databricks6.2 Aggregating and Summarizing Data

Aggregating and summarizing data

Now let's look at some powerful functions we can use to aggregate and summarize data. In this notebook, we will continue to work with hihger-order functions; this time we will apply them to arrays containing numerical data. Also, we will work with additional functions in Spark SQL that can be helpful when presenting data.

In this notebook, you will:

- Apply higher-order functions to numeric data
- Use a `PIVOT` command to create Pivot tables
- Use `ROLLUP` and `CUBE` modifiers to generate subtotals
- Use window functions to perform operations on a group of rows
- Use Databricks visualization tools to visualize and share data

Run the cell below to set up our 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**

```
res4: Boolean = false
```

```
res5: Boolean = false
```

Higher-order functions and numerical data

Each of the higher-order functions we worked with in the last lesson can also be used with numerical data. In this lesson, we demonstrate how each of the functions in the previous lesson work with numeric data, as well as explore some powerful new higher-order functions.

Run the next two cells to create and describe the table we will be working with. You may recognize this table from a previous lesson. Recall that it contains data measuring environmental variability in a collection of data centers. The table `DeviceData` contains the `temps` and `co2Level` arrays we use to demonstrate higher-order functions.

```
DROP TABLE IF EXISTS DCDataRaw;
CREATE TABLE DCDataRaw
USING parquet
OPTIONS (
    PATH "/mnt/training/iot-devices/data-centers/2019-q2-q3"
    );

CREATE TABLE IF NOT EXISTS DeviceData
USING parquet
WITH ExplodeSource
AS
  (
  SELECT
  dc_id,
  to_date(date) AS date,
  EXPLODE (source)
  FROM DCDataRaw
  )
SELECT
  dc_id,
  key device_type,
  date,
  value.description,
  value.ip,
  value.temps,
  value.co2_level co2Level

FROM ExplodeSource;
```

```
OK
```

```
--borra el directorio asociado al archivo dbfs
%fs rm -r dbfs:/user/hive/warehouse/devicedata
```

```
res7: Boolean = true
```

```
DESCRIBE DeviceData;
```

	col_name	data_type	comment
1	dc_id	string	null
2	device_type	string	null
3	date	date	null
4	description	string	null
5	ip	string	null
6	temps	array<int>	null
7	co2Level	array<int>	null

Showing all 7 rows.

Preview data

Let's take a look a sample fo the data so that we con better understand the array values.

```
SELECT
  temps,
  co2Level
FROM DeviceData
TABLESAMPLE (1 ROWS)
```

	temps	co2Level
1	▸ [16, 13, 19, 11, 9, 23, 18, 13, 18, 17, 12, 12]	▸ [1196, 1360, 1125, 1206, 1342, 1198]

Showing all 1 rows.

Filter

Filter operates on arrays containing numeric data just the same as those with text data. In this case, let's imagine that we want to collect all temperatures above a given threshold. Run the cell below to view the example.

```
SELECT
  temps,
  FILTER(temps, t -> t > 18) highTemps
FROM DeviceData
```

	temps	highTemps
1	▸ [16, 13, 19, 11, 9, 23, 18, 13, 18, 17, 12, 12]	▸ [19, 23]
2	▸ [26, 17, 19, 13, 9, 12, 10, 12, 1, 13, 16, 12]	▸ [26, 19]
3	▸ [11, 13, 19, 8, 14, 16, 13, 14, 14, 9, 7, 12]	▸ [19]
4	▸ [20, 18, 20, 18, 11, 14, 17, 24, 17, 15, 19, 22]	▸ [20, 20, 24, 19, 22]

Truncated results, showing first 1000 rows.

Exists

Exists operates on arrays containing numeric data just the same as those with text data. Let's say that we want to flag the records whose temperatures have exceeded a given value. Run the cell below to view the example.

```
SELECT
  temps,
  EXISTS(temps, t -> t > 23) highTempsFlag
FROM DeviceData
```

	temps	highTempsFlag
1	▸ [16, 13, 19, 11, 9, 23, 18, 13, 18, 17, 12, 12]	false
2	▸ [26, 17, 19, 13, 9, 12, 10, 12, 1, 13, 16, 12]	true
3	▸ [11, 13, 19, 8, 14, 16, 13, 14, 14, 9, 7, 12]	false

4	▸ [20, 18, 20, 18, 11, 14, 17, 24, 17, 15, 19, 22]	true	

Truncated results, showing first 1000 rows.

