# Contents

Learn how to use Azure Data Lake Analytics to run big data analysis jobs that scale to massive data sets. Tutorials and other documentation show you how to create and manage batch, real-time, and interactive analytics jobs, and how to query using the U SQL language.

### Learn about Data Lake Analytics

### Azure Data Lake Analytics Video Library

### Get started with Azure Data Lake Analytics using the Azure portal

## Quickstarts

| | | | | |
|---|---|---|---|---|
| Portal | Visual Studio | Visual Studio Code | PowerShell | Azure CLI |

## Manage Data Lake Analytics

| | | | | |
|---|---|---|---|---|
| Portal | Python | Azure CLI | .NET SDK | Python SDK |

## Reference

### Command-Line

PowerShell

Azure CLI

### Languages

.NET

Node.js

Python

U-SQL

### REST

REST API

# What is Azure Data Lake Analytics?

9/24/2018 • 2 minutes to read • Edit Online

Azure Data Lake Analytics is an on-demand analytics job service that simplifies big data. Instead of deploying, configuring, and tuning hardware, you write queries to transform your data and extract valuable insights. The analytics service can handle jobs of any scale instantly by setting the dial for how much power you need. You only pay for your job when it is running, making it cost-effective.

### Dynamic scaling

Data Lake Analytics dynamically provisions resources and lets you do analytics on terabytes to petabytes of data. You pay only for the processing power used. As you increase or decrease the size of data stored or the amount of compute resources used, you don't have to rewrite code.

### Develop faster, debug, and optimize smarter using familiar tools

Data Lake Analytics deep integrates with Visual Studio. You can use familiar tools to run, debug, and tune your code. Visualizations of your U-SQL jobs let you see how your code runs at scale, so you can easily identify performance bottlenecks and optimize costs.

### U-SQL: simple and familiar, powerful, and extensible

Data Lake Analytics includes U-SQL, a query language that extends the familiar, simple, declarative nature of SQL with the expressive power of C#. The U-SQL language uses the same distributed runtime that powers Microsoft's internal exabyte-scale data lake. SQL and .NET developers can now process and analyze their data with the skills they already have.

### Integrates seamlessly with your IT investments

Data Lake Analytics uses your existing IT investments for identity, management, and security. This approach simplifies data governance and makes it easy to extend your current data applications. Data Lake Analytics is integrated with Active Directory for user management and permissions and comes with built-in monitoring and auditing.

### Affordable and cost effective

Data Lake Analytics is a cost-effective solution for running big data workloads. You pay on a per-job basis when data is processed. No hardware, licenses, or service-specific support agreements are required. The system automatically scales up or down as the job starts and completes, so you never pay for more than what you need. Learn more about controlling costs and saving money.

### Works with all your Azure data

Data Lake Analytics works with Azure Data Lake Store for the highest performance, throughput, and parallelization and works with Azure Storage blobs, Azure SQL Database, Azure Warehouse.

### Next steps

- Get Started with Data Lake Analytics using Azure portal | Azure PowerShell | CLI
- Manage Azure Data Lake Analytics using Azure portal | Azure PowerShell | CLI | Azure .NET SDK | Node.js
- How to control costs and save money with Data Lake Analytics

# Get started with Azure Data Lake Analytics using the Azure portal

9/18/2018 • 2 minutes to read • Edit Online

This article describes how to use the Azure portal to create Azure Data Lake Analytics accounts, define jobs in U-SQL, and submit jobs to the Data Lake Analytics service.

## Prerequisites

Before you begin this tutorial, you must have an **Azure subscription**. See Get Azure free trial.

## Create a Data Lake Analytics account

Now, you will create a Data Lake Analytics and an Azure Data Lake Storage Gen1 account at the same time. This step is simple and only takes about 60 seconds to finish.

1. Sign on to the Azure portal.
2. Click **Create a resource** > **Data + Analytics** > **Data Lake Analytics**.
3. Select values for the following items:
   - **Name**: Name your Data Lake Analytics account (Only lower case letters and numbers allowed).
   - **Subscription**: Choose the Azure subscription used for the Analytics account.
   - **Resource Group**. Select an existing Azure Resource Group or create a new one.
   - **Location**. Select an Azure data center for the Data Lake Analytics account.
   - **Data Lake Storage Gen1**: Follow the instruction to create a new Data Lake Storage Gen1 account, or select an existing one.
4. Optionally, select a pricing tier for your Data Lake Analytics account.
5. Click **Create**.

## Your first U-SQL script

The following text is a very simple U-SQL script. All it does is define a small dataset within the script and then write that dataset out to the default Data Lake Storage Gen1 account as a file called `/data.csv`.

```
@a  =
    SELECT * FROM
        (VALUES
            ("Contoso", 1500.0),
            ("Woodgrove", 2700.0)
        ) AS
              D( customer, amount );
OUTPUT @a
    TO "/data.csv"
    USING Outputters.Csv();
```

## Submit a U-SQL job

1. From the Data Lake Analytics account, select **New Job**.
2. Paste in the text of the preceding U-SQL script. Name the job.
3. Select **Submit** button to start the job.

4. Monitor the **Status** of the job, and wait until the job status changes to **Succeeded**.
5. Select the **Data** tab, then select the **Outputs** tab. Select the output file named `data.csv` and view the output data.

## See also

- To get started developing U-SQL applications, see Develop U-SQL scripts using Data Lake Tools for Visual Studio.
- To learn U-SQL, see Get started with Azure Data Lake Analytics U-SQL language.
- For management tasks, see Manage Azure Data Lake Analytics using Azure portal.

# Develop U-SQL scripts by using Data Lake Tools for Visual Studio

8/27/2018 • 2 minutes to read • Edit Online

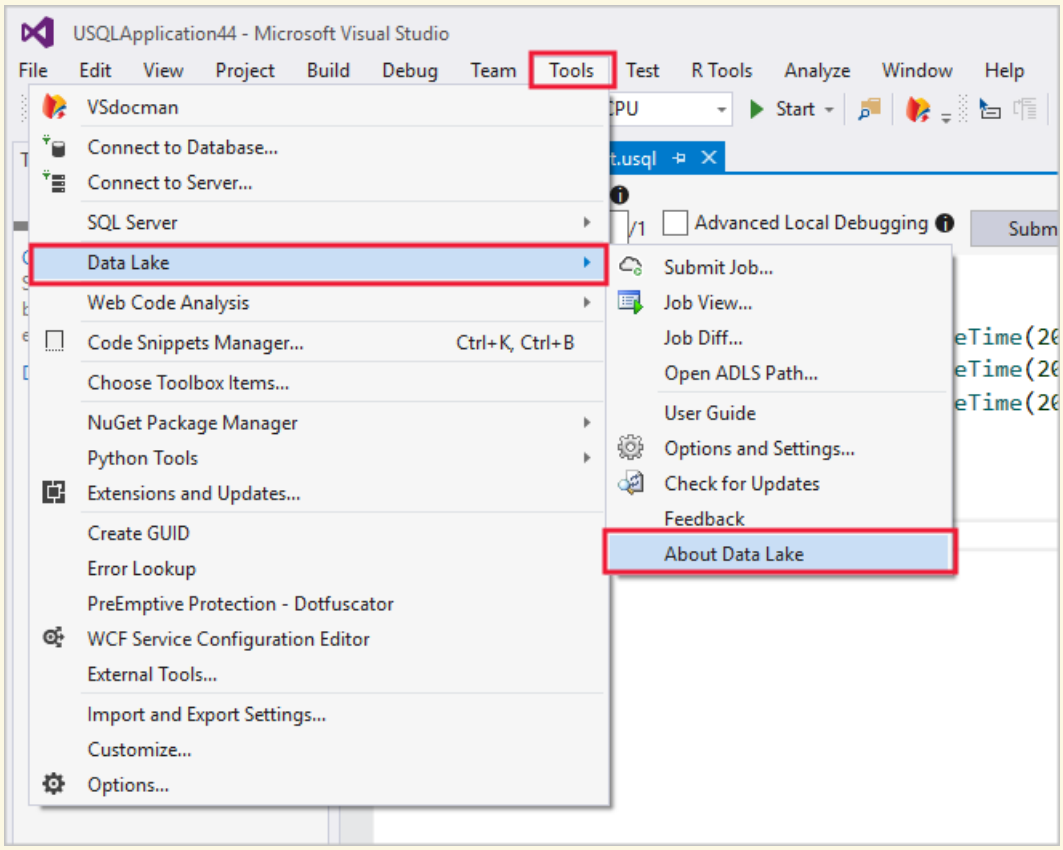
Learn how to use Visual Studio to create Azure Data Lake Analytics accounts, define jobs in U-SQL, and submit jobs to the Data Lake Analytics service. For more information about Data Lake Analytics, see Azure Data Lake Analytics overview.

**IMPORTANT**

Microsoft recommends you upgrade to Azure Data Lake Tools for Visual Studio version 2.3.3000.4 or later. The previous versions are no longer available for download and are now deprecated.

**What do I need to do?**

1. Check if you are using an earlier version than 2.3.3000.4 of Azure Data Lake Tools for Visual Studio.




2. If your version is an earlier version of 2.3.3000.4, update your Azure Data Lake Tools for Visual Studio by visiting the download center:

   - For Visual Studio 2017
   - For Visual Studio 2013 and 2015

## Prerequisites

- **Visual Studio**: All editions except Express are supported.
  - Visual Studio 2017
  - Visual Studio 2015

  - Visual Studio 2013
- **Microsoft Azure SDK for .NET** version 2.7.1 or later. Install it by using the Web platform installer.
- A **Data Lake Analytics** account. To create an account, see Get Started with Azure Data Lake Analytics using Azure portal.

## Install Azure Data Lake Tools for Visual Studio

This tutorial requires that Data Lake Tools for Visual Studio is installed. Follow the installation instructions.

## Connect to an Azure Data Lake Analytics account

1. Open Visual Studio.
2. Open Server Explorer by selecting **View** > **Server Explorer**.
3. Right-click **Azure**. Then select **Connect to Microsoft Azure Subscription** and follow the instructions.
4. In Server Explorer, select **Azure** > **Data Lake Analytics**. You see a list of your Data Lake Analytics accounts.

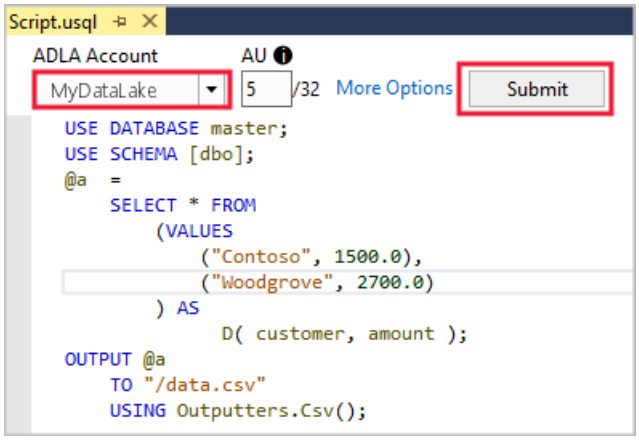
## Write your first U-SQL script

The following text is a simple U-SQL script. It defines a small dataset and writes that dataset to the default Data Lake Store as a file called `/data.csv`.

```
USE DATABASE master;
USE SCHEMA dbo;
@a  =
    SELECT * FROM
        (VALUES
            ("Contoso", 1500.0),
            ("Woodgrove", 2700.0)
        ) AS
              D( customer, amount );
OUTPUT @a
    TO "/data.csv"
    USING Outputters.Csv();
```

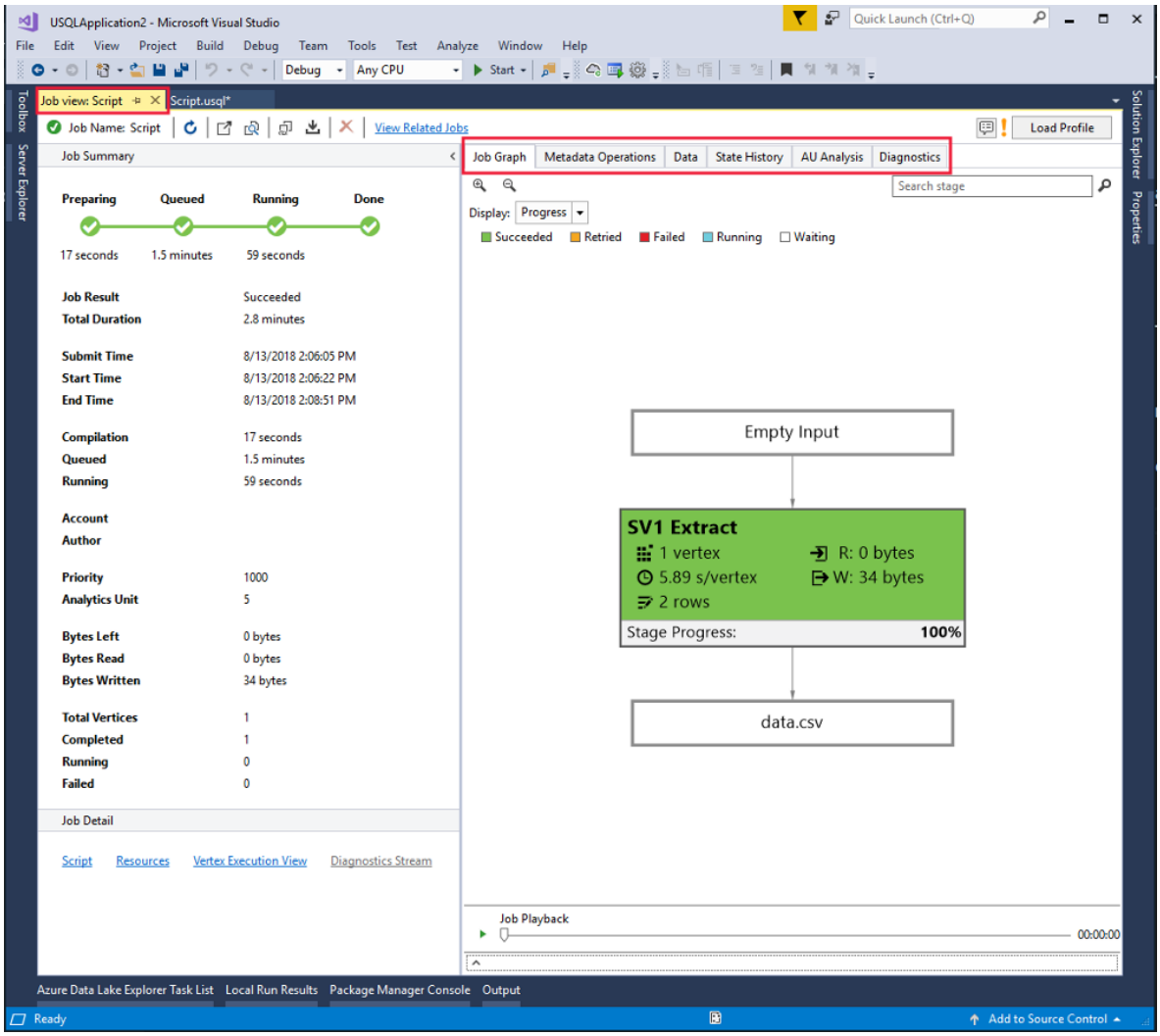
## Submit a Data Lake Analytics job

1. Select **File** > **New** > **Project**.
2. Select the **U-SQL Project** type, and then click **OK**. Visual Studio creates a solution with a **Script.usql** file.
3. Paste the previous script into the **Script.usql** window.
4. In the upper-left corner of the **Script.usql** window, specify the Data Lake Analytics account.




