databricks5.1 Managing Nested Data

Managing Nested Data with Spark SQL

In this notebook, you'll be digging into mock data from a group of data centers. A **data center** is a dedicated space where computing and networking equipment is set up to collect, store, process, and distribute data. The continuous operation of centers like this can be crucial to maintaining continuity in business, so environmental conditions must be closely monitored.

This example uses mock data from 4 different data centers, each with four different kinds of sensors that periodically collect temperature and CO_2 level readings. Temperature and CO_2 levels are stored as arrays where temperature is collected 12 times per day and CO_2 level is collected 6 times per day.

Run the following queries to learn about how to work with and manage nested data in Spark SQL.
In this notebook, you will:

- Work with hierarchical data

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
```

Create table

The Databricks File System (DBFS) (https://docs.databricks.com/data/databricks-file-system.html) is a distributed file system mounted into a Databricks workspace and available on Databricks clusters. In practice, this will allow you to access data that has been mounted to your workspace and interact with that storage using directories and file names instead of storage urls. In this lesson, we'll use data from datasets in object storage that has been mounted to the DBFS. We will create a table and explore some of the optional arguments available to us.

The cell below begins with a `DROP TABLE IF EXISTS`; command. This means that if a table by the given name exists, it will be dropped. If it does not exist, this command does nothing. This will keep our notebook **idempotent**, meaning it could be run more than once without throwing errors or introducing extra files.

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

OK

View metadata and "Detailed Table Information"

In a previous lesson, we used the `DESCRIBE` command to view metadata. Run the command below to see the output when we attach the optional keyword `EXTENDED`.

You can find the same information about the schema at the top. Notice that one of our columns contains a `MapType` column, and, within that, a `StructType` field. When working with structured data, like parquet files, and semi-structured data, like JSON files, you will frequently encounter complex data types, like `MapType`, `StructType`, and `ArrayType`.

In this example, the `MapType` column holds a JSON object that has a `string` as its **key** and a `struct` field as the **value**. As you work through this notebook, we will unnest and explore that data. Learn more about the data types you will be working with in Spark SQL in the associated docs (https://spark.apache.org/docs/latest/sql-ref-datatypes.html).

Detailed Table Information contains information about the table's database name, original source file type and location, and more.

```
DESCRIBE EXTENDED DCDataRaw;
```

	col_name	data_type
1	dc_id	string
2	date	string
3	source	map<string,struct<description:string,ip:string,id:int,temps:array<
4		
5	# Detailed Table Information	
6	Database	default
7	Table	dcdataraw

Showing all 17 rows.

View a sample

It may you help understand the data if we view a few rows. Instead of simply returning the top rows, we can get a random sampling of rows using the function `RAND()` to return random rows and the `LIMIT` keyword to set the number of rows we want to see.

```
SELECT * FROM DCDataRaw
ORDER BY RAND()
LIMIT 3;
```

	dc_id	date	source
1	dc-103	2019-11-29	▸ {"sensor-igauge": {"description": "Sensor attached to the con 21, 13, 17, 23, 11, 19, 7, 11, 16, 19], "co2_level": [1705, 1630, 1 ipad attached to carbon cylinders", "ip": "20.164.47.183", "id": 2 "co2_level": [1288, 1259, 1207, 1314, 1327, 1466]}, "sensor-ine

			"170.215.68.213", "id": 30, "temps": [16, 15, 10, 12, 12, 19, 10, 1523]}, "sensor-istick": {"description": "Sensor embedded in exl "temps": [13, 24, 19, 15, 10, 16, 14, 26, 15, 23, 12, 24], "co2_le
2	dc-102	2019-10-16	▸ {"sensor-igauge": {"description": "Sensor attached to the con 19, 25, 20, 16, 21, 13, 18, 24, 20], "co2_level": [1468, 1355, 15 ipad attached to carbon cylinders", "ip": "87.223.119.226", "id": "co2_level": [1351, 1537, 1221, 1529, 1439, 1307]}, "sensor-ine "286.280.238.44", "id": 21, "temps": [25, 19, 20, 23, 23, 23, 21, 1513]}, "sensor-istick": {"description": "Sensor embedded in exl "temps": [22, 16, 5, 16, 11, 17, 17, 12, 15, 13, 17, 17], "co2_lev
3	dc-101	2019-08-05	▸ {"sensor-igauge": {"description": "Sensor attached to the con 15, 15, 13, 9, 13, 24, 14, 21, 18], "co2_level": [1202, 1224, 112 attached to carbon cylinders", "ip": "193.189.270.101", "id": 17, "co2_level": [1002, 1085, 1217, 1079, 1085, 1154]}, "sensor-ine "189.88.273.219", "id": 21, "temps": [17, 18, 19, 24, 18, 15, 19, 1176]}, "sensor-istick": {"description": "Sensor embedded in exl [28, 21, 16, 24, 26, 26, 21, 20, 28, 25, 19, 21], "co2_level": [113

Showing all 3 rows.

Explode a nested object

We can observe from the output that the `source` column contains a nested object with named `key-value` pairs. We'll use `EXPLODE` to get a closer look at the data in that column.

`EXPLODE` is used with arrays and elements of a map expression. When used with an array, it splits the elements into multiple rows. Used with a map, as in this example, it splits the elements of a map into multiple rows and columns and uses the default names, `key` and `value`, to name the new columns. This data structure is mapped such that each `key`, the name of a certain device, holds an object, `value`, containing information about that device.

