subquery** in SQL is a query within a query. They are useful for performing an operations in multiple steps. In

```
SELECT
  *
FROM
  (
    SELECT
      authors, title,
      FILTER(categories, category -> category = "Engineering Blog") AS blogType
    FROM
      DatabricksBlog
  )
WHERE
  size(blogType) > 0
```

	authors
1	["Tathagata Das"]
2	["Michael Armbrust", "Reynold Xin"]
3	["Patrick Wendell"]
4	["Ali Ghodsi", "Ahir Reddy"]
5	["Jai Ranganathan", "Matei Zaharia"]
6	["Ion Stoica"]
7	["Ahir Reddy", "Reynold Xin"]

Showing all 141 rows.

Exists

Exists (https://spark.apache.org/docs/latest/api/sql/#exists) tests whether a statement is true for one or more elements in an array. Let's say we want to flag all blog posts with `"Company Blog"` in the categories field. I can use the `EXISTS` function to mark which entries include that category.

Let's dissect this line of code to better understand the function:

```
EXISTS (categories, c -> c = "Company Blog") companyFlag
```

`EXISTS` : the name of the higher-order function
`categories` : the name of our input array
`c` : the name of the iterator variable. You choose this name and then use it in the lambda function. It iterates over the array, cycling each value into the function one at a time. Note that we're using the same kind as references as in the previous command, but we name the iterator with a single letter
`->` : Indicates the start of a function
`c = "Engineering Blog"` : This is the function. Each value is checked to see if it **is the same as** the value `"Company Blog"`. If it is, it gets flagged into the new column, `companyFlag`

```
SELECT
  categories,
  EXISTS (categories, c -> c = "Company Blog") companyFlag
FROM DatabricksBlog
```

	categories	companyFlag
1	["Company Blog", "Partners"]	true
2	["Apache Spark", "Engineering Blog", "Machine Learning"]	false
3	["Company Blog", "Partners"]	true
4	["Apache Spark", "Engineering Blog"]	false
5	["Apache Spark", "Engineering Blog"]	false
6	["Apache Spark", "Ecosystem", "Engineering Blog"]	false
7	["Company Blog", "Customers"]	true

Showing all 365 rows.

Transform

Transform (https://spark.apache.org/docs/latest/api/sql/#transform) uses the provided function to transform all elements of an array. SQL's built-in functions are designed to operate on a single, simple data type within a cell. They cannot process array values.

Transform can be particularly useful when you want to apply an existing function to each element in an array. In this case, we want to rewrite all of the names in the `categories` column in lowercase.

Let's dissect this line of code to better understand the function:

```
TRANSFORM(categories, cat -> LOWER(cat)) lwrCategories
```

`TRANSFORM` : the name of the higher-order function
`categories` : the name of our input array
`cat` : the name of the iterator variable. You choose this name and then use it in the lambda function. It iterates over the array, cycling each value into the function one at a time. Note that we're using the same kind as references as in the previous command, but we name the iterator with a new variable
`->` : Indicates the start of a function
`LOWER(cat)` : This is the function. For each value in the input array, the built-in function `LOWER()` is applied to transform the word to lowercase.

```
SELECT
  categories, TRANSFORM(categories, cat -> LOWER(cat)) lwrCategories
FROM DatabricksBlog
```

	categories	lwrCategories
1	["Company Blog", "Partners"]	["company blog",
2	["Apache Spark", "Engineering Blog", "Machine Learning"]	["apache spark",
3	["Company Blog", "Partners"]	["company blog",
4	["Apache Spark", "Engineering Blog"]	["apache spark",
5	["Apache Spark", "Engineering Blog"]	["apache spark",
6	["Apache Spark", "Ecosystem", "Engineering Blog"]	["apache spark",
7	["Company Blog", "Customers"]	["company blog",

Showing all 365 rows.

```
%run ../Includes/Classroom-Cleanup
```