# Perform data engineering with Azure Synapse Apache Spark Pools

Apache Spark is an open source parallel processing framework for large-scale data processing and analytics. Spark has become extremely popular in "big data" processing scenarios, and is available in multiple platform implementations; including Azure HDInsight, Azure Databricks, and Azure Synapse Analytics.

This module explores how you can use Spark in Azure Synapse Analytics to ingest, process, and analyze data from a data lake. While the core techniques and code described in this module are common to all Spark implementations, the integrated tools and ability to work with Spark in the same environment as other Synapse analytical runtimes are specific to Azure Synapse Analytics.

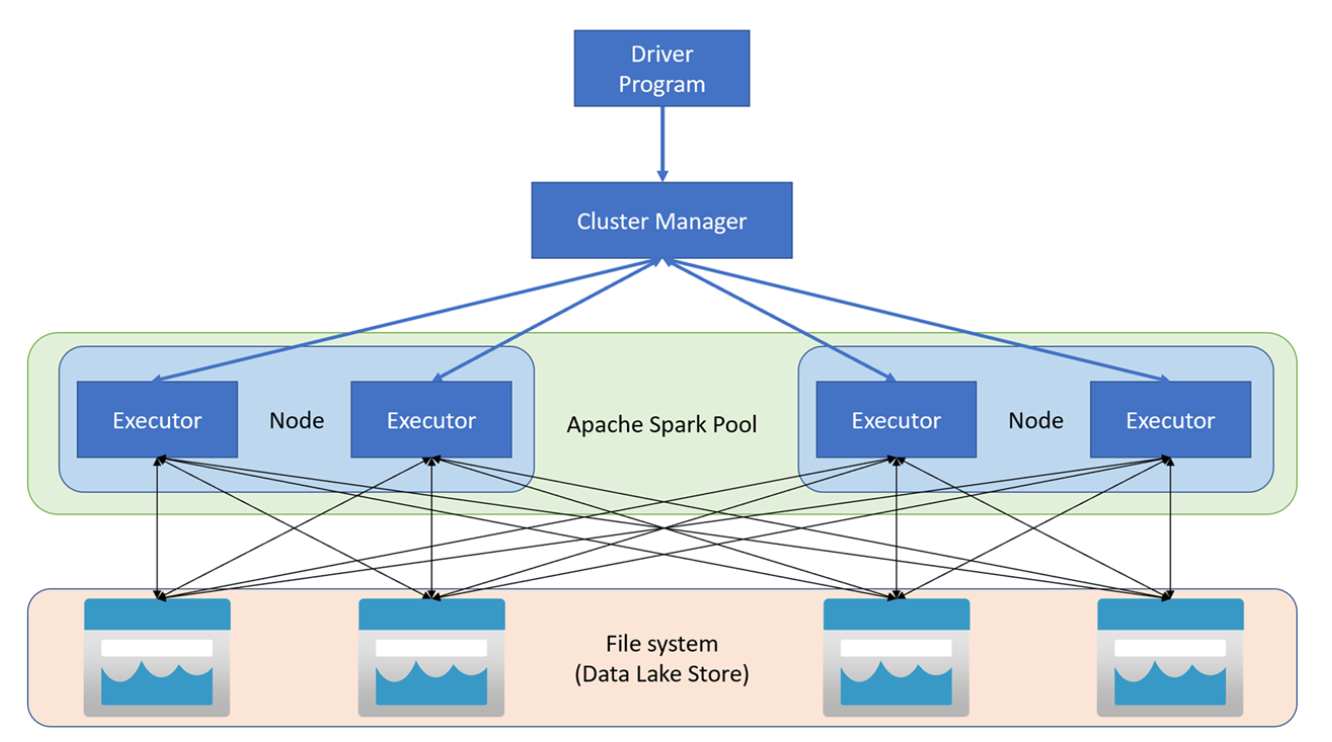
After completing this module, you'll be able to:

* Identify core features and capabilities of Apache Spark.
* Configure a Spark pool in Azure Synapse Analytics.
* Run code to load, analyze, and visualize data in a Spark notebook.

## **How Spark works**

Apache Spark applications run as independent sets of processes on a cluster, coordinated by the SparkContext object in your main program (called the driver program). The SparkContext connects to the cluster manager, which allocates resources across applications using an implementation of Apache Hadoop YARN. Once connected, Spark acquires executors on nodes in the cluster to run your application code.

The SparkContext runs the main function and parallel operations on the cluster nodes, and then collects the results of the operations. The nodes read and write data from and to the file system and cache transformed data in-memory as Resilient Distributed Datasets (RDDs).



The SparkContext is responsible for converting an application to a directed acyclic graph (DAG). The graph consists of individual tasks that get executed within an executor process on the nodes. Each application gets its own executor processes, which stay up for the duration of the whole application and run tasks in multiple threads.

## Spark pools in Azure Synapse Analytics

In Azure Synapse Analytics, a cluster is implemented as a Spark pool, which provides a runtime for Spark operations. You can create one or more Spark pools in an Azure Synapse Analytics workspace [by using the Azure portal](https://learn.microsoft.com/en-us/azure/synapse-analytics/quickstart-create-apache-spark-pool-portal), or [in Azure Synapse Studio](https://learn.microsoft.com/en-us/azure/synapse-analytics/quickstart-create-apache-spark-pool-studio). When defining a Spark pool, you can specify configuration options for the pool, including:

* A name for the spark pool.
* The size of virtual machine (VM) used for the nodes in the pool, including the option to use [hardware accelerated GPU-enabled nodes](https://learn.microsoft.com/en-us/azure/synapse-analytics/quickstart-create-apache-gpu-pool-portal).
* The number of nodes in the pool, and whether the pool size is fixed or individual nodes can be brought online dynamically to auto-scale the cluster; in which case, you can specify the minimum and maximum number of active nodes.
* The version of the Spark Runtime to be used in the pool; which dictates the versions of individual components such as Python, Java, and others that get installed.

**Tip**

* For more information about Spark pool configuration options, see [**Apache Spark pool configurations in Azure Synapse Analytics**](https://learn.microsoft.com/en-us/azure/synapse-analytics/spark/apache-spark-pool-configurations) in the Azure Synapse Analytics documentation.

Spark pools in an Azure Synapse Analytics Workspace are serverless - they start on-demand and stop when idle.

