

[Before you start](#)

[Provision an Azure Synapse Analytics workspace](#)

[Create delta tables](#)

[Create catalog tables](#)

[Use delta tables for streaming data](#)

[Query a delta table from a serverless SQL pool](#)

[Delete Azure resources](#)

# Use Delta Lake with Spark in Azure Synapse Analytics

Delta Lake is an open source project to build a transactional data storage layer on top of a data lake. Delta Lake adds support for relational semantics for both batch and streaming data operations, and enables the creation of a *Lakehouse* architecture in which Apache Spark can be used to process and query data in tables that are based on underlying files in the data lake.

This lab will take approximately **40** minutes to complete.

## Before you start

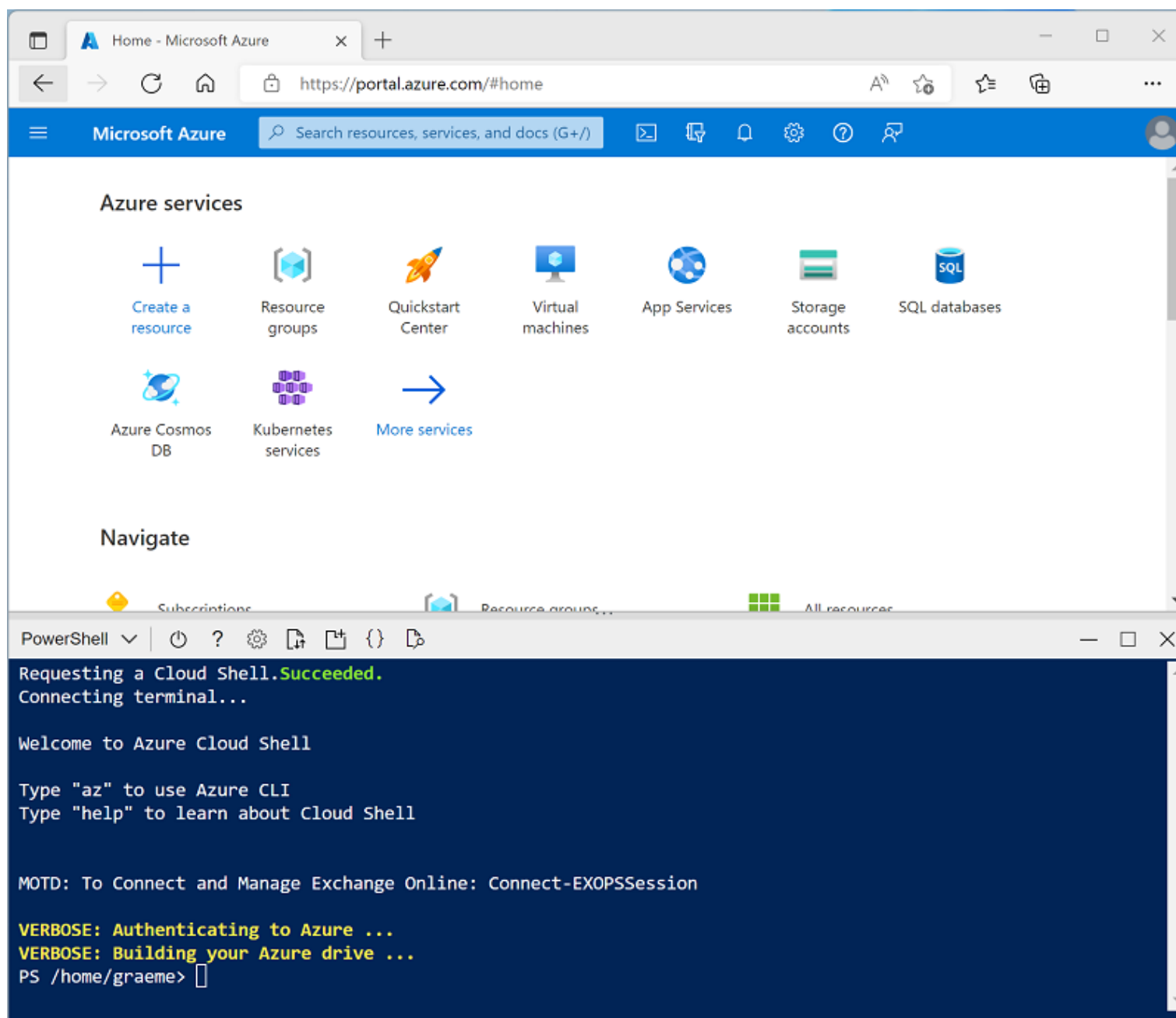
You'll need an [Azure subscription](#) in which you have administrative-level access.

