

# Case Study Introduction and Goals

---

In the lab, we introduce Spark and Spark SQL as a solution that allows us to use common SQL syntax when working with structured or semi-structured data. In this lab, you will use Spark SQL on Databricks to practice common design patterns for efficient table creation and use built-in functions that can help you explore, manipulate, and aggregate nested data.

By the end of this lab, you will be able to:

- Create tables using different data sources and other optional parameters.
- Use common design patterns to create and modify tables and views.
- Use built-in functions and features of Spark SQL to manage and manipulate nested objects.
- Use pivot, roll-up, cube, and window functions to aggregate data.
- Create a dashboard with data visualization.

# Case Study Setup

---

## Investigate Data Center Data



*Photo of a data center*

In this lesson, we'll use mock data center data for the practical exercises. 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 CO2 level readings. Temperature and CO2 levels are stored as nested arrays where temperature is collected 12 times per day and CO2 level is collected 6 times per day.

For simplicity, we are providing you with a data file that you can upload to your workspace. In practice, your company workspace may be connected to an existing object storage system, and you will be able to pull current data into tables for analysis.

In this lab, you will be presented with a series of Spark SQL commands that you can use in a Databricks notebook. The steps outlined in this lesson will help you set up your notebook so that you can experiment with these commands.

## Step 1: Download sample data

Download this [file](#) and upload to the databricks. Once you read the file, you should see structure of the data in the file as below.

```
dc_id: string
date: string
▼ source: map
  key: string
  ▼ value: struct
    description: string
    ip: string
    id: integer
    ▼ temps: array
      element: integer
    ▼ co2_level: array
      element: integer
```



## Step 2: Lab setup



We'll organize the work for this lab in a single Databricks notebook. To get started, login to your workspace, start your cluster, and create a new notebook. If necessary, use the images below to guide you through the process.

### Login to Databricks



## Sign In to Databricks

 student@databricks.com 

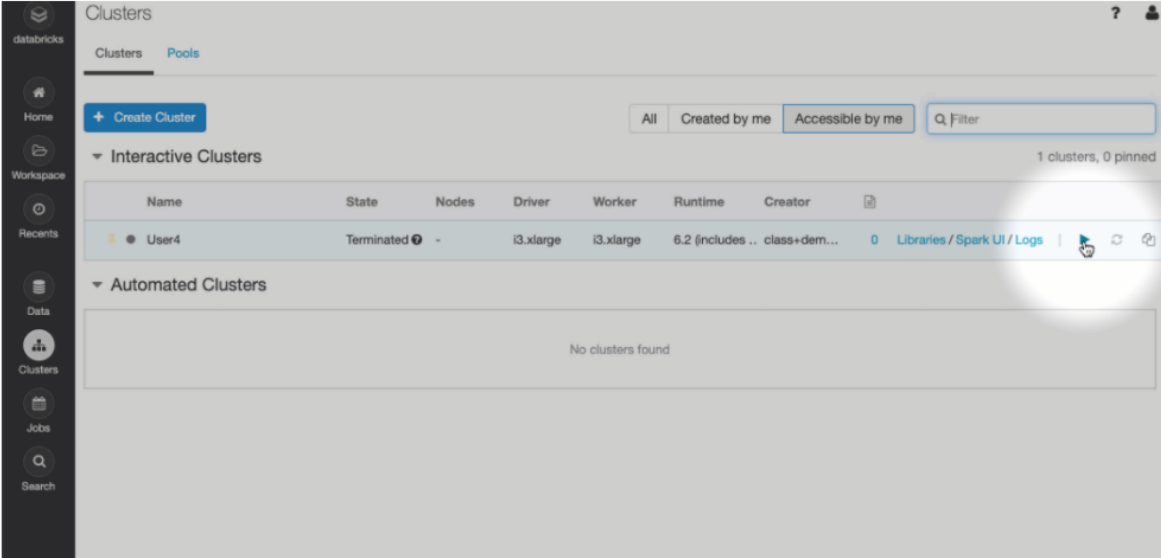
 ..... 

[Forgot Password?](#)

Sign In

[Privacy Policy](#) | [Terms of Use](#)

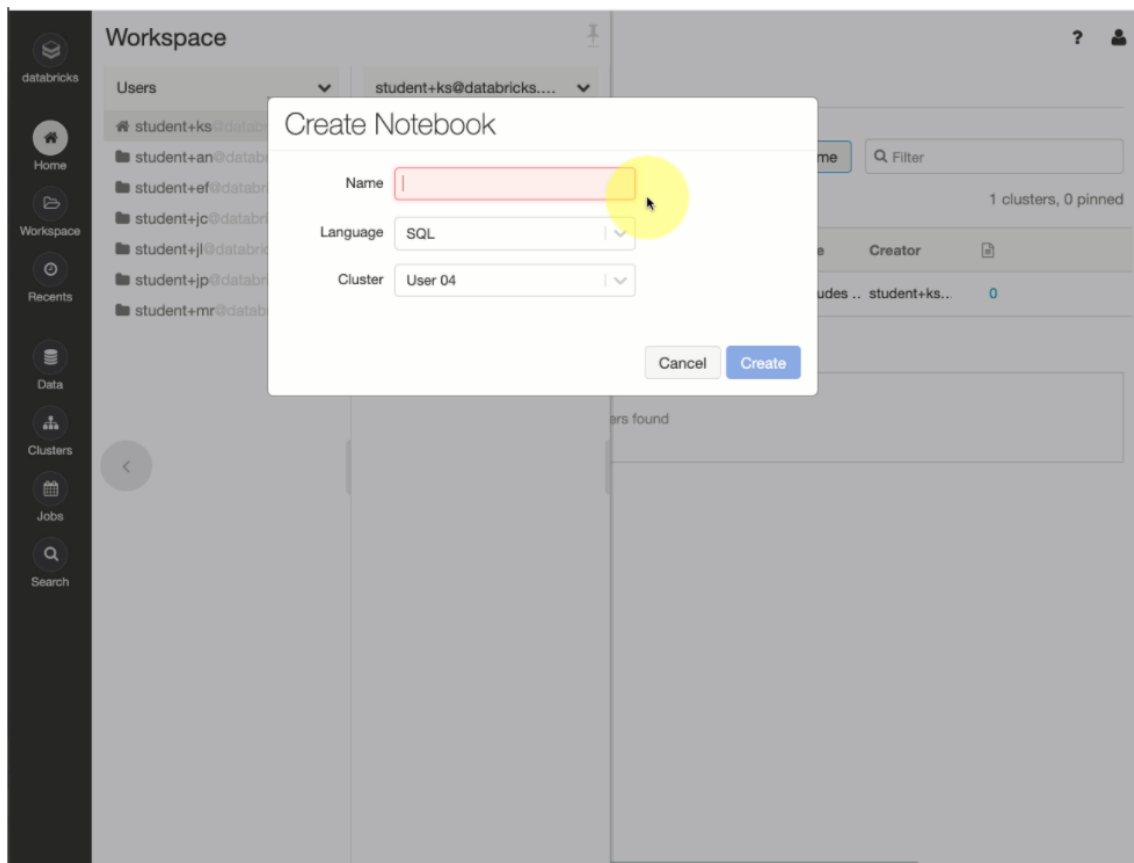
## Start your Cluster



The screenshot shows the Databricks Clusters management interface. On the left is a sidebar with navigation icons for Home, Workspace, Recents, Data, Clusters, Jobs, and Search. The main header includes the 'Clusters' tab and a 'Pools' tab. Below the header, there's a '+ Create Cluster' button and filter tabs for 'All', 'Created by me', and 'Accessible by me'. A search bar labeled 'Filter' is also present. The main content area is divided into two sections: 'Interactive Clusters' and 'Automated Clusters'. The 'Interactive Clusters' section shows a table with one cluster, 'User4', which is in a 'Terminated' state. The table columns are Name, State, Nodes, Driver, Worker, Runtime, and Creator. The 'Automated Clusters' section below it shows 'No clusters found'.