Transform

When using `TRANSFORM` with numeric data, we can apply any built-in function meant to work with a single value or we can name our own set of operations to be applied to each value in the array. This data includes temperature readings taken in Celsius. Each row contains an array of 12 temperature readings. We can use `TRANSFORM` to convert each element of each array to Fahrenheit. To convert from Celsius to Fahrenheit, multiply the temperature in Celsius by 9, divide by 5, and then add 32.

Let's dissect the code below to better understand the function:

`TRANSFORM(temps, t -> ((t * 9) div 5) + 32 ) temps_F`

`TRANSFORM` : the name of the higher-order function
`temps` : the name of our input array
`t` : 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 the function
`((t * 9) div 5) + 32` : This is the function. For each value in the input array, the value is multipled by 9 and then divided by 5. Then, we add 32. This is the formula for converting from Celcius to Fahrenheit. Recall that TRANSFORM takes an array, an iterator, and an anonymous function as input. In the code below, temps is the column

```
SELECT
  temps temps_C,
  TRANSFORM (temps, t -> ((t * 9) div 5) + 32 ) temps_F
FROM DeviceData;
```

	temps_C	temps_F	
1	▸ [16, 13, 19, 11, 9, 23, 18, 13, 18, 17, 12, 12]	▸ [60, 55, 66, 51, 48, 73, 64, 55, 64, 62, 53, 53]	
2	▸ [26, 17, 19, 13, 9, 12, 10, 12, 1, 13, 16,	▸ [78, 62, 66, 55, 48, 53, 50, 53, 33, 55,	

	12]	60, 53]	
3	▸ [11, 13, 19, 8, 14, 16, 13, 14, 14, 9, 7, 12]	▸ [51, 55, 66, 46, 57, 60, 55, 57, 57, 48, 44, 53]	
4	▸ [20, 18, 20, 18, 11, 14, 17, 24, 17, 15, 19, 22]	▸ [68, 64, 68, 64, 51, 57, 62, 75, 62, 59, 66, 71]	

Truncated results, showing first 1000 rows.

Reduce

`REDUCE` is more advanced than `TRANSFORM`; it takes two lambda functions. You can use it to reduce the elements of an array to a single value by merging the elements into a buffer, and applying a finishing function on the final buffer.

We will use the reduce function to find an average value, by day, for our CO_2 readings. Take a closer look at the individual pieces of the `REDUCE` function by reviewing the list below.

```
REDUCE(co2_level, 0, (c, acc) -> c + acc, acc ->(acc div size(co2_level)))
```

`co2_level` is the input array
`0` is the starting point for the buffer. Remember, we have to hold a temporary buffer value each time a new value is added to from the array; we start at zero in this case to get an accurate sum of the values in the list.
`(c, acc)` is the list of arguments we'll use for this function. It may be helpful to think of `acc` as the buffer value and `c` as the value that gets added to the buffer.
`c + acc` is the buffer function. As the function iterates over the list, it holds the total (`acc`) and adds the next value in the list (`c`)

```
CREATE OR REPLACE TEMPORARY VIEW Co2LevelsTemporary
AS
  SELECT
    dc_id,
    device_type,
    co2Level,
    REDUCE(co2Level, 0, (c, acc) -> c + acc, acc ->(acc div size(co2Level))) as
averageCo2Level

  FROM DeviceData
  SORT BY averageCo2Level DESC;

SELECT * FROM Co2LevelsTemporary
```

	dc_id	device_type	co2Level	averageCo2Level
1	dc-103	sensor-istick	▸ [1819, 1705, 1658, 1753, 1616, 1871]	1737
2	dc-103	sensor-ipad	▸ [1617, 1607, 1835, 1783, 1726, 1568]	1689
3	dc-103	sensor-inest	▸ [1595, 1684, 1682, 1631, 1688, 1754]	1672
4	dc-103	sensor-inest	▸ [1633, 1651, 1753, 1913, 1526, 1552]	1671

Truncated results, showing first 1000 rows.

Other higher-order functions

There are many built-in functions designed to work with array type data and well as other higher-order functions to explore. You can import this notebook (https://docs.databricks.com/_static/notebooks/apache-spark-2.4-functions.html?_ga=2.12496948.1216795462.1586360468-278368669.1586265166) for a list of examples.