## Provision an Azure Synapse Analytics workspace

You'll need an Azure Synapse Analytics workspace with access to data lake storage and an Apache Spark pool that you can use to query and process files in the data lake.

In this exercise, you'll use a combination of a PowerShell script and an ARM template to provision an Azure Synapse Analytics workspace.

1. Sign into the [Azure portal](#) at `https://portal.azure.com`.
2. Use the `[>]` button to the right of the search bar at the top of the page to create a new Cloud Shell in the Azure portal, selecting a **PowerShell** environment and creating storage if prompted. The cloud shell provides a command line interface in a pane at the bottom of the Azure portal, as shown here:



⚠ **Note:** If you have previously created a cloud shell that uses a *Bash* environment, use the the drop-down menu at the top left of the cloud shell pane to change it to **PowerShell**.

- Note that you can resize the cloud shell by dragging the separator bar at the top of the pane, or by using the —, □, and X icons at the top right of the pane to minimize, maximize, and close the pane. For more information about using the Azure Cloud Shell, see the [Azure Cloud Shell documentation](#).

- In the PowerShell pane, enter the following commands to clone this repo:

Code	Copy
<pre>rm -r dp-000 -f git clone https://github.com/MicrosoftLearning/mslearn-synapse dp-000</pre>	

- After the repo has been cloned, enter the following commands to change to the folder for this lab and run the **setup.ps1** script it contains:

Code	Copy
<pre>cd dp-000/Allfiles/Labs/05 ./setup.ps1</pre>	

- If prompted, choose which subscription you want to use (this will only happen if you have access to multiple Azure subscriptions).

- When prompted, enter a suitable password to be set for your Azure Synapse SQL pool.

⚠ **Note:** Be sure to remember this password!

- Wait for the script to complete - this typically takes around 10 minutes, but in some cases may take longer. While you are waiting, review the [What is Delta Lake](#) article in the Azure Synapse Analytics documentation.

## Create delta tables

The script provisions an Azure Synapse Analytics workspace and an Azure Storage account to host the data lake, then uploads a data file to the data lake.

### Explore the data in the data lake

- After the script has completed, in the Azure portal, go to the **dp000-xxxxxxx** resource group that it created, and select your Synapse workspace.
- In the **Overview** page for your Synapse workspace, in the **Open Synapse Studio** card, select **Open** to open Synapse Studio in a new browser tab; signing in if prompted.
- On the left side of Synapse Studio, use the » icon to expand the menu - this reveals the different pages within Synapse Studio that you'll use to manage resources and perform data analytics tasks.
- On the **Data** page, view the **Linked** tab and verify that your workspace includes a link to your Azure Data Lake Storage Gen2 storage account, which should have a name similar to **synapsexxxxxxx (Primary - datalakexxxxxxx)**.
- Expand your storage account and verify that it contains a file system container named **files**.
- Select the **files** container, and note that it contains a folder named **products**. This folder contains the data you are going to work with in this exercise.
- Open the **products** folder, and observe that it contains a file named **products.csv**.
- Select **products.csv**, and then in the **New notebook** list on the toolbar, select **Load to DataFrame**.
- In the **Notebook 1** pane that opens, in the **Attach to** list, select the **sparkxxxxxxx** Spark pool and ensure that the **Language** is set to **PySpark (Python)**.