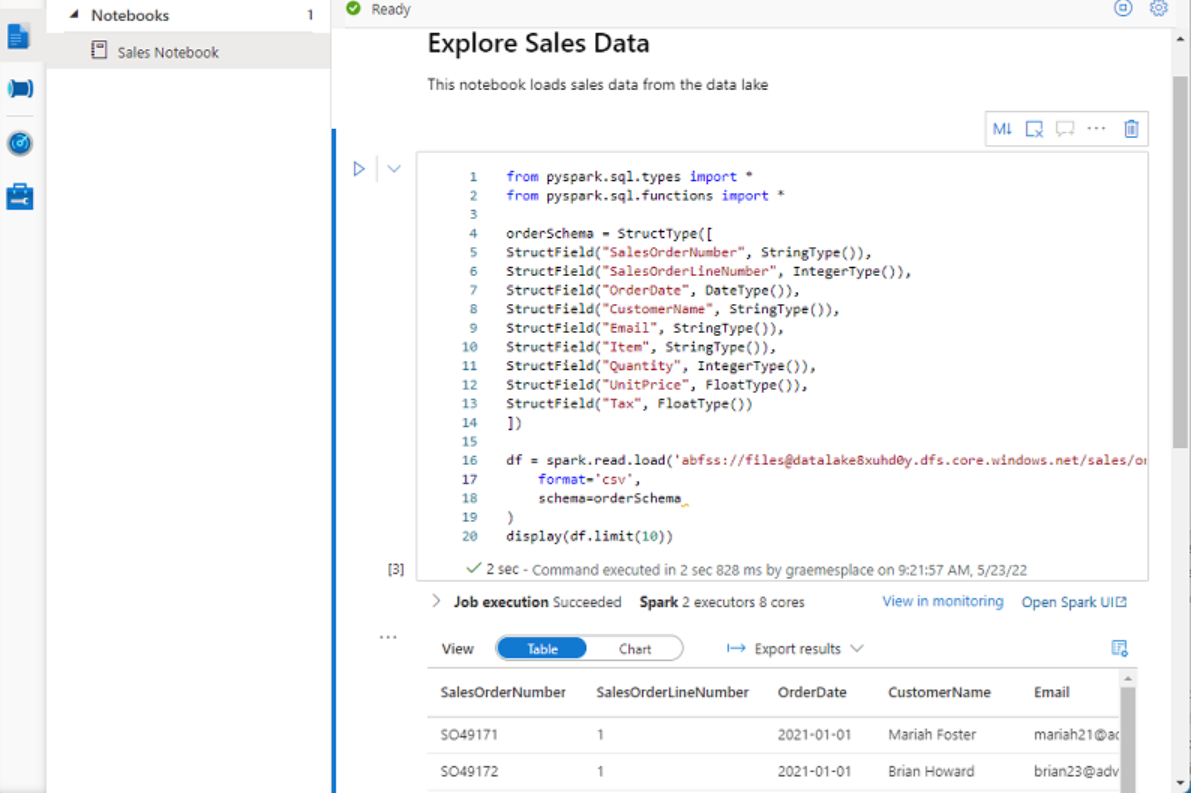
# Use Spark in Azure Synapse Analytics

You can run many different kinds of application on Spark, including code in Python or Scala scripts, Java code compiled as a Java Archive (JAR), and others. Spark is commonly used in two kinds of workload:

* Batch or stream processing jobs to ingest, clean, and transform data - often running as part of an automated pipeline.
* Interactive analytics sessions to explore, analyze, and visualize data.

## **Running Spark code in notebooks**

Azure Synapse Studio includes an integrated notebook interface for working with Spark. Notebooks provide an intuitive way to combine code with Markdown notes, commonly used by data scientists and data analysts. The look and feel of the integrated notebook experience within Azure Synapse Studio is similar to that of Jupyter notebooks - a popular open source notebook platform.



Note

While usually used interactively, notebooks can be included in automated pipelines and run as an unattended script.

Notebooks consist of one or more cells, each containing either code or markdown. Code cells in notebooks have some features that can help you be more productive, including:

Syntax highlighting and error support.

Code auto-completion.

Interactive data visualizations.

The ability to export results.

**Tip**

To learn more about working with notebooks in Azure Synapse Analytics, see the [**Create, develop, and maintain Synapse notebooks in Azure Synapse Analytics**](https://learn.microsoft.com/en-us/azure/synapse-analytics/spark/apache-spark-development-using-notebooks) article in the Azure Synapse Analytics documentation.

## **Accessing data from a Synapse Spark pool**

You can use Spark in Azure Synapse Analytics to work with data from various sources, including:

* A data lake based on the primary storage account for the Azure Synapse Analytics workspace.
* A data lake based on storage defined as a linked service in the workspace.
* A dedicated or serverless SQL pool in the workspace.
* An Azure SQL or SQL Server database (using the Spark connector for SQL Server)
* An Azure Cosmos DB analytical database defined as a linked service and configured using Azure Synapse Link for Cosmos DB.
* An Azure Data Explorer Kusto database defined as a linked service in the workspace.
* An external Hive metastore defined as a linked service in the workspace.

One of the most common uses of Spark is to work with data in a data lake, where you can read and write files in multiple commonly used formats, including delimited text, Parquet, Avro, and others.

One of the benefits of using Spark is that you can write and run code in various programming languages, enabling you to use the programming skills you already have and to use the most appropriate language for a given task. The default language in a new Azure Synapse Analytics Spark notebook is PySpark - a Spark-optimized version of Python, which is commonly used by data scientists and analysts due to its strong support for data manipulation and visualization. Additionally, you can use languages such as Scala (a Java-derived language that can be used interactively) and SQL (a variant of the commonly used SQL language included in the Spark SQL library to work with relational data structures). Software engineers can also create compiled solutions that run on Spark using frameworks such as Java and Microsoft .NET.

**Note**

We won't explore Spark catalog tables in depth in this module, but it's worth taking the time to highlight a few key points:

* You can create an empty table by using the spark.catalog.createTable method. Tables are metadata structures that store their underlying data in the storage location associated with the catalog. Deleting a table also deletes its underlying data.
* You can save a dataframe as a table by using its saveAsTable method.
* You can create an *external* table by using the spark.catalog.createExternalTable method. External tables define metadata in the catalog but get their underlying data from an external storage location; typically a folder in a data lake. Deleting an external table does not delete the underlying data.