Name	State	Nodes	Driver	Worker	Runtime	Creator
User4	Terminated	-	i3.xlarge	i3.xlarge	6.2 (includes .. class+dem...	0 Libraries / Spark UI / Logs

## Create a SQL Language Notebook



### Step 3: Create and specify the database

A Databricks database is a collection of tables that can be available globally or locally. A global table is accessible across all clusters and registered to a Hive [metastore](#). A local table is not accessible from other clusters and not registered in the Hive metastore. You may recall working with temporary views in Fundamentals of SQL on Databricks. A temporary view is an example of a local table.

In this step, we will name the database where we will store the data that we upload and the tables that we will create later on in the lesson. Naming the database helps keep our shared workspace neat and allows you and your colleagues to work on your own tables. Because multiple people within your workspace may be following this same lesson, it's important to keep your tables separate.

Copy and paste the following code into the first cell in your notebook. Replace with your name. Notice in the example image, we used user1 as the name.

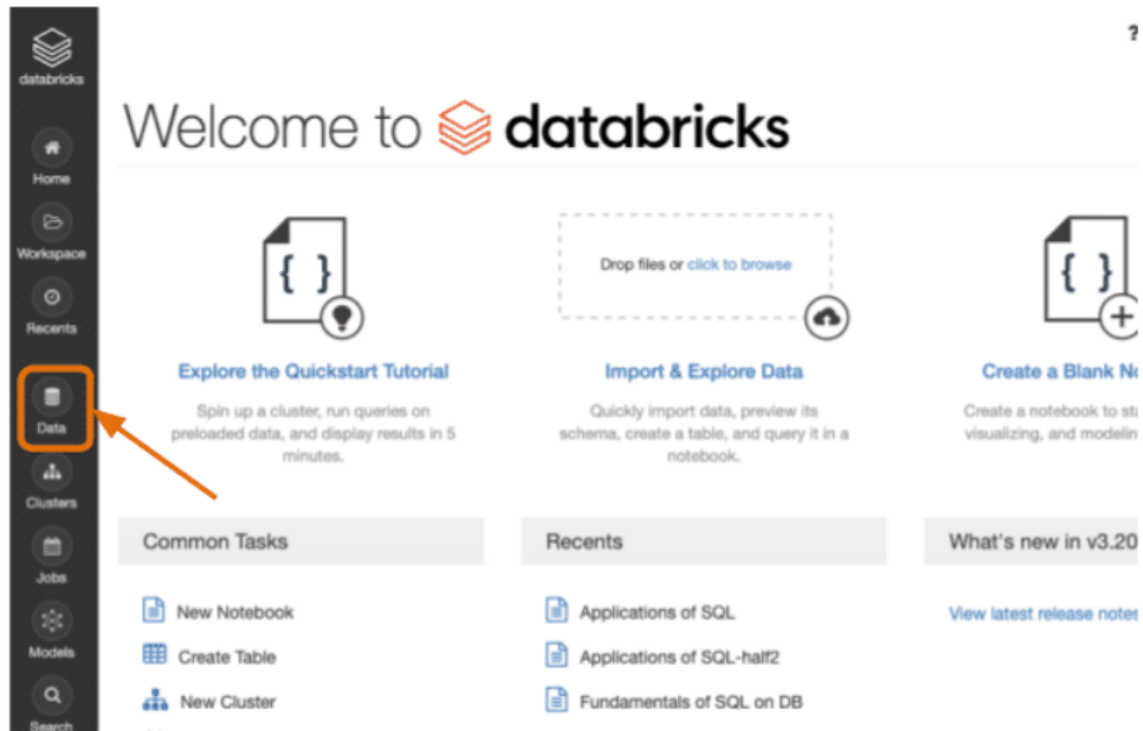
```
1 CREATE DATABASE IF NOT EXISTS dbacademy_user1;
2 USE dbacademy_user1;
```

OK

```
CREATE DATABASE IF NOT EXISTS dbacademy_<YOURNAMEHERE>;
USE dbacademy_<YOURNAMEHERE>;
```

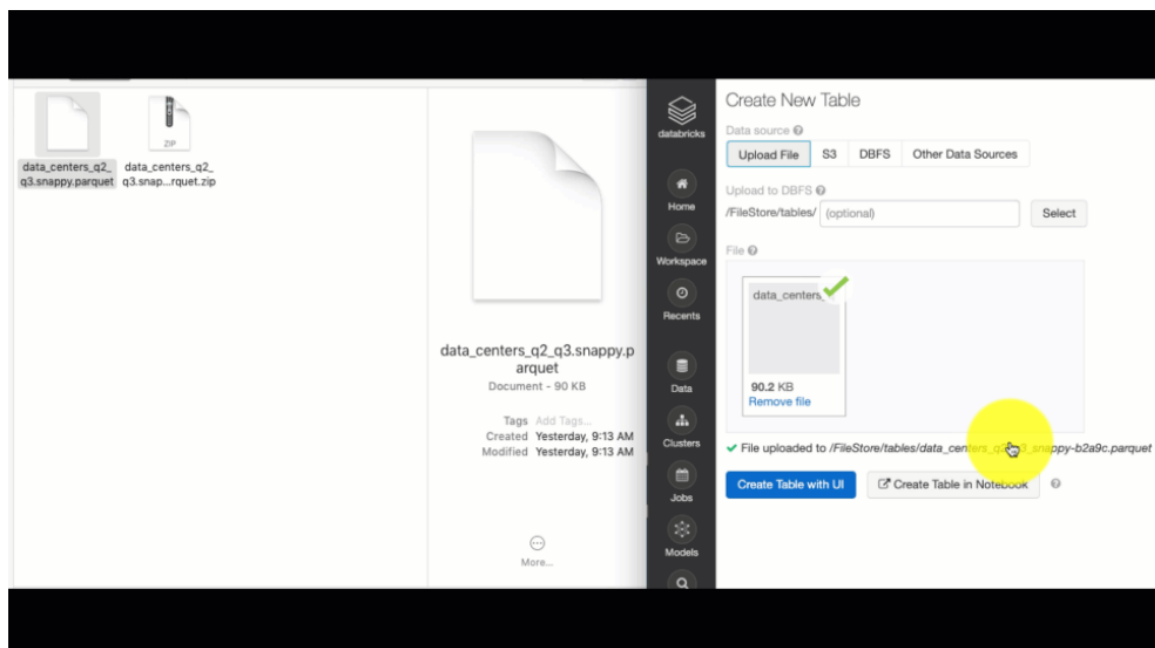
## Step 4: Data

Now we are ready to upload data to your Databricks database. Click on the Data tab to get started. Then, click on Add Data.



### 4.1 Add Data

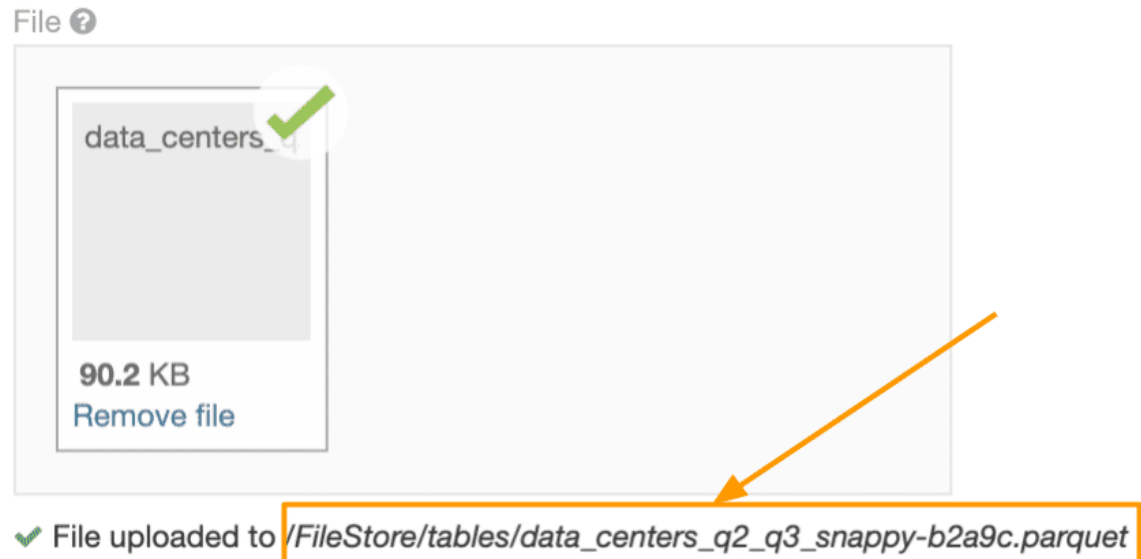
Drag your unzipped parquet file into the file dialog.