10. Review the code in the first (and only) cell in the notebook, which should look like this:

Code

Copy

```
%%pyspark
df =
spark.read.load('abfss://files@datalakexxxxxxx.dfs.core.windows.net/products/products
format='csv'
## If header exists uncomment line below
##, header=True
)
display(df.limit(10))
```

11. Uncomment the `,header=True` line (because the `products.csv` file has the column headers in the first line), so your code looks like this:

Code

Copy

```
%%pyspark
df =
spark.read.load('abfss://files@datalakexxxxxxx.dfs.core.windows.net/products/products
format='csv'
## If header exists uncomment line below
, header=True
)
display(df.limit(10))
```

12. Use the ▶ icon to the left of the code cell to run it, and wait for the results. The first time you run a cell in a notebook, the Spark pool is started - so it may take a minute or so to return any results. Eventually, the results should appear below the cell, and they should be similar to this:

ProductID	ProductName	Category	ListPrice
771	Mountain-100 Silver, 38	Mountain Bikes	3399.9900
772	Mountain-100 Silver, 42	Mountain Bikes	3399.9900
...	...	...	...

Load the file data into a delta table

1. Under the results returned by the first code cell, use the + **Code** button to add a new code cell. Then enter the following code in the new cell and run it:

Code

Copy

```
delta_table_path = "/delta/products-delta"
df.write.format("delta").save(delta_table_path)
```

2. On the **files** tab, use the ↑ icon in the toolbar to return to the root of the **files** container, and note that a new folder named **delta** has been created. Open this folder and the **products-delta** table it contains, where you should see the parquet format file(s) containing the data.

3. Return to the **Notebook 1** tab and add another new code cell. Then, in the new cell, add the following code and run it:

Code

Copy

```

from delta.tables import *
from pyspark.sql.functions import *

# Create a deltaTable object
deltaTable = DeltaTable.forPath(spark, delta_table_path)


# Update the table (reduce price of product 771 by 10%)
deltaTable.update(
    condition = "ProductID == 771",
    set = { "ListPrice": "ListPrice * 0.9" })

# View the updated data as a dataframe
deltaTable.toDF().show(10)

```

The data is loaded into a **DeltaTable** object and updated. You can see the update reflected in the query results.

4. Add another new code cell with the following code and run it:

Code	 Copy
<pre> new_df = spark.read.format("delta").load(delta_table_path) new_df.show(10) </pre>	


The code loads the delta table data into a data frame from its location in the data lake, verifying that the change you made via a **DeltaTable** object has been persisted.

5. Modify the code you just ran as follows, specifying the option to use the *time travel* feature of delta lake to view a previous version of the data.

Code	 Copy
<pre> new_df = spark.read.format("delta").option("versionAsOf", 0).load(delta_table_path) new_df.show(10) </pre>	

When you run the modified code, the results show the original version of the data.

6. Add another new code cell with the following code and run it:

Code	 Copy
<pre> deltaTable.history(10).show(20, False, True) </pre>	

The history of the last 20 changes to the table is shown - there should be two (the original creation, and the update you made.)

## Create catalog tables

So far you've worked with delta tables by loading data from the folder containing the parquet files on which the table is based. You can define *catalog tables* that encapsulate the data and provide a named table entity that you can reference in SQL code. Spark supports two kinds of catalog tables for delta lake:

- *External* tables that are defined by the path to the parquet files containing the table data.
- *Managed* tables, that are defined in the Hive metastore for the Spark pool.

Create an external table

1. In a new code cell, add and run the following code:

Code

Copy

```
spark.sql("CREATE DATABASE AdventureWorks")
spark.sql("CREATE TABLE AdventureWorks.ProductsExternal USING DELTA LOCATION
'{0}'".format(delta_table_path))
spark.sql("DESCRIBE EXTENDED AdventureWorks.ProductsExternal").show(truncate=False)
```

This code creates a new database named **AdventureWorks** and then creates an external table named **ProductsExternal** in that database based on the path to the parquet files you defined previously. It then displays a description of the table's properties. Note that the **Location** property is the path you specified.

2. Add a new code cell, and then enter and run the following code:

Sql

Copy

```
%%sql

USE AdventureWorks;

SELECT * FROM ProductsExternal;
```

The code uses SQL to switch context to the **AdventureWorks** database (which returns no data) and then query the **ProductsExternal** table (which returns a resultset containing the products data in the Delta Lake table).

Create a managed table

1. In a new code cell, add and run the following code:

Code

Copy

```
df.write.format("delta").saveAsTable("AdventureWorks.ProductsManaged")
spark.sql("DESCRIBE EXTENDED AdventureWorks.ProductsManaged").show(truncate=False)
```

This code creates a managed table named **ProductsManaged** based on the DataFrame you originally loaded from the **products.csv** file (before you updated the price of product 771). You do not specify a path for the parquet files used by the table - this is managed for you in the Hive metastore, and shown in the **Location** property in the table description (in the **files/synapse/workspaces/synapsexxxxxx/warehouse** path).

2. Add a new code cell, and then enter and run the following code:

Sql

Copy

```
%%sql

USE AdventureWorks;

SELECT * FROM ProductsManaged;
```

The code uses SQL to query the **ProductsManaged** table.

Compare external and managed tables

1. In a new code cell, add and run the following code:

Sql

Copy

```
%%sql

USE AdventureWorks;

SHOW TABLES;
```

This code lists the tables in the **AdventureWorks** database.

2. Modify the code cell as follows, add run it:

Sql

Copy

```
%%sql

USE AdventureWorks;

DROP TABLE IF EXISTS ProductsExternal;
DROP TABLE IF EXISTS ProductsManaged;
```

This code drops the tables from the metastore.

- 3. Return to the **files** tab and view the **files/delta/products-delta** folder. Note that the data files still exist in this location. Dropping the external table has removed the table from the metastore, but left the data files intact.
- 4. View the **files/synapse/workspaces/synapsexxxxxxx/warehouse** folder, and note that there is no folder for the **ProductsManaged** table data. Dropping a managed table removes the table from the metastore and also deletes the table’s data files.

Create a table using SQL

1. Add a new code cell, and then enter and run the following code:

Sql

Copy

```
%%sql

USE AdventureWorks;

CREATE TABLE Products
USING DELTA
LOCATION '/delta/products-delta';
```

2. Add a new code cell, and then enter and run the following code:

Sql

Copy

```
%%sql

USE AdventureWorks;

SELECT * FROM Products;
```

Observe that the new catalog table was created for the existing Delta Lake table folder, which reflects the changes that were made previously.

Use delta tables for streaming data

Delta lake supports streaming data. Delta tables can be a *sink* or a *source* for data streams created using the Spark Structured Streaming API. In this example, you'll use a delta table as a sink for some streaming data in a simulated internet of things (IoT) scenario.

- 1. Return to the **Notebook 1** tab and add a new code cell. Then, in the new cell, add the following code and run it:

Code

Copy

```
from notebookutils import mssparkutils
from pyspark.sql.types import *
from pyspark.sql.functions import *

# Create a folder
inputPath = '/data/'
mssparkutils.fs.mkdirs(inputPath)

# Create a stream that reads data from the folder, using a JSON schema
jsonSchema = StructType([
    StructField("device", StringType(), False),
    StructField("status", StringType(), False)
])
iotstream = spark.readStream.schema(jsonSchema).option("maxFilesPerTrigger",
1).json(inputPath)

# Write some event data to the folder
device_data = '''{"device":"Dev1","status":"ok"}
{"device":"Dev1","status":"ok"}
{"device":"Dev1","status":"ok"}
{"device":"Dev2","status":"error"}
{"device":"Dev1","status":"ok"}
{"device":"Dev1","status":"error"}
{"device":"Dev2","status":"ok"}
{"device":"Dev2","status":"error"}
{"device":"Dev1","status":"ok"}'''
mssparkutils.fs.put(inputPath + "data.txt", device_data, True)
print("Source stream created...")
```

Ensure the message *Source stream created...* is printed. The code you just ran has created a streaming data source based on a folder to which some data has been saved, representing readings from hypothetical IoT devices.

- 2. In a new code cell, add and run the following code:

Code

Copy

```
# Write the stream to a delta table
delta_stream_table_path = '/delta/iotdevicedata'
checkpointpath = '/delta/checkpoint'
deltastream = iotstream.writeStream.format("delta").option("checkpointLocation",
checkpointpath).start(delta_stream_table_path)
print("Streaming to delta sink...")
```

This code writes the streaming device data in delta format.

- 3. In a new code cell, add and run the following code:

Code


Copy



```
# Read the data in delta format into a dataframe
df = spark.read.format("delta").load(delta_stream_table_path)
display(df)
```


This code reads the streamed data in delta format into a dataframe. Note that the code to load streaming data is no different to that used to load static data from a delta folder.

4. In a new code cell, add and run the following code:

Code	 Copy
<pre># create a catalog table based on the streaming sink spark.sql("CREATE TABLE IotDeviceData USING DELTA LOCATION '{0}'".format(delta_stream_table_path))</pre>	

This code creates a catalog table named **IotDeviceData** (in the **default** database) based on the delta folder. Again, this code is the same as would be used for non-streaming data.

5. In a new code cell, add and run the following code:

Sql	 Copy
<pre>%%sql  SELECT * FROM IotDeviceData;</pre>	


This code queries the **IotDeviceData** table, which contains the device data from the streaming source.

6. In a new code cell, add and run the following code:

Code	 Copy
<pre># Add more data to the source stream more_data = '''{"device":"Dev1","status":"ok"} {"device":"Dev1","status":"ok"} {"device":"Dev1","status":"ok"} {"device":"Dev1","status":"ok"} {"device":"Dev1","status":"error"} {"device":"Dev2","status":"error"} {"device":"Dev1","status":"ok"}'''  mssparkutils.fs.put(inputPath + "more-data.txt", more_data, True)</pre>	

This code writes more hypothetical device data to the streaming source.

7. In a new code cell, add and run the following code:

Sql	 Copy
<pre>%%sql  SELECT * FROM IotDeviceData;</pre>	

This code queries the **IotDeviceData** table again, which should now include the additional data that was added to the streaming source.

8. In a new code cell, add and run the following code:



Code

```
deltastream.stop()
```

Copy

This code stops the stream.

## Query a delta table from a serverless SQL pool

In addition to Spark pools, Azure Synapse Analytics includes a built-in serverless SQL pool. You can use the relational database engine in this pool to query delta tables using SQL.

- 1. In the **files** tab, browse to the **files/delta** folder.
- 2. Select the **products-delta** folder, and on the toolbar, in the **New SQL script** drop-down list, select **Select TOP 100 rows**.
- 3. In the **Select TOP 100 rows** pane, in the **File type** list, select **Delta format** and then select **Apply**.
- 4. Review the SQL code that is generated, which should look like this:

Sql

```
-- This is auto-generated code
SELECT
    TOP 100 *
FROM
    OPENROWSET(
        BULK 'https://datalakexxxxxxx.dfs.core.windows.net/files/delta/products-
delta/',
        FORMAT = 'DELTA'
    ) AS [result]
```

Copy

- 5. Use the **Run** icon to run the script, and review the results. They should look similar to this:

ProductID	ProductName	Category	ListPrice
771	Mountain-100 Silver, 38	Mountain Bikes	3059.991
772	Mountain-100 Silver, 42	Mountain Bikes	3399.9900
...	...	...	...

This demonstrates how you can use a serverless SQL pool to query delta format files that were created using Spark, and use the results for reporting or analysis.

- 6. Replace the query with the following SQL code:

Sql

```
USE AdventureWorks;

SELECT * FROM Products;
```

Copy

- 7. Run the code and observe that you can also use the serverless SQL pool to query Delta Lake data in catalog tables that are defined the Spark metastore.

## Delete Azure resources

If you’ve finished exploring Azure Synapse Analytics, you should delete the resources you’ve created to avoid unnecessary Azure costs.

- 1. Close the Synapse Studio browser tab and return to the Azure portal.

2. On the Azure portal, on the **Home** page, select **Resource groups**.
3. Select the **dp000-xxxxxxx** resource group for your Synapse Analytics workspace (not the managed resource group), and verify that it contains the Synapse workspace, storage account, and Spark pool for your workspace.
4. At the top of the **Overview** page for your resource group, select **Delete resource group**.
5. Enter the **dp000-xxxxxxx** resource group name to confirm you want to delete it, and select **Delete**.

After a few minutes, your Azure Synapse workspace resource group and the managed workspace resource group associated with it will be deleted.