**Build data analytics solutions using Azure Synapse serverless SQL pools**

Azure Synapse Analytics includes serverless SQL pools, which are tailored for querying data in a data lake. With a serverless SQL pool you can use SQL code to query data in files of various common formats without needing to load the file data into database storage. This capability helps data analysts and data engineers analyze and process file data in the data lake using a familiar data processing language, without the need to create or maintain a relational database store.

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

* Identify capabilities and use cases for serverless SQL pools in Azure Synapse Analytics
* Query CSV, JSON, and Parquet files using a serverless SQL pool
* Create external database objects in a serverless SQL pool

**[Use Azure Synapse serverless SQL pool to query files in a data lake](https://learn.microsoft.com/en-us/training/modules/query-data-lake-using-azure-synapse-serverless-sql-pools/)**

**Understand Azure Synapse serverless SQL pool capabilities and use cases**

Azure Synapse SQL is a distributed query system in Azure Synapse Analytics that offers two kinds of runtime environments:

* **Serverless SQL pool**: on-demand SQL query processing, primarily used to work with data in a data lake.
* **Dedicated SQL pool**: Enterprise-scale relational database instances used to host data warehouses in which data is stored relational tables.

The benefits of using serverless SQL pool include:

* A familiar Transact-SQL syntax to query data in place without the need to copy or load data into a specialized store.
* Integrated connectivity from a wide range of business intelligence and ad-hoc querying tools, including the most popular drivers.
* Distributed query processing that is built for large-scale data, and computational functions - resulting in fast query performance.
* Built-in query execution fault-tolerance, resulting in high reliability and success rates even for long-running queries involving large data sets.
* No infrastructure to setup or clusters to maintain. A built-in endpoint for this service is provided within every Azure Synapse workspace, so you can start querying data as soon as the workspace is created.
* No charge for resources reserved, you're only charged for the data processed by queries you run i.e., provides a pay-per-query endpoint to query the data in your data lake

## **When to use serverless SQL pools**

Serverless SQL pool is tailored for querying the data residing in the data lake, so in addition to eliminating management burden, it eliminates a need to worry about ingesting the data into the system. You just point the query to the data that is already in the lake and run it.

Synapse SQL serverless resource model is great for unplanned or "bursty" workloads that can be processed using the always-on serverless SQL endpoint in your Azure Synapse Analytics workspace. Using the serverless pool helps when you need to know exact cost for each query executed to monitor and attribute costs.

**Note**

Serverless SQL pool is an analytics system and is **not recommended for OLTP workloads** such as databases used by applications to store transactional data. Workloads that require millisecond response times and are looking to pinpoint a single row in a data set are not good fit for serverless SQL pool.

Common use cases for serverless SQL pools include:

* **Data exploration**: Data exploration involves browsing the data lake to get initial insights about the data, and is easily achievable with Azure Synapse Studio. You can browse through the files in your linked data lake storage, and use the built-in serverless SQL pool to automatically generate a SQL script to select TOP 100 rows from a file or folder just as you would do with a table in SQL Server. From there, you can apply projections, filtering, grouping, and most of the operation over the data as if the data were in a regular SQL Server table.
* **Data transformation**: While Azure Synapse Analytics provides great data transformations capabilities with Synapse Spark, some data engineers might find data transformation easier to achieve using SQL. Serverless SQL pool enables you to perform SQL-based data transformations; either interactively or as part of an automated data pipeline.
* **Logical data warehouse**: After your initial exploration of the data in the data lake, you can define external objects such as tables and views in a serverless SQL database. The data remains stored in the data lake files, but are abstracted by a relational schema that can be used by client applications and analytical tools to query the data as they would in a relational database hosted in SQL Server.

# Query files using a serverless SQL pool

You can use a serverless SQL pool to query data files in various common file formats, including:

* Delimited text, such as comma-separated values (CSV) files.
* JavaScript object notation (JSON) files.
* Parquet files.

The basic syntax for querying is the same for all of these types of file, and is built on the OPENROWSET SQL function; which generates a tabular rowset from data in one or more files. For example, the following query could be used to extract data from CSV files.

SELECT TOP 100 \*

FROM OPENROWSET(

BULK 'https://mydatalake.blob.core.windows.net/data/files/\*.csv',

FORMAT = 'csv') AS rows

The OPENROWSET function includes more parameters that determine factors such as:

* The schema of the resulting rowset
* Additional formatting options for delimited text files.

**Tip**

You'll find the full syntax for the OPENROWSET function in the [**Azure Synapse Analytics documentation**](https://learn.microsoft.com/en-us/azure/synapse-analytics/sql/develop-openrowset#syntax).

The output from OPENROWSET is a rowset to which an alias must be assigned. In the previous example, the alias **rows** is used to name the resulting rowset.

The **BULK** parameter includes the full URL to the location in the data lake containing the data files. This can be an individual file, or a folder with a wildcard expression to filter the file types that should be included. The **FORMAT** parameter specifies the type of data being queried. The example above reads delimited text from all .csv files in the **files** folder.

**Note**

This example assumes that the user has access to the files in the underlying store, If the files are protected with a SAS key or custom identity, you would need to [**create a server-scoped credential**](https://learn.microsoft.com/en-us/azure/synapse-analytics/sql/develop-storage-files-storage-access-control?tabs=shared-access-signature#server-scoped-credential).

As seen in the previous example, you can use wildcards in the **BULK** parameter to include or exclude files in the query. The following list shows a few examples of how this can be used:

* https://mydatalake.blob.core.windows.net/data/files/file1.csv: Only include *file1.csv* in the *files* folder.
* https://mydatalake.blob.core.windows.net/data/files/file\*.csv: All .csv files in the *files* folder with names that start with "file".
* https://mydatalake.blob.core.windows.net/data/files/\*: All files in the *files* folder.
* https://mydatalake.blob.core.windows.net/data/files/\*\*: All files in the *files* folder, and recursively its subfolders.

You can also specify multiple file paths in the **BULK** parameter, separating each path with a comma.

## **Querying delimited text files**

Delimited text files are a common file format within many businesses. The specific formatting used in delimited files can vary, for example:

* With and without a header row.
* Comma and tab-delimited values.
* Windows and Unix style line endings.
* Non-quoted and quoted values, and escaping characters.

Regardless of the type of delimited file you're using, you can read data from them by using the OPENROWSET function with the **csv** FORMAT parameter, and other parameters as required to handle the specific formatting details for your data. For example:

SELECT TOP 100 \*

FROM OPENROWSET(

BULK 'https://mydatalake.blob.core.windows.net/data/files/\*.csv',

FORMAT = 'csv',

PARSER\_VERSION = '2.0',

FIRSTROW = 2) AS rows

The **PARSER\_VERSION** is used to determine how the query interprets the text encoding used in the files. Version 1.0 is the default and supports a wide range of file encodings, while version 2.0 supports fewer encodings but offers better performance. The **FIRSTROW** parameter is used to skip rows in the text file, to eliminate any unstructured preamble text or to ignore a row containing column headings.

Additional parameters you might require when working with delimited text files include:

* FIELDTERMINATOR - the character used to separate field values in each row. For example, a tab-delimited file separates fields with a TAB (*\t*) character. The default field terminator is a comma (*,*).
* ROWTERMINATOR - the character used to signify the end of a row of data. For example, a standard Windows text file uses a combination of a carriage return (CR) and line feed (LF), which is indicated by the code *\n*; while UNIX-style text files use a single line feed character, which can be indicated using the code *0x0a*.
* FIELDQUOTE - the character used to enclose quoted string values. For example, to ensure that the comma in the address field value *126 Main St, apt 2* isn't interpreted as a field delimiter, you might enclose the entire field value in quotation marks like this: *"126 Main St, apt 2"*. The double-quote (") is the default field quote character.

**Tip**

For details of additional parameters when working with delimted text files, refer to the [**Azure Synapse Analytics documentation**](https://learn.microsoft.com/en-us/azure/synapse-analytics/sql/develop-openrowset#syntax).

### **Specifying the rowset schema**

It's common for delimited text files to include the column names in the first row. The OPENROWSET function can use this to define the schema for the resulting rowset, and automatically infer the data types of the columns based on the values they contain. For example, consider the following delimited text:

product\_id,product\_name,list\_price

123,Widget,12.99

124,Gadget,3.99

The data consists of the following three columns:

* **product\_id** (integer number)
* **product\_name** (string)
* **list\_price** (decimal number)

You could use the following query to extract the data with the correct column names and appropriately inferred SQL Server data types (in this case INT, NVARCHAR, and DECIMAL)

SELECT TOP 100 \*

FROM OPENROWSET(

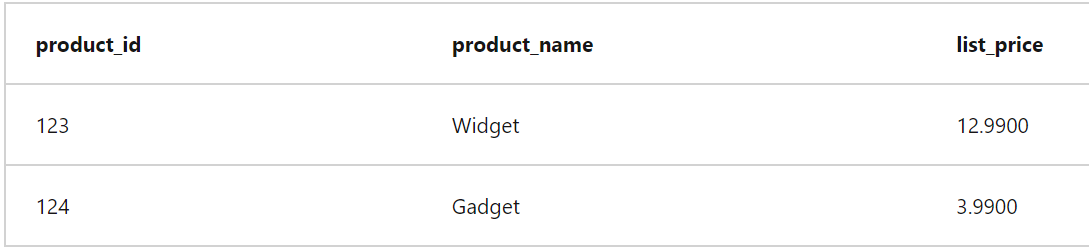
BULK 'https://mydatalake.blob.core.windows.net/data/files/\*.csv',

FORMAT = 'csv',

PARSER\_VERSION = '2.0',

HEADER\_ROW = TRUE) AS rows

The **HEADER\_ROW** parameter (which is only available when using parser version 2.0) instructs the query engine to use the first row of data in each file as the column names, like this:



Now consider the following data:

123,Widget,12.99

124,Gadget,3.99

This time, the file doesn't contain the column names in a header row; so while the data types can still be inferred, the column names will be set to **C1**, **C2**, **C3**, and so on.

| **C1** | **C2** | **C3** |
| --- | --- | --- |
| 123 | Widget | 12.9900 |
| 124 | Gadget | 3.9900 |

To specify explicit column names and data types, you can override the default column names and inferred data types by providing a schema definition in a **WITH** clause, like this:

SELECT TOP 100 \*

FROM OPENROWSET(

BULK 'https://mydatalake.blob.core.windows.net/data/files/\*.csv',

FORMAT = 'csv',

PARSER\_VERSION = '2.0')