## 4.2 Copy the link

You can use this link to programmatically create your table. Copy this link and for use in the next lesson.

**HINT:** Paste your link into a note-keeping app or empty document so that you can refer to it in the next lesson.



# Prepare Data

## Step 1: Programmatically create a new table

We can use the link generated during the manual data upload process to programmatically create a table. You'll use the path you copied in the the previous step for this one.

Copy and paste the code below into your notebook. Substitute your generated link as the *PATH*.

Notice that this cell starts with a *DROP TABLE* command. This will keep our notebook idempotent, meaning it could be run more that once without throwing errors or introducing extra files and/or records.

```
DROP TABLE IF EXISTS dc_data_raw;  
CREATE TABLE dc_data_raw  
USING parquet  
OPTIONS (  
  PATH "PLACE/YOUR/FILE/PATH/HERE"  
);
```

**USING parquet:** Parquet is a common format for many big data applications because it can be read quickly and easily split into partitions. Recall that Spark's speed and power relies on its ability to parallelize large workloads by concurrently working on small chunks of data. Learn more about how Parquet compares to CSV and JSON file format [here](#)

```
DROP TABLE IF EXISTS dc_data_raw;  
CREATE TABLE dc_data_raw  
USING parquet  
OPTIONS (  
  PATH "PLACE/YOUR/FILE/PATH/HERE"  
);
```

OK

## Step 2: View Table

Now let's take a look at some sample data from our table.

```
SELECT * FROM dc_data_raw LIMIT 1;
```

```
SELECT * FROM dc_data_raw LIMIT 1;
```

dc_id	date	source
dc-101	2019-04-01	<div>» {"sensor-igauge":{"description":"Sensor attached to the container ceilings","ip":"34.232.238.223","id":17,"temps": [16,31,22,23,25,24,20,18,20,23,16,18],"co2_level":[1041,1317,1122,864,1005,1048]},"sensor-ipad":{"description":"Sensor ipad attached to carbon cylinders","ip":"107.208.111.63","id":22,"temps": [7,10,15,12,10,13,9,11,15,13,10,14],"co2_level": [1000,1009,999,1249,905,1088]},"sensor-inest":{"description":"Sensor attached to the factory ceilings","ip":"231.24.15.160","id":30,"temps": [26,20,21,21,15,11,12,22,17,22,16,19],"co2_level": [1000,1180,1187,1136,1175,1059]},"sensor-istick":{"description":"Sensor embedded in exhaust pipes in the ceilings","ip":"272.112.280.252","id":33,"temps": [7,8,17,14,14,2,11,19,11,6,14,16],"co2_level": [1232,1246,1237,1142,1249,1218]}}</div>





# Examine and Explode Nested Objects

## Step 1: Describe Detail

We can examine our table using the *DESCRIBE* command. In this exercise, we use the [optional parameter](#) *DETAIL* to get a detailed look at how and where our table is stored. Click on the information buttons below to explore each field.

```
DESCRIBE DETAIL dc_data_raw;
```

**DESCRIBE** DETAIL dc\_data\_raw;

format	id	name	description	location	createdAt	lastModified	partiti
parquet	null	dbacademy_user1.dc_data_raw		dbfs:/FileStore/tables /data_centers_q2_q3_snappy- b2a9c.parquet	2020-06-05T12:58:32.000+0000	null	[]



## Step 2: Explode

When we examine the table, we can see that the source column contains nested values. Now, we'll use the *EXPLODE* command to get a look at how that column is structured.

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

**SELECT** EXPLODE (source)  
**FROM** dc\_data\_raw;

key	value
sensor-igauge	» {"description":"Sensor attached to the container ceilings","ip":"34.232.238.223","id":17,"temps": [16,31,22,23,25,24,20,18,20,23,16,18],"co2_level":[1041,1317,1122,864,1005,1048]}
sensor-ipad	» {"description":"Sensor ipad attached to carbon cylinders","ip":"107.208.111.63","id":22,"temps": [7,10,15,12,10,13,9,11,15,13,10,14],"co2_level": [1000,1009,999,1249,905,1088]}
sensor-ineet	» {"description":"Sensor attached to the factory ceilings","ip":"231.24.15.160","id":30,"temps": [26,20,21,21,15,11,12,22,17,22,16,19],"co2_level": [1000,1180,1187,1136,1175,1059]}
sensor-istick	» {"description":"Sensor embedded in exhaust pipes in the ceilings","ip":"272.112.280.252","id":33,"temps": [7,8,17,14,14,2,11,19,11,6,14,16],"co2_level": [1232,1246,1237,1142,1249,1218]}
sensor-igauge	» {"description":"Sensor attached to the container ceilings","ip":"47.132.238.88","id":17,"temps": [12,16,9,10,14,13,15,13,8,18,16,9],"co2_level": [1306,1290,1253,1343,1304,1327]}

Showing the first 1000 rows.



# Manage Tables and Views

---

Now that we have a sense of how our data is organized, we can creating different tables and views that will help us get a sense of what the data is telling us about the environmental conditions at these data centers.

## Step 1: Common Table Expressions (CTE)

Common Table Expressions are supported in Spark SQL. A CTE provides a temporary result set that you can use in a *SELECT* statement. These are different from a temporary view in that they cannot be used beyond the scope of a single query.

We will use the CTE to get a closer look at the nested data without writing a new table or view. Remember that some of our fields hold array data about temperature and CO2 data in our data centers. Flattening this table will allow us to work directly with those values.

```
WITH explode_source
AS
(
  SELECT
    dc_id,
    to_date(date) AS date,
    EXplode (source)
  FROM dc_data_raw
)
SELECT key,
       dc_id,
       date,
       value.description,
       value.ip,
       value.temps,
       value.co2_level
FROM explode_source;
```

```

WITH explode_source
AS
(
  SELECT
    dc_id,
    to_date(date) AS date,
    EXplode (source)
  FROM dc_data_raw
)
SELECT key,
dc_id,
date,
value.description,
value.ip,
value.temps,
value.co2_level
FROM explode_source;

```

key	dc_id	date	description	ip	temps	co2_level
sensor-igauge	dc-101	2019-04-01	Sensor attached to the container ceilings	34.232.238.223	» [16,31,22,23,25,24,20,18,20,23,16,18]	» [1041,1317,1122,864,1005,1048]
sensor-ipad	dc-101	2019-04-01	Sensor ipad attached to carbon cylinders	107.208.111.63	» [7,10,15,12,10,13,9,11,15,13,10,14]	» [1000,1009,999,1249,905,1088]
sensor-inest	dc-101	2019-04-01	Sensor attached to the factory ceilings	231.24.15.160	» [26,20,21,21,15,11,12,22,17,22,16,19]	» [1000,1180,1187,1136,1175,1059]
sensor-istick	dc-101	2019-04-01	Sensor embedded in exhaust pipes in the ceilings	272.112.280.252	» [7,8,17,14,14,2,11,19,11,6,14,16]	» [1232,1246,1237,1142,1249,1218]
sensor-	dc-101	2019-04-02	Sensor attached to the	47.132.238.88	» [12,16,9,10,14,13,15,13,8,18,16,9]	» [1306,1290,1253,1343,1304,1327]

Showing the first 1000 rows.



