**Query files using a serverless SQL pool**

Completed100 XP

* 10 minutes

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.

SQLCopy

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:

SQLCopy

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 delimited 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:

textCopy

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)

SQLCopy

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:

Expand table

| **product\_id** | **product\_name** | **list\_price** |
| --- | --- | --- |
| 123 | Widget | 12.9900 |
| 124 | Gadget | 3.9900 |

Now consider the following data:

textCopy

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.

Expand table

| **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:

SQLCopy

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:

Expand table

| **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:

JSONCopy

{

"product\_id": 123,

"product\_name": "Widget",

"list\_price": 12.99

}

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

SQLCopy

SELECT doc

FROM

OPENROWSET(

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

FORMAT = 'csv',

FIELDTERMINATOR ='0x0b',

FIELDQUOTE = '0x0b',

ROWTERMINATOR = '0x0b'

) WITH (doc NVARCHAR(MAX)) as rows

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:

Expand table

| **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:

SQLCopy

SELECT JSON\_VALUE(doc, '$.product\_name') AS product,

JSON\_VALUE(doc, '$.list\_price') AS price

FROM

OPENROWSET(

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

FORMAT = 'csv',

FIELDTERMINATOR ='0x0b',

FIELDQUOTE = '0x0b',

ROWTERMINATOR = '0x0b'

) WITH (doc NVARCHAR(MAX)) as rows

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

Expand table

| **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:

SQLCopy

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:

* /orders
  + /year=2020
    - /month=1
      * /01012020.parquet
      * /02012020.parquet
      * ...
    - /month=2
      * /01022020.parquet
      * /02022020.parquet
      * ...
    - ...
  + /year=2021
    - /month=1
      * /01012021.parquet
      * /02012021.parquet
      * ...
    - ...

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

SQLCopy

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.