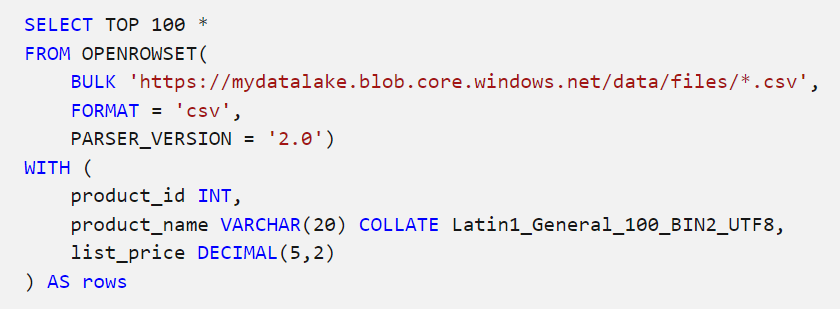
WITH (

product\_id INT,

product\_name VARCHAR(20) COLLATE Latin1\_General\_100\_BIN2\_UTF8,

list\_price DECIMAL(5,2)

) AS rows



This query produces the expected results:

| **product\_id** | **product\_name** | **list\_price** |
| --- | --- | --- |
| 123 | Widget | 12.99 |
| 124 | Gadget | 3.99 |

**Tip**

When working with text files, you may encounter some incompatibility with UTF-8 encoded data and the collation used in the **master** database for the serverless SQL pool. To overcome this, you can specify a compatible collation for individual VARCHAR columns in the schema. See the [**troubleshooting guidance**](https://learn.microsoft.com/en-us/azure/synapse-analytics/troubleshoot/reading-utf8-text) for more details.

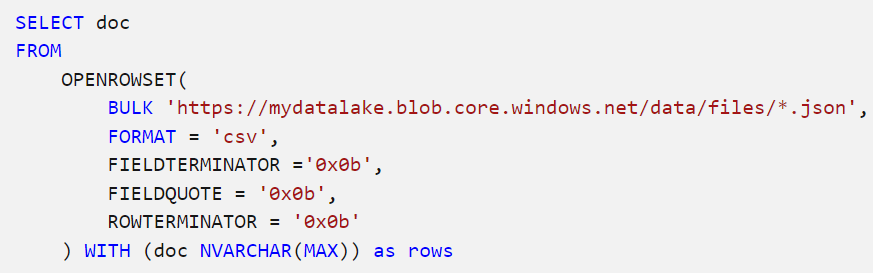
## **Querying JSON files**

JSON is a popular format for web applications that exchange data through REST interfaces or use NoSQL data stores such as Azure Cosmos DB. So, it's not uncommon to persist data as JSON documents in files in a data lake for analysis.

For example, a JSON file that defines an individual product might look like this:



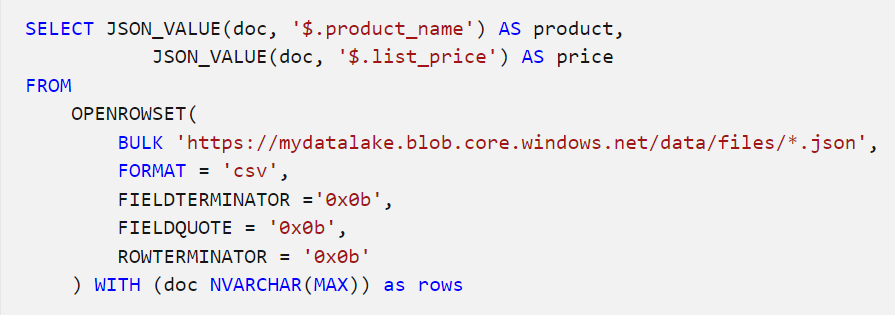
To return product data from a folder containing multiple JSON files in this format, you could use the following SQL query:



OPENROWSET has no specific format for JSON files, so you must use csv format with **FIELDTERMINATOR**, **FIELDQUOTE**, and **ROWTERMINATOR** set to 0x0b, and a schema that includes a single NVARCHAR(MAX) column. The result of this query is a rowset containing a single column of JSON documents, like this:

| **doc** |
| --- |
| {"product\_id":123,"product\_name":"Widget","list\_price": 12.99} |
| {"product\_id":124,"product\_name":"Gadget","list\_price": 3.99} |

To extract individual values from the JSON, you can use the JSON\_VALUE function in the SELECT statement, as shown here:



This query would return a rowset similar to the following results:

| **product** | **price** |
| --- | --- |
| Widget | 12.99 |
| Gadget | 3.99 |

## **Querying Parquet files**

Parquet is a commonly used format for big data processing on distributed file storage. It's an efficient data format that is optimized for compression and analytical querying.

In most cases, the schema of the data is embedded within the Parquet file, so you only need to specify the **BULK** parameter with a path to the file(s) you want to read, and a **FORMAT** parameter of parquet; like this:

SELECT TOP 100 \*

FROM OPENROWSET(

BULK 'https://mydatalake.blob.core.windows.net/data/files/\*.\*',

FORMAT = 'parquet') AS rows

## Query partitioned data

It's common in a data lake to partition data by splitting across multiple files in subfolders that reflect partitioning criteria. This enables distributed processing systems to work in parallel on multiple partitions of the data, or to easily eliminate data reads from specific folders based on filtering criteria. For example, suppose you need to efficiently process sales order data, and often need to filter based on the year and month in which orders were placed. You could partition the data using folders, like this:

To create a query that filters the results to include only the orders for January and February 2020, you could use the following code:



To create a query that filters the results to include only the orders for January and February 2020, you could use the following code:

SELECT \*

FROM OPENROWSET(

BULK 'https://mydatalake.blob.core.windows.net/data/orders/year=\*/month=\*/\*.\*',

FORMAT = 'parquet') AS orders

WHERE orders.filepath(1) = '2020'

AND orders.filepath(2) IN ('1','2');

The numbered filepath parameters in the WHERE clause reference the wildcards in the folder names in the BULK path -so the parameter 1 is the \* in the *year=\** folder name, and parameter 2 is the \* in the *month=\** folder name.

@@@@@@@@@ EXTRA@@@@@@@@

An external table points to data located in Hadoop, Azure Storage blob, or Azure Data Lake Storage. External tables are used to read data from files or write data to files in Azure Storage. With Synapse SQL, you can use external tables to read external data using dedicated SQL pool or serverless SQL pool.

Depending on the type of the external data source, you can use two types of external tables:

* **Hadoop external tables** that you can use to read and export data in various data formats such as CSV, Parquet, and ORC. Hadoop external tables are available in dedicated SQL pools, but they aren't available in serverless SQL pools.
* **Native external tables** that you can use to read and export data in various data formats such as CSV and Parquet. Native external tables are available in serverless SQL pools, and they are in **public preview** in dedicated SQL pools. Writing/exporting data using CETAS and the native external tables is available only in the serverless SQL pool, but not in the dedicated SQL pools.

The key differences between Hadoop and native external tables are presented in the following table:

| **External table type** | **Hadoop** | **Native** |
| --- | --- | --- |
| Dedicated SQL pool | Available | Only Parquet tables are available in **public preview**. |
| Serverless SQL pool | Not available | Available |
| Supported formats | Delimited/CSV, Parquet, ORC, Hive RC, and RC | Serverless SQL pool: Delimited/CSV, Parquet, and [Delta Lake](https://learn.microsoft.com/en-us/azure/synapse-analytics/sql/query-delta-lake-format) Dedicated SQL pool: Parquet (preview) |
| [Folder partition elimination](https://learn.microsoft.com/en-us/azure/synapse-analytics/sql/develop-tables-external-tables?tabs=hadoop#folder-partition-elimination) | No | Partition elimination is available only in the partitioned tables created on Parquet or CSV formats that are synchronized from Apache Spark pools. You might create external tables on Parquet partitioned folders, but the partitioning columns will be inaccessible and ignored, while the partition elimination will not be applied. Do not create [external tables on Delta Lake folders](https://learn.microsoft.com/en-us/azure/synapse-analytics/sql/create-use-external-tables#delta-tables-on-partitioned-folders) because they are not supported. Use [Delta partitioned views](https://learn.microsoft.com/en-us/azure/synapse-analytics/sql/create-use-views#delta-lake-partitioned-views) if you need to query partitioned Delta Lake data. |
| [File elimination](https://learn.microsoft.com/en-us/azure/synapse-analytics/sql/develop-tables-external-tables?tabs=hadoop#file-elimination) (predicate pushdown) | No | Yes in serverless SQL pool. For the string pushdown, you need to use Latin1\_General\_100\_BIN2\_UTF8 collation on the VARCHAR columns to enable pushdown. For more information on collations, refer to [Collation types supported for Synapse SQL](https://learn.microsoft.com/en-us/azure/synapse-analytics/sql/reference-collation-types). |
| Custom format for location | No | Yes, using wildcards like /year=\*/month=\*/day=\* for Parquet or CSV formats. Custom folder paths are not available in Delta Lake. In the serverless SQL pool you can also use recursive wildcards /logs/\*\* to reference Parquet or CSV files in any sub-folder beneath the referenced folder. |
| Recursive folder scan | Yes | Yes. In serverless SQL pools must be specified /\*\* at the end of the location path. In Dedicated pool the folders are always scanned recursively. |
| Storage authentication | Storage Access Key(SAK), Azure Active Directory passthrough, Managed identity, custom application Azure Active Directory identity | [Shared Access Signature(SAS)](https://learn.microsoft.com/en-us/azure/synapse-analytics/sql/develop-storage-files-storage-access-control?tabs=shared-access-signature), [Azure Active Directory passthrough](https://learn.microsoft.com/en-us/azure/synapse-analytics/sql/develop-storage-files-storage-access-control?tabs=user-identity), [Managed identity](https://learn.microsoft.com/en-us/azure/synapse-analytics/sql/develop-storage-files-storage-access-control?tabs=managed-identity), [Custom application Azure AD identity](https://learn.microsoft.com/en-us/azure/synapse-analytics/sql/develop-storage-files-storage-access-control?tabs=service-principal). |
| Column mapping | Ordinal - the columns in the external table definition are mapped to the columns in the underlying Parquet files by position. | Serverless pool: by name. The columns in the external table definition are mapped to the columns in the underlying Parquet files by column name matching. Dedicated pool: ordinal matching. The columns in the external table definition are mapped to the columns in the underlying Parquet files by position. |
| CETAS (exporting/transformation) | Yes | CETAS with the native tables as a target works only in the serverless SQL pool. You cannot use the dedicated SQL pools to export data using native tables. |

**Note**

The native external tables are the recommended solution in the pools where they are generally available. If you need to access external data, always use the native tables in serverless pools. In dedicated pools, you should switch to the native tables for reading Parquet files once they are in GA. Use the Hadoop tables only if you need to access some types that are not supported in native external tables (for example - ORC, RC), or if the native version is not available.