## Step 2: Create Table As Select (CTAS)

Use the *CREATE TABLE AS SELECT* pattern to create a new table based on the output of a *SELECT* statement. Notice that this example uses our common table expression within the *CREATE TABLE* command to *EXPLODE* the nested data, and reorganize it into columns.

This will create a permanent table in our database. Be sure to click on the info buttons to understand the optional parameters included here.

```

DROP TABLE IF EXISTS device_data;

CREATE TABLE device_data
USING delta
PARTITIONED BY (device_type)
WITH explode_source
AS
(
  SELECT
    dc_id,
    to_date(date) AS date,
    EXplode (source)
  FROM dc_data_raw
)
SELECT
  dc_id,
  key `device_type`,
  date,

```

```

    value.description,
    value.ip,
    value.temps,
    value.co2_level
FROM explode_source;

SELECT * FROM device_data

```

```
DROP TABLE IF EXISTS device_data;
```

```

CREATE TABLE device_data
USING delta
PARTITIONED BY (device_type)
WITH explode_source
AS

```

```

(
  SELECT
    dc_id,
    to_date(date) AS date,
    EXPLODE (source)
  FROM dc_data_raw
)

```

```

SELECT
  dc_id,
  key `device_type`,
  date,
  value.description,
  value.ip,
  value.temps,
  value.co2_level

```

```
FROM explode_source;
```

```
SELECT * FROM device_data
```

dc_id	device_type	date	description	ip	temps	co2_level
dc-101	sensor-igauge	2019-04-01	Sensor attached to the container ceilings	34.232.238.223	» [16,31,22,23,25,24,20,18,20,23,16,18]	» [1041,1317,1122,864,1005,1048]
dc-101	sensor-igauge	2019-04-02	Sensor attached to the container ceilings	47.132.238.88	» [12,16,9,10,14,13,15,13,8,18,16,9]	» [1306,1290,1253,1343,1304,1327]
dc-101	sensor-igauge	2019-04-03	Sensor attached to the container ceilings	221.34.28.81	» [11,10,12,14,7,11,15,10,16,10,8,18]	» [1088,1207,1119,1067,1141,1113]
dc-101	sensor-igauge	2019-04-04	Sensor attached to the container ceilings	60.33.298.275	» [13,15,12,15,13,12,11,20,14,13,9,13]	» [1228,1238,1234,1196,1326,1296]
dc-101	sensor-igauge	2019-04-05	Sensor attached to the container ceilings	280.81.156.34	» [19,16,15,17,20,21,15,11,18,15,16,19]	» [1246,1663,1493,1336,1420,1437]

Showing the first 1000 rows.



### Step 3: Describe Extended

Tables store two important pieces of information: the data within the table and metadata, which is information about the data in the table.

We can use the *DESCRIBE EXTENDED* to display detailed information about the table including the database, table type, and location of the directory where the data is stored.

```
DESCRIBE EXTENDED device_data
```

DESCRIBE EXTENDED device\_data

col_name	data_type	comment
dc_id	string	
device_type	string	
date	date	
description	string	
ip	string	
temps	array<int>	
co2_level	array<int>	



## Step 4: Cache Table

The table *device\_data* contains all the information we'll need for the remainder of the lesson. One useful way to optimize your queries is by caching tables that you will reuse over and over. Caching places the contents of a table into the memory (or local disk) of your cluster. This can make subsequent reads faster. Keep in mind that caching may be an expensive operation as it needs to first cache all of the data before it can operate on it. Caching is useful if you plan to query the same data repeatedly.

```
CACHE TABLE device_data
```

CACHE TABLE device\_data

OK

# Working with Array Data

Now that you've flattened your table, you can easily access the array values that are stored in the temperature and CO2 level columns. That brings us to our next problem: complex data types (like arrays) can be very difficult to work with in structured tables. Most structured data includes one simple value per row in each column of a table. Most of our built-in functions (like ROUND, AVG, MAX) are built to work with data stored in that manner.

Typically, there are two ways people usually approach this kind of data:

1. Explode the nested structure, apply a transformation, and then collect the values to recreate the original structure
2. Build a user-defined function

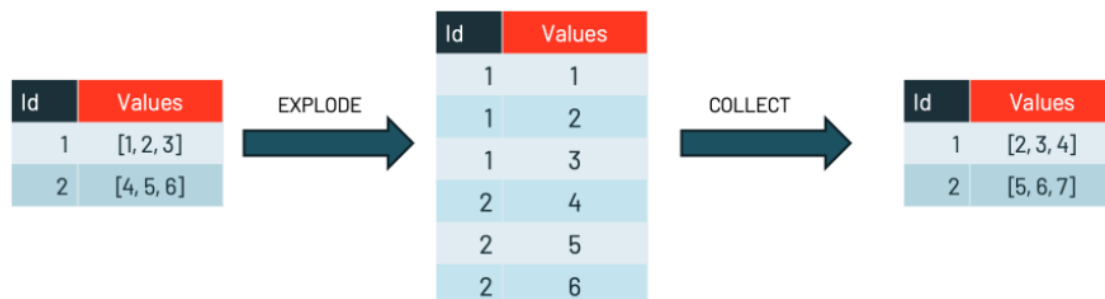
First, let's take a look at how these strategies can work.

## 1) Explode and Collect

One common method for working with array data is to explode your table, transform each value, and then collect all values back into the original structure. This can be highly inefficient and error prone, in addition to being computationally expensive.

### Explode and Collect

- TASK: Add 1 to each element in an array



## 2) Build a user-defined function

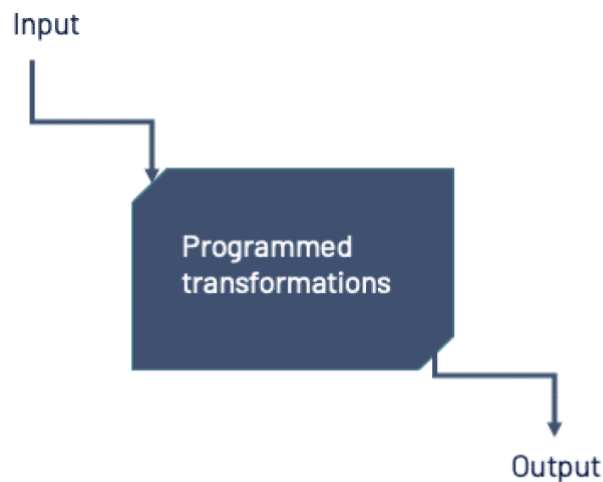
User-defined functions (UDFs) can be written in Python or Scala, and applied to tables in SQL.

Pros	Cons
Not as error prone as unpacking and repacking complex data	Need to be written in Scala or Python
Faster than the previously described method	Not optimized / slower than built-in functions

Spark SQL offers a more robust solution for dealing with this kind of data. Built-in functions designed to work with complex data, like arrays, make it easy to manipulate the data directly and have been optimized to work in parallel. Read on to better understand how they work.

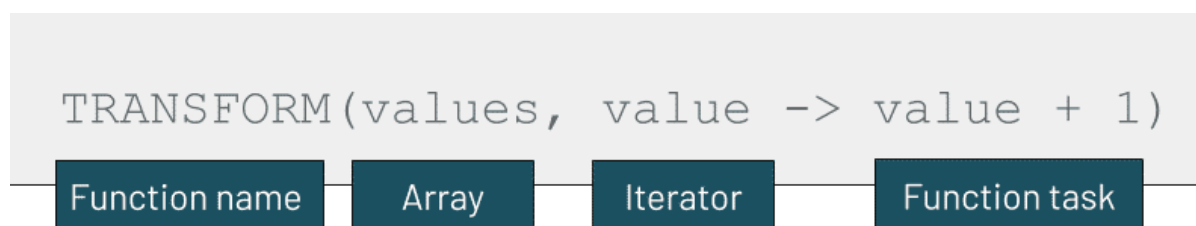
### What are functions?