```
SELECT EXPLODE (source)
FROM DCDataRaw;
```

	key	value
1	sensor-igauge	▸ {"description": "Sensor attached to the container ceilings", "ip": "43.48.56.53 18, 17, 12, 12], "co2_level": [1196, 1360, 1125, 1206, 1342, 1198]}
2	sensor-ipad	▸ {"description": "Sensor ipad attached to carbon cylinders", "ip": "171.59.65.2 12, 1, 13, 16, 12], "co2_level": [1002, 988, 1137, 1171, 1094, 1206]}
3	sensor-inest	▸ {"description": "Sensor attached to the factory ceilings", "ip": "189.125.254.1

4	sensor-istick	▸ {"description": "Sensor embedded in exhaust pipes in the ceilings", "ip": "27 14, 17, 24, 17, 15, 19, 22], "co2_level": [1098, 1113, 1161, 981, 967, 939]}
5	sensor-igauge	▸ {"description": "Sensor attached to the container ceilings", "ip": "286.139.18 -2, 3], "co2_level": [1494, 1385, 1378, 1335, 1480, 1107]}
6	sensor-ipad	▸ {"description": "Sensor ipad attached to carbon cylinders", "ip": "55.80.146.1 10, 12, 14, 12], "co2_level": [1312, 1386, 1375, 1359, 1396, 1308]}
7	sensor-inest	▸ {"description": "Sensor attached to the factory ceilings", "ip": "127.239.249.2

Truncated results, showing first 1000 rows.

```
SELECT
EXPLODE (value)
FROM
(SELECT EXPLODE (source)
FROM DCDataRaw);
```

```
Error in SQL statement: AnalysisException: cannot resolve 'explode(__auto_ge
nerated_subquery_name.`value`)' due to data type mismatch: input to function
explode should be array or map type, not struct<description:string,ip:strin
g,id:int,temps:array<int>,co2_level:array<int>>; line 2 pos 0;
'Project [unresolvedalias(explode(value#1853), None)]
+- SubqueryAlias __auto_generated_subquery_name
   +- Project [key#1852, value#1853]
      +- Generate explode(source#1827), false, [key#1852, value#1853]
         +- SubqueryAlias spark_catalog.default.dcdataraw
            +- Relation[dc_id#1825,date#1826,source#1827] parquet
```

Common Table Expressions

Common Table Expressions (CTE) are supported in Spark SQL. A CTE provides a temporary result set which you can then use in a `SELECT` statement. These are different from temporary views in that they cannot be used beyond the scope of a single query. In this case, we will use the CTE to get a closer look at the nested data without writing a new table or view. CTEs use the `WITH` clause to start defining the expression.

Notice that after we explode the source column, we can access individual properties in the `value` field by using dot notation with the property name.

```
WITH ExplodeSource  -- specify the name of the result set we will query
AS
(                   -- wrap a SELECT statement in parentheses
  SELECT            -- this is the temporary result set you will query
    dc_id,
    to_date(date) AS date,
    EXPLODE (source)
  FROM
    DCDataRaw
)
SELECT              -- write a select statment to query the result set
  key,
  dc_id,
  date,
  value.description,
  value.ip,
  value.temps,
  value.co2_level
FROM                -- this query is coming from the CTE we named
  ExplodeSource;
```

	key	dc_id	date	description
1	sensor-igauge	dc-101	2019-07-01	Sensor attached to the container ceilings
2	sensor-ipad	dc-101	2019-07-01	Sensor ipad attached to carbon cylinders
3	sensor-inest	dc-101	2019-07-01	Sensor attached to the factory ceilings
4	sensor-istick	dc-101	2019-07-01	Sensor embedded in exhaust pipes in the c

Truncated results, showing first 1000 rows.

Create Table as Select (CTAS)

CTEs like those in the cell above are temporary and cannot be queried again. In the next cell, we demonstrate how you create a table using the common table expression syntax.

In Spark SQL, you can populate a new table with input data from a `SELECT` statement. The following is an example where we create a new table, `DeviceData`, using the CTE syntax we used in the previous cell. In this example, we rename the `key` column to `device_type`.

```
DROP TABLE IF EXISTS DeviceData;
CREATE TABLE DeviceData
USING parquet
WITH ExplodeSource                          -- The start of the CTE from the last
cell
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

FROM ExplodeSource;
```

OK

Run a `SELECT` all to view the new table.

```
SELECT * FROM DeviceData
```

	dc_id	device_type	date	description
1	dc-101	sensor-igauge	2019-07-01	Sensor attached to the container ceilings
2	dc-101	sensor-ipad	2019-07-01	Sensor ipad attached to carbon cylinders

3	dc-101	sensor-inest	2019-07-01	Sensor attached to the factory ceilings
4	dc-101	sensor-istick	2019-07-01	Sensor embedded in exhaust pipes in the c

Truncated results, showing first 1000 rows.

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