NOTE:

You can create external tables in Synapse SQL pools via the following steps:

1. [CREATE EXTERNAL DATA SOURCE](https://learn.microsoft.com/en-us/azure/synapse-analytics/sql/develop-tables-external-tables?tabs=hadoop#create-external-data-source) to reference an external Azure storage and specify the credential that should be used to access the storage.
2. [CREATE EXTERNAL FILE FORMAT](https://learn.microsoft.com/en-us/azure/synapse-analytics/sql/develop-tables-external-tables?tabs=hadoop#create-external-file-format) to describe format of CSV or Parquet files.
3. [CREATE EXTERNAL TABLE](https://learn.microsoft.com/en-us/azure/synapse-analytics/sql/develop-tables-external-tables?tabs=hadoop#create-external-table) on top of the files placed on the data source with the same file format.

NOTE:

Creates an external file format object that defines external data stored in Azure Blob Storage or Azure Data Lake Storage. Creating an external file format is a prerequisite for creating an external table. The complete documentation is [here](https://learn.microsoft.com/en-us/sql/t-sql/statements/create-external-file-format-transact-sql?view=azure-sqldw-latest&preserve-view=true).

By creating an external file format, you specify the actual layout of the data referenced by an external table.

For more info can refer to:

<https://learn.microsoft.com/en-us/azure/synapse-analytics/sql/develop-tables-external-tables?tabs=hadoop>

@@@@@@@@@@@ extra ends here @@@@@@@@

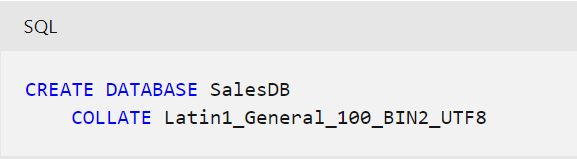
# Create external database objects

You can use the OPENROWSET function in SQL queries that run in the default **master** database of the built-in serverless SQL pool to explore data in the data lake. However, sometimes you may want to create a custom database that contains some objects that make it easier to work with external data in the data lake that you need to query frequently.

## **Creating a database**

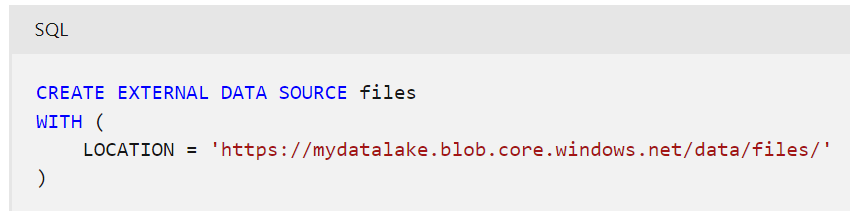
You can create a database in a serverless SQL pool just as you would in a SQL Server instance. You can use the graphical interface in Synapse Studio, or a CREATE DATABASE statement. One consideration is to set the collation of your database so that it supports conversion of text data in files to appropriate Transact-SQL data types.

The following example code creates a database named salesDB with a collation that makes it easier to import UTF-8 encoded text data into VARCHAR columns.

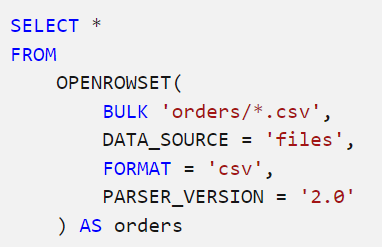


## **Creating an external data source**

You can use the OPENROWSET function with a BULK path to query file data from your own database, just as you can in the **master** database; but if you plan to query data in the same location frequently, it's more efficient to define an external data source that references that location. For example, the following code creates a data source named files for the hypothetical https://mydatalake.blob.core.windows.net/data/files/ folder:

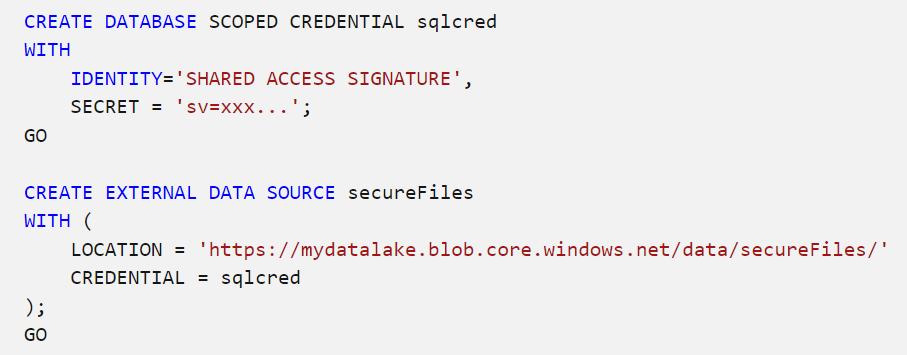


One benefit of an external data source, is that you can simplify an OPENROWSET query to use the combination of the data source and the relative path to the folders or files you want to query:



In this example, the **BULK** parameter is used to specify the relative path for all .csv files in the **orders** folder, which is a subfolder of the **files** folder referenced by the data source.

Another benefit of using a data source is that you can assign a credential for the data source to use when accessing the underlying storage, enabling you to provide access to data through SQL without permitting users to access the data directly in the storage account. For example, the following code creates a credential that uses a shared access signature (SAS) to authenticate against the underlying Azure storage account hosting the data lake.

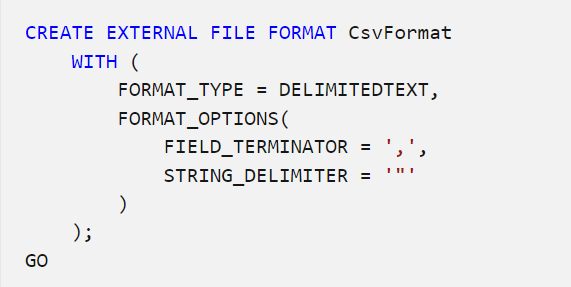


**Tip**

In addition to SAS authentication, you can define credentials that use *managed identity* (the Azure Active Directory identity used by your Azure Synapse workspace), a specific Azure Active Directory principal, or passthrough authentication based on the identity of the user running the query (which is the default type of authentication). To learn more about using credentials in a serverless SQL pool, see the [**Control storage account access for serverless SQL pool in Azure Synapse Analytics**](https://learn.microsoft.com/en-us/azure/synapse-analytics/sql/develop-storage-files-storage-access-control) article in Azure Synapse Analytics documentation.

## **Creating an external file format**

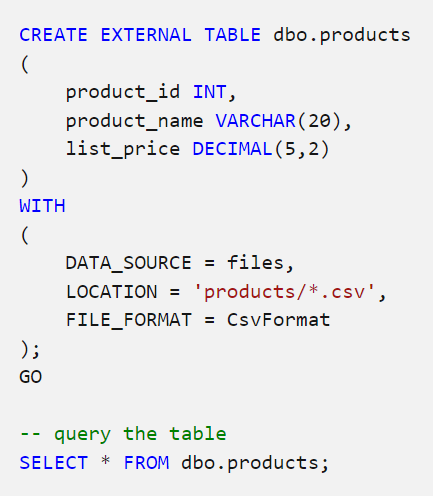
While an external data source simplifies the code needed to access files with the OPENROWSET function, you still need to provide format details for the file being access; which may include multiple settings for delimited text files. You can encapsulate these settings in an external file format, like this:



After creating file formats for the specific data files you need to work with, you can use the file format to create external tables, as discussed next.

## **Creating an external table**

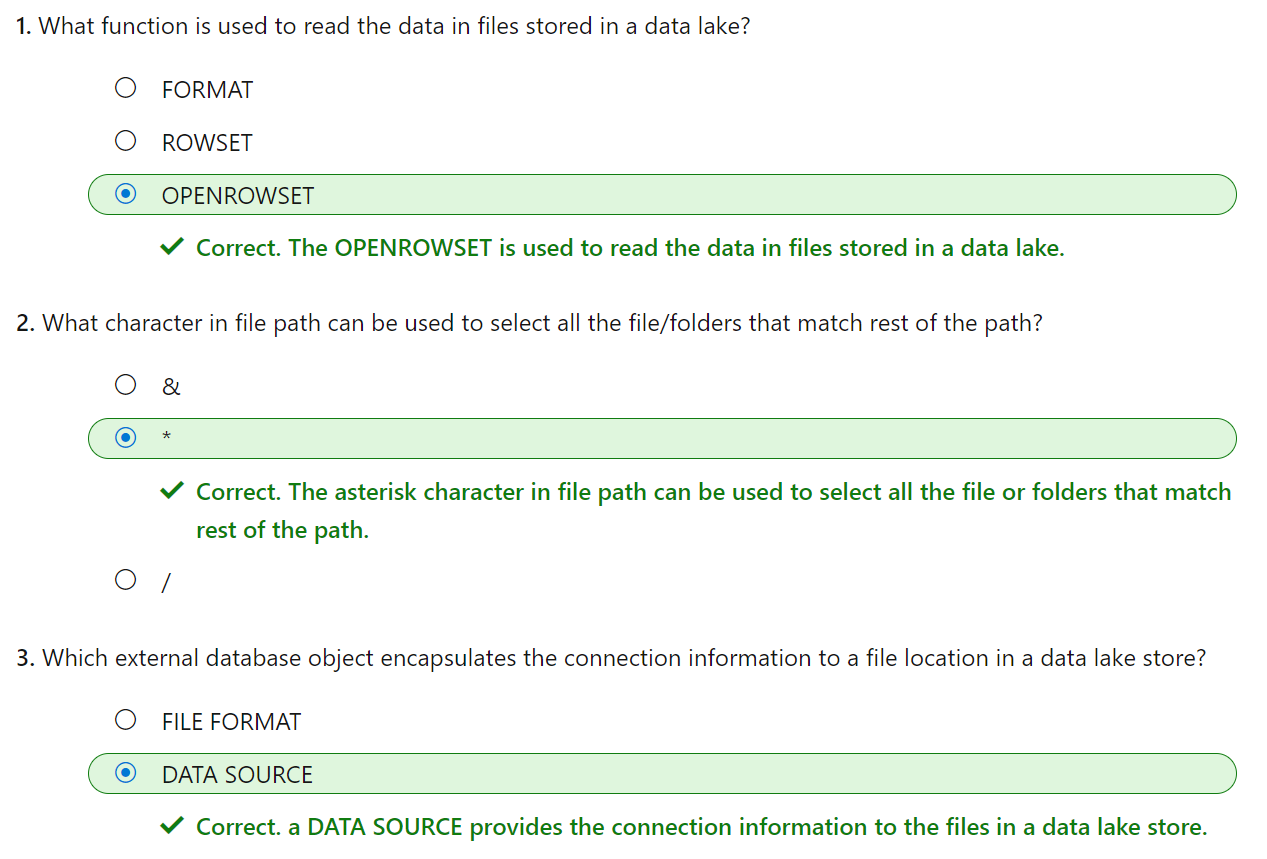
When you need to perform a lot of analysis or reporting from files in the data lake, using the OPENROWSET function can result in complex code that includes data sources and file paths. To simplify access to the data, you can encapsulate the files in an external table; which users and reporting applications can query using a standard SQL SELECT statement just like any other database table. To create an external table, use the CREATE EXTERNAL TABLE statement, specifying the column schema as for a standard table, and including a WITH clause specifying the external data source, relative path, and external file format for your data.