Pivot tables: Example 1

Pivot tables are supported in Spark SQL. A pivot table allows you to transform rows into columns and group by any data field. Let's take a closer look at our query.

`SELECT * FROM ()`: The `SELECT` statement inside the parentheses in the input for this table. Note that it takes two columns from the view `Co2LevelsTemporary`
`PIVOT`: The first argument in the clause is an aggregate function and the column to be aggregated. Then, we specify the pivot column in the `FOR` subclause. The `IN` operator contains the pivot column values.

```
SELECT * FROM (
  SELECT device_type, averageCo2Level
  FROM Co2LevelsTemporary
)
PIVOT (
  ROUND(AVG(averageCo2Level), 2) avg_co2
  FOR device_type IN ('sensor-ipad', 'sensor-inest',
    'sensor-istick', 'sensor-igauge')
  );
```

	sensor-ipad	sensor-inest	sensor-istick	sensor-igauge
1	1245.98	1250.41	1244.86	1247.56

Showing all 1 rows.

Pivot Tables: Example 2

In this example, we again pull data from our larger table `DeviceData`. Within the subquery, we create the `month` column and use the `REDUCE` function to create the `averageCo2Level` column.

In the pivot, we take the average of of the `averageCo2Level` values grouped by month. Notice that we rename the month columns from their number to the english abbreviations.

Learn more about pivot tables in this blog post (https://databricks.com/blog/2018/11/01/sql-pivot-converting-rows-to-columns.html).

```
SELECT
  *
FROM
  (
    SELECT
      month(date) month,
      REDUCE(co2Level, 0, (c, acc) -> c + acc, acc ->(acc div size(co2Level)))
averageCo2Level
    FROM
      DeviceData
  )
```

	month	averageCo2Level
1	7	1237
2	7	1099
3	7	1233
4	7	1043
5	7	1363
6	7	1356
7	7	1091

Truncated results, showing first 1000 rows.

```
SELECT
  *
FROM
  (
    SELECT
      month(date) month,
      REDUCE(co2Level, 0, (c, acc) -> c + acc, acc ->(acc div size(co2Level)))
averageCo2Level
    FROM
      DeviceData
  ) PIVOT (
    avg(averageCo2Level) avg FOR month IN (7 JUL, 8 AUG, 9 SEPT, 10 OCT, 11
NOV)
  )
```

	JUL	AUG	SEPT	OCT
1	1242.8850806451612	1250.8649193548388	1245.1229166666667	1249.29838709677

Showing all 1 rows.

```
SELECT
  COALESCE(dc_id, "All data centers") AS dc_id,
  COALESCE(device_type, "All devices") AS device_type,
  ROUND(AVG(averageCo2Level))  AS avgCo2Level
FROM Co2LevelsTemporary
GROUP BY ROLLUP (dc_id, device_type)
ORDER BY dc_id, device_type;
```

	dc_id	device_type	avgCo2Level
1	All data centers	All devices	1247
2	dc-101	All devices	1197
3	dc-101	sensor-igauge	1202
4	dc-101	sensor-inest	1197
5	dc-101	sensor-ipad	1194
6	dc-101	sensor-istick	1196
7	dc-102	All devices	1296

Showing all 21 rows.

Cube

`CUBE` is also an operator used with the `GROUP BY` clause. Similar to `ROLLUP`, you can use `CUBE` to generate summary values for sub-elements grouped by column value. `CUBE` is different than `ROLLUP` in that it will also generate subtotals for all combinations of grouping columns specified in the `GROUP BY` clause.

Notice that the output for the example below shows some of additional values generated in this query. Data from `"All data centers"` has been aggregated across device types for all centers.

```
SELECT
  COALESCE(dc_id, "All data centers") AS dc_id,
  COALESCE(device_type, "All devices") AS device_type,
  ROUND(AVG(averageCo2Level))  AS avgCo2Level
FROM Co2LevelsTemporary
GROUP BY CUBE (dc_id, device_type)
ORDER BY dc_id, device_type;
```

	dc_id	device_type	avgCo2Level
1	All data centers	All devices	1247
2	All data centers	sensor-igauge	1248
3	All data centers	sensor-inest	1250
4	All data centers	sensor-ipad	1246
5	All data centers	sensor-istick	1245
6	dc-101	All devices	1197
7	dc-101	sensor-igauge	1202

Showing all 25 rows.

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