Functions are reusable pieces of code that perform an action. Standard ANSI SQL includes many built-in functions like AVG(), COUNT(), MIN(), and MAX(), for example. Each of these built-in functions does a job - it performs a series of steps and returns a value. They are designed to work efficiently with a high volume of records, but they generally require that each column contains a single value.



### Higher-order functions

Higher order functions are a simple extension to SQL that allows you to write a function to apply to each element of an array. This kind of function is called an anonymous or lambda function. Let's see how this works by going over a basic transformation. If we wanted to add 1 to each value in a column of arrays, we could use a function like the one below. Click on the info buttons to learn more about each part of this command.



**Function Name:** TRANSFORM is the name of this higher-order function. It takes an array, and iterator variable, and an anonymous function as input. This function will transform each element of the array according to the task assigned in the parentheses.

**Array:** In your function, this will be the name of the column that you want to transform. For this higher-order function, this column must have array-type values.

Iterator: `value` is the iterator variable we're using in this function. It can have any name. We use it to traverse the array. The iterator will temporarily hold the value of each element. Notice that it appears in the function body to define how we want each element to be transformed.

Function Task (Function Body): This is where we define the transformation we want for each element in the array. This function will add 1 to each value in the array.



# Applied Higher Order Functions

In this section, you'll use higher order functions to manipulate arrays of integer data. You should use the same notebook that you began earlier in this lesson.

## Step 1: Transform

For our data, the temperature is given in degrees Celsius. We can use the *TRANSFORM* function to convert each element of the array to Fahrenheit. To convert from Celsius to Fahrenheit, you multiply the temperature in Celsius by 9, divide by 5, and then add 32.

In this section, you'll use higher order functions to manipulate arrays of integer data. You should use the same notebook that you began earlier in this lesson.

## Step 1: Transform

For our data, the temperature is given in degrees Celsius. We can use the *TRANSFORM* function to convert each element of the array to Fahrenheit. To convert from Celsius to Fahrenheit, you multiply the temperature in Celsius by 9, divide by 5, and then add 32.

```
SELECT
  dc_id,
  device_type,
  temps,
  TRANSFORM (temps, t -> ((t * 9) div 5) + 32 ) AS `temps_F`
FROM device_data;
```

```
SELECT
  dc_id,
  device_type,
  temps,
  TRANSFORM (temps, t -> ((t * 9) div 5) + 32 ) AS `temps_F`
FROM device_data;
```

dc_id	device_type	temps	temps_F
dc-101	sensor-igauge	» [19,4,9,12,11,7,5,9,5,6,12,11]	» [66,39,48,53,51,44,41,48,41,42,53,51]
dc-101	sensor-ipad	» [8,11,12,10,9,12,21,10,8,17,19,18]	» [46,51,53,50,48,53,69,50,46,62,66,64]
dc-101	sensor-inest	» [18,22,21,14,17,20,18,19,23,24,18,21]	» [64,71,69,57,62,68,64,66,73,75,64,69]
dc-101	sensor-istick	» [9,9,23,13,14,13,18,19,18,12,19,15]	» [48,48,73,55,57,55,64,66,64,53,66,59]
dc-101	sensor-igauge	» [26,20,24,23,19,25,22,20,25,23,17,21]	» [78,68,75,73,66,77,71,68,77,73,62,69]
dc-101	sensor-ipad	» [11,9,9,13,11,19,5,12,11,6,16,9]	» [51,48,48,55,51,66,41,53,51,42,60,48]
dc-101	sensor-inest	» [13,10,14,18,10,16,12,11,12,15,24,18]	» [55,50,57,64,50,60,53,51,53,59,75,64]
dc-101	sensor-istick	» [26,17,13,14,13,19,20,16,11,12,15,16]	» [78,62,55,57,55,66,68,60,51,53,59,60]

Showing the first 1000 rows.



**Select :** We are selecting columns `dc_id` , `device_type` , `temps` (original) and `temps` (transformed) to show up in our table

**Transform:** Notice that `temps` is the input column that contains an array. The argument we pass into the function is `t` . This stands for a single temperature reading. The function applies the necessary arithmetic operations to convert from Celsius to Fahrenheit.

**Result:** We have a new column in our results, `temps_F`, which shows the array of converted temperature readings.

## Step 2: Reduce

This function is more advanced than others; 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 CO2 readings. Take a closer look at the individual pieces of the *REDUCE* function by reviewing the info buttons on the image below.

This data will be useful in subsequent lessons, so we'll create a temporary view in this step so that we can access it later on.

```
CREATE OR REPLACE TEMPORARY VIEW co2_levels_temporary
AS
  SELECT
    dc_id,
    device_type,
    co2_level,
    REDUCE(co2_level, 0, (c, acc) -> c + acc, acc -> (acc div size(co2_level))) as
average_co2_level
  FROM device_data
  SORT BY average_co2_level DESC;

SELECT * FROM co2_levels_temporary
```

```
CREATE OR REPLACE TEMPORARY VIEW co2_levels_temporary
AS
  SELECT
    dc_id,
    device_type,
    co2_level,
    REDUCE(co2_level, 0, (c, acc) -> c + acc, acc -> (acc div size(co2_level))) as average_co2_level
  FROM device_data
  SORT BY average_co2_level DESC;

SELECT * FROM co2_levels_temporary
```

dc_id	device_type	co2_level	average_co2_level
dc-103	sensor-igauge	▶[1873,1787,1822,1630,1764,1880]	1792
dc-103	sensor-inest	▶[1713,1675,1568,1597,1751,1872]	1696
dc-103	sensor-igauge	▶[1723,1575,1551,1677,1701,1670]	1649
dc-103	sensor-igauge	▶[1745,1695,1453,1563,1659,1573]	1614
dc-103	sensor-igauge	▶[1692,1638,1529,1695,1610,1503]	1611
dc-103	sensor-istick	▶[1694,1773,1636,1557,1545,1461]	1611
dc-103	sensor-inest	▶[1384,1565,1742,1533,1701,1743]	1611
dc-103	sensor-igauge	▶[1495,1674,1710,1416,1673,1634]	1600
dc-103	sensor-inest	▶[1434,1550,1640,1670,1440,1570]	1595

Showing the first 1000 rows.



Reduce:

- `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 will 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`)
- `acc div size(co2_level)` is the finishing function. Once we have the sum of all numbers in the array, we divide by the number of elements to find the average

## Other Higher-Order Functions

There are a lot of higher-order functions (and other built-in functions) that you can use to work with array data! Check out [more examples here](#).