By creating a database that contains the external objects discussed in this unit, you can provide a relational database layer over files in a data lake, making it easier for many data analysts and reporting tools to access the data by using standard SQL query semantics.

# Exercise - Query files using a serverless SQL pool

<https://microsoftlearning.github.io/DP-500-Azure-Data-Analyst/Instructions/labs/01-analyze-data-with-sql.html>



Summary:

Serverless SQL pools enable you to easily query files in data lake. You can query various file formats CSV, JSON, Parquet, and create external database objects to provide a relational abstraction layer over the raw files.

In this module, you've learned how to:

* Identify capabilities and use cases for serverless SQL pools in Azure Synapse Analytics
* Query CSV, JSON, and Parquet files using a serverless SQL pool
* Create external database objects in a serverless SQL pool

[**Use Azure Synapse serverless SQL pools to transform data in a data lake**](https://learn.microsoft.com/en-us/training/modules/use-azure-synapse-serverless-sql-pools-for-transforming-data-lake/)

While SQL is commonly used by data analysts to query data and support analytical and reporting workloads, data engineers often need to use SQL to *transform* data; often as part of a data ingestion pipeline or extract, transform, and load (ETL) process.

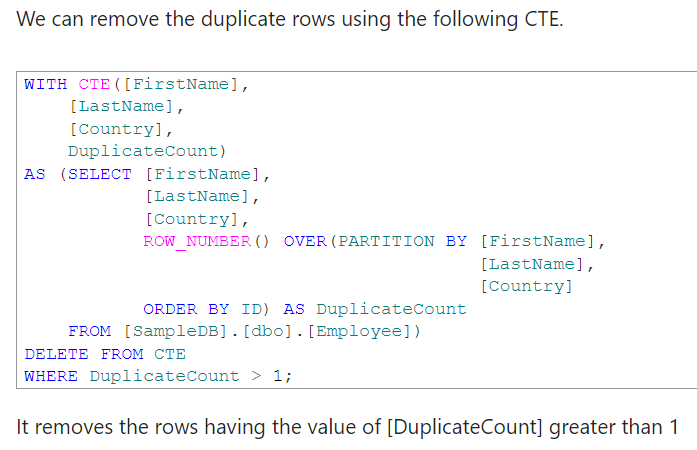
In this module, you'll learn how to use CREATE EXTERNAL TABLE AS SELECT (CETAS) statements to transform data, and store the results in files in a data lake that can be queried through a relational table in a serverless SQL database or processed directly from the file system.

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

* Use a CREATE EXTERNAL TABLE AS SELECT (CETAS) statement to transform data.
* Encapsulate a CETAS statement in a stored procedure.
* Include a data transformation stored procedure in a pipeline.

**NOTE: CTE is different from CET**

CTE 🡪 Common Table Expression



CET 🡪 CREATE EXTERNAL TABLE

@@@@@@@

# Transform data files with the CREATE EXTERNAL TABLE AS SELECT statement

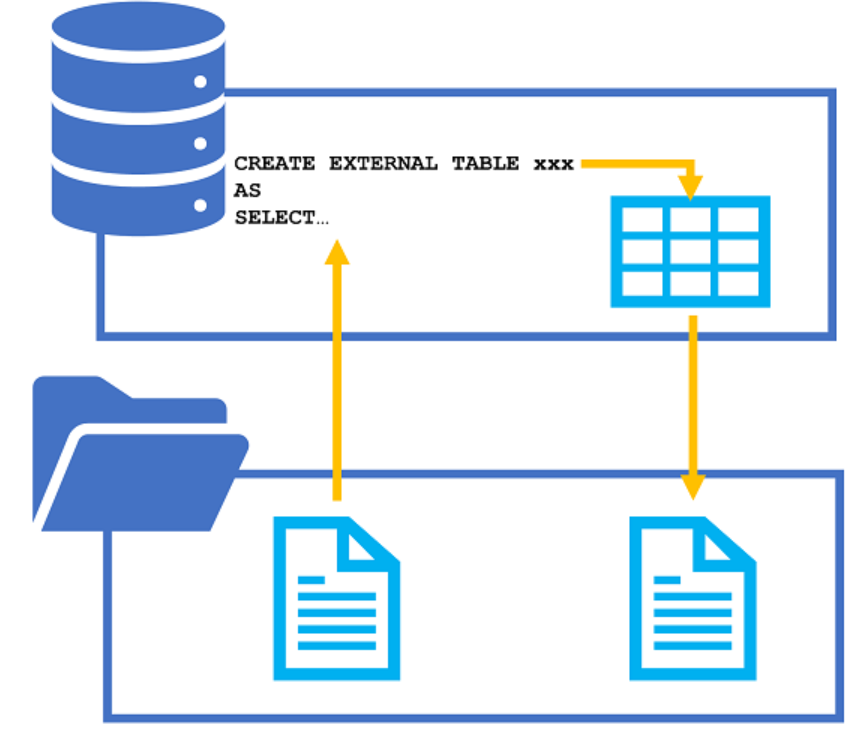
The SQL language includes many features and functions that enable you to manipulate data. For example, you can use SQL to:

* Filter rows and columns in a dataset.
* Rename data fields and convert between data types.
* Calculate derived data fields.
* Manipulate string values.
* Group and aggregate data.

Azure Synapse serverless SQL pools can be used to run SQL statements that transform data and persist the results as a file in a data lake for further processing or querying. If you're familiar with Transact-SQL syntax, you can craft a SELECT statement that applies the specific transformation you're interested in, and store the results of the SELECT statement in a selected file format with a metadata table schema that can be queried using SQL.

You can use a CREATE EXTERNAL TABLE AS SELECT (CETAS) statement in a dedicated SQL pool or serverless SQL pool to persist the results of a query in an external table, which stores its data in a file in the data lake.

The CETAS statement includes a SELECT statement that queries and manipulates data from any valid data source (which could be an existing table or view in a database, or an OPENROWSET function that reads file-based data from the data lake). The results of the SELECT statement are then persisted in an external table, which is a metadata object in a database that provides a relational abstraction over data stored in files. The following diagram illustrates the concept visually:



By applying this technique, you can use SQL to extract and transform data from files or tables, and store the transformed results for downstream processing or analysis. Subsequent operations on the transformed data can be performed against the relational table in the SQL pool database or directly against the underlying data files.

## **Creating external database objects to support CETAS**

To use CETAS expressions, you must create the following types of object in a database for either a serverless or dedicated SQL pool. When using a serverless SQL pool, create these objects in a custom database (created using the CREATE DATABASE statement), not the **built-in** database.

### **External data source**

An external data source encapsulates a connection to a file system location in a data lake. You can then use this connection to specify a relative path in which the data files for the external table created by the CETAS statement are saved.

If the source data for the CETAS statement is in files in the same data lake path, you can use the same external data source in the OPENROWSET function used to query it. Alternatively, you can create a separate external data source for the source files or use a fully qualified file path in the OPENROWSET function.

To create an external data source, use the CREATE EXTERNAL DATA SOURCE statement, as shown in this example:

CREATE EXTERNAL DATA SOURCE files

WITH (

LOCATION = 'https://mydatalake.blob.core.windows.net/data/files/'

);

The previous example assumes that users running queries that use the external data source will have sufficient permissions to access the files. An alternative approach is to encapsulate a credential in the external data source so that it can be used to access file data without granting all users permissions to read it directly:

CREATE DATABASE SCOPED CREDENTIAL storagekeycred

WITH

IDENTITY='SHARED ACCESS SIGNATURE',

SECRET = 'sv=xxx...';

CREATE EXTERNAL DATA SOURCE secureFiles

WITH (

LOCATION = 'https://mydatalake.blob.core.windows.net/data/secureFiles/'

CREDENTIAL = storagekeycred

);

In addition to SAS authentication, you can define credentials that use managed identity (the Azure Active Directory identity used by your Azure Synapse workspace), a specific Azure Active Directory principal, or passthrough authentication based on the identity of the user running the query (which is the default type of authentication). To learn more about using credentials in a serverless SQL pool, see the [**Control storage account access for serverless SQL pool in Azure Synapse Analytics**](https://learn.microsoft.com/en-us/azure/synapse-analytics/sql/develop-storage-files-storage-access-control) article in Azure Synapse Analytics documentation.

### External file format

The CETAS statement creates a table with its data stored in files. You must specify the format of the files you want to create as an external file format.

To create an external file format, use the CREATE EXTERNAL FILE FORMAT statement, as shown in this example:

CREATE EXTERNAL FILE FORMAT ParquetFormat

WITH (

FORMAT\_TYPE = PARQUET,

DATA\_COMPRESSION = 'org.apache.hadoop.io.compress.SnappyCodec'

);

**Tip**

In this example, the files will be saved in Parquet format. You can also create external file formats for other types of file. See [**CREATE EXTERNAL FILE FORMAT (Transact-SQL)**](https://learn.microsoft.com/en-us/sql/t-sql/statements/create-external-file-format-transact-sql) for details.

After creating an external data source and external file format, you can use the CETAS statement to transform data and stored the results in an external table.