5. In the upper-left corner of the **Script.usql** window, select **Submit**.

6. After the job submission, the **Job view** tab opens to show the job progress. To see the latest job status and refresh the screen, click **Refresh**.




- **Job Summary** shows the summary of the job.
- **Job Graph** visualizes the progress of the job.
- **MetaData Operations** shows all the actions that were taken on the U-SQL catalog.
- **Data** shows all the inputs and outputs.
- **State History** shows the timeline and state details.
- **AU Analysis** shows how many AUs were used in the job and explore simulations of different AUs allocation strategies.
- **Diagnostics** provides an advanced analysis for job execution and performance optimization.

## Check job status

1. In Server Explorer, select **Azure** > **Data Lake Analytics**.
2. Expand the Data Lake Analytics account name.
3. Double-click **Jobs**.
4. Select the job that you previously submitted.

## See the job output

1. In Server Explorer, browse to the job you submitted.
2. Click the **Data** tab.
3. In the **Job Outputs** tab, select the `"/data.csv"` file.

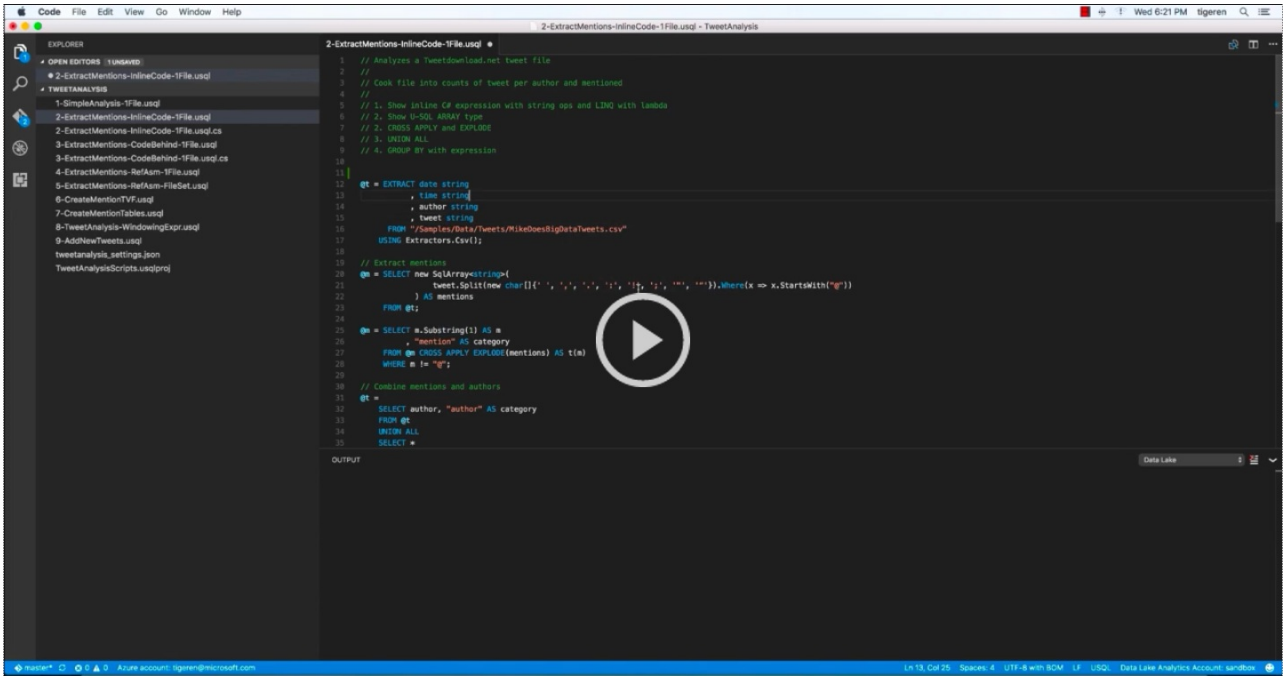
## Next steps

- Run U-SQL scripts on your own workstation for testing and debugging
- Debug C# code in U-SQL jobs using Azure Data Lake Tools for Visual Studio Code
- Use the Azure Data Lake Tools for Visual Studio Code

# Use Azure Data Lake Tools for Visual Studio Code

9/14/2018 • 14 minutes to read • Edit Online

In this article, learn how you can use Azure Data Lake Tools for Visual Studio Code (VS Code) to create, test, and run U-SQL scripts. The information is also covered in the following video:



## Prerequisites

Azure Data Lake Tools for VS Code supports Windows, Linux, and macOS. U-SQL local run and local debug works only in Windows.

- Visual Studio Code

For MacOS and Linux:

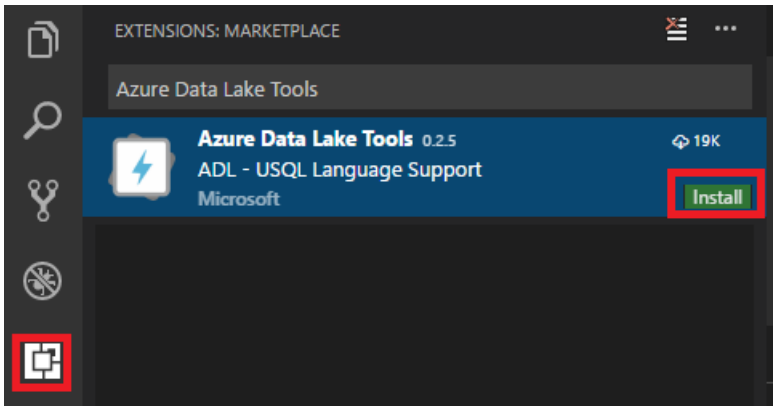
- .NET Core SDK 2.0
- Mono 5.2.x

## Install Azure Data Lake Tools

After you install the prerequisites, you can install Azure Data Lake Tools for VS Code.

**To install Azure Data Lake Tools**

1. Open Visual Studio Code.
2. Select **Extensions** in the left pane. Enter **Azure Data Lake Tools** in the search box.
3. Select **Install** next to **Azure Data Lake Tools**.




After a few seconds, the **Install** button changes to **Reload**.

4. Select **Reload** to activate the **Azure Data Lake Tools** extension.
5. Select **Reload Window** to confirm. You can see **Azure Data Lake Tools** in the **Extensions** pane.

## Activate Azure Data Lake Tools

Create a .usql file or open an existing .usql file to activate the extension.

## Work with U-SQL

To work with U-SQL, you need open either a U-SQL file or a folder.

**To open the sample script**

Open the command palette (Ctrl+Shift+P) and enter **ADL: Open Sample Script**. It opens another instance of this sample. You can also edit, configure, and submit a script on this instance.

**To open a folder for your U-SQL project**

1. From Visual Studio Code, select the **File** menu, and then select **Open Folder**.
2. Specify a folder, and then select **Select Folder**.
3. Select the **File** menu, and then select **New**. An Untitled-1 file is added to the project.
4. Enter the following code in the Untitled-1 file:

```
@departments  =
    SELECT * FROM
        (VALUES
            (31,    "Sales"),
            (33,    "Engineering"),
            (34,    "Clerical"),
            (35,    "Marketing")
        ) AS
              D( DepID, DepName );

OUTPUT @departments
    TO "/Output/departments.csv"
USING Outputters.Csv();
```

The script creates a departments.csv file with some data included in the /output folder.

5. Save the file as **myUSQL.usql** in the opened folder.

**To compile a U-SQL script**

1. Select Ctrl+Shift+P to open the command palette.
2. Enter **ADL: Compile Script**. The compile results appear in the **Output** window. You can also right-click a script

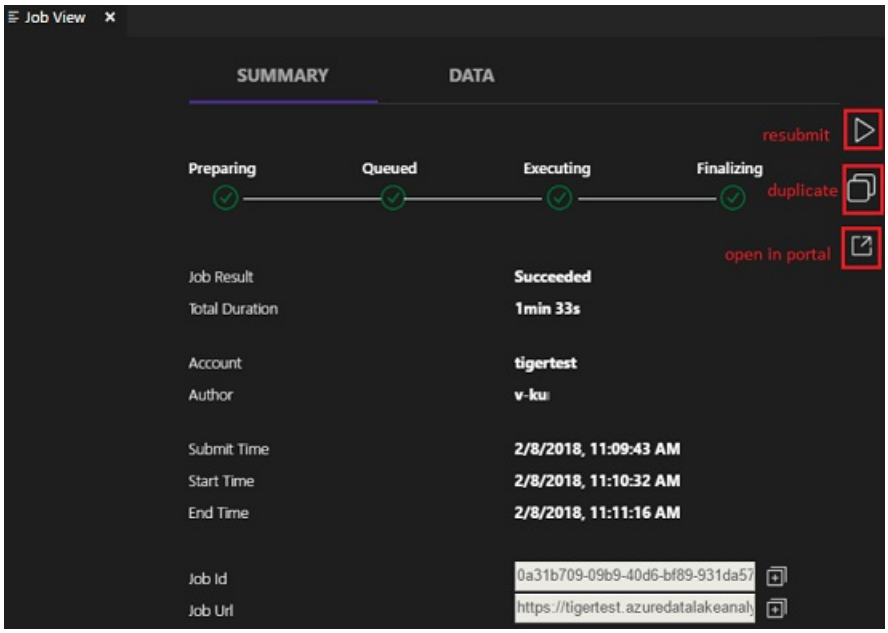
file, and then select **ADL: Compile Script** to compile a U-SQL job. The compilation result appears in the **Output** pane.

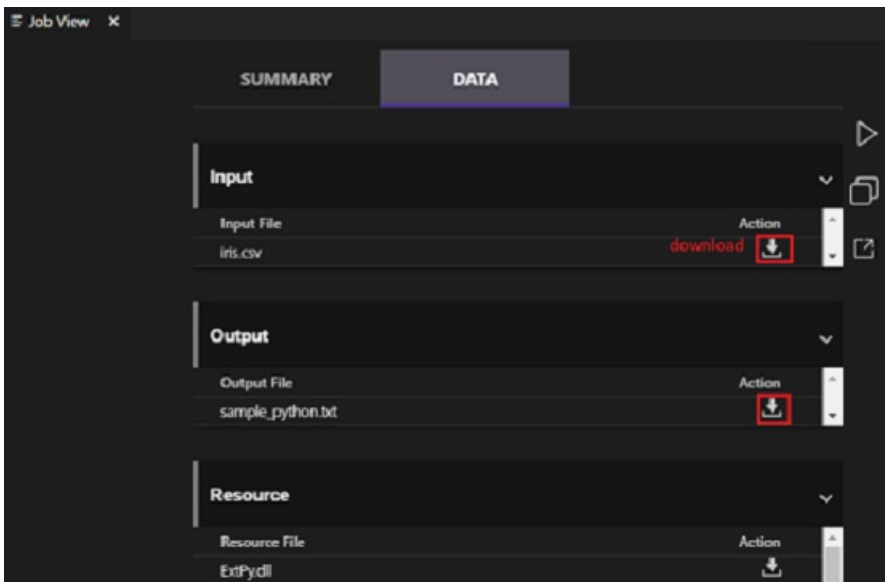
**To submit a U-SQL script**

1. Select Ctrl+Shift+P to open the command palette.
2. Enter **ADL: Submit Job**. You can also right-click a script file, and then select **ADL: Submit Job**.

After you submit a U-SQL job, the submission logs appear in the **Output** window in VS Code. The job view appears in the right pane. If the submission is successful, the job URL appears too. You can open the job URL in a web browser to track the real-time job status.

On the job view's **SUMMARY** tab, you can see the job details. Main functions include resubmit a script, duplicate a script, and open in the portal. On the job view's **DATA** tab, you can refer to the input files, output files, and resource files. Files can be downloaded to the local computer.



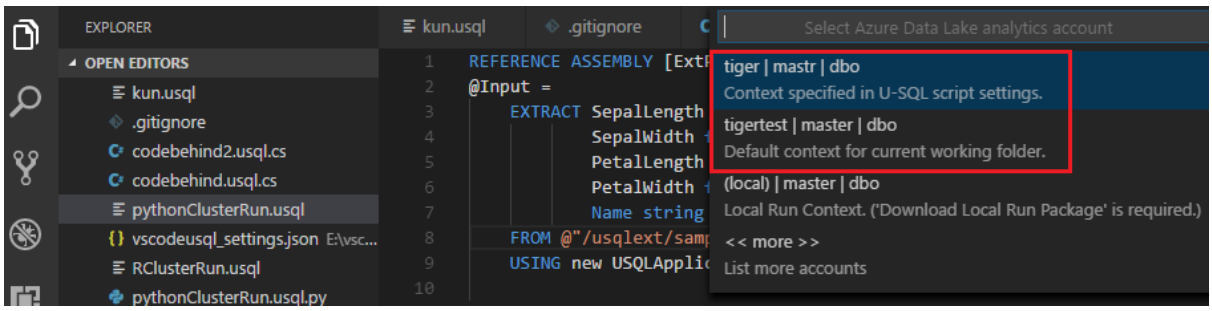




**To set the default context**

You can set the default context to apply this setting to all script files if you have not set parameters for files individually.

1. Select Ctrl+Shift+P to open the command palette.
2. Enter **ADL: Set Default Context**. Or right-click the script editor and select **ADL: Set Default Context**.

3. Choose the account, database, and schema that you want. The setting is saved to the xxx_settings.json configuration file.




**To set script parameters**

1. Select Ctrl+Shift+P to open the command palette.
2. Enter **ADL: Set Script Parameters**.
3. The xxx_settings.json file is opened with the following properties:

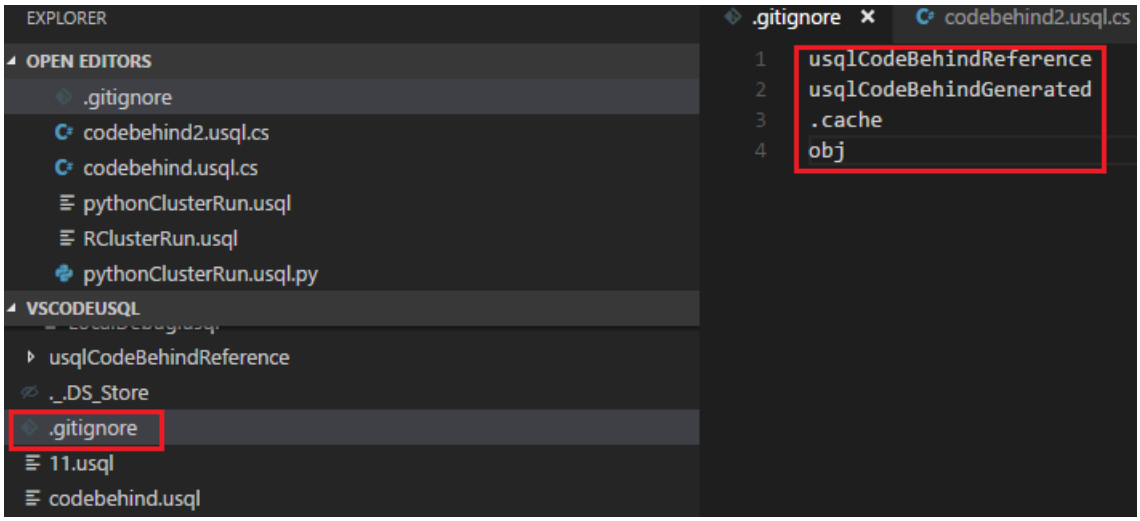
- **account**: An Azure Data Lake Analytics account under your Azure subscription that's needed to compile and run U-SQL jobs. You need configure the computer account before you compile and run U-SQL jobs.
- **database**: A database under your account. The default is **master**.
- **schema**: A schema under your database. The default is **dbo**.
- **optionalSettings**:
  - **priority**: The priority range is from 1 to 1000, with 1 as the highest priority. The default value is **1000**.
  - **degreeOfParallelism**: The parallelism range is from 1 to 150. The default value is the maximum parallelism allowed in your Azure Data Lake Analytics account.

> **NOTE**
> After you save the configuration, the account, database, and schema information appear on the status bar at the lower-left corner of the corresponding .usql file if you don't have a default context set up.

**To set Git ignore**

1. Select Ctrl+Shift+P to open the command palette.
2. Enter **ADL: Set Git Ignore**.

   - If you don't have a **.gitIgnore** file in your VS Code working folder, a file named **.gitIgnore** is created in your folder. Four items (**usqlCodeBehindReference**, **usqlCodeBehindGenerated**, **.cache**, **obj**) are added in the file by default. You can make more updates if needed.
   - If you already have a **.gitIgnore** file in your VS Code working folder, the tool adds four items (**usqlCodeBehindReference**, **usqlCodeBehindGenerated**, **.cache**, **obj**) in your **.gitIgnore** file if the four items were not included in the file.




## Work with code-behind files: C Sharp, Python, and R

Azure Data Lake Tools supports multiple custom codes. For instructions, see Develop U-SQL with Python, R, and C Sharp for Azure Data Lake Analytics in VS Code.

## Work with assemblies

For information on developing assemblies, see Develop U-SQL assemblies for Azure Data Lake Analytics jobs.

You can use Data Lake Tools to register custom code assemblies in the Data Lake Analytics catalog.

**To register an assembly**

You can register the assembly through the **ADL: Register Assembly** or **ADL: Register Assembly (Advanced)** command.

**To register through the ADL: Register Assembly command**

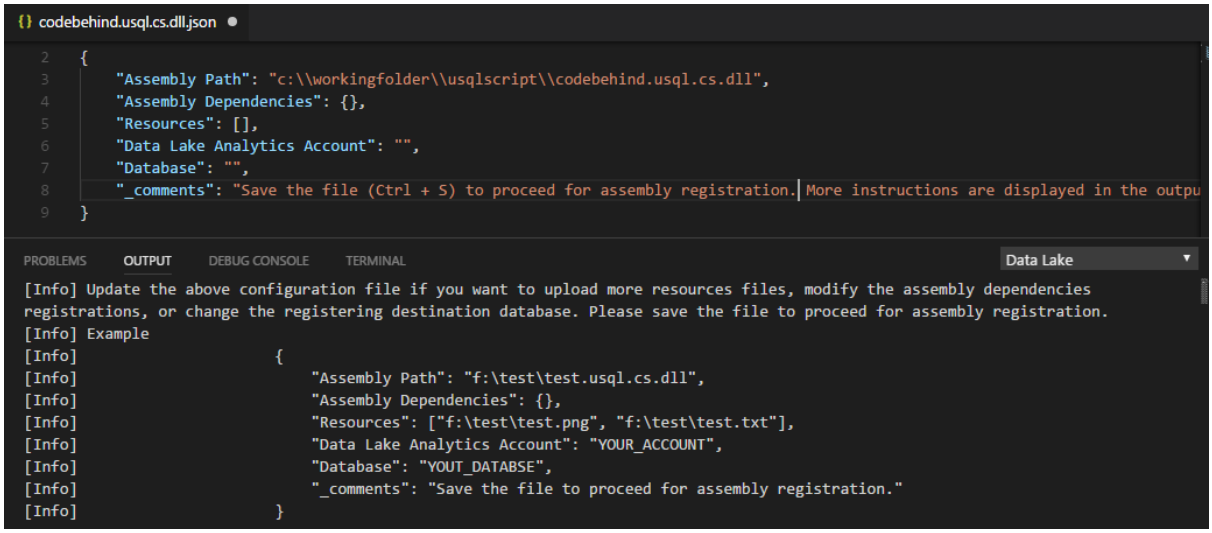
1. Select Ctrl+Shift+P to open the command palette.
2. Enter **ADL: Register Assembly**.
3. Specify the local assembly path.
4. Select a Data Lake Analytics account.
5. Select a database.

The portal is opened in a browser and displays the assembly registration process.

A more convenient way to trigger the **ADL: Register Assembly** command is to right-click the .dll file in File Explorer.

**To register through the ADL: Register Assembly (Advanced) command**

1. Select Ctrl+Shift+P to open the command palette.
2. Enter **ADL: Register Assembly (Advanced)**.
3. Specify the local assembly path.
4. The JSON file is displayed. Review and edit the assembly dependencies and resource parameters, if needed. Instructions are displayed in the **Output** window. To proceed to the assembly registration, save (Ctrl+S) the JSON file.




**NOTE**

- Azure Data Lake Tools autodetects whether the DLL has any assembly dependencies. The dependencies are displayed in the JSON file after they're detected.
- You can upload your DLL resources (for example, .txt, .png, and .csv) as part of the assembly registration.

Another way to trigger the **ADL: Register Assembly (Advanced)** command is to right-click the .dll file in File Explorer.

The following U-SQL code demonstrates how to call an assembly. In the sample, the assembly name is *test*.

```
REFERENCE ASSEMBLY [test];

@a =
    EXTRACT
        Iid int,
    Starts DateTime,
    Region string,
    Query string,
    DwellTime int,
    Results string,
    ClickedUrls string
    FROM @"Sample/SearchLog.txt"
    USING Extractors.Tsv();

@d =
    SELECT DISTINCT Region
    FROM @a;

@d1 =
    PROCESS @d
    PRODUCE
        Region string,
    Mkt string
    USING new USQLApplication_codebehind.MyProcessor();

OUTPUT @d1
    TO @"Sample/SearchLogtest.txt"
    USING Outputters.Tsv();
```

## Use U-SQL local run and local debug for Windows users

U-SQL local run tests your local data and validates your script locally before your code is published to Data Lake Analytics. You can use the local debug feature to complete the following tasks before your code is submitted to Data Lake Analytics:

- Debug your C# code-behind.
- Step through the code.
- Validate your script locally.

The local run and local debug feature only works in Windows environments, and is not supported on macOS and Linux-based operating systems.

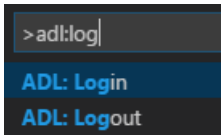
For instructions on local run and local debug, see U-SQL local run and local debug with Visual Studio Code.

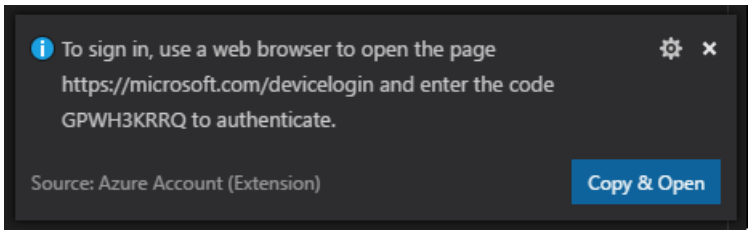
## Connect to Azure

Before you can compile and run U-SQL scripts in Data Lake Analytics, you must connect to your Azure account.

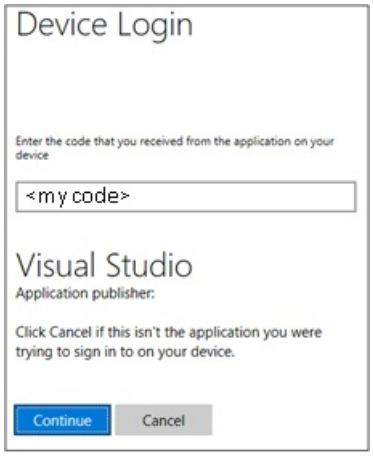
**To connect to Azure by using a command**

1. Select Ctrl+Shift+P to open the command palette.
2. Enter **ADL: Login**. The login information appears on the lower right.






3. Select **Copy & Open** to open the login webpage. Paste the code into the box, and then select **Continue**.



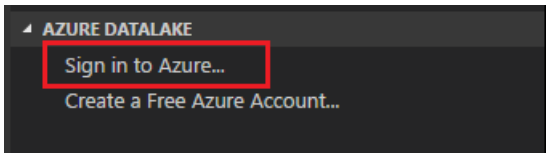

4. Follow the instructions to sign in from the webpage. When you're connected, your Azure account name appears on the status bar in the lower-left corner of the VS Code window.

> **NOTE**
>
> - Data Lake Tools automatically signs you in the next time if you don't sign out.
> - If your account has two factors enabled, we recommend that you use phone authentication rather than using a PIN.

To sign out, enter the command **ADL: Logout**.

**To connect to Azure from the explorer**

Expand **AZURE DATALAKE**, select **Sign in to Azure**, and then follow step 3 and step 4 of To connect to Azure by using a command.



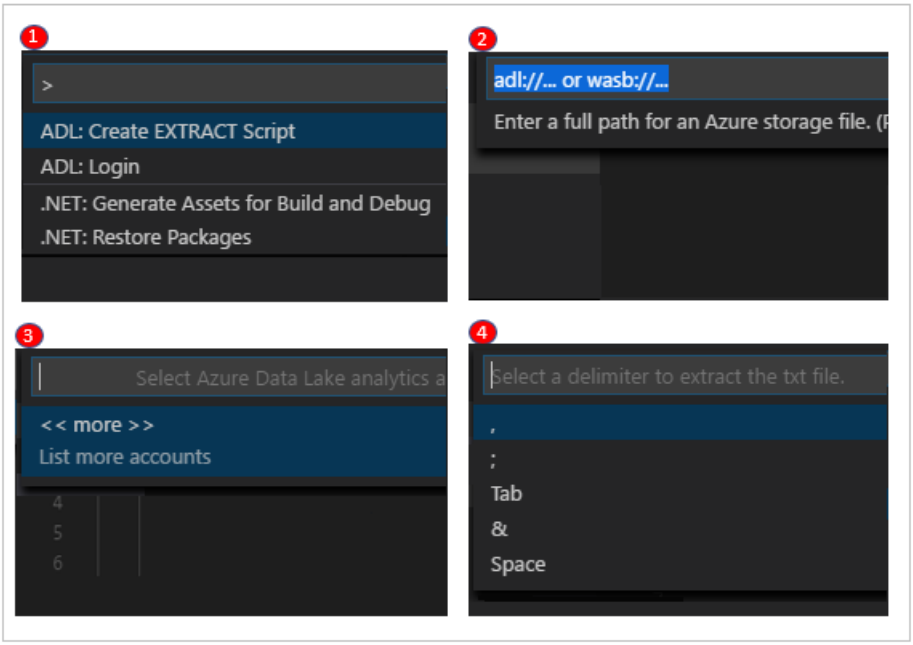

You can't sign out from the explorer. To sign out, see To connect to Azure by using a command.

## Create an extraction script

You can create an extraction script for .csv, .tsv, and .txt files by using the command **ADL: Create EXTRACT Script** or from the Azure Data Lake explorer.

**To create an extraction script by using a command**

1. Select Ctrl+Shift+P to open the command palette, and enter **ADL: Create EXTRACT Script**.
2. Specify the full path for an Azure Storage file, and select the Enter key.
3. Select one account.
4. For a .txt file, select a delimiter to extract the file.




The extraction script is generated based on your entries. For a script that cannot detect the columns, choose one from the two options. If not, only one script will be generated.

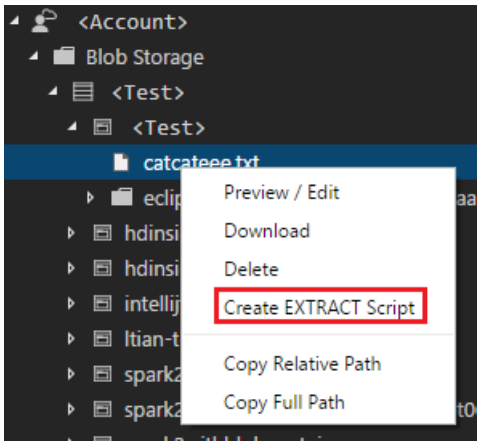
```
/********** File Information **********
        File Name: text.txt
        Modified: Wed May 16 2018 00:04:09 GMT-0700 (Pacific Daylight Time)
        File Size: 51 bytes
        Path: adl://<myaccount>.azuredatalakestore.net/my/text.txt
**************************************/

// Please choose your extract script from the two options below.
// EXTRACT WITH HEADER ROW
@input =
    EXTRACT [1 1 1 1 1 1 1 1] string
    FROM "/my/text.txt"
    USING Extractors.Text(skipFirstNRows:1, delimiter:',', quoting:true);
// For more information on U-SQL extractor parameters, please see https://msdn.mi


//EXTRACT WITHOUT HEADER ROW
@input =
    EXTRACT [Column_0] string
    FROM "/my/text.txt"
    USING Extractors.Text(delimiter:',', quoting:true);
// For more information on U-SQL extractor parameters, please see https://msdn.mi
```

**To create an extraction script from the explorer**

Another way to create the extraction script is through the right-click (shortcut) menu on the .csv, .tsv, or .txt file in Azure Data Lake Store or Azure Blob storage.




## Integrate with Azure Data Lake Analytics through a command

You can access Azure Data Lake Analytics resources to list accounts, access metadata, and view analytics jobs.

**To list the Azure Data Lake Analytics accounts under your Azure subscription**

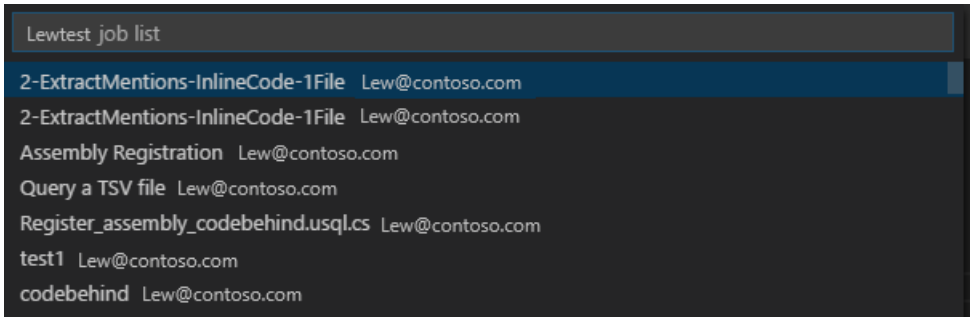
1. Select Ctrl+Shift+P to open the command palette.
2. Enter **ADL: List Accounts**. The accounts appear in the **Output** pane.

**To access Azure Data Lake Analytics metadata**

1. Select Ctrl+Shift+P, and then enter **ADL: List Tables**.
2. Select one of the Data Lake Analytics accounts.
3. Select one of the Data Lake Analytics databases.
4. Select one of the schemas. You can see the list of tables.

**To view Azure Data Lake Analytics jobs**

1. Open the command palette (Ctrl+Shift+P) and select **ADL: Show Jobs**.
2. Select a Data Lake Analytics or local account.
3. Wait for the job list to appear for the account.
4. Select a job from the job list. Data Lake Tools opens the job view in the right pane and displays some information in the VS Code output.




## Integrate with Azure Data Lake Store through a command

You can use Azure Data Lake Store-related commands to:

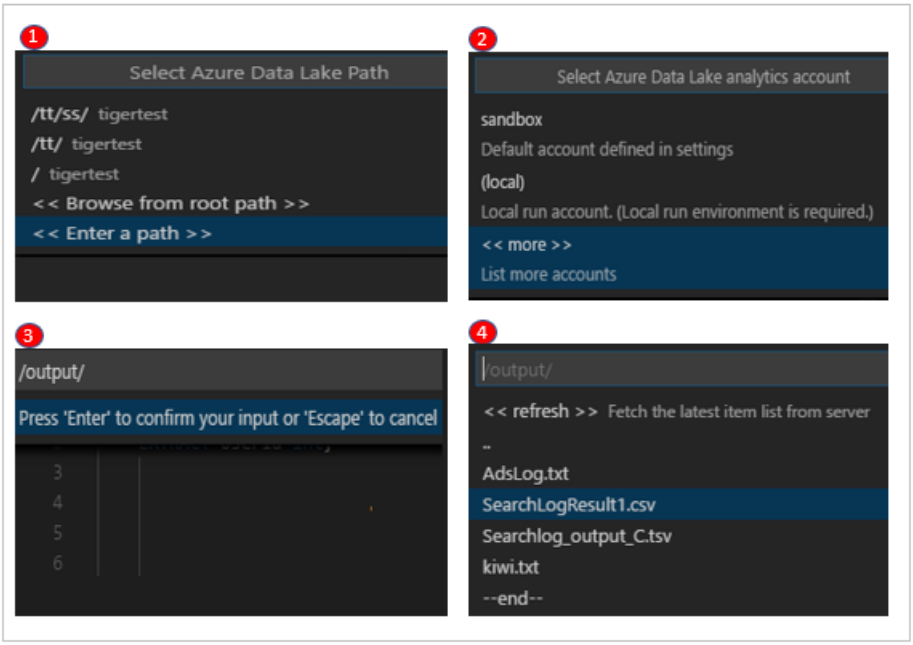
- Browse through the Azure Data Lake Store resources
- Preview the Azure Data Lake Store file
- Upload the file directly to Azure Data Lake Store in VS Code
- Download the file directly from Azure Data Lake Store in VS Code

**List the storage path**

**To list the storage path through the command palette**

1. Right-click the script editor and select **ADL: List Path**.
2. Choose the folder in the list, or select **Enter a path** or **Browse from root path**. (We're using **Enter a path** as an example.)
3. Select your Data Lake Analytics account.
4. Browse to or enter the storage folder path (for example, /output/).

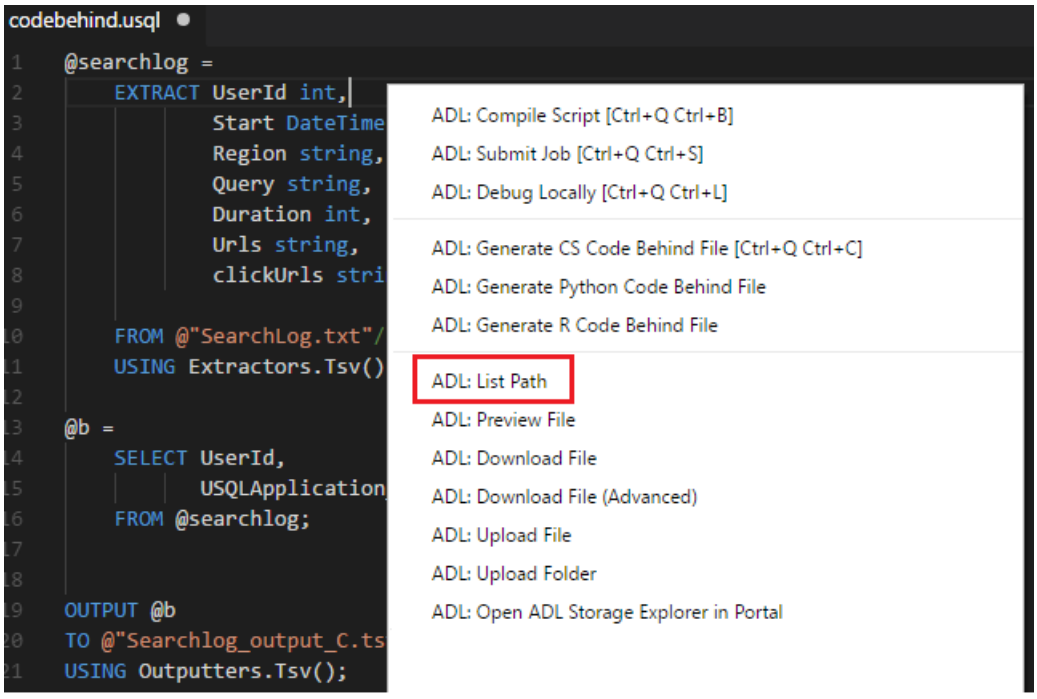
The command palette lists the path information based on your entries.




A more convenient way to list the relative path is through the shortcut menu.

**To list the storage path through the shortcut menu**

Right-click the path string and select **List Path**.

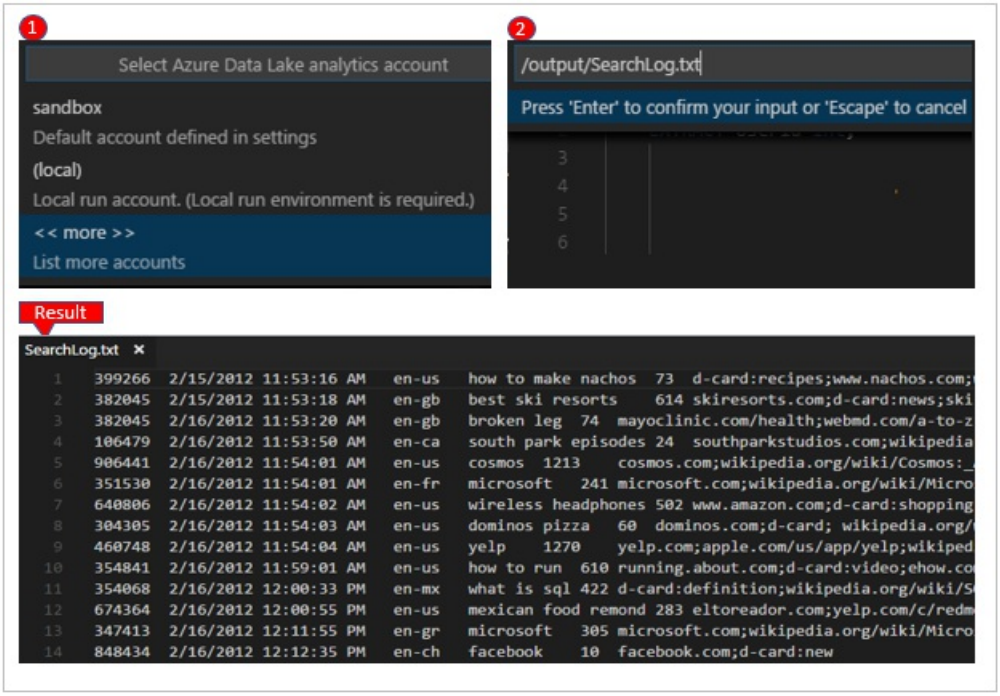



**Preview the storage file**

1. Right-click the script editor and select **ADL: Preview File**.
2. Select your Data Lake Analytics account.

3. Enter an Azure Storage file path (for example, /output/SearchLog.txt).

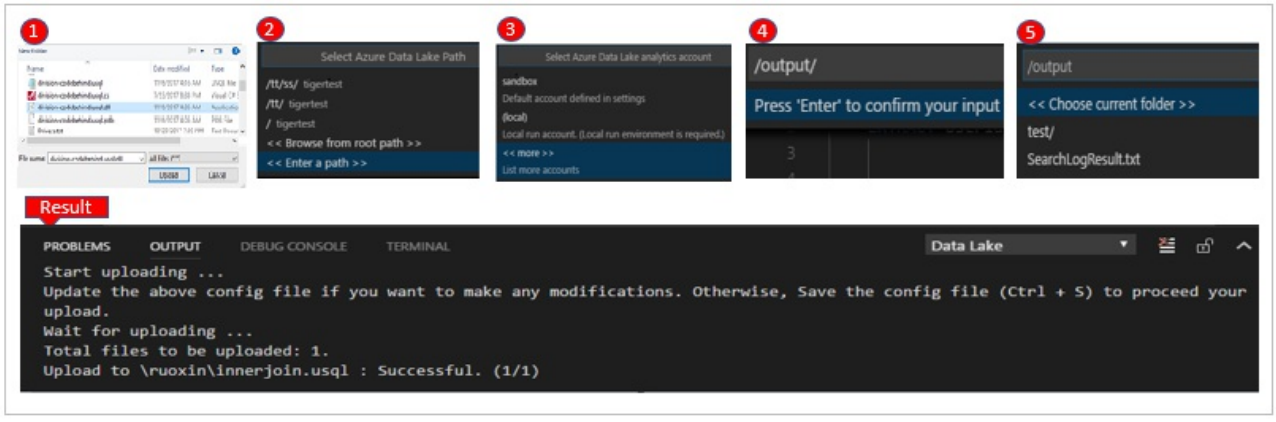
The file opens in VS Code.




Another way to preview the file is through the shortcut menu on the file's full path or the file's relative path in the script editor.

### Upload a file or folder

1. Right-click the script editor and select **Upload File** or **Upload Folder**.
2. Choose one file or multiple files if you selected **Upload File**, or choose the whole folder if you selected **Upload Folder**. Then select **Upload**.
3. Choose the storage folder in the list, or select **Enter a path** or **Browse from root path**. (We're using **Enter a path** as an example.)
4. Select your Data Lake Analytics account.
5. Browse to or enter the storage folder path (for example, /output/).
6. Select **Choose Current Folder** to specify your upload destination.




Another way to upload files to storage is through the shortcut menu on the file's full path or the file's relative path in the script editor.

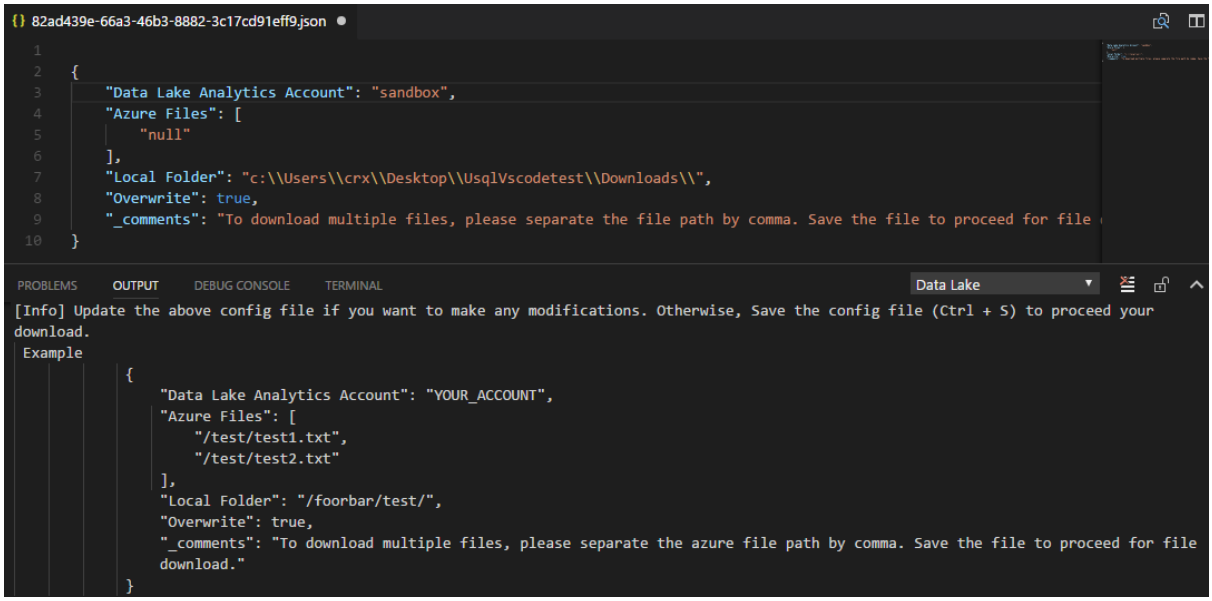
You can monitor the upload status.

### Download a file

You can download a file by using the command **ADL: Download File** or **ADL: Download File (Advanced)**.

**To download a file through the ADL: Download File (Advanced) command**

1. Right-click the script editor, and then select **Download File (Advanced)**.
2. VS Code displays a JSON file. You can enter file paths and download multiple files at the same time. Instructions are displayed in the **Output** window. To proceed to download the file or files, save (Ctrl+S) the JSON file.



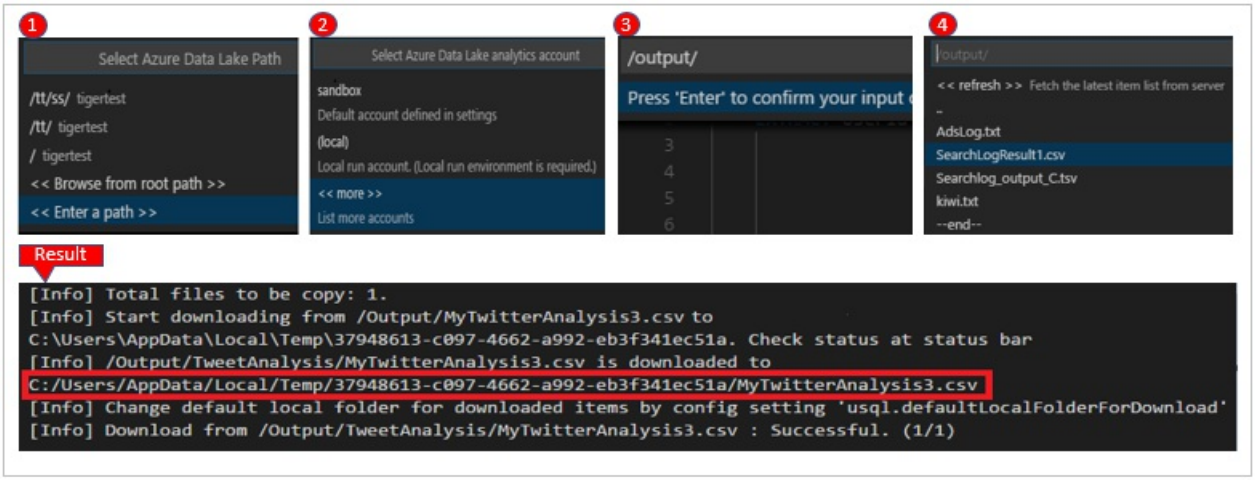

The **Output** window displays the file download status.

```
[Info] Wait for downloading ...
[Info] Total files to be copy: 2.
[Info] /ru/SearchLogResult1.txt is downloaded to
C:/Users/AppData/Local/Temp/8f9d91b6-4750-420a-ba3d-605a7606a33c/SearchLogResult1.txt
[Info] Change default local folder for downloaded items by config setting 'usql.defaultLocalFolderForDownload'
[Info] Download from /ru/SearchLogResult1.txt : Successful. (1/2)
[Info] /ru/SearchLog.txt is downloaded to C:/Users/crx/AppData/Local/Temp/8f9d91b6-4750-420a-ba3d-605a7606a33c/SearchLog.txt
[Info] Change default local folder for downloaded items by config setting 'usql.defaultLocalFolderForDownload'
[Info] Download from /ru/SearchLog.txt : Successful. (2/2)
```

You can monitor the download status.

**To download a file through the ADL: Download File command**

1. Right-click the script editor, select **Download File**, and then select the destination folder from the **Select Folder** dialog box.
2. Choose the folder in the list, or select **Enter a path** or **Browse from root path**. (We're using **Enter a path** as an example.)
3. Select your Data Lake Analytics account.
4. Browse to or enter the storage folder path (for example, /output/), and then choose a file to download.

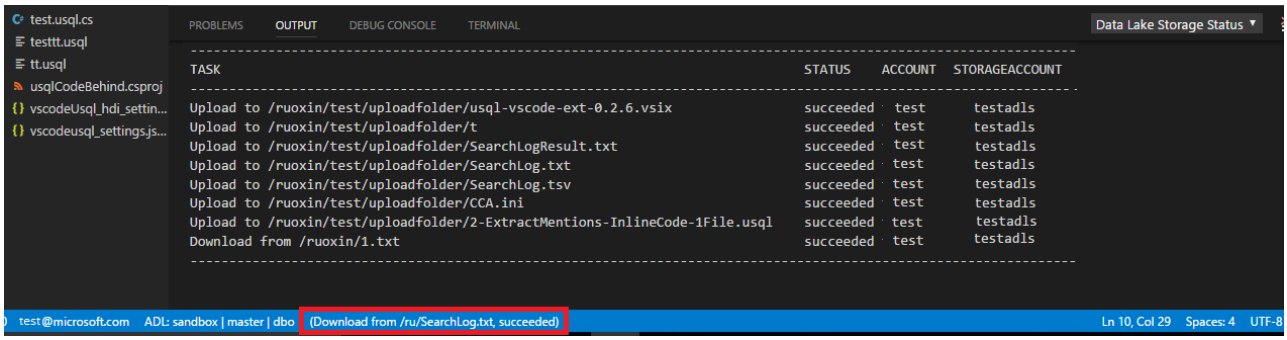



Another way to download storage files is through the shortcut menu on the file's full path or the file's relative path in the script editor.

You can monitor the download status.

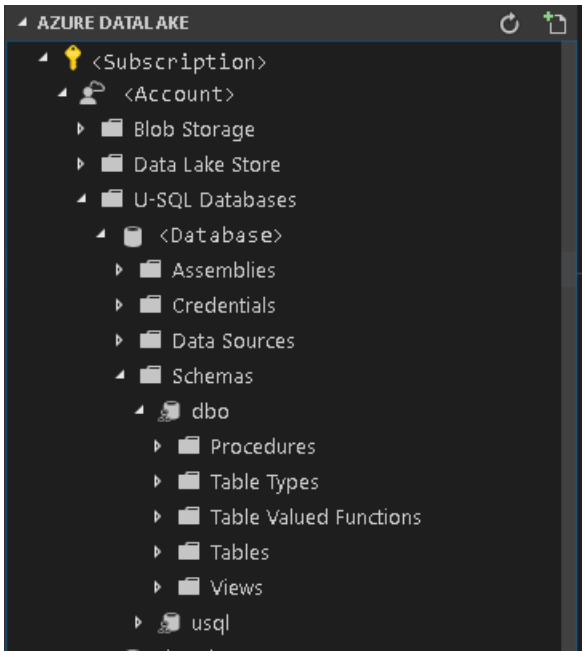
### Check storage tasks' status

The upload and download status appears on the status bar. Select the status bar, and then the status appears on the **OUTPUT** tab.




# Integrate with Azure Data Lake Analytics from the explorer

After you log in, all the subscriptions for your Azure account are listed in the left pane, under **AZURE DATALAKE**.



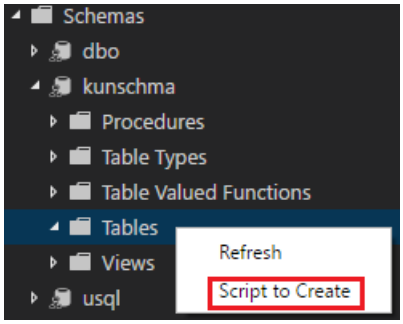

### Data Lake Analytics metadata navigation

Expand your Azure subscription. Under the **U-SQL Databases** node, you can browse through your U-SQL database and view folders like **Schemas**, **Credentials**, **Assemblies**, **Tables**, and **Index**.

#### Data Lake Analytics metadata entity management

Expand **U-SQL Databases**. You can create a database, schema, table, table type, index, or statistic by right-clicking the corresponding node, and then selecting **Script to Create** on the shortcut menu. On the opened script page, edit the script according to your needs. Then submit the job by right-clicking it and selecting **ADL: Submit Job**.

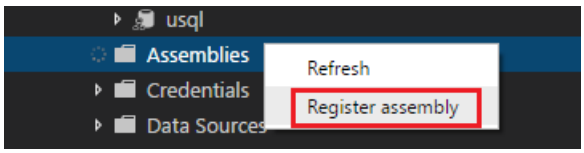
After you finish creating the item, right-click the node and then select **Refresh** to show the item. You can also delete the item by right-clicking it and then selecting **Delete**.





#### Data Lake Analytics assembly registration

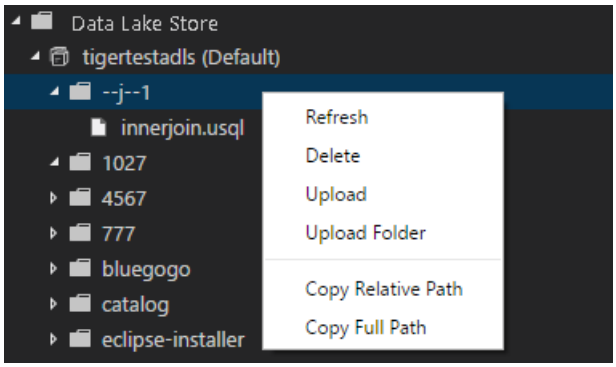
You can register an assembly in the corresponding database by right-clicking the **Assemblies** node, and then selecting **Register assembly**.



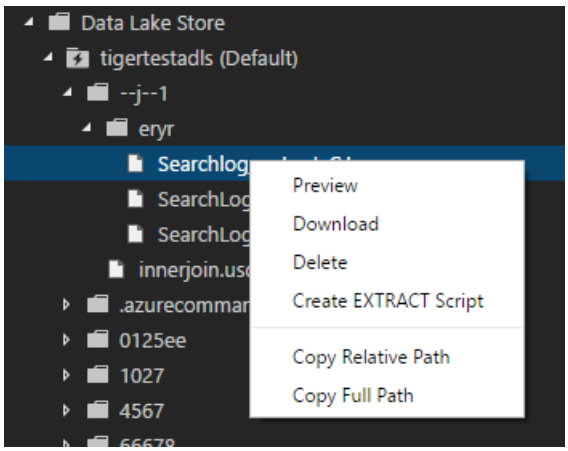

## Integrate with Azure Data Lake Store from the explorer

Browse to **Data Lake Store**:

- You can right-click the folder node and then use the **Refresh**, **Delete**, **Upload**, **Upload Folder**, **Copy Relative Path**, and **Copy Full Path** commands on the shortcut menu.



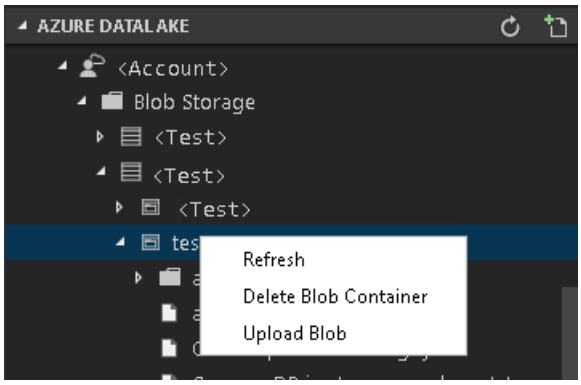

- You can right-click the file node and then use the **Preview**, **Download**, **Delete**, **Create EXTRACT Script** (available only for CSV, TSV, and TXT files), **Copy Relative Path**, and **Copy Full Path** commands on the shortcut menu.



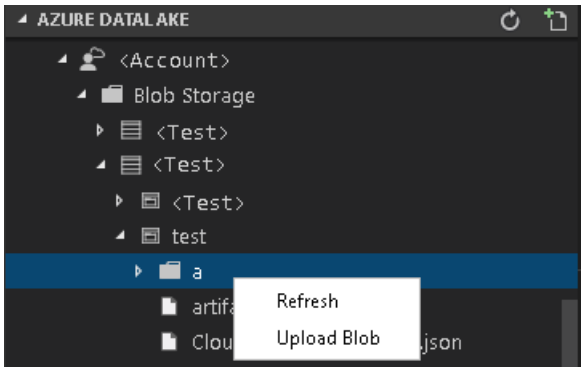

## Integrate with Azure Blob storage from the explorer

Browse to Blob storage:

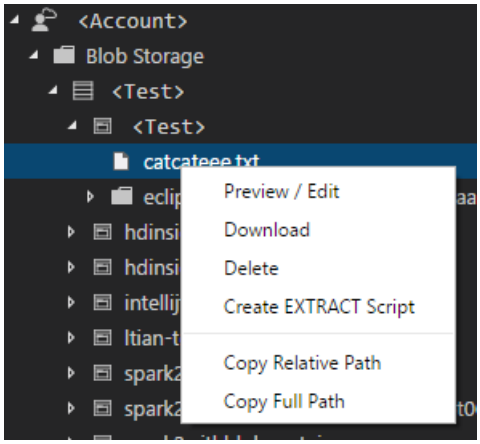
- You can right-click the blob container node and then use the **Refresh**, **Delete Blob Container**, and **Upload Blob** commands on the shortcut menu.




- You can right-click the folder node and then use the **Refresh** and **Upload Blob** commands on the shortcut menu.




- You can right-click the file node and then use the **Preview/Edit**, **Download**, **Delete**, **Create EXTRACT Script** (available only for CSV, TSV, and TXT files), **Copy Relative Path**, and **Copy Full Path** commands on the shortcut menu.




## Open the Data Lake explorer in the portal

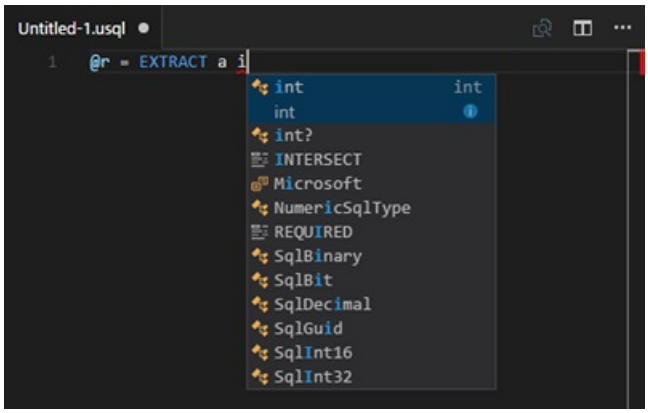
1. Select Ctrl+Shift+P to open the command palette.
2. Enter **Open Web Azure Storage Explorer** or right-click a relative path or the full path in the script editor, and then select **Open Web Azure Storage Explorer**.
3. Select a Data Lake Analytics account.

Data Lake Tools opens the Azure Storage path in the Azure portal. You can find the path and preview the file from the web.

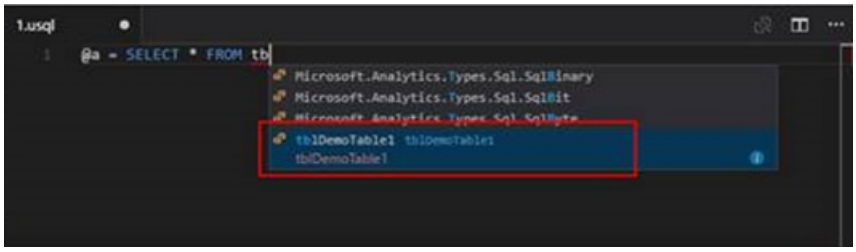
## Additional features

Data Lake Tools for VS Code supports the following features:

- **IntelliSense autocomplete**: Suggestions appear in pop-up windows around items like keywords, methods, and variables. Different icons represent different types of objects:
  - Scala data type
  - Complex data type
  - Built-in UDTs
  - .NET collection and classes
  - C# expressions
  - Built-in C# UDFs, UDOs, and UDAAGs
  - U-SQL functions
  - U-SQL windowing functions




- **IntelliSense autocomplete on Data Lake Analytics metadata**: Data Lake Tools downloads the Data Lake Analytics metadata information locally. The IntelliSense feature automatically populates objects from the Data Lake Analytics metadata. These objects include the database, schema, table, view, table-valued function, procedures, and C# assemblies.




- **IntelliSense error marker**: Data Lake Tools underlines editing errors for U-SQL and C#.
- **Syntax highlights**: Data Lake Tools uses colors to differentiate items like variables, keywords, data types, and functions.

```
@searchlog =
    EXTRACT UserId int,
            Start DateTime,
            Region string,
            Query string,
            Duration int?,
            Urls string,
            ClickedUrls string
    FROM "/Samples/Data/SearchLog.tsv"
    USING Extractors.Tsv();

OUTPUT @searchlog
TO "/output/SearchLogResult1.csv"
USING Outputters.Csv();
```

> **NOTE**
>
> We recommend that you upgrade to Azure Data Lake Tools for Visual Studio version 2.3.3000.4 or later. The previous versions are no longer available for download and are now deprecated.

## Next steps

- Develop U-SQL with Python, R, and C Sharp for Azure Data Lake Analytics in VS Code
- U-SQL local run and local debug with Visual Studio Code
- Tutorial: Get started with Azure Data Lake Analytics
- Tutorial: Develop U-SQL scripts by using Data Lake Tools for Visual Studio

# Get started with Azure Data Lake Analytics using Azure PowerShell

8/27/2018 • 2 minutes to read • Edit Online

Learn how to use Azure PowerShell to create Azure Data Lake Analytics accounts and then submit and run U-SQL jobs. For more information about Data Lake Analytics, see Azure Data Lake Analytics overview.

## Prerequisites

Before you begin this tutorial, you must have the following information:

- **An Azure Data Lake Analytics account**. See Get started with Data Lake Analytics.
- **A workstation with Azure PowerShell**. See How to install and configure Azure PowerShell.

## Log in to Azure

This tutorial assumes you are already familiar with using Azure PowerShell. In particular, you need to know how to log in to Azure. See the Get started with Azure PowerShell if you need help.

To log in with a subscription name:

```
Connect-AzureRmAccount -SubscriptionName "ContosoSubscription"
```

Instead of the subscription name, you can also use a subscription id to log in:

```
Connect-AzureRmAccount -SubscriptionId "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx"
```

If successful, the output of this command looks like the following text:

```
Environment           : AzureCloud
Account               : joe@contoso.com
TenantId              : "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx"
SubscriptionId        : "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx"
SubscriptionName      : ContosoSubscription
CurrentStorageAccount :
```

## Preparing for the tutorial

The PowerShell snippets in this tutorial use these variables to store this information:

```
$rg = "<ResourceGroupName>"
$adls = "<DataLakeStoreAccountName>"
$adla = "<DataLakeAnalyticsAccountName>"
$location = "East US 2"
```

## Get information about a Data Lake Analytics account

```
Get-AdlAnalyticsAccount -ResourceGroupName $rg -Name $adla
```

## Submit a U-SQL job

Create a PowerShell variable to hold the U-SQL script.

```
$script = @"
@a  =
    SELECT * FROM
        (VALUES
            ("Contoso", 1500.0),
            ("Woodgrove", 2700.0)
        ) AS
              D( customer, amount );
OUTPUT @a
    TO "/data.csv"
    USING Outputters.Csv();

"@
```

Submit the script text with the `Submit-AdlJob` cmdlet and the `-Script` parameter.

```
$job = Submit-AdlJob -Account $adla -Name "My Job" -Script $script
```

As an alternative, you can submit a script file using the `-ScriptPath` parameter:

```
$filename = "d:\test.usql"
$script | out-File $filename
$job = Submit-AdlJob -Account $adla -Name "My Job" -ScriptPath $filename
```

Get the status of a job with `Get-AdlJob`.

```
$job = Get-AdlJob -Account $adla -JobId $job.JobId
```

Instead of calling Get-AdlJob over and over until a job finishes, use the `Wait-AdlJob` cmdlet.

```
Wait-AdlJob -Account $adla -JobId $job.JobId
```

Download the output file using `Export-AdlStoreItem`.

```
Export-AdlStoreItem -Account $adls -Path "/data.csv" -Destination "C:\data.csv"
```

## See also

- To see the same tutorial using other tools, click the tab selectors on the top of the page.
- To learn U-SQL, see Get started with Azure Data Lake Analytics U-SQL language.
- For management tasks, see Manage Azure Data Lake Analytics using Azure portal.

# Get started with Azure Data Lake Analytics using Azure CLI

9/24/2018 • 4 minutes to read • Edit Online

This article describes how to use the Azure CLI command-line interface to create Azure Data Lake Analytics accounts, submit USQL jobs, and catalogs. The job reads a tab separated values (TSV) file and converts it into a comma-separated values (CSV) file.

## Prerequisites

Before you begin, you need the following items:

- **An Azure subscription**. See Get Azure free trial.
- This article requires that you are running the Azure CLI version 2.0 or later. If you need to install or upgrade, see Install Azure CLI.

## Log in to Azure

To log in to your Azure subscription:

```
azurecli
az login
```

You are requested to browse to a URL, and enter an authentication code. And then follow the instructions to enter your credentials.

Once you have logged in, the login command lists your subscriptions.

To use a specific subscription:

```
az account set --subscription <subscription id>
```

## Create Data Lake Analytics account

You need a Data Lake Analytics account before you can run any jobs. To create a Data Lake Analytics account, you must specify the following items:

- **Azure Resource Group**. A Data Lake Analytics account must be created within an Azure Resource group. Azure Resource Manager enables you to work with the resources in your application as a group. You can deploy, update, or delete all of the resources for your application in a single, coordinated operation.

To list the existing resource groups under your subscription:

```
az group list
```

To create a new resource group:

```
az group create --name "<Resource Group Name>" --location "<Azure Location>"
```

- **Data Lake Analytics account name**. Each Data Lake Analytics account has a name.
- **Location**. Use one of the Azure data centers that supports Data Lake Analytics.
- **Default Data Lake Store account**: Each Data Lake Analytics account has a default Data Lake Store account.

To list the existing Data Lake Store account:

```
az dls account list
```

To create a new Data Lake Store account:

```
az dls account create --account "<Data Lake Store Account Name>" --resource-group "<Resource Group Name>"
```

Use the following syntax to create a Data Lake Analytics account:

```
az dla account create --account "<Data Lake Analytics Account Name>" --resource-group "<Resource Group Name>"
--location "<Azure location>" --default-data-lake-store "<Default Data Lake Store Account Name>"
```

After creating an account, you can use the following commands to list the accounts and show account details:

```
az dla account list
az dla account show --account "<Data Lake Analytics Account Name>"
```

## Upload data to Data Lake Store

In this tutorial, you process some search logs. The search log can be stored in either Data Lake store or Azure Blob storage.

The Azure portal provides a user interface for copying some sample data files to the default Data Lake Store account, which include a search log file. See Prepare source data to upload the data to the default Data Lake Store account.

To upload files using Azure CLI, use the following commands:

```
az dls fs upload --account "<Data Lake Store Account Name>" --source-path "<Source File Path>" --destination-
path "<Destination File Path>"
az dls fs list --account "<Data Lake Store Account Name>" --path "<Path>"
```

Data Lake Analytics can also access Azure Blob storage. For uploading data to Azure Blob storage, see Using the Azure CLI with Azure Storage.

## Submit Data Lake Analytics jobs

The Data Lake Analytics jobs are written in the U-SQL language. To learn more about U-SQL, see Get started with U-SQL language and U-SQL language eence.

**To create a Data Lake Analytics job script**

Create a text file with following U-SQL script, and save the text file to your workstation:

```
@a  =
    SELECT * FROM
        (VALUES
            ("Contoso", 1500.0),
            ("Woodgrove", 2700.0)
        ) AS
              D( customer, amount );
OUTPUT @a
    TO "/data.csv"
    USING Outputters.Csv();
```

This U-SQL script reads the source data file using **Extractors.Tsv()**, and then creates a csv file using **Outputters.Csv()**.

Don't modify the two paths unless you copy the source file into a different location. Data Lake Analytics creates the output folder if it doesn't exist.

It is simpler to use relative paths for files stored in default Data Lake Store accounts. You can also use absolute paths. For example:

```
adl://<Data LakeStorageAccountName>.azuredatalakestore.net:443/Samples/Data/SearchLog.tsv
```

You must use absolute paths to access files in linked Storage accounts. The syntax for files stored in linked Azure Storage account is:

```
wasb://<BlobContainerName>@<StorageAccountName>.blob.core.windows.net/Samples/Data/SearchLog.tsv
```

> **NOTE**
> Azure Blob container with public blobs are not supported.
> Azure Blob container with public containers are not supported.

**To submit jobs**

Use the following syntax to submit a job.

```
az dla job submit --account "<Data Lake Analytics Account Name>" --job-name "<Job Name>" --script "<Script
Path and Name>"
```

For example:

```
az dla job submit --account "myadlaaccount" --job-name "myadlajob" --script @"C:\DLA\myscript.txt"
```

**To list jobs and show job details**

```
azurecli
az dla job list --account "<Data Lake Analytics Account Name>"
az dla job show --account "<Data Lake Analytics Account Name>" --job-identity "<Job Id>"
```

**To cancel jobs**

```
az dla job cancel --account "<Data Lake Analytics Account Name>" --job-identity "<Job Id>"
```

## Retrieve job results

After a job is completed, you can use the following commands to list the output files, and download the files:

```
az dls fs list --account "<Data Lake Store Account Name>" --source-path "/Output" --destination-path "
<Destination>"
az dls fs preview --account "<Data Lake Store Account Name>" --path "/Output/SearchLog-from-Data-Lake.csv"
az dls fs preview --account "<Data Lake Store Account Name>" --path "/Output/SearchLog-from-Data-Lake.csv" --
length 128 --offset 0
az dls fs download --account "<Data Lake Store Account Name>" --source-path "/Output/SearchLog-from-Data-
Lake.csv" --destination-path "<Destination Path and File Name>"
```

For example:

```
az dls fs download --account "myadlsaccount" --source-path "/Output/SearchLog-from-Data-Lake.csv" --
destination-path "C:\DLA\myfile.csv"
```

## Next steps

- To see the Data Lake Analytics Azure CLI reference document, see Data Lake Analytics.
- To see the Data Lake Store Azure CLI reference document, see Data Lake Store.
- To see a more complex query, see Analyze Website logs using Azure Data Lake Analytics.

# Manage Azure Data Lake Analytics using the Azure portal

8/27/2018 • 5 minutes to read • Edit Online

This article describes how to manage Azure Data Lake Analytics accounts, data sources, users, and jobs by using the Azure portal.

## Manage Data Lake Analytics accounts

**Create an account**

1. Sign in to the Azure portal.
2. Click **Create a resource** > **Intelligence + analytics** > **Data Lake Analytics**.
3. Select values for the following items:
   a. **Name**: The name of the Data Lake Analytics account.
   b. **Subscription**: The Azure subscription used for the account.
   c. **Resource Group**: The Azure resource group in which to create the account.
   d. **Location**: The Azure datacenter for the Data Lake Analytics account.
   e. **Data Lake Store**: The default store to be used for the Data Lake Analytics account. The Azure Data Lake Store account and the Data Lake Analytics account must be in the same location.
4. Click **Create**.

**Delete a Data Lake Analytics account**

Before you delete a Data Lake Analytics account, delete its default Data Lake Store account.

1. In the Azure portal, go to your Data Lake Analytics account.
2. Click **Delete**.
3. Type the account name.
4. Click **Delete**.

## Manage data sources

Data Lake Analytics supports the following data sources:

- Data Lake Store
- Azure Storage

You can use Data Explorer to browse data sources and perform basic file management operations.

**Add a data source**

1. In the Azure portal, go to your Data Lake Analytics account.
2. Click **Data Sources**.
3. Click **Add Data Source**.

   - To add a Data Lake Store account, you need the account name and access to the account to be able to query it.
   - To add Azure Blob storage, you need the storage account and the account key. To find them, go to the storage account in the portal.

## Set up firewall rules

You can use Data Lake Analytics to further lock down access to your Data Lake Analytics account at the network level. You can enable a firewall, specify an IP address, or define an IP address range for your trusted clients. After you enable these measures, only clients that have the IP addresses within the defined range can connect to the store.

If other Azure services, like Azure Data Factory or VMs, connect to the Data Lake Analytics account, make sure that **Allow Azure Services** is turned **On**.

**Set up a firewall rule**

1. In the Azure portal, go to your Data Lake Analytics account.
2. On the menu on the left, click **Firewall**.

## Add a new user

You can use the **Add User Wizard** to easily provision new Data Lake users.

1. In the Azure portal, go to your Data Lake Analytics account.
2. On the left, under **Getting Started**, click **Add User Wizard**.
3. Select a user, and then click **Select**.
4. Select a role, and then click **Select**. To set up a new developer to use Azure Data Lake, select the **Data Lake Analytics Developer** role.
5. Select the access control lists (ACLs) for the U-SQL databases. When you're satisfied with your choices, click **Select**.
6. Select the ACLs for files. For the default store, don't change the ACLs for the root folder "/" and for the /system folder. Click **Select**.
7. Review all your selected changes, and then click **Run**.
8. When the wizard is finished, click **Done**.

## Manage Role-Based Access Control

Like other Azure services, you can use Role-Based Access Control (RBAC) to control how users interact with the service.

The standard RBAC roles have the following capabilities:

- **Owner**: Can submit jobs, monitor jobs, cancel jobs from any user, and configure the account.
- **Contributor**: Can submit jobs, monitor jobs, cancel jobs from any user, and configure the account.
- **Reader**: Can monitor jobs.

Use the Data Lake Analytics Developer role to enable U-SQL developers to use the Data Lake Analytics service. You can use the Data Lake Analytics Developer role to:

- Submit jobs.
- Monitor job status and the progress of jobs submitted by any user.
- See the U-SQL scripts from jobs submitted by any user.
- Cancel only your own jobs.

**Add users or security groups to a Data Lake Analytics account**

1. In the Azure portal, go to your Data Lake Analytics account.
2. Click **Access control (IAM)** > **Add**.
3. Select a role.
4. Add a user.

5. Click **OK**.

> **NOTE**
> If a user or a security group needs to submit jobs, they also need permission on the store account. For more information, see Secure data stored in Data Lake Store.

# Manage jobs

### Submit a job

1. In the Azure portal, go to your Data Lake Analytics account.
2. Click **New Job**. For each job, configure:
   a. **Job Name**: The name of the job.
   b. **Priority**: Lower numbers have higher priority. If two jobs are queued, the one with lower priority value runs first.
   c. **Parallelism**: The maximum number of compute processes to reserve for this job.
3. Click **Submit Job**.

### Monitor jobs

1. In the Azure portal, go to your Data Lake Analytics account.
2. Click **View All Jobs**. A list of all the active and recently finished jobs in the account is shown.
3. Optionally, click **Filter** to help you find the jobs by **Time Range**, **Job Name**, and **Author** values.

### Monitoring pipeline jobs

Jobs that are part of a pipeline work together, usually sequentially, to accomplish a specific scenario. For example, you can have a pipeline that cleans, extracts, transforms, aggregates usage for customer insights. Pipeline jobs are identified using the "Pipeline" property when the job was submitted. Jobs scheduled using ADF V2 will automatically have this property populated.

To view a list of U-SQL jobs that are part of pipelines:

1. In the Azure portal, go to your Data Lake Analytics accounts.
2. Click **Job Insights**. The "All Jobs" tab will be defaulted, showing a list of running, queued, and ended jobs.
3. Click the **Pipeline Jobs** tab. A list of pipeline jobs will be shown along with aggregated statistics for each pipeline.

### Monitoring recurring jobs

A recurring job is one that has the same business logic but uses different input data every time it runs. Ideally, recurring jobs should always succeed, and have relatively stable execution time; monitoring these behaviors will help ensure the job is healthy. Recurring jobs are identified using the "Recurrence" property. Jobs scheduled using ADF V2 will automatically have this property populated.

To view a list of U-SQL jobs that are recurring:

1. In the Azure portal, go to your Data Lake Analytics accounts.
2. Click **Job Insights**. The "All Jobs" tab will be defaulted, showing a list of running, queued, and ended jobs.
3. Click the **Recurring Jobs** tab. A list of recurring jobs will be shown along with aggregated statistics for each recurring job.

# Next steps

- Overview of Azure Data Lake Analytics

- Manage Azure Data Lake Analytics by using Azure PowerShell
- Manage Azure Data Lake Analytics using policies

# Manage Azure Data Lake Analytics using the Azure Command-line Interface (CLI)

8/27/2018 • 4 minutes to read • Edit Online

Learn how to manage Azure Data Lake Analytics accounts, data sources, users, and jobs using the Azure CLI. To see management topics using other tools, click the tab select above.

**Prerequisites**

Before you begin this tutorial, you must have the following resources:

- An Azure subscription. See Get Azure free trial.
- Azure CLI. See Install and configure Azure CLI.
  - Download and install the **pre-release** Azure CLI tools in order to complete this demo.
- Authenticate by using the `az login` command and select the subscription that you want to use. For more information on authenticating using a work or school account, see Connect to an Azure subscription from the Azure CLI.

  ```
  az login
  az account set --subscription <subscription id>
  ```

  You can now access the Data Lake Analytics and Data Lake Store commands. Run the following command to list the Data Lake Store and Data Lake Analytics commands:

  ```
  az dls -h
  az dla -h
  ```

## Manage accounts

Before running any Data Lake Analytics jobs, you must have a Data Lake Analytics account. Unlike Azure HDInsight, you don't pay for an Analytics account when it is not running a job. You only pay for the time when it is running a job. For more information, see Azure Data Lake Analytics Overview.

**Create accounts**

Run the following command to create a Data Lake account,

```
az dla account create --account "<Data Lake Analytics account name>" --location "<Location Name>" --resource-group "<Resource Group Name>" --default-data-lake-store "<Data Lake Store account name>"
```

**Update accounts**

The following command updates the properties of an existing Data Lake Analytics Account

```
az dla account update --account "<Data Lake Analytics Account Name>" --firewall-state "Enabled" --query-store-retention 7
```

**List accounts**

List Data Lake Analytics accounts within a specific resource group

```
az dla account list "<Resource group name>"
```

# Get details of an account

```
az dla account show --account "<Data Lake Analytics account name>" --resource-group "<Resource group name>"
```

**Delete an account**

```
az dla account delete --account "<Data Lake Analytics account name>" --resource-group "<Resource group name>"
```

# Manage data sources

Data Lake Analytics currently supports the following two data sources:

- Azure Data Lake Store
- Azure Storage

When you create an Analytics account, you must designate an Azure Data Lake Storage account to be the default storage account. The default Data Lake storage account is used to store job metadata and job audit logs. After you have created an Analytics account, you can add additional Data Lake Storage accounts and/or Azure Storage account.

**Find the default Data Lake Store account**

You can view the default Data Lake Store account used by running the `az dla account show` command. Default account name is listed under the defaultDataLakeStoreAccount property.

```
az dla account show --account "<Data Lake Analytics account name>"
```

**Add additional Blob storage accounts**

```
az dla account blob-storage add --access-key "<Azure Storage Account Key>" --account "<Data Lake Analytics account name>" --storage-account-name "<Storage account name>"
```

> **NOTE**
> Only Blob storage short names are supported. Don't use FQDN, for example "myblob.blob.core.windows.net".

**Add additional Data Lake Store accounts**

The following command updates the specified Data Lake Analytics account with an additional Data Lake Store account:

```
az dla account data-lake-store add --account "<Data Lake Analytics account name>" --data-lake-store-account-name "<Data Lake Store account name>"
```

**Update existing data source**

To update an existing Blob storage account key:

```
az dla account blob-storage update --access-key "<New Blob Storage Account Key>" --account "<Data Lake
Analytics account name>" --storage-account-name "<Data Lake Store account name>"
```

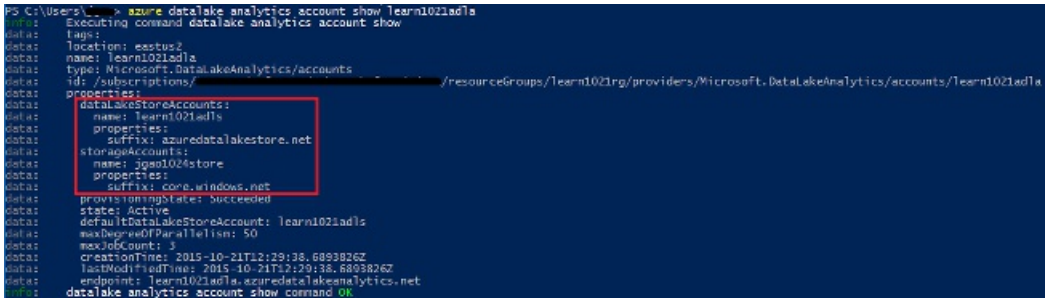
**List data sources:**

To list the Data Lake Store accounts:

```
az dla account data-lake-store list --account "<Data Lake Analytics account name>"
```

To list the Blob storage account:

```
az dla account blob-storage list --account "<Data Lake Analytics account name>"
```




**Delete data sources:**

To delete a Data Lake Store account:

```
az dla account data-lake-store delete --account "<Data Lake Analytics account name>" --data-lake-store-
account-name "<Azure Data Lake Store account name>"
```

To delete a Blob storage account:

```
az dla account blob-storage delete --account "<Data Lake Analytics account name>" --storage-account-name "
<Data Lake Store account name>"
```

# Manage jobs

You must have a Data Lake Analytics account before you can create a job. For more information, see Manage Data Lake Analytics accounts.

**List jobs**

```
az dla job list --account "<Data Lake Analytics account name>"
```

```
PS C:\Users\[redacted]> azure datalake analytics job list -n learn1021adla
info:    Executing command datalake analytics job list
+ Retrieving job list for account: learn1021adla
data:    stateAuditRecords:
data:    jobId: f92869fb-6023-4e2b-be9f-03e38a610bd4
data:    name: Bad path
data:    type: USql
data:    submitter: [redacted]
data:    degreeOfParallelism: 1
data:    priority: 1000
data:    submitTime: Tue, 27 Oct 2015 15:08:11 GMT
data:    startTime: Tue, 27 Oct 2015 15:08:57 GMT
data:    endTime: Tue, 27 Oct 2015 15:09:42 GMT
data:    state: Ended
data:    result: Succeeded
data:    ----------------------------------------------------------------------------------------
data:    stateAuditRecords:
data:    jobId: b881436c-a901-4f2b-ae80-71c0c5259237
data:    name: Bad path
data:    type: USql
data:    submitter: jgao@microsoft.com
data:    degreeOfParallelism: 1
data:    priority: 1000
data:    submitTime: Tue, 27 Oct 2015 14:38:31 GMT
data:    endTime: Tue, 27 Oct 2015 14:38:42 GMT
data:    state: Ended
data:    result: Failed
data:    ----------------------------------------------------------------------------------------
info:    datalake analytics job list command OK
```

### Get job details

```
az dla job show --account "<Data Lake Analytics account name>" --job-identity "<Job Id>"
```

### Submit jobs

> **NOTE**
>
> The default priority of a job is 1000, and the default degree of parallelism for a job is 1.

```
az dla job submit --account "<Data Lake Analytics account name>" --job-name "<Name of your job>" --script
"<Script to submit>"
```

### Cancel jobs

Use the list command to find the job id, and then use cancel to cancel the job.

```
az dla job cancel --account "<Data Lake Analytics account name>" --job-identity "<Job Id>"
```

## Pipelines and recurrences

### Get information about pipelines and recurrences

Use the `az dla job pipeline` commands to see the pipeline information previously submitted jobs.

```
az dla job pipeline list --account "<Data Lake Analytics Account Name>"

az dla job pipeline show --account "<Data Lake Analytics Account Name>" --pipeline-identity "<Pipeline ID>"
```

Use the `az dla job recurrence` commands to see the recurrence information for previously submitted jobs.

```
az dla job recurrence list --account "<Data Lake Analytics Account Name>"

az dla job recurrence show --account "<Data Lake Analytics Account Name>" --recurrence-identity "<Recurrence
ID>"
```

## See also

- Overview of Microsoft Azure Data Lake Analytics
- Get started with Data Lake Analytics using Azure portal
- Manage Azure Data Lake Analytics using Azure portal
- Monitor and troubleshoot Azure Data Lake Analytics jobs using Azure portal

# Manage Azure Data Lake Analytics using Azure PowerShell

9/24/2018 • 8 minutes to read • Edit Online

This article describes how to manage Azure Data Lake Analytics accounts, data sources, users, and jobs by using Azure PowerShell.

## Prerequisites

To use PowerShell with Data Lake Analytics, collect the following pieces of information:

- **Subscription ID**: The ID of the Azure subscription that contains your Data Lake Analytics account.
- **Resource group**: The name of the Azure resource group that contains your Data Lake Analytics account.
- **Data Lake Analytics account name**: The name of your Data Lake Analytics account.
- **Default Data Lake Store account name**: Each Data Lake Analytics account has a default Data Lake Store account.
- **Location**: The location of your Data Lake Analytics account, such as "East US 2" or other supported locations.

The PowerShell snippets in this tutorial use these variables to store this information

```
$subId = "<SubscriptionId>"
$rg = "<ResourceGroupName>"
$adla = "<DataLakeAnalyticsAccountName>"
$adls = "<DataLakeStoreAccountName>"
$location = "<Location>"
```

## Log in to Azure

**Log in using interactive user authentication**

Log in using a subscription ID or by subscription name

```
# Using subscription id
Connect-AzureRmAccount -SubscriptionId $subId

# Using subscription name
Connect-AzureRmAccount -SubscriptionName $subname
```

## Saving authentication context

The `Connect-AzureRmAccount` cmdlet always prompts for credentials. You can avoid being prompted by using the following cmdlets:

```
# Save login session information
Save-AzureRmProfile -Path D:\profile.json

# Load login session information
Select-AzureRmProfile -Path D:\profile.json
```

### Log in using a Service Principal Identity (SPI)

```
$tenantid = "XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX"
$spi_appname = "appname"
$spi_appid = "XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX"
$spi_secret = "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"

$pscredential = New-Object System.Management.Automation.PSCredential ($spi_appid, (ConvertTo-SecureString
$spi_secret -AsPlainText -Force))
Login-AzureRmAccount -ServicePrincipal -TenantId $tenantid -Credential $pscredential -Subscription $subid
```

## Manage accounts

### List accounts

```
# List Data Lake Analytics accounts within the current subscription.
Get-AdlAnalyticsAccount

# List Data Lake Analytics accounts within a specific resource group.
Get-AdlAnalyticsAccount -ResourceGroupName $rg
```

### Create an account

Every Data Lake Analytics account requires a default Data Lake Store account that it uses for storing logs. You can reuse an existing account or create an account.

```
# Create a data lake store if needed, or you can re-use an existing one
New-AdlStore -ResourceGroupName $rg -Name $adls -Location $location
New-AdlAnalyticsAccount -ResourceGroupName $rg -Name $adla -Location $location -DefaultDataLake $adls
```

### Get account information

Get details about an account.

```
Get-AdlAnalyticsAccount -Name $adla
```

### Check if an account exists

```
Test-AdlAnalyticsAccount -Name $adla
```

## Manage data sources

Azure Data Lake Analytics currently supports the following data sources:

- Azure Data Lake Store
- Azure Storage

Every Data Lake Analytics account has a default Data Lake Store account. The default Data Lake Store account is used to store job metadata and job audit logs.

### Find the default Data Lake Store account

```
$adla_acct = Get-AdlAnalyticsAccount -Name $adla
$dataLakeStoreName = $adla_acct.DefaultDataLakeAccount
```

You can find the default Data Lake Store account by filtering the list of datasources by the `IsDefault` property:

```
Get-AdlAnalyticsDataSource -Account $adla  | ? { $_.IsDefault }
```

### Add a data source

```
# Add an additional Storage (Blob) account.
$AzureStorageAccountName = "<AzureStorageAccountName>"
$AzureStorageAccountKey = "<AzureStorageAccountKey>"
Add-AdlAnalyticsDataSource -Account $adla -Blob $AzureStorageAccountName -AccessKey $AzureStorageAccountKey

# Add an additional Data Lake Store account.
$AzureDataLakeStoreName = "<AzureDataLakeStoreAccountName"
Add-AdlAnalyticsDataSource -Account $adla -DataLakeStore $AzureDataLakeStoreName
```

### List data sources

```
# List all the data sources
Get-AdlAnalyticsDataSource -Account $adla

# List attached Data Lake Store accounts
Get-AdlAnalyticsDataSource -Account $adla | where -Property Type -EQ "DataLakeStore"

# List attached Storage accounts
Get-AdlAnalyticsDataSource -Account $adla | where -Property Type -EQ "Blob"
```

## Submit U-SQL jobs

### Submit a string as a U-SQL job

```
$script = @"
@a  =
    SELECT * FROM
        (VALUES
            ("Contoso", 1500.0),
            ("Woodgrove", 2700.0)
        ) AS D( customer, amount );
OUTPUT @a
    TO "/data.csv"
    USING Outputters.Csv();
"@

$scriptpath = "d:\test.usql"
$script | Out-File $scriptpath

Submit-AdlJob -AccountName $adla -Script $script -Name "Demo"
```

### Submit a file as a U-SQL job

```
$scriptpath = "d:\test.usql"
$script | Out-File $scriptpath
Submit-AdlJob -AccountName $adla –ScriptPath $scriptpath -Name "Demo"
```

### List jobs

The output includes the currently running jobs and those jobs that have recently completed.

```
Get-AdlJob -Account $adla
```

### List the top N jobs

By default the list of jobs is sorted on submit time. So the most recently submitted jobs appear first. By default, The ADLA account remembers jobs for 180 days, but the Get-AdlJob cmdlet by default returns only the first 500. Use -Top parameter to list a specific number of jobs.

```
$jobs = Get-AdlJob -Account $adla -Top 10
```

### List jobs by job state

Using the `-State` parameter. You can combine any of these values:

- `Accepted`
- `Compiling`
- `Ended`
- `New`
- `Paused`
- `Queued`
- `Running`
- `Scheduling`
- `Start`

```
# List the running jobs
Get-AdlJob -Account $adla -State Running

# List the jobs that have completed
Get-AdlJob -Account $adla -State Ended

# List the jobs that have not started yet
Get-AdlJob -Account $adla -State Accepted,Compiling,New,Paused,Scheduling,Start
```

### List jobs by job result

Use the `-Result` parameter to detect whether ended jobs completed successfully. It has these values:

- Cancelled
- Failed
- None
- Succeeded

```
# List Successful jobs.
Get-AdlJob -Account $adla -State Ended -Result Succeeded

# List Failed jobs.
Get-AdlJob -Account $adla -State Ended -Result Failed
```

### List jobs by job submitter

The `-Submitter` parameter helps you identify who submitted a job.

```
Get-AdlJob -Account $adla -Submitter "joe@contoso.com"
```

### List jobs by submission time

The `-SubmittedAfter` is useful in filtering to a time range.

```
# List  jobs submitted in the last day.
$d = [DateTime]::Now.AddDays(-1)
Get-AdlJob -Account $adla -SubmittedAfter $d

# List  jobs submitted in the last seven day.
$d = [DateTime]::Now.AddDays(-7)
Get-AdlJob -Account $adla -SubmittedAfter $d
```

### Get job status

Get the status of a specific job.

```
Get-AdlJob -AccountName $adla -JobId $job.JobId
```

### Cancel a job

```
Stop-AdlJob -Account $adla -JobID $jobID
```

### Wait for a job to finish

Instead of repeating `Get-AdlAnalyticsJob` until a job finishes, you can use the `Wait-AdlJob` cmdlet to wait for the job to end.

```
Wait-AdlJob -Account $adla -JobId $job.JobId
```

## Analyzing job history

Using Azure PowerShell to analyze the history of jobs that have run in Data Lake analytics is a powerful technique. You can use it to gain insights into usage and cost. You can learn more by looking at the Job History Analysis sample repo

## List job pipelines and recurrences

Use the `Get-AdlJobPipeline` cmdlet to see the pipeline information previously submitted jobs.

```
$pipelines = Get-AdlJobPipeline -Account $adla
$pipeline = Get-AdlJobPipeline -Account $adla -PipelineId "<pipeline ID>"
```

Use the `Get-AdlJobRecurrence` cmdlet to see the recurrence information for previously submitted jobs.

```
$recurrences = Get-AdlJobRecurrence -Account $adla

$recurrence = Get-AdlJobRecurrence -Account $adla -RecurrenceId "<recurrence ID>"
```

## Manage compute policies

### List existing compute policies

The `Get-AdlAnalyticsComputePolicy` cmdlet retrieves info about compute policies for a Data Lake Analytics account.

```
$policies = Get-AdlAnalyticsComputePolicy -Account $adla
```

### Create a compute policy

The `New-AdlAnalyticsComputePolicy` cmdlet creates a new compute policy for a Data Lake Analytics account. This example sets the maximum AUs available to the specified user to 50, and the minimum job priority to 250.

```
$userObjectId = (Get-AzureRmAdUser -SearchString "garymcdaniel@contoso.com").Id

New-AdlAnalyticsComputePolicy -Account $adla -Name "GaryMcDaniel" -ObjectId $objectId -ObjectType User -
MaxDegreeOfParallelismPerJob 50 -MinPriorityPerJob 250
```

## Manage files

### Check for the existence of a file.

```
Test-AdlStoreItem -Account $adls -Path "/data.csv"
```

### Uploading and downloading

Upload a file.

```
Import-AdlStoreItem -AccountName $adls -Path "c:\data.tsv" -Destination "/data_copy.csv"
```

Upload an entire folder recursively.

```
Import-AdlStoreItem -AccountName $adls -Path "c:\myData\" -Destination "/myData/" -Recurse
```

Download a file.

```
Export-AdlStoreItem -AccountName $adls -Path "/data.csv" -Destination "c:\data.csv"
```

Download an entire folder recursively.

```
Export-AdlStoreItem -AccountName $adls -Path "/" -Destination "c:\myData\" -Recurse
```

> **NOTE**
> If the upload or download process is interrupted, you can attempt to resume the process by running the cmdlet again with the `-Resume` flag.

## Manage the U-SQL catalog

The U-SQL catalog is used to structure data and code so they can be shared by U-SQL scripts. The catalog enables the highest performance possible with data in Azure Data Lake. For more information, see Use U-SQL catalog.

### List items in the U-SQL catalog

```
# List U-SQL databases
Get-AdlCatalogItem -Account $adla -ItemType Database

# List tables within a database
Get-AdlCatalogItem -Account $adla -ItemType Table -Path "database"

# List tables within a schema.
Get-AdlCatalogItem -Account $adla -ItemType Table -Path "database.schema"
```

### List all the assemblies the U-SQL catalog

```
$dbs = Get-AdlCatalogItem -Account $adla -ItemType Database

foreach ($db in $dbs)
{
    $asms = Get-AdlCatalogItem -Account $adla -ItemType Assembly -Path $db.Name

    foreach ($asm in $asms)
    {
        $asmname = "[" + $db.Name + "].[" + $asm.Name + "]"
        Write-Host $asmname
    }
}
```

### Get details about a catalog item

```
# Get details of a table
Get-AdlCatalogItem  -Account $adla -ItemType Table -Path "master.dbo.mytable"

# Test existence of a U-SQL database.
Test-AdlCatalogItem  -Account $adla -ItemType Database -Path "master"
```

### Store credentials in the catalog

Within a U-SQL database, create a credential object for a database hosted in Azure. Currently, U-SQL credentials are the only type of catalog item that you can create through PowerShell.

```
$dbName = "master"
$credentialName = "ContosoDbCreds"
$dbUri = "https://contoso.database.windows.net:8080"

New-AdlCatalogCredential -AccountName $adla `
          -DatabaseName $db `
          -CredentialName $credentialName `
          -Credential (Get-Credential) `
          -Uri $dbUri
```

## Manage firewall rules

### List firewall rules

```
Get-AdlAnalyticsFirewallRule -Account $adla
```

### Add a firewall rule

```
$ruleName = "Allow access from on-prem server"
$startIpAddress = "<start IP address>"
$endIpAddress = "<end IP address>"

Add-AdlAnalyticsFirewallRule -Account $adla -Name $ruleName -StartIpAddress $startIpAddress -EndIpAddress
$endIpAddress
```

### Modify a firewall rule

```
Set-AdlAnalyticsFirewallRule -Account $adla -Name $ruleName -StartIpAddress $startIpAddress -EndIpAddress
$endIpAddress
```

### Remove a firewall rule

```
Remove-AdlAnalyticsFirewallRule -Account $adla -Name $ruleName
```

### Allow Azure IP addresses

```
Set-AdlAnalyticsAccount -Name $adla -AllowAzureIpState Enabled
```

```
Set-AdlAnalyticsAccount -Name $adla -FirewallState Enabled
Set-AdlAnalyticsAccount -Name $adla -FirewallState Disabled
```

# Working with Azure

### Get details of AzureRm errors

```
Resolve-AzureRmError -Last
```

### Verify if you are running as an Administrator on your Windows machine

```
function Test-Administrator
{
    $user = [Security.Principal.WindowsIdentity]::GetCurrent();
    $p = New-Object Security.Principal.WindowsPrincipal $user
    $p.IsInRole([Security.Principal.WindowsBuiltinRole]::Administrator)
}
```

### Find a TenantID

From a subscription name:

```
function Get-TenantIdFromSubcriptionName( [string] $subname )
{
    $sub = (Get-AzureRmSubscription -SubscriptionName $subname)
    $sub.TenantId
}

Get-TenantIdFromSubcriptionName "ADLTrainingMS"
```

From a subscription ID:

```
function Get-TenantIdFromSubcriptionId( [string] $subid )
{
    $sub = (Get-AzureRmSubscription -SubscriptionId $subid)
    $sub.TenantId
}

$subid = "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx"
Get-TenantIdFromSubcriptionId $subid
```

From a domain address such as "contoso.com"

```
function Get-TenantIdFromDomain( $domain )
{
    $url = "https://login.windows.net/" + $domain + "/.well-known/openid-configuration"
    return (Invoke-WebRequest $url|ConvertFrom-Json).token_endpoint.Split('/')[3]
}

$domain = "contoso.com"
Get-TenantIdFromDomain $domain
```

**List all your subscriptions and tenant IDs**

```
$subs = Get-AzureRmSubscription
foreach ($sub in $subs)
{
    Write-Host $sub.Name "("  $sub.Id ")"
    Write-Host "`tTenant Id" $sub.TenantId
}
```

## Create a Data Lake Analytics account using a template

You can also use an Azure Resource Group template using the following sample: Create a Data Lake Analytics account using a template

## Next steps

- Overview of Microsoft Azure Data Lake Analytics
- Get started with Data Lake Analytics using the Azure portal | Azure PowerShell | Azure CLI
- Manage Azure Data Lake Analytics using Azure portal | Azure PowerShell | CLI

# Manage Azure Data Lake Analytics a .NET app

8/27/2018 • 7 minutes to read • Edit Online

This article describes how to manage Azure Data Lake Analytics accounts, data sources, users, and jobs using an app written using the Azure .NET SDK.

## Prerequisites

- **Visual Studio 2015, Visual Studio 2013 update 4, or Visual Studio 2012 with Visual C++ Installed**.
- **Microsoft Azure SDK for .NET version 2.5 or above**. Install it using the Web platform installer.
- **Required NuGet Packages**

### Install NuGet packages

| PACKAGE | VERSION |
| --- | --- |
| Microsoft.Rest.ClientRuntime.Azure.Authentication | 2.3.1 |
| Microsoft.Azure.Management.DataLake.Analytics | 3.0.0 |
| Microsoft.Azure.Management.DataLake.Store | 2.2.0 |
| Microsoft.Azure.Management.ResourceManager | 1.6.0-preview |
| Microsoft.Azure.Graph.RBAC | 3.4.0-preview |

You can install these packages via the NuGet command line with the following commands:

```
Install-Package -Id Microsoft.Rest.ClientRuntime.Azure.Authentication  -Version 2.3.1
Install-Package -Id Microsoft.Azure.Management.