### **Using the Spark SQL API to query data**

You can use the Spark SQL API in code written in any language to query data in the catalog. For example, the following PySpark code uses a SQL query to return data from the **products** view as a dataframe.

bikes\_df = spark.sql("SELECT ProductID, ProductName, ListPrice \

FROM products \

WHERE Category IN ('Mountain Bikes', 'Road Bikes')")

display(bikes\_df)

The results from the code example would look similar to the following table:

| **ProductName** | **ListPrice** |
| --- | --- |
| Mountain-100 Silver, 38 | 3399.9900 |
| Road-750 Black, 52 | 539.9900 |

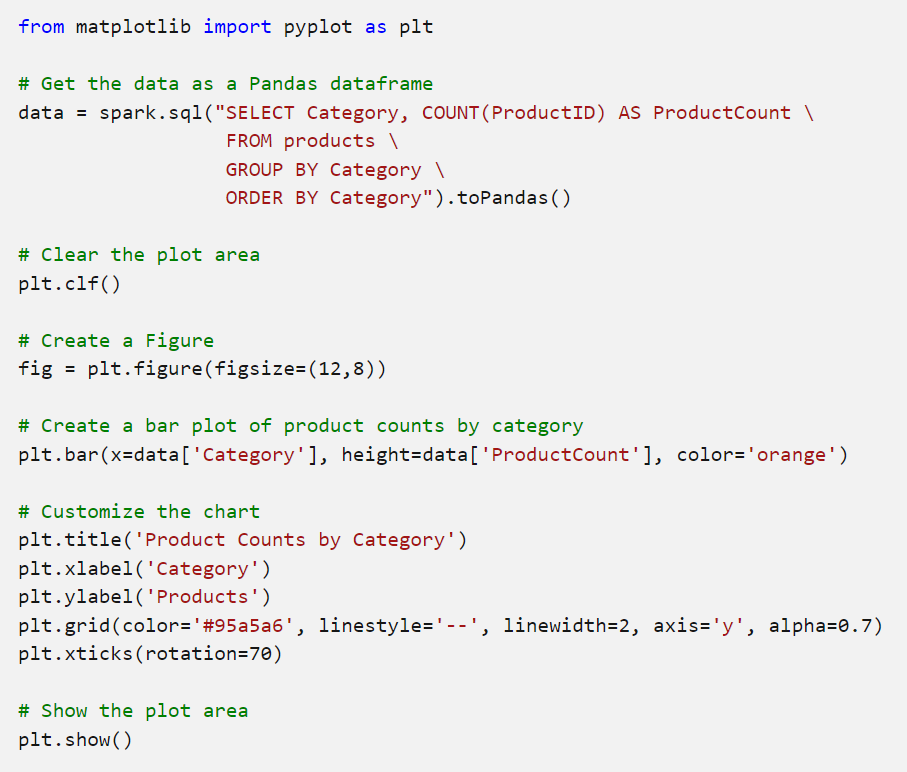
The SQL code example returns a resultset that is automatically displayed in the notebook as a table, like the one below:

| **Category** | **ProductCount** |
| --- | --- |
| Bib-Shorts | 3 |
| Bike Racks | 1 |
| Bike Stands | 1 |
| ... | ... |

## **Using graphics packages in code**

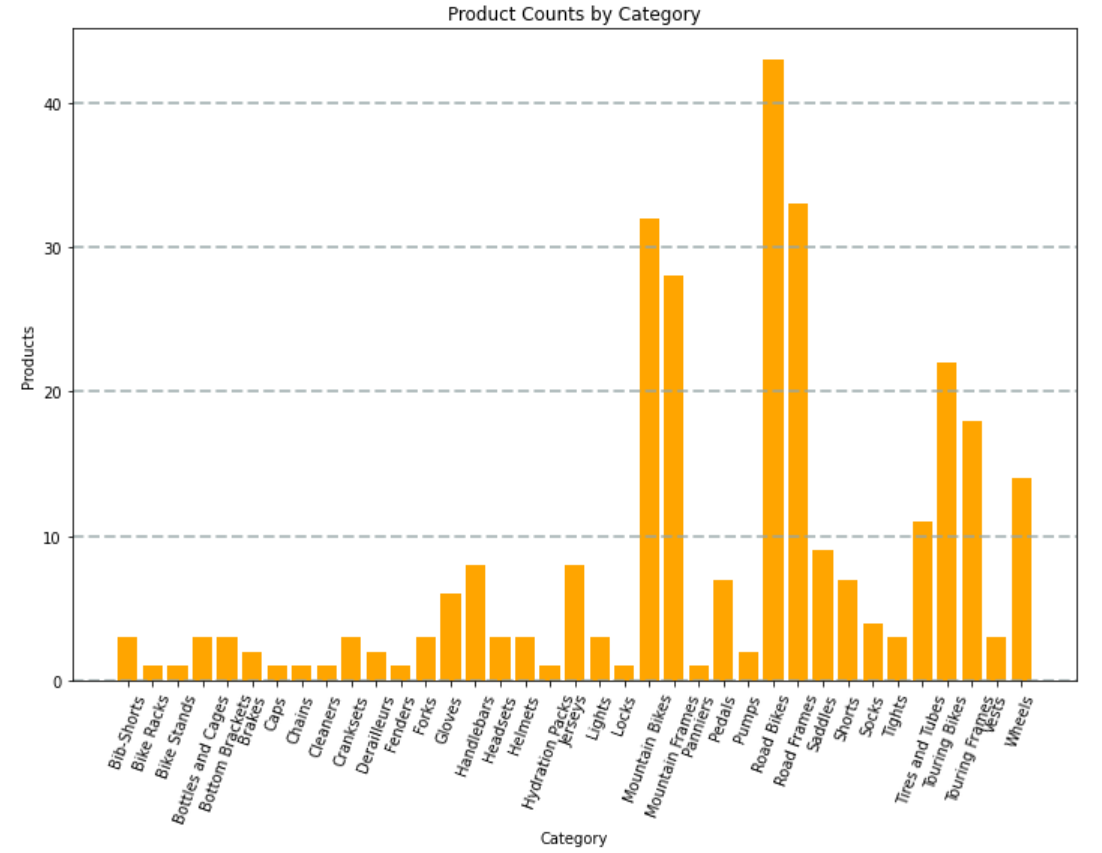
There are many graphics packages that you can use to create data visualizations in code. In particular, Python supports a large selection of packages; most of them built on the base **Matplotlib** library. The output from a graphics library can be rendered in a notebook, making it easy to combine code to ingest and manipulate data with inline data visualizations and markdown cells to provide commentary.

For example, you could use the following PySpark code to aggregate data from the hypothetical products data explored previously in this module, and use Matplotlib to create a chart from the aggregated data.