For example, suppose the source data you want to transform consists of sales orders in comma-delimited text files that are stored in a folder in a data lake. You want to filter the data to include only orders that are marked as "special order", and save the transformed data as Parquet files in a different folder in the same data lake. You could use the same external data source for both the source and destination folders as shown in this example:

CREATE EXTERNAL TABLE SpecialOrders

WITH (

-- details for storing results

LOCATION = 'special\_orders/',

DATA\_SOURCE = files,

FILE\_FORMAT = ParquetFormat

)

AS

SELECT OrderID, CustomerName, OrderTotal

FROM

OPENROWSET(

-- details for reading source files

BULK 'sales\_orders/\*.csv',

DATA\_SOURCE = 'files',

FORMAT = 'CSV',

PARSER\_VERSION = '2.0',

HEADER\_ROW = TRUE

) AS source\_data

WHERE OrderType = 'Special Order';

The LOCATION and BULK parameters in the previous example are relative paths for the results and source files respectively. The paths are relative to the file system location referenced by the **files** external data source.

An important point to understand is that you ***must*** use an external data source to specify the location where the transformed data for the external table is to be saved. When file-based source data is stored in the same folder hierarchy, you can use the same external data source. Otherwise, you can use a second data source to define a connection to the source data or use the fully qualified path, as shown in this example:

CREATE EXTERNAL TABLE SpecialOrders

WITH (

-- details for storing results

LOCATION = 'special\_orders/',

DATA\_SOURCE = files,

FILE\_FORMAT = ParquetFormat

)

AS

SELECT OrderID, CustomerName, OrderTotal

FROM

OPENROWSET(

-- details for reading source files

BULK 'https://mystorage.blob.core.windows.net/data/sales\_orders/\*.csv',

FORMAT = 'CSV',

PARSER\_VERSION = '2.0',

HEADER\_ROW = TRUE

) AS source\_data

WHERE OrderType = 'Special Order';

## Dropping external tables

If you no longer need the external table containing the transformed data, you can drop it from the database my using the DROP EXTERNAL TABLE statement, as shown here:

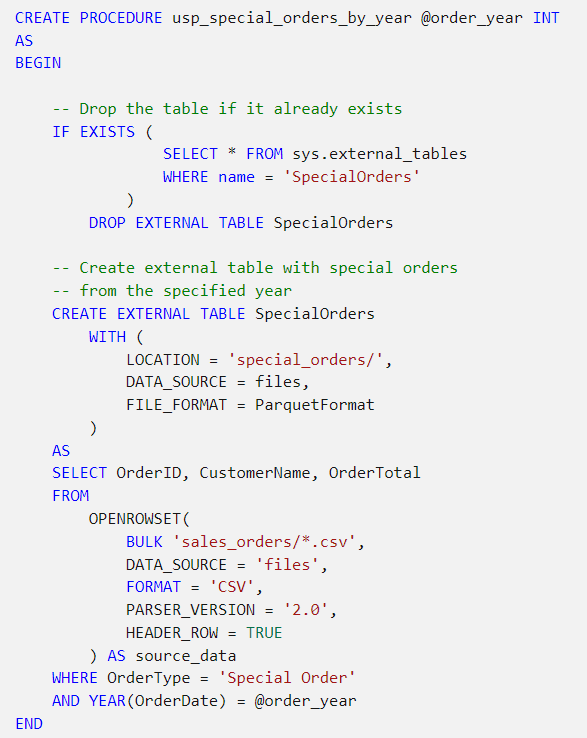
DROP EXTERNAL TABLE SpecialOrders;

However, it's important to understand that external tables are a metadata abstraction over the files that contain the actual data. Dropping an external table does ***not*** delete the underlying files.

# Encapsulate data transformations in a stored procedure

While you can run a CREATE EXTERNAL TABLE AS SELECT (CETAS) statement in a script whenever you need to transform data, it's good practice to encapsulate the transformation operation in stored procedure. This approach can make it easier to operationalize data transformations by enabling you to supply parameters, retrieve outputs, and include additional logic in a single procedure call.

For example, the following code creates a stored procedure that drops the external table if it already exists before recreating it with order data for the specified year:



**Note**

As discussed previously, dropping an existing external table does not delete the folder containing its data files. You must explicitly delete the target folder if it exists before running the stored procedure, or an error will occur.

In addition to encapsulating Transact-SQL logic, stored procedures also provide the following benefits:

## **Reduces client to server network traffic**

The commands in a procedure are executed as a single batch of code; which can significantly reduce network traffic between the server and client because only the call to execute the procedure is sent across the network.

## **Provides a security boundary**

Multiple users and client programs can perform operations on underlying database objects through a procedure, even if the users and programs don't have direct permissions on those underlying objects. The procedure controls what processes and activities are performed and protects the underlying database objects; eliminating the requirement to grant permissions at the individual object level and simplifies the security layers.

## **Eases maintenance**

Any changes in the logic or file system locations involved in the data transformation can be applied only to the stored procedure; without requiring updates to client applications or other calling functions.

## **Improved performance**

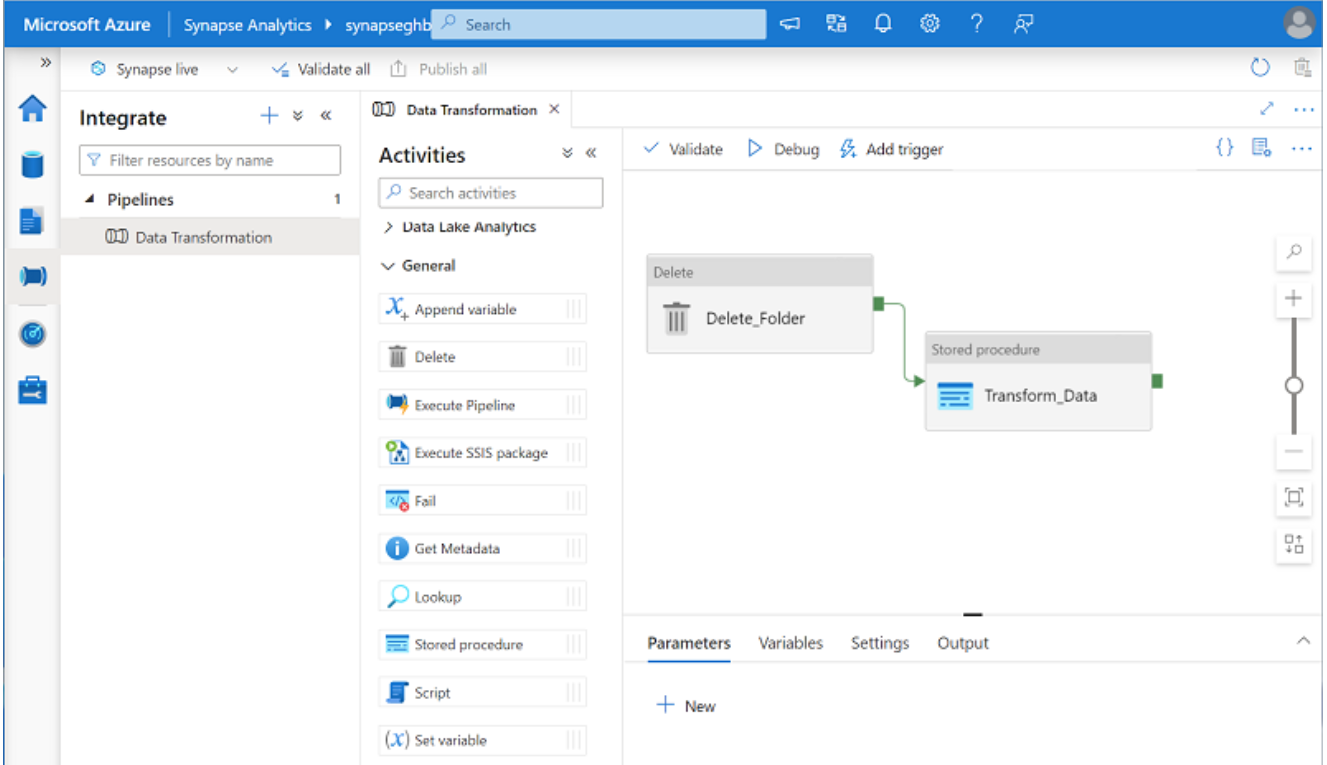
Stored procedures are compiled the first time they're executed, and the resulting execution plan is held in the cache and reused on subsequent runs of the same stored procedure. As a result, it takes less time to process the procedure.

# Include a data transformation stored procedure in a pipeline

Encapsulating a CREATE EXTERNAL TABLE AS SELECT (CETAS) statement in a stored procedure makes it easier for you to operationalize data transformations that you may need to perform repeatedly. In Azure Synapse Analytics and Azure Data Factory, you can create pipelines that connect to *linked services*, including Azure Data Lake Store Gen2 storage accounts that host data lake files, and serverless SQL pools; enabling you to call your stored procedures as part of an overall data extract, transform, and load (ETL) pipeline.

For example, you can create a pipeline that includes the following activities:

* A **Delete** activity that deletes the target folder for the transformed data in the data lake if it already exists.
* A **Stored procedure** activity that connects to your serverless SQL pool and runs the stored procedure that encapsulates your CETAS operation.

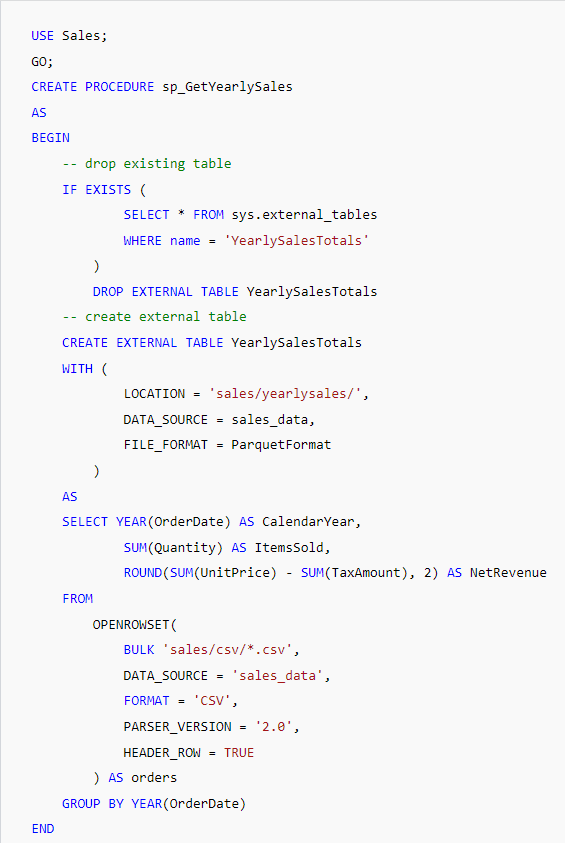


Creating a pipeline for the data transformation enables you to schedule the operation to run at specific times or based on specific events (such as new files being added to the source storage location).