# Pivots, Rollups, and Cubes

## Step 1: Use Pivot Tables

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 try viewing our data by device to see which type emits the most CO2 on average.

```
SELECT * FROM (  
  SELECT device_type, average_co2_level  
  FROM co2_levels_temporary  
)  
PIVOT (  
  ROUND(AVG(average_co2_level), 2) AS avg_co2  
  FOR device_type IN ('sensor-ipad', 'sensor-inest',  
    'sensor-istick', 'sensor-igauge')  
);
```

```
SELECT * FROM (  
  SELECT device_type, average_co2_level  
  FROM co2_levels_temporary  
)  
PIVOT (  
  ROUND(AVG(average_co2_level), 2) AS avg_co2  
  FOR device_type IN ('sensor-ipad', 'sensor-inest',  
    'sensor-istick', 'sensor-igauge')  
);
```

sensor-ipad	sensor-inest	sensor-istick	sensor-igauge
1249.18	1240.24	1248.05	1251.05



**SELECT:** The select statement inside the parentheses is the input.

**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.

## Step 2: Rollups

Rollups are operators used with the *GROUP BY* clause. They allow you to summarize data based on the columns passed to the *ROLLUP* operator. In this example, we've calculated the average temperature across all data centers and across all devices as well as the overall average at each data center (*dc\_id*)

```

SELECT
    dc_id,
    device_type,
    ROUND(AVG(average_co2_level)) AS avg_co2_level
FROM co2_levels_temporary
GROUP BY ROLLUP (dc_id, device_type)
ORDER BY dc_id, device_type

```

```

SELECT
    dc_id,
    device_type,
    ROUND(AVG(average_co2_level)) AS avg_co2_level
FROM co2_levels_temporary
GROUP BY ROLLUP (dc_id, device_type)
ORDER BY dc_id, device_type

```

dc_id	device_type	avg_co2_level
null	null	1247
dc-101	null	1198
dc-101	sensor-igauge	1205
dc-101	sensor-inest	1198
dc-101	sensor-ipad	1204
dc-101	sensor-istick	1187
dc-102	null	1292
dc-102	sensor-igauge	1302

The results of this query include average co2 levels, grouped by dc\_id and device\_type. Notice that we introduced null values for the aggregated records.

- In the first row, we see the aggregate average CO2 level across all data centers and devices.
- In subsequent rows, the null value appears in the device\_type column, and shows the aggregate average CO2 level in a single data center, across all devices.

### Step 3: Coalesce

We can use the *COALESCE* function to substitute a new title in for each null value we introduced. When we *GROUP BY dc\_id*, the resulting aggregated value is for "All data centers". The null values in the *device\_type* column is the aggregated value across "All devices". The table shows average CO2 levels across all data centers and all devices.

```

SELECT
    COALESCE(dc_id, "All data centers") AS dc_id,
    COALESCE(device_type, "All devices") AS device_type,
    ROUND(AVG(average_co2_level)) AS avg_co2_level
FROM co2_levels_temporary
GROUP BY ROLLUP (dc_id, device_type)
ORDER BY dc_id, device_type

```

```

SELECT
  COALESCE(dc_id, "All data centers") AS dc_id,
  COALESCE(device_type, "All devices") AS device_type,
  ROUND(AVG(average_co2_level)) AS avg_co2_level
FROM co2_levels_temporary
GROUP BY ROLLUP (dc_id, device_type)
ORDER BY dc_id, device_type

```

dc_id	device_type	avg_co2_level
All data centers	All devices	1247
dc-101	All devices	1198
dc-101	sensor-igauge	1205
dc-101	sensor-inest	1198
dc-101	sensor-ipad	1204
dc-101	sensor-istick	1187
dc-102	All devices	1292
dc-102	sensor-igauge	1302



Notice that the null values in the previous chart have been replaced with more descriptive names.

## Step 4: 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(average_co2_level)) AS avg_co2_level
FROM co2_levels_temporary
GROUP BY CUBE (dc_id, device_type)
ORDER BY dc_id, device_type;

```

```

SELECT
  COALESCE(dc_id, "All data centers") AS dc_id,
  COALESCE(device_type, "All devices") AS device_type,
  ROUND(AVG(average_co2_level)) AS avg_co2_level
FROM co2_levels_temporary
GROUP BY CUBE (dc_id, device_type)
ORDER BY dc_id, device_type;

```

dc_id	device_type	avg_co2_level
All data centers	All devices	1247
All data centers	sensor-igauge	1251
All data centers	sensor-inest	1240
All data centers	sensor-ipad	1249
All data centers	sensor-istick	1248
dc-101	All devices	1198
dc-101	sensor-igauge	1205
dc-101	sensor-inest	1198



# Partitions and Window Functions

## Step 1: New Partitioned Table

For the next set of exercises, we will create and use a table called `avg_temps`. You may recognize this query from a previous exercise. This table includes temperature readings taken over entire days as well as the calculated value `avg_daily_temp_C`.

Notice that this table has been *PARTITIONED BY* the column `device_type`. The result of this kind of partitioning is that the table is stored in separate files. This may speed up subsequent queries that can filter out certain partitions. These are **not** the same partitions we refer to when discussing basic Spark architecture.

```
CREATE TABLE IF NOT EXISTS avg_temps
USING delta
PARTITIONED BY (device_type)
AS
SELECT
  dc_id,
  date,
  temps,
  REDUCE(temps, 0, (t, acc) -> t + acc, acc -> (acc div size(temps))) as
avg_daily_temp_C,
  device_type
FROM device_data;

SELECT * FROM avg_temps;
```

```
1 CREATE TABLE IF NOT EXISTS avg_temps
2 USING delta
3 PARTITIONED BY (device_type)
4 AS
5 SELECT
6   dc_id,
7   date,
8   temps,
9   REDUCE(temps, 0, (t, acc) -> t + acc, acc -> (acc div size(temps))) as avg_daily_temp_C,
10  device_type
11 FROM device_data;
12
13 SELECT * FROM avg_temps;
```

▶ (4) Spark Jobs

dc_id	date	temps	avg_daily_temp_C	device_type
dc-101	2019-04-01	[7,8,17,14,14,2,11,19,11,6,14,16]	11	sensor-istick
dc-101	2019-04-02	[10,7,13,7,18,11,12,9,5,14,19,14]	11	sensor-istick
dc-101	2019-04-03	[17,15,19,21,25,28,16,23,21,17,23,	21	sensor-istick
dc-101	2019-04-04	[9,1,6,9,15,12,11,17,17,9,16,3]	10	sensor-istick
dc-101	2019-04-05	[12,16,16,18,17,9,9,21,13,11,18,1	14	sensor-istick
dc-101	2019-04-06	[17,19,8,15,14,17,11,19,16,12,16,1	14	sensor-istick
dc-101	2019-04-07	[9,13,19,11,13,11,3,13,10,13,15,1	11	sensor-istick

Showing the first 1000 rows.



## Step 2: Check Partitioning

Use the command `SHOW PARTITIONS` to see how your data is partitioned. In this case, we can verify that the data has been partitioned according to device type.

```
SHOW PARTITIONS avg_temps
```

## Step 3: Window Functions

Window functions calculate an aggregate value for every input row of a table based on a group of rows selected by the user, also called **the frame**. To use window functions, we need to mark that a function is used as a window by adding an `OVER` clause after a supported function in SQL. Within the `OVER` clause, you specify which rows are included in the frame associated with this window.

In the example, the function we will use is `AVG`. We define the window associated with this function with `OVER(PARTITION BY ...)`. The results show that the average monthly temperature is calculated for a data center on a given date. The `WHERE` clause at the end of this query is included to show one month of data from a single data center.

```
SELECT
  dc_id,
  date,
  avg_daily_temp_C,
  AVG(avg_daily_temp_C)
  OVER (PARTITION BY month(date), dc_id) AS average_monthly_temp
FROM avg_temps
WHERE month(date) = "8" and dc_id = "dc-102";
```

```
SELECT
  dc_id,
  date,
  avg_daily_temp_C,
  AVG(avg_daily_temp_C)
  OVER (PARTITION BY month(date), dc_id) AS average_monthly_temp
FROM avg_temps
WHERE month(date) = "8" and dc_id = "dc-102";
```

dc_id	date	avg_daily_temp_C	average_monthly_temp
dc-102	2019-08-01	18	15.661290322580646
dc-102	2019-08-02	20	15.661290322580646
dc-102	2019-08-03	9	15.661290322580646
dc-102	2019-08-04	16	15.661290322580646
dc-102	2019-08-05	18	15.661290322580646
dc-102	2019-08-06	9	15.661290322580646
dc-102	2019-08-07	14	15.661290322580646
dc-102	2019-08-08	23	15.661290322580646
dc-102	2019-08-09	24	15.661290322580646





## Step 4: CTE with Window Functions

You can run a window function on a common table expression. This can be useful when we want to avoid writing a new table to disk. In this example, we create the common table expression *diff\_chart* including the window function that finds the average monthly temperature for each data center.

```
WITH diff_chart AS
(
  SELECT
    dc_id,
    date,
    avg_daily_temp_C,
    AVG(avg_daily_temp_C)
      OVER (PARTITION BY month(date), dc_id) AS average_monthly_temp_C
  FROM avg_temps
)
SELECT
  dc_id,
  date,
  avg_daily_temp_C,
  average_monthly_temp_C,
  avg_daily_temp_C - ROUND(average_monthly_temp_C) AS degree_diff
FROM diff_chart
```

```
WITH diff_chart AS
(
  SELECT
    dc_id,
    date,
    avg_daily_temp_C,
    AVG(avg_daily_temp_C)
      OVER (PARTITION BY month(date), dc_id) AS average_monthly_temp_C
  FROM avg_temps
)
SELECT
  dc_id,
  date,
  avg_daily_temp_C,
  average_monthly_temp_C,
  avg_daily_temp_C - ROUND(average_monthly_temp_C) AS degree_diff
FROM diff_chart
```

dc_id	date	avg_daily_temp_C	average_monthly_temp_C	degree_diff
dc-103	2019-08-01	15	14.89516129032258	0
dc-103	2019-08-02	13	14.89516129032258	-2
dc-103	2019-08-03	11	14.89516129032258	-4
dc-103	2019-08-04	16	14.89516129032258	1
dc-103	2019-08-05	18	14.89516129032258	3
dc-103	2019-08-06	22	14.89516129032258	7
dc-103	2019-08-07	21	14.89516129032258	6
dc-103	2019-08-08	14	14.89516129032258	-1
dc-103	2019-08-09	18	14.89516129032258	3

Showing the first 1000 rows.



# Sharing Visualizations

Now that we've prepared the data, let's review the built-in visualization tools and our options for sharing insights.

## Create Visualization

Now let's create a chart to display the results of the last query.

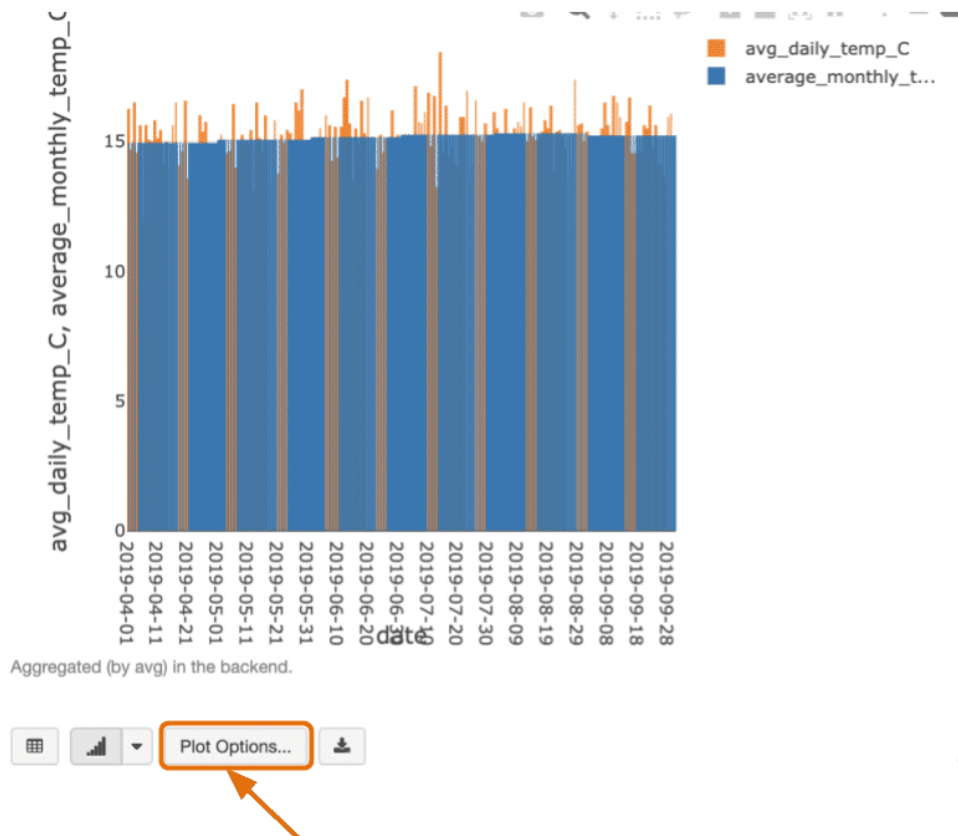
### Step 1: Click the Chart icon

dc_id ▼	date ▼	avg_daily_temp_C ▼	average
dc-104	2019-05-01	15	14.637
dc-104	2019-05-02	11	14.637
dc-104	2019-05-03	22	14.637
dc-104	2019-05-04	21	14.637
dc-104	2019-05-05	15	14.637
dc-104	2019-05-06	16	14.637
dc-104	2019-05-07	21	14.637
dc-104	2019-05-08	13	14.637
dc-104	2019-05-09	6	14.637

Showing the first 1000 rows.



### Step 2: Click Plot Options



### Step 3: Customize plot

Drag fields from the All fields into the

- Drag the **date** field into the Keys dialogue
- Drag **avg\_daily\_temp\_C** and **average\_monthly\_temp\_C** into the Values dialogue.
- Select **AVG** as the Aggregation type.
- Select **Line Chart** from the Display type drop-down menu.
- Click Apply

### Customize Plot

All fields:

- dc\_id
- date
- avg\_daily\_temp...
- average\_month...
- degree\_diff
- <id>

Keys:

date

Series groupings:

Values:

avg\_daily\_temp...

average\_monthl...

Showing sample based on the first 1000 rows. Apply to aggregate over all results.

Y-axis Range

(1000, 8000)

Show Points ☐

Log Scale ☐

Aggregation: 

AVG

Display type: 

Line chart

☐ Global color consistency

Cancel

Apply

## Summary

Your chart shows temperature deviation from the monthly average by day. Now, let's explore some options for sharing visualizations.

## Dashboards

Dashboards allow you to publish graphs and visualizations derived from notebook output and share them in a presentation format with your organization. This part of the lesson will show you how to create and share a dashboard. We'll practice adding one chart to a dashboard as an example. In practice, you'll likely add multiple charts from your notebook to a single dashboard.

## Create a New Dashboard

You can add the contents of any cell to a dashboard. Follow steps illustrated here to get started.

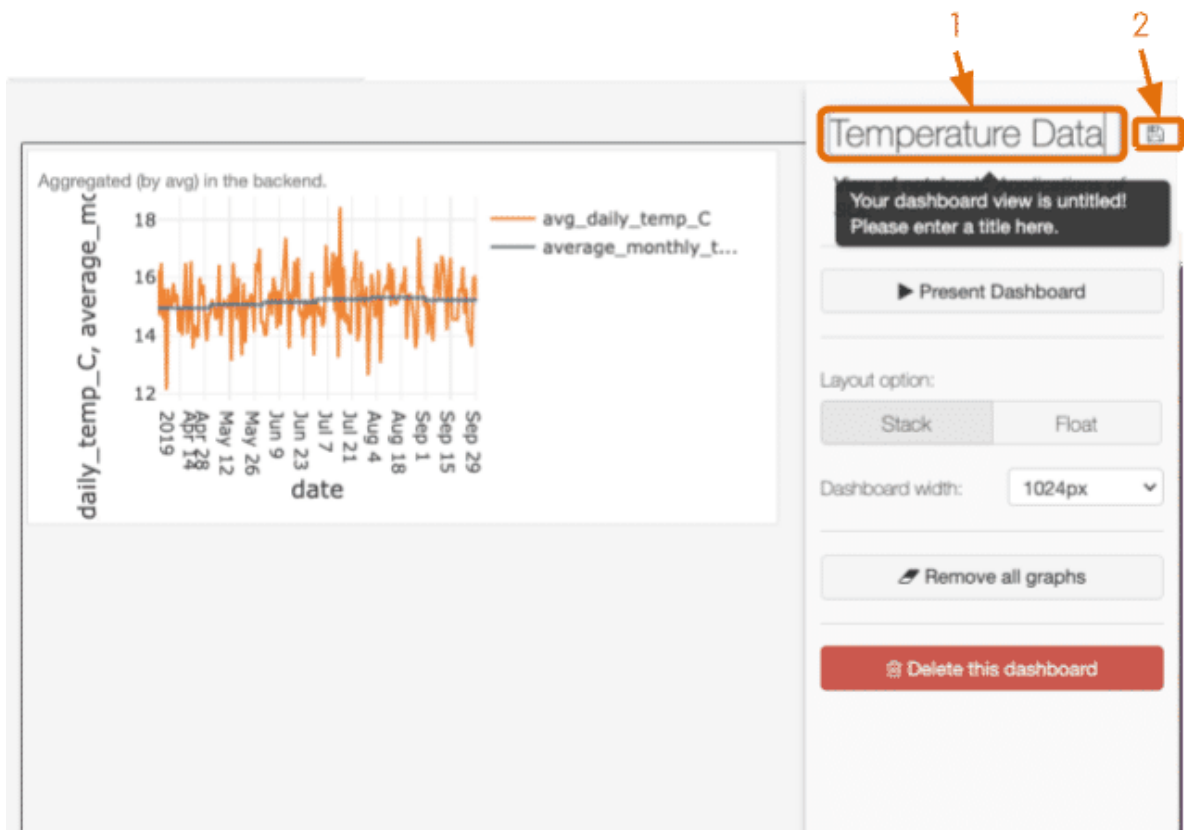
### Step 1: Create New Dashboard

Select the chart icon in the upper right corner of the cell. Then, select **Add to New Dashboard**



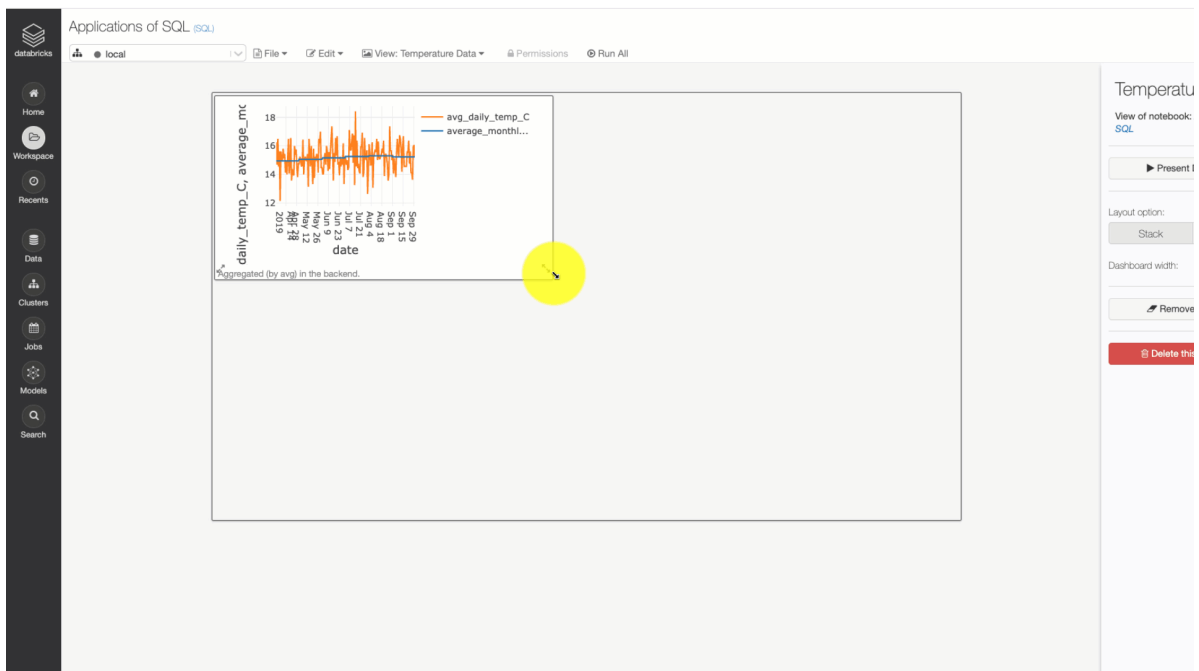
### Step 2: Select a title

Select a descriptive title for your dashboard. Then click the disk icon to save.



### Step 3: Resize your chart

Resize your chart by dragging the right corner.



### Step 4: Present Dashboard

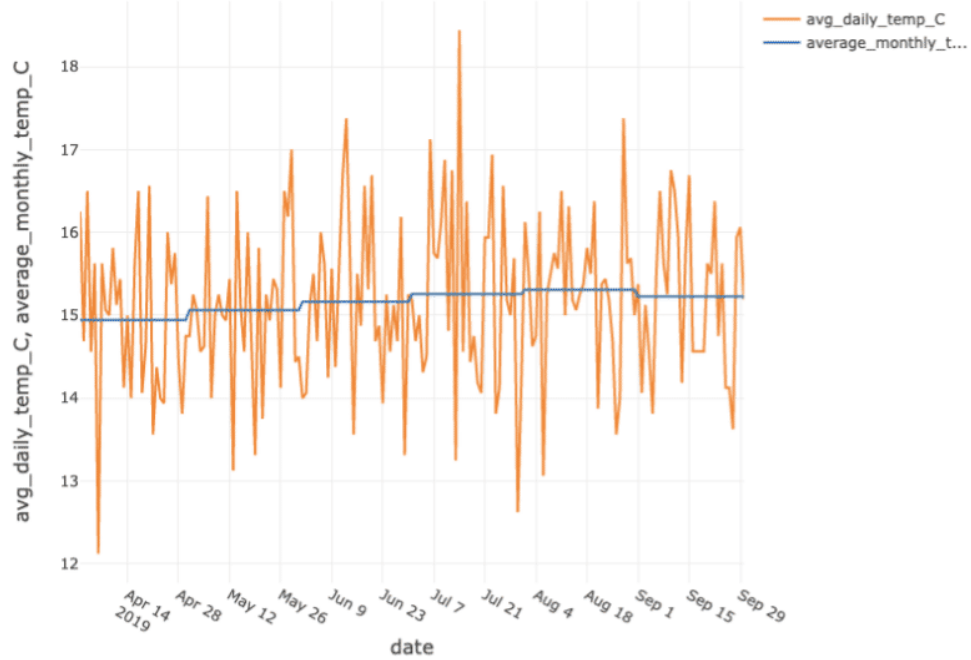
You can share a link to your dashboard with anyone who has access to your notebook. You can use the Update button to pull fresh data from your notebook.

Exit

# Temperature Data

Update

Aggregated (by avg) in the backend.



## Summary

Dashboards can be useful for sharing insights with your team. [Review the documentation](#) to see how you can make dashboards part of your team's workflow.

# Summary and Next Steps

---

Congratulations! You have completed the lab

By now, you should be comfortable:

- Programmatically creating tables using different data sources and other optional parameters
  - Recall that you programmatically created a table using Parquet and later converted that table to Delta
- Using common design patterns for create and modify tables and views
  - Remember working with CTEs to view data without storing a table, and using the CTAS design pattern to create a permanent table from a CTE
- Use higher-order functions to work with complex data types
  - We used *TRANSFORM* and *REDUCE* to transform and summarize array data
- Create a dashboard with data visualization
  - You used built-in visualization tools to view data
  - You created a dashboard that you could share with your organization