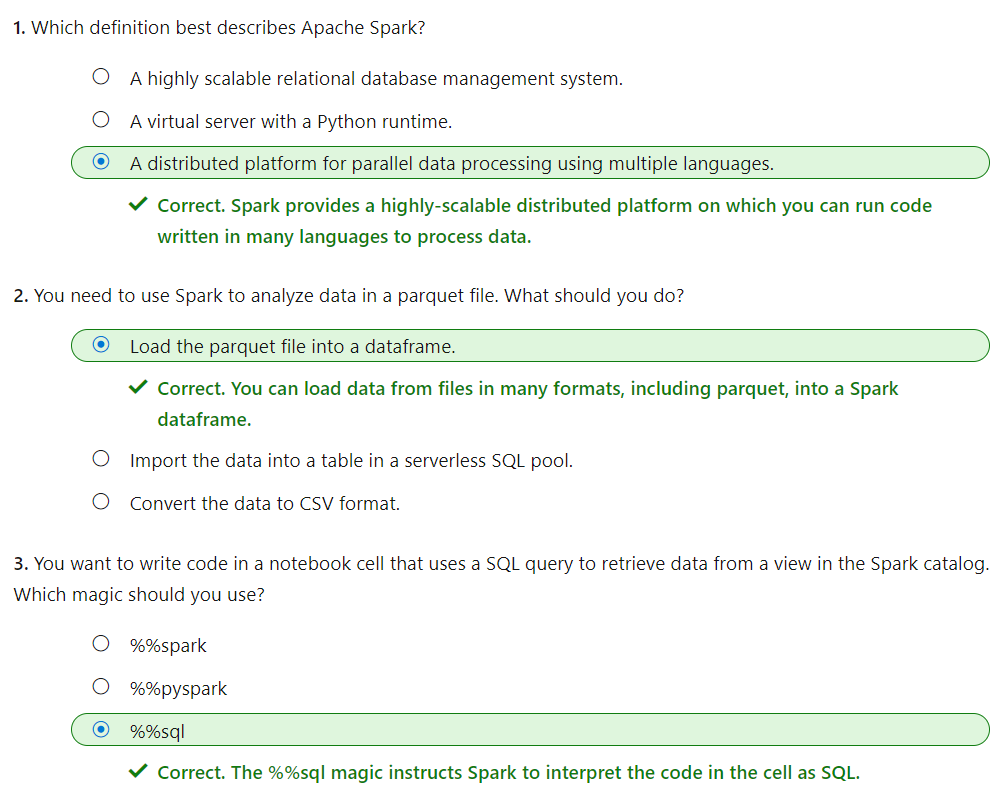
The Matplotlib library requires data to be in a Pandas dataframe rather than a Spark dataframe, so the **toPandas** method is used to convert it. The code then creates a figure with a specified size and plots a bar chart with some custom property configuration before showing the resulting plot.

The chart produced by the code would look similar to the following image:



You can use the Matplotlib library to create many kinds of chart; or if preferred, you can use other libraries such as **Seaborn** to create highly customized charts.

<https://microsoftlearning.github.io/DP-500-Azure-Data-Analyst/Instructions/labs/02-analyze-files-with-Spark.html>



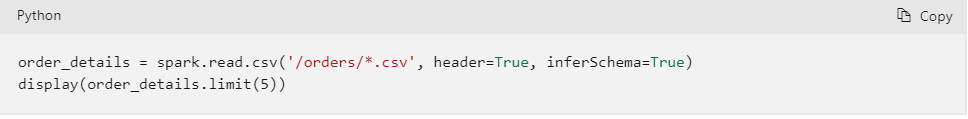
# Transform data with Spark in Azure Synapse Analytics

Apache Spark provides a powerful platform for performing data cleansing and transformation tasks on large volumes of data. By using the Spark *dataframe* object, you can easily load data from files in a data lake and perform complex modifications. You can then save the transformed data back to the data lake for downstream processing or ingestion into a data warehouse.

Azure Synapse Analytics provides Apache Spark pools that you can use to run Spark workloads to transform data as part of a data ingestion and preparation workload. You can use natively supported notebooks to write and run code on a Spark pool to prepare data for analysis. You can then use other Azure Synapse Analytics capabilities such as SQL pools to work with the transformed data.

# Modify and save dataframes

Apache Spark provides the dataframe object as the primary structure for working with data. You can use dataframes to query and transform data, and persist the results in a data lake. To load data into a dataframe, you use the **spark.read** function, specifying the file format, path, and optionally the schema of the data to be read. For example, the following code loads data from all .csv files in the **orders** folder into a dataframe named **order\_details** and then displays the first five records.



## Transform the data structure

After loading the source data into a dataframe, you can use the dataframe object's methods and Spark functions to transform it. Typical operations on a dataframe include:

* Filtering rows and columns
* Renaming columns
* Creating new columns, often derived from existing ones
* Replacing null or other values

In the following example, the code uses the split function to separate the values in the **CustomerName** column into two new columns named **FirstName** and **LastName**. Then it uses the drop method to delete the original **CustomerName** column.



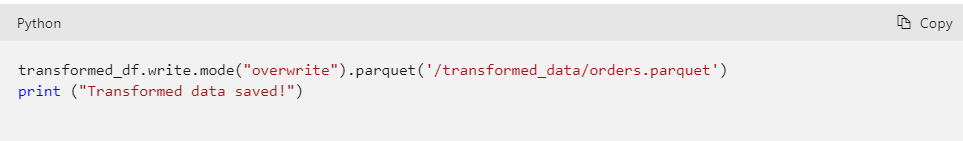
|  |
| --- |
| from pyspark.sql.functions import split, col  # Create the new FirstName and LastName fields  transformed\_df = order\_details.withColumn("FirstName", split(col("CustomerName"), " ").getItem(0)).withColumn("LastName", split(col("CustomerName"), " ").getItem(1))  # Remove the CustomerName field  transformed\_df = transformed\_df.drop("CustomerName")  display(transformed\_df.limit(5)) |

You can use the full power of the Spark SQL library to transform the data by filtering rows, deriving, removing, renaming columns, and any applying other required data modifications.

## Save the transformed data

After your dataFrame is in the required structure, you can save the results to a supported format in your data lake.

The following code example saves the dataFrame into a parquet file in the data lake, replacing any existing file of the same name.



**Note**

The Parquet format is typically preferred for data files that you will use for further analysis or ingestion into an analytical store. Parquet is a very efficient format that is supported by most large scale data analytics systems. In fact, sometimes your data transformation requirement may simply be to convert data from another format (such as CSV) to Parquet!

# Partition data files

Partitioning is an optimization technique that enables spark to maximize performance across the worker nodes. More performance gains can be achieved when filtering data in queries by eliminating unnecessary disk IO.

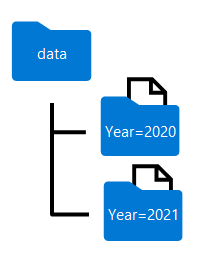
## Partition the output file

To save a dataframe as a partitioned set of files, use the **partitionBy** method when writing the data.

The following example creates a derived **Year** field. Then uses it to partition the data.



The folder names generated when partitioning a dataframe include the partitioning column name and value in a ***column=value*** format, as shown here:



**Note**

You can partition the data by multiple columns, which results in a hierarchy of folders for each partitioning key. For example, you could partition the order in the example by year and month, so that the folder hierarchy includes a folder for each year value, which in turn contains a subfolder for each month value.

## Filter parquet files in a query

When reading data from parquet files into a dataframe, you have the ability to pull data from any folder within the hierarchical folders. This filtering process is done with the use of explicit values and wildcards against the partitioned fields.

In the following example, the following code will pull the sales orders, which were placed in 2020.



**Note**

The partitioning columns specified in the file path are omitted in the resulting dataframe. The results produced by the example query would not include a **Year** column - all rows would be from 2020.

# Transform data with SQL

The SparkSQL library, which provides the dataframe structure also enables you to use SQL as a way of working with data. With this approach, You can query and transform data in dataframes by using SQL queries, and persist the results as tables.

**Note**

Tables are metadata abstractions over files. The data is not stored in a relational table, but the table provides a relational layer over files in the data lake.

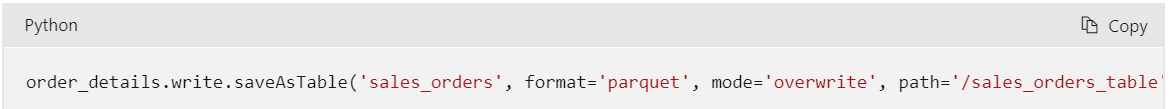
## Define tables and views

Table definitions in Spark are stored in the metastore, a metadata layer that encapsulates relational abstractions over files. External tables are relational tables in the metastore that reference files in a data lake location that you specify. You can access this data by querying the table or by reading the files directly from the data lake.

**Note**

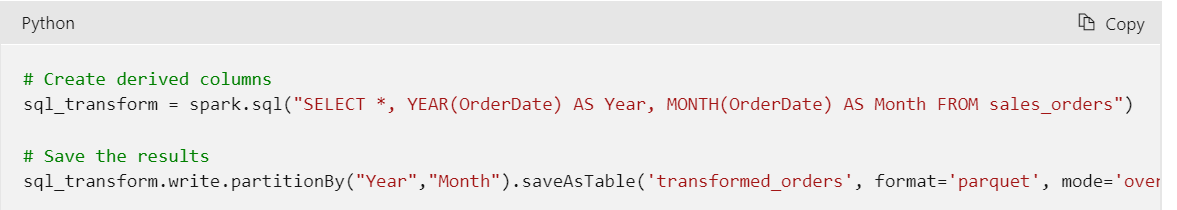
External tables are "loosely bound" to the underlying files and deleting the table does not delete the files. This allows you to use Spark to do the heavy lifting of transformation then persist the data in the lake. After this is done you can drop the table and downstream processes can access these optimized structures. You can also define managed tables, for which the underlying data files are stored in an internally manages storage location associated with the metastore. Manages tables are "tightly-bound" to the files, and dropping a managed table deletes the associated files.

The following code example saves a dataframe (loaded from CSV files) as an external table name **sales\_orders**. The files are stored in the **/sales\_orders\_table** folder in the data lake.



## Use SQL to query and transform the data

After defining a table, you can use of SQL to query and transform its data. The following code creates two new derived columns named **Year** and **Month** and then creates a new table transformed\_orders with the new derived columns added.

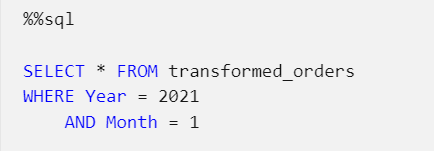


|  |
| --- |
| # Create derived columns  sql\_transform = spark.sql("SELECT \*, YEAR(OrderDate) AS Year, MONTH(OrderDate) AS Month FROM sales\_orders")  # Save the results  sql\_transform.write.partitionBy("Year","Month").saveAsTable('transformed\_orders', format='parquet', mode='overwrite', path='/transformed\_orders\_table') |

The data files for the new table are stored in a hierarchy of folders with the format of **Year=\*NNNN\* / Month=\*N\***, with each folder containing a parquet file for the corresponding orders by year and month.

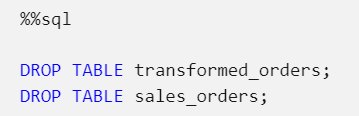
## Query the metastore

Because this new table was created in the metastore, you can use SQL to query it directly with the %%sql magic key in the first line to indicate that the SQL syntax will be used as shown in the following script:



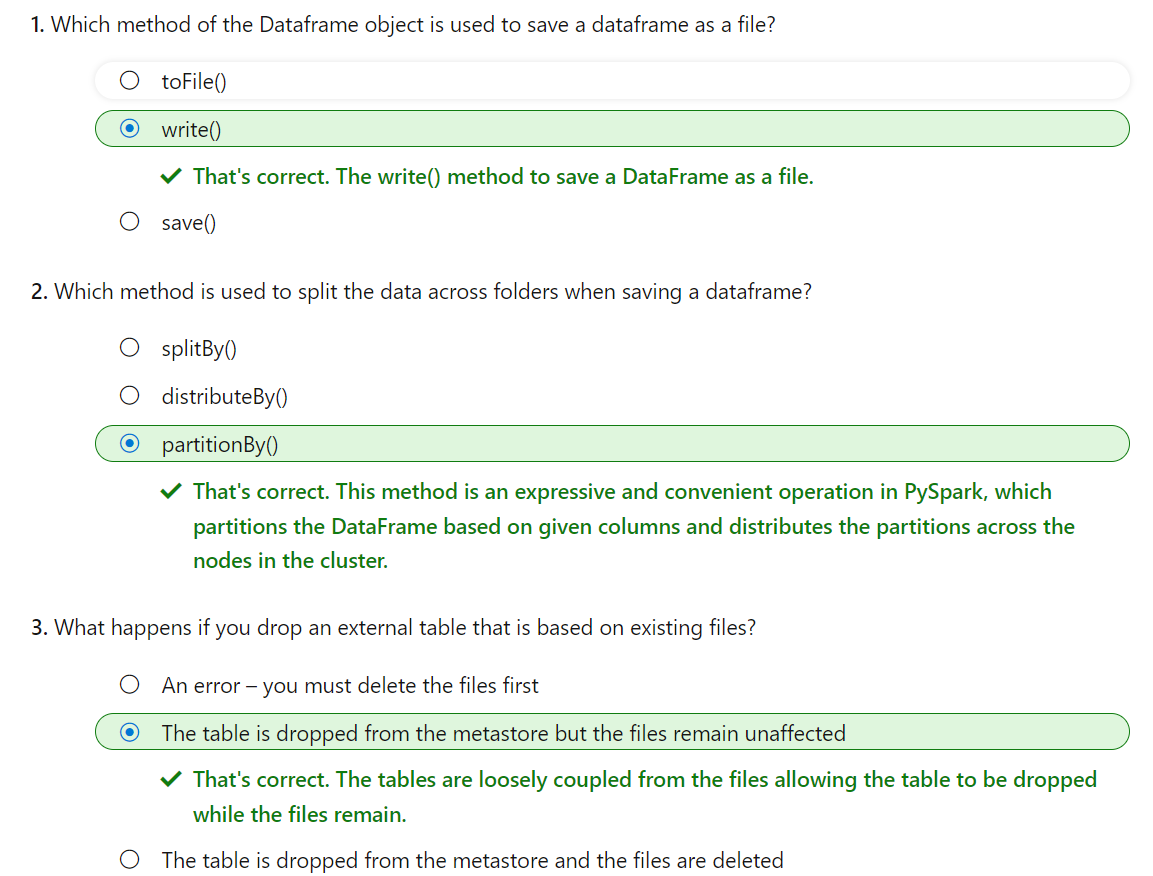
## Drop tables

When working with external tables, you can use the DROP command to delete the table definitions from the metastore without affecting the files in the data lake. This approach enables you to clean up the metastore after using SQL to transform the data, while making the transformed data files available to downstream data analysis and ingestion processes.



# Exercise: Transform data with Spark in Azure Synapse Analytics

<https://microsoftlearning.github.io/dp-203-azure-data-engineer/Instructions/Labs/06-Transform-Data-with-Spark.html>



# Use Delta Lake in Azure Synapse Analytics

Delta Lake is an open source relational storage area for Spark that you can use to implement a data lakehouse architecture in Azure Synapse Analytics.

## Learning objectives

In this module, you'll learn how to:

* Describe core features and capabilities of Delta Lake.
* Create and use Delta Lake tables in a Synapse Analytics Spark pool.
* Create Spark catalog tables for Delta Lake data.
* Use Delta Lake tables for streaming data.
* Query Delta Lake tables from a Synapse Analytics SQL pool.

Linux foundation Delta Lake is an open-source storage layer for Spark that enables relational database capabilities for batch and streaming data. By using Delta Lake, you can implement a data lakehouse architecture in Spark to support SQL\_based data manipulation semantics with support for transactions and schema enforcement. The result is an analytical data store that offers many of the advantages of a relational database system with the flexibility of data file storage in a data lake.

# Understand Delta Lake

Delta Lake is an open-source storage layer that adds relational database semantics to Spark-based data lake processing. Delta Lake is supported in Azure Synapse Analytics Spark pools for PySpark, Scala, and .NET code.

The benefits of using Delta Lake in a Synapse Analytics Spark pool include:

* **Relational tables that support querying and data modification**. With Delta Lake, you can store data in tables that support CRUD (create, read, update, and delete) operations. In other words, you can select, insert, update, and delete rows of data in the same way you would in a relational database system.
* **Support for ACID transactions**. Relational databases are designed to support transactional data modifications that provide atomicity (transactions complete as a single unit of work), consistency (transactions leave the database in a consistent state), isolation (in-process transactions can't interfere with one another), and durability (when a transaction completes, the changes it made are persisted). Delta Lake brings this same transactional support to Spark by implementing a transaction log and enforcing serializable isolation for concurrent operations.
* **Data versioning and time travel**. Because all transactions are logged in the transaction log, you can track multiple versions of each table row and even use the time travel feature to retrieve a previous version of a row in a query.
* **Support for batch and streaming data**. While most relational databases include tables that store static data, Spark includes native support for streaming data through the Spark Structured Streaming API. Delta Lake tables can be used as both sinks (destinations) and sources for streaming data.
* **Standard formats and interoperability**. The underlying data for Delta Lake tables is stored in Parquet format, which is commonly used in data lake ingestion pipelines. Additionally, you can use the serverless SQL pool in Azure Synapse Analytics to query Delta Lake tables in SQL.

**Tip**

For more information about Delta Lake in Azure Synapse Analytics, see [**What is Delta Lake**](https://learn.microsoft.com/en-us/azure/synapse-analytics/spark/apache-spark-what-is-delta-lake) in the Azure Synapse Analytics documentation.

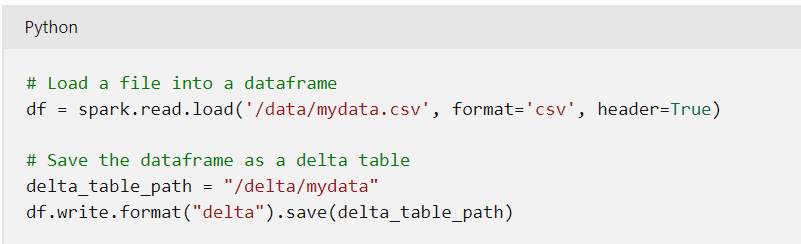
# Create Delta Lake tables

Delta lake is built on tables, which provide a relational storage abstraction over files in a data lake.

## Creating a Delta Lake table from a dataframe

One of the easiest ways to create a Delta Lake table is to save a dataframe in the delta format, specifying a path where the data files and related metadata information for the table should be stored.

For example, the following PySpark code loads a dataframe with data from an existing file, and then saves that dataframe to a new folder location in delta format:

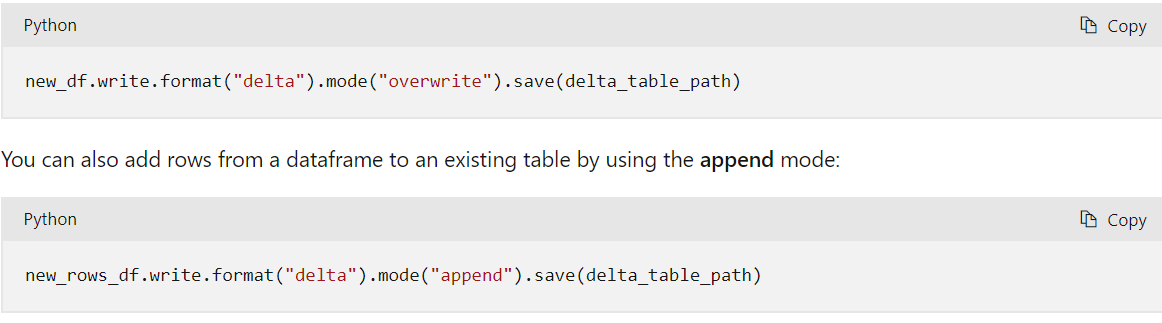


After saving the delta table, the path location you specified includes parquet files for the data (regardless of the format of the source file you loaded into the dataframe) and a **\_delta\_log** folder containing the transaction log for the table.

**Note**

The transaction log records all data modifications to the table. By logging each modification, transactional consistency can be enforced and versioning information for the table can be retained.

You can replace an existing Delta Lake table with the contents of a dataframe by using the **overwrite** mode, as shown here:



## Making conditional updates

While you can make data modifications in a dataframe and then replace a Delta Lake table by overwriting it, a more common pattern in a database is to insert, update or delete rows in an existing table as discrete transactional operations. To make such modifications to a Delta Lake table, you can use the **DeltaTable** object in the Delta Lake API, which supports update, delete, and merge operations. For example, you could use the following code to update the **price** column for all rows with a **category** column value of "Accessories":



The data modifications are recorded in the transaction log, and new parquet files are created in the table folder as required.

**Tip**

For more information about using the Data Lake API, see [**the Delta Lake API documentation**](https://docs.delta.io/latest/delta-apidoc.html)

## Querying a previous version of a table

Delta Lake tables support versioning through the transaction log. The transaction log records modifications made to the table, noting the timestamp and version number for each transaction. You can use this logged version data to view previous versions of the table - a feature known as time travel.

You can retrieve data from a specific version of a Delta Lake table by reading the data from the delta table location into a dataframe, specifying the version required as a versionAsOf option:



# Create catalog tables

So far we've considered Delta Lake table instances created from dataframes and modified through the Delta Lake API. You can also define Delta Lake tables as catalog tables in the Hive metastore for your Spark pool, and work with them using SQL.

## External vs managed tables

Tables in a Spark catalog, including Delta Lake tables, can be managed or external; and it's important to understand the distinction between these kinds of table.

* A managed table is defined without a specified location, and the data files are stored within the storage used by the metastore. Dropping the table not only removes its metadata from the catalog, but also deletes the folder in which its data files are stored.
* An external table is defined for a custom file location, where the data for the table is stored. The metadata for the table is defined in the Spark catalog. Dropping the table deletes the metadata from the catalog, but doesn't affect the data files.

## Creating catalog tables

There are several ways to create catalog tables.

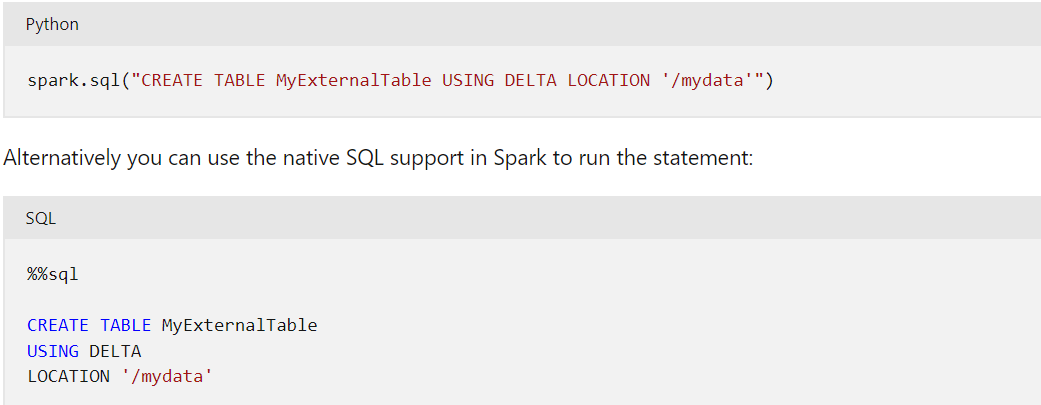
### Creating a catalog table from a dataframe

You can create managed tables by writing a dataframe using the saveAsTable operation as shown in the following examples:



### Creating a catalog table using SQL

You can also create a catalog table by using the CREATE TABLE SQL statement with the USING DELTA clause, and an optional LOCATION parameter for external tables. You can run the statement using the SparkSQL API, like the following example:

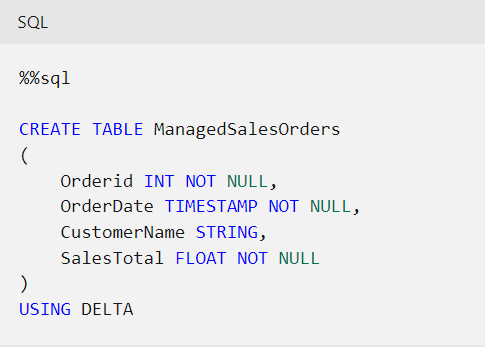


**Tip**

The CREATE TABLE statement returns an error if a table with the specified name already exists in the catalog. To mitigate this behavior, you can use a CREATE TABLE IF NOT EXISTS statement or the CREATE OR REPLACE TABLE statement.

#### Defining the table schema

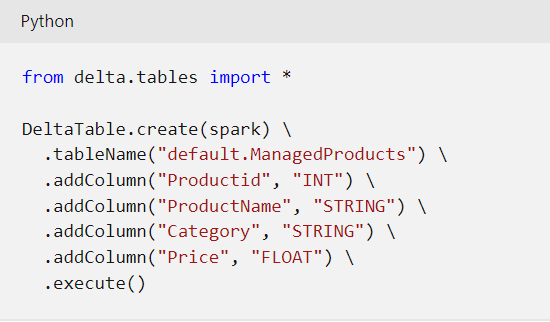
In all of the examples so far, the table is created without an explicit schema. In the case of tables created by writing a dataframe, the table schema is inherited from the dataframe. When creating an external table, the schema is inherited from any files that are currently stored in the table location. However, when creating a new managed table, or an external table with a currently empty location, you define the table schema by specifying the column names, types, and nullability as part of the CREATE TABLE statement; as shown in the following example:



When using Delta Lake, table schemas are enforced - all inserts and updates must comply with the specified column nullability and data types.

### Using the DeltaTableBuilder API

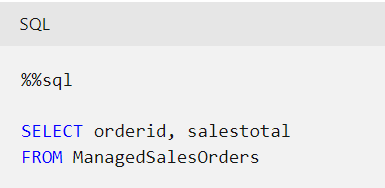
You can use the DeltaTableBuilder API (part of the Delta Lake API) to create a catalog table, as shown in the following example:



Similarly to the CREATE TABLE SQL statement, the create method returns an error if a table with the specified name already exists. You can mitigate this behavior by using the createIfNotExists or createOrReplace method.

## Using catalog tables

You can use catalog tables like tables in any SQL-based relational database, querying and manipulating them by using standard SQL statements. For example, the following code example uses a SELECT statement to query the **ManagedSalesOrders** table:



**Tip**

For more information about working with Delta Lake, see [**Table batch reads and writes**](https://docs.delta.io/latest/delta-batch.html)in the Delta Lake documentation.

# Use Delta Lake with streaming data

All of the data we've explored up to this point has been static data in files. However, many data analytics scenarios involve streaming data that must be processed in near real time. For example, you might need to capture readings emitted by internet-of-things (IoT) devices and store them in a table as they occur.

## Spark Structured Streaming

A typical stream processing solution involves constantly reading a stream of data from a source, optionally processing it to select specific fields, aggregate and group values, or otherwise manipulate the data, and writing the results to a sink.

Spark includes native support for streaming data through Spark Structured Streaming, an API that is based on a boundless dataframe in which streaming data is captured for processing. A Spark Structured Streaming dataframe can read data from many different kinds of streaming source, including network ports, real time message brokering services such as Azure Event Hubs or Kafka, or file system locations.

**Tip**

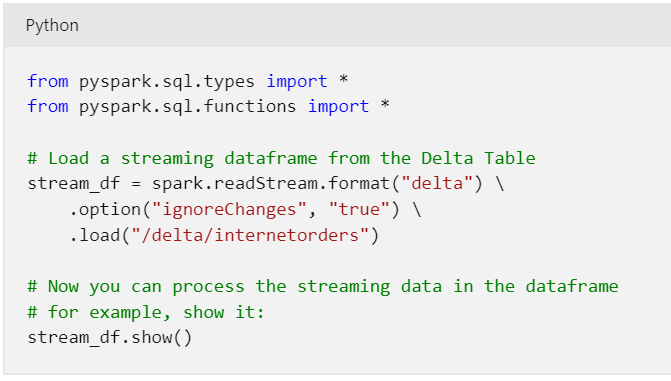
For more information about Spark Structured Streaming, see [**Structured Streaming Programming Guide**](https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html) in the Spark documentation.

## Streaming with Delta Lake tables

You can use a Delta Lake table as a source or a sink for Spark Structured Streaming. For example, you could capture a stream of real time data from an IoT device and write the stream directly to a Delta Lake table as a sink - enabling you to query the table to see the latest streamed data. Or, you could read a Delta Table as a streaming source, enabling you to constantly report new data as it is added to the table.

### Using a Delta Lake table as a streaming source

In the following PySpark example, a Delta Lake table is used to store details of Internet sales orders. A stream is created that reads data from the Delta Lake table folder as new data is appended.



**Note**

When using a Delta Lake table as a streaming source, only append operations can be included in the stream. Data modifications will cause an error unless you specify the ignoreChanges or ignoreDeletes option.

After reading the data from the Delta Lake table into a streaming dataframe, you can use the Spark Structured Streaming API to process it. In the example above, the dataframe is simply displayed; but you could use Spark Structured Streaming to aggregate the data over temporal windows (for example to count the number of orders placed every minute) and send the aggregated results to a downstream process for near-real-time visualization.

### Using a Delta Lake table as a streaming sink

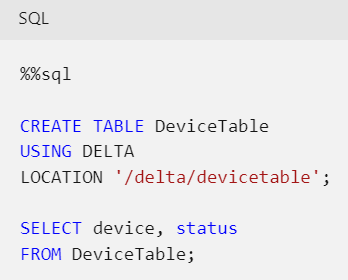
In the following PySpark example, a stream of data is read from JSON files in a folder. The JSON data in each file contains the status for an IoT device in the format {"device":"Dev1","status":"ok"} New data is added to the stream whenever a file is added to the folder. The input stream is a boundless dataframe, which is then written in delta format to a folder location for a Delta Lake table.



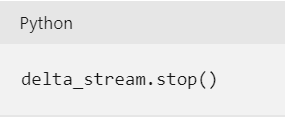
**Note**

The checkpointLocation option is used to write a checkpoint file that tracks the state of the stream processing. This file enables you to recover from failure at the point where stream processing left off.

After the streaming process has started, you can query the Delta Lake table to which the streaming output is being written to see the latest data. For example, the following code creates a catalog table for the Delta Lake table folder and queries it:



To stop the stream of data being written to the Delta Lake table, you can use the stop method of the streaming query:



**Tip**

For more information about using Delta Lake tables for streaming data, see [**Table streaming reads and writes**](https://docs.delta.io/latest/delta-streaming.html)in the Delta Lake documentation.

# Use Delta Lake in a SQL pool

Delta Lake is designed as a transactional, relational storage layer for Apache Spark; including Spark pools in Azure Synapse Analytics. However, Azure Synapse Analytics also includes a serverless SQL pool runtime that enables data analysts and engineers to run SQL queries against data in a data lake or a relational database.

**Note**

You can only query data from Delta Lake tables in a serverless SQL pool; you can't update, insert, or delete data.

## Querying delta formatted files with OPENROWSET

The serverless SQL pool in Azure Synapse Analytics includes support for reading delta format files; enabling you to use the SQL pool to query Delta Lake tables. This approach can be useful in scenarios where you want to use Spark and Delta tables to process large quantities of data, but use the SQL pool to run queries for reporting and analysis of the processed data.

In the following example, a SQL SELECT query reads delta format data using the OPENROWSET function.



You could run this query in a serverless SQL pool to retrieve the latest data from the Delta Lake table stored in the specified file location.

You could also create a database and add a data source that encapsulates the location of your Delta Lake data files, as shown in this example:

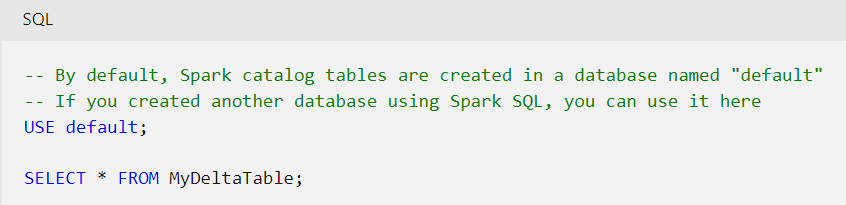


**Note**

When working with Delta Lake data, which is stored in Parquet format, it's generally best to create a database with a UTF-8 based collation in order to ensure string compatibility.

## Querying catalog tables

The serverless SQL pool in Azure Synapse Analytics has shared access to databases in the Spark metastore, so you can query catalog tables that were created using Spark SQL. In the following example, a SQL query in a serverless SQL pool queries a catalog table that contains Delta Lake data:



**Tip**

For more information about using Delta Tables from a serverless SQL pool, see [**Query Delta Lake files using serverless SQL pool in Azure Synapse Analytics**](https://learn.microsoft.com/en-us/azure/synapse-analytics/sql/query-delta-lake-format) in the Azure Synapse Analytics documentation.

# Exercise - Use Delta Lake in Azure Synapse Analytics

<https://microsoftlearning.github.io/dp-203-azure-data-engineer/Instructions/Labs/07-Use-delta-lake.html>