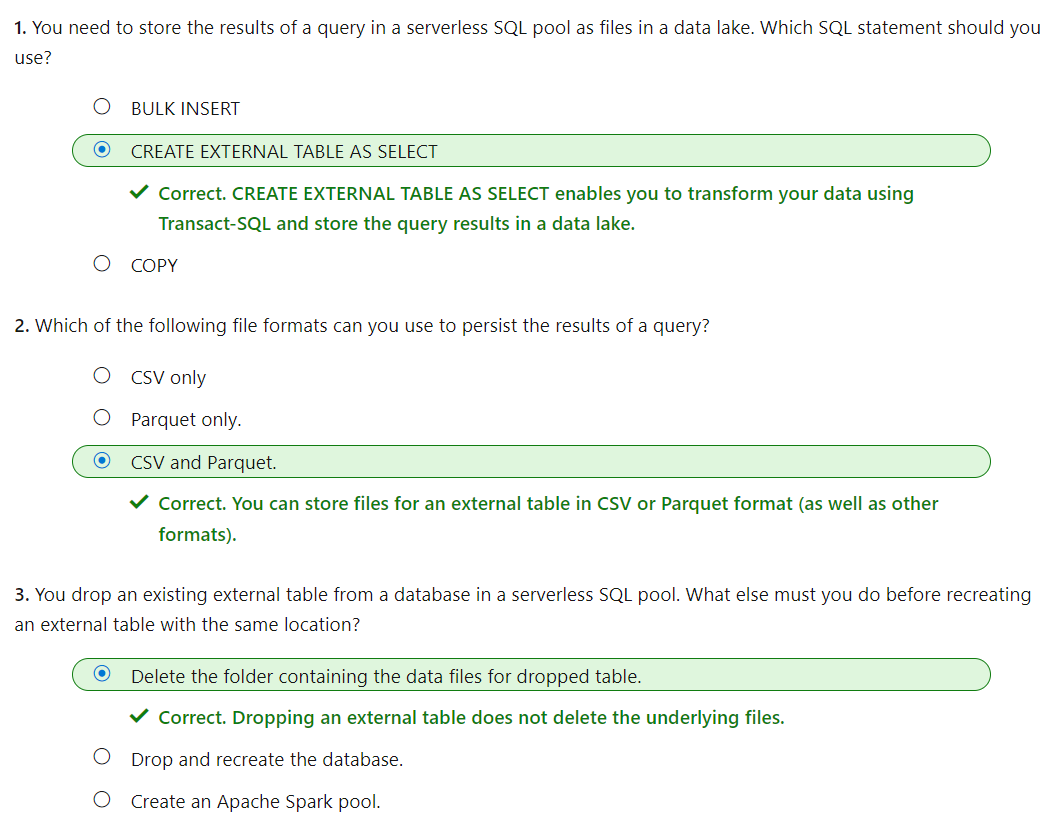
**Tip**

For more information about using the **Stored procedure** activity in a pipeline, see [**Transform data by using the SQL Server Stored Procedure activity in Azure Data Factory or Synapse Analytics**](https://learn.microsoft.com/en-us/azure/data-factory/transform-data-using-stored-procedure) in the Azure Data Factory documentation.

<https://microsoftlearning.github.io/dp-203-azure-data-engineer/Instructions/Labs/03-Transform-data-with-sql.html>



EXEC sp\_GetYearlySales;



# Create a lake database in Azure Synapse Analytics

Data analysts and engineers often find themselves forced to choose between the flexibility of storing data files in a data lake, with the advantages of a structured schema in a relational database. *Lake databases* in Azure Synapse Analytics provide a way to combine these two approaches and benefit from an explicit relational schema of tables, views, and relationships that is decoupled from file-based storage.

In this module, you'll learn how to:

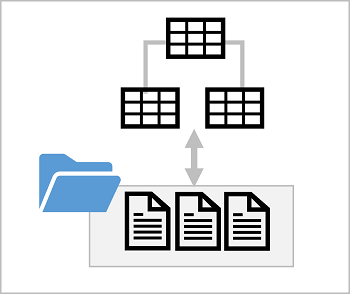
* Understand lake database concepts and components
* Describe database templates in Azure Synapse Analytics
* Create a lake database

# Understand lake database concepts

In a traditional relational database, the database schema is composed of tables, views, and other objects. Tables in a relational database define the entities for which data is stored - for example, a retail database might include tables for products, customers, and orders. Each entity consists of a set of attributes that are defined as columns in the table, and each column has a name and a data type. The data for the tables is stored in the database, and is tightly coupled to the table definition; which enforces data types, nullability, key uniqueness, and referential integrity between related keys. All queries and data manipulations must be performed through the database system.

In a data lake, there is no fixed schema. Data is stored in files, which may be structured, semi-structured, or unstructured. Applications and data analysts can work directly with the files in the data lake using the tools of their choice; without the constraints of a relational database system.

A lake database provides a relational metadata layer over one or more files in a data lake. You can create a lake database that includes definitions for tables, including column names and data types as well as relationships between primary and foreign key columns. The tables reference files in the data lake, enabling you to apply relational semantics to working with the data and querying it using SQL. However, the storage of the data files is decoupled from the database schema; enabling more flexibility than a relational database system typically offers.



## Lake database schema

You can create a lake database in Azure Synapse Analytics, and define the tables that represent the entities for which you need to store data. You can apply proven data modeling principles to create relationships between tables and use appropriate naming conversions for tables, columns, and other database objects.

Azure Synapse Analytics includes a graphical database design interface that you can use to model complex database schema, using many of the same best practices for database design that you would apply to a traditional database.

## Lake database storage

The data for the tables in your lake database is stored in the data lake as Parquet or CSV files. The files can be managed independently of the database tables, making it easier to manage data ingestion and manipulation with a wide variety of data processing tools and technologies.

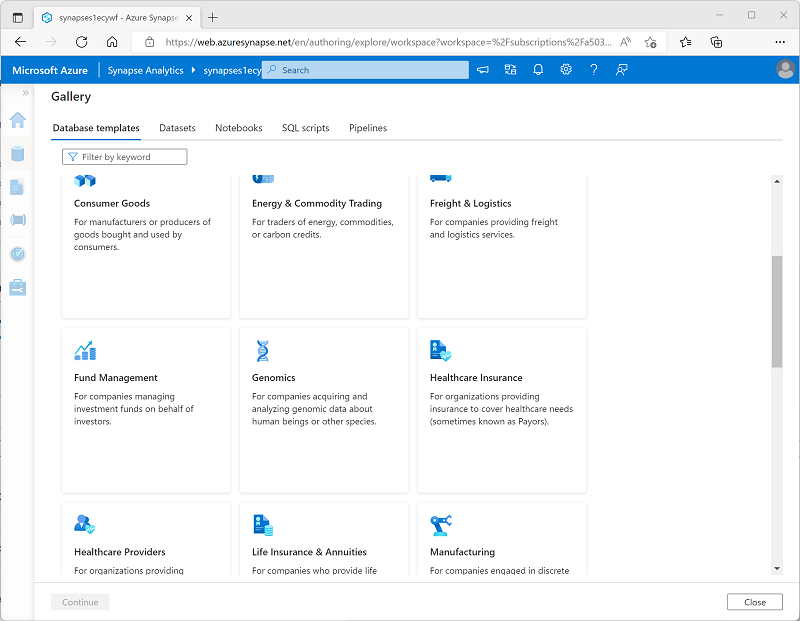
## Lake database compute

To query and manipulate the data through the tables you have defined, you can use an Azure Synapse serverless SQL pool to run SQL queries or an Azure Synapse Apache Spark pool to work with the tables using the Spark SQL API.

# Explore database templates

You can create a Lake database from an empty schema, to which you add definitions for tables and the relationships between them. However, Azure Synapse Analytics provides a comprehensive collection of database templates that reflect common schemas found in multiple business scenarios; including:

* Agriculture
* Automotive
* Banking
* Consumer goods
* Energy and commodity trading
* Freight and logistics
* Fund management
* Healthcare insurance
* Healthcare provider
* Manufacturing
* Retail
* *and many others...*



You can use one of the enterprise database templates as the starting point for creating your lake database, or you can start with a blank schema and add and modify tables from the templates as required.

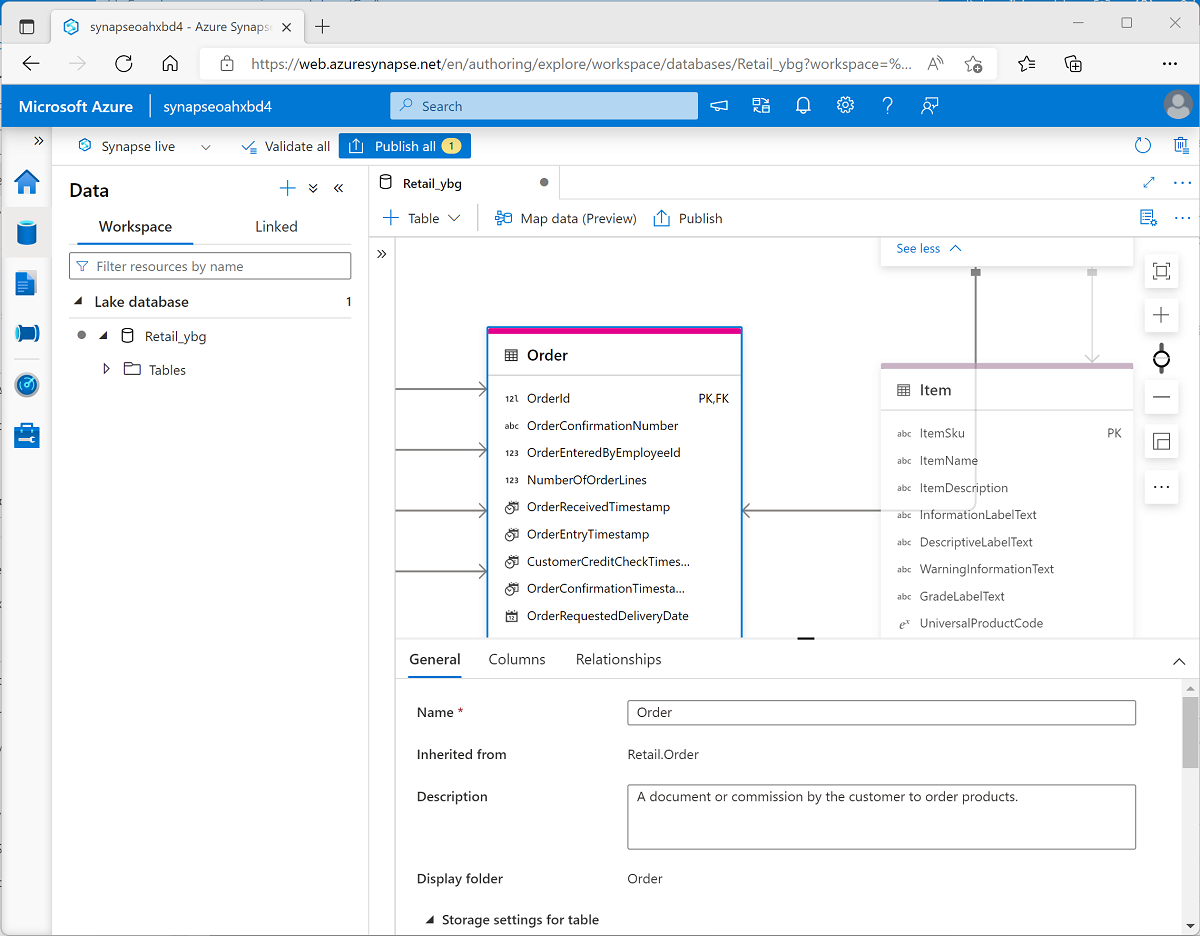
# Create a lake database

You can create a lake database using the lake database designer in Azure Synapse Studio. Start by adding a new lake database on the **Data** page, selecting a template from the gallery or starting with a blank lake database; and then add and customize tables using the visual database designer interface.

As you create each table, you can specify the type and location of the files you want to use to store the underlying data, or you can create a table from existing files that are already in the data lake. In most cases, it's advisable to store all of the database files in a consistent format within the same root folder in the data lake.

## Database designer

The database designer interface in Azure Synapse Studio provides a drag-and-drop surface on which you can edit the tables in your database and the relationships between them.

[](https://learn.microsoft.com/en-us/training/wwl-data-ai/create-metadata-objects-azure-synapse-serverless-sql-pools/media/database-designer.png#lightbox)

Using the database designer, you can define the schema for your database by adding or removing tables and:

* Specifying the name and storage settings for each table.
* Specifying the names, key usage, nullability, and data types for each column.
* Defining relationships between key columns in tables.

When your database schema is ready for use, you can publish the database and start using it.

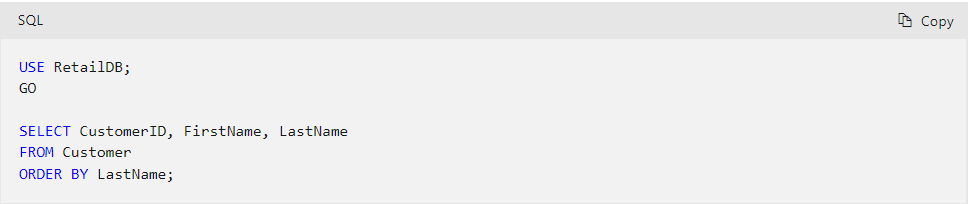
# Use a lake database

After creating a lake database, you can store data files that match the table schemas in the appropriate folders in the data lake, and query them using SQL.

## Using a serverless SQL pool

You can query a lake database in a SQL script by using a serverless SQL pool.

For example, suppose a lake database named **RetailDB** contains an **Customer** table. You could query it using a standard SELECT statement like this:

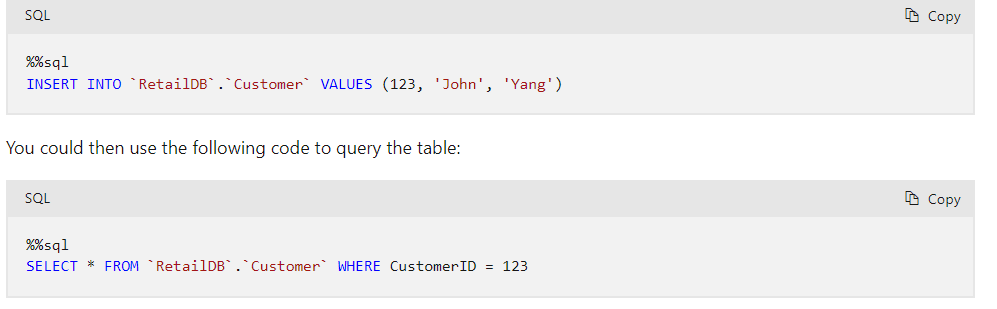


There is no need to use an OPENROWSET function or include any additional code to access the data from the underlying file storage. The serverless SQL pool handles the mapping to the files for you.

## Using an Apache Spark pool

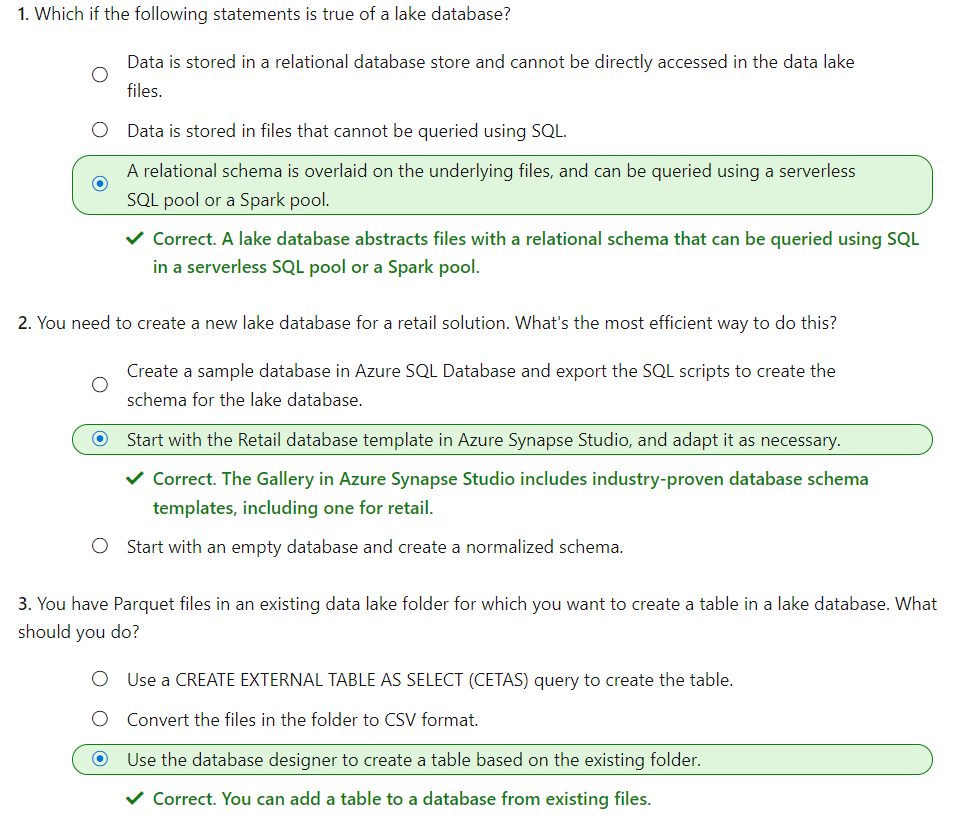
In addition to using a serverless SQL pool, you can work with lake database tables using Spark SQL in an Apache Spark pool.

For example, you could use the following code to insert a new customer record into the **Customer** table.



# Exercise - Analyze data in a lake database

<https://microsoftlearning.github.io/dp-203-azure-data-engineer/Instructions/Labs/04-Create-a-Lake-Database.html>



# Secure data and manage users in Azure Synapse serverless SQL pools

In this lesson, you will learn how you can set up security when using Azure Synapse serverless SQL pools

After the completion of this lesson, you will be able to:

* Choose an authentication method in Azure Synapse serverless SQL pools
* Manage users in Azure Synapse serverless SQL pools
* Manage user permissions in Azure Synapse serverless SQL pools

# Choose an authentication method in Azure Synapse serverless SQL pools

Serverless SQL pool authentication refers to how users prove their identity when connecting to the endpoint. Two types of authentication are supported:

* **SQL Authentication**

This authentication method uses a username and password.

* **Azure Active Directory Authentication**

This authentication method uses identities managed by Azure Active Directory. For Azure AD users, multi-factor authentication can be enabled. Use Active Directory authentication (integrated security) whenever possible.

## Authorization

Authorization refers to what a user can do within a serverless SQL pool database and is controlled by your user account's database role memberships and object-level permissions.

If SQL Authentication is used, the SQL user exists only in the serverless SQL pool and permissions are scoped to the objects in the serverless SQL pool. Access to securable objects in other services (such as Azure Storage) can't be granted to a SQL user directly since it only exists in scope of serverless SQL pool. The SQL user needs get authorization to access the files in the storage account.

If Azure Active Directory authentication is used, a user can sign in to a serverless SQL pool and other services, like Azure Storage, and can grant permissions to the Azure Active Directory user.

## Access to storage accounts

A user that is logged into the serverless SQL pool service must be authorized to access and query the files in Azure Storage. Serverless SQL pool supports the following authorization types:

* Anonymous access

To access publicly available files placed on Azure storage accounts that allow anonymous access.

* Shared access signature (SAS)

Provides delegated access to resources in storage account. With a SAS, you can grant clients access to resources in storage account, without sharing account keys. A SAS gives you granular control over the type of access you grant to clients who have the SAS: validity interval, granted permissions, acceptable IP address range, acceptable protocol (https/http).

* Managed Identity.

Is a feature of Azure Active Directory (Azure AD) that provides Azure services for serverless SQL pool. Also, it deploys an automatically managed identity in Azure AD. This identity can be used to authorize the request for data access in Azure Storage. Before accessing the data, the Azure Storage administrator must grant permissions to Managed Identity for accessing the data. Granting permissions to Managed Identity is done the same way as granting permission to any other Azure AD user.

* User Identity

Also known as "pass-through", is an authorization type where the identity of the Azure AD user that logged into serverless SQL pool is used to authorize access to the data. Before accessing the data, Azure Storage administrator must grant permissions to Azure AD user for accessing the data. This authorization type uses the Azure AD user that logged into serverless SQL pool, therefore it's not supported for SQL user types.

Supported authorization types for database users can be found in the table below:

| **Authorization type** | **SQL user** | **Azure AD user** |
| --- | --- | --- |
| User Identity | Not supported | Supported |
| SAS | Supported | Supported |
| Managed Identity | Not supported | Supported |

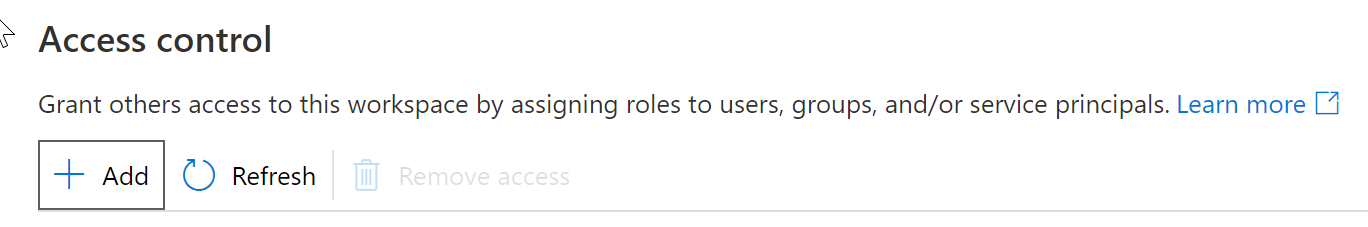
Supported storage and authorization types can be found in the table below:

| **Authorization type** | **Blob Storage** | **ADLS Gen1** | **ADLS Gen2** |
| --- | --- | --- | --- |
| User Identity | Supported - SAS token can be used to access storage that is not protected with firewall | Not supported | Supported - SAS token can be used to access storage that is not protected with firewall |
| SAS | Supported | Supported | Supported |
| Managed Identity | Supported | Supported | Supported |

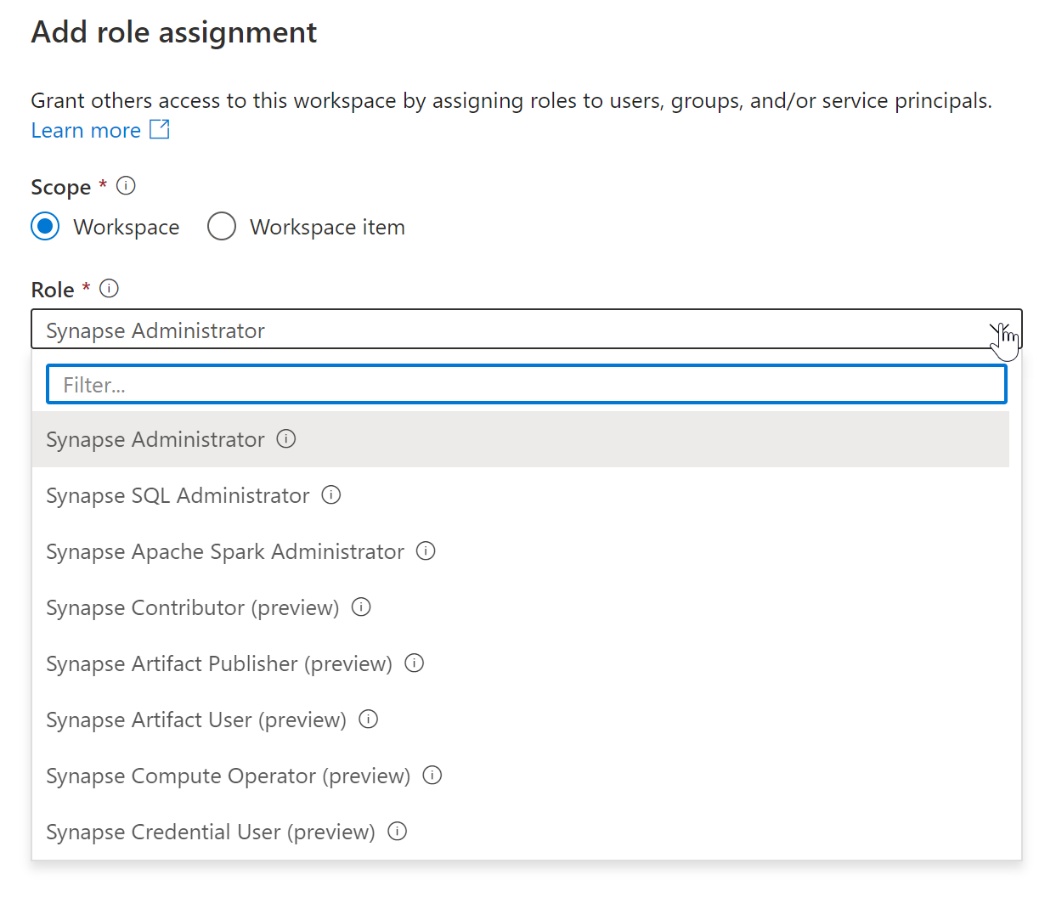
# Manage users in Azure Synapse serverless SQL pools

You can give administrator privileges to a user to Azure Synapse serverless SQL pool. To do this you should open the Azure Synapse workspace and do the following steps:

1. Go to **Manage** menu
2. Go to **Access control**
3. Click on **Add**



1. Choose **Synapse Administrator**

[](https://learn.microsoft.com/en-us/training/wwl-data-ai/secure-data-manage-users-azure-synapse-serverless-sql-pools/media/azure-synapse-workspace-admin.png#lightbox)

1. Select a User or Security group (a security group is the recommended option here)
2. Click **Apply**

Now this user or group is the administrator of the Azure Synapse workspace and serverless SQL pool.

# Manage user permissions in Azure Synapse serverless SQL pools

To secure data, Azure Storage implements an access control model that supports both Azure role-based access control (Azure RBAC) and Portable Operating System Interface for Unix (POSIX) like access control lists (ACLs).

You can associate a security principal with an access level for files and directories. These associations are captured in an access control list (ACL). Each file and directory in your storage account has an access control list. When a security principal attempts an operation on a file or directory, An ACL check determines whether that security principal (user, group, service principal, or managed identity) has the correct permission level to perform the operation.

There are two kinds of access control lists:

* **Access ACLs**

Controls access to an object. Files and directories both have access ACLs.

* **Default ACLs**

Are templates of ACLs associated with a directory that determine the access ACLs for any child items that are created under that directory. Files do not have default ACLs.

Both access ACLs and default ACLs have the same structure.

The permissions on a container object are Read, Write, and Execute, and they can be used on files and directories as shown in the following table:

**Levels of permissions**

| **Permission** | **File** | **Directory** |
| --- | --- | --- |
| Read (R) | Can read the contents of a file | Requires Read and Execute to list the contents of the directory |
| Write (W) | Can write or append to a file | Requires Write and Execute to create child items in a directory |
| Execute (X) | Does not mean anything in the context of Data Lake Storage Gen2 | Required to traverse the child items of a directory |

## Guidelines in setting up ACLs

Always use Azure Active Directory security groups as the assigned principal in an ACL entry. Resist the opportunity to directly assign individual users or service principals. Using this structure will allow you to add and remove users or service principals without the need to reapply ACLs to an entire directory structure. Instead, you can just add or remove users and service principals from the appropriate Azure AD security group.

There are many ways to set up groups. For example, imagine that you have a directory named **/LogData** which holds log data that is generated by your server. Azure Data Factory (ADF) ingests data into that folder. Specific users from the service engineering team will upload logs and manage other users of this folder, and various Databricks clusters will analyze logs from that folder.

To enable these activities, you could create a LogsWriter group and a LogsReader group. Then, you could assign permissions as follows:

* Add the LogsWriter group to the ACL of the **/LogData** directory with rwx permissions.
* Add the LogsReader group to the ACL of the **/LogData** directory with r-x permissions.
* Add the service principal object or Managed Service Identity (MSI) for ADF to the LogsWriters group.
* Add users in the service engineering team to the LogsWriter group.
* Add the service principal object or MSI for Databricks to the LogsReader group.

If a user in the service engineering team leaves the company, you could just remove them from the LogsWriter group. If you did not add that user to a group, but instead, you added a dedicated ACL entry for that user, you would have to remove that ACL entry from the **/LogData** directory. You would also have to remove the entry from all subdirectories and files in the entire directory hierarchy of the **/LogData** directory.

## Roles necessary for serverless SQL pool users

For users which need **read only** access you should assign role named **Storage Blob Data Reader**.

For users which need **read/write** access you should assign role named **Storage Blob Data Contributor**. Read/Write access is needed if user should have access to create external table as select (CETAS).

**Note**

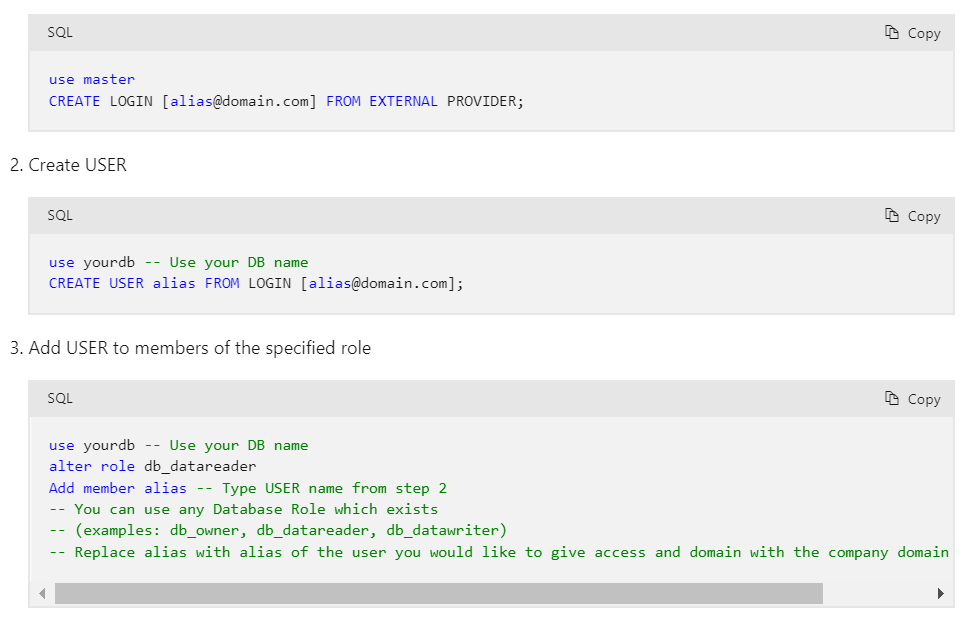
If user has a role Owner or Contributor, that role is not enough. Azure Data Lake Storage gen 2 has super-roles which should be assigned.

## Database level permission

To provide more granular access to the user, you should use Transact-SQL syntax to create logins and users.

To grant access to a user to a single serverless SQL pool database, follow the steps in this example:

1. Create LOGIN



## Server level permission

1. To grant full access to a user to all serverless SQL pool databases, follow the step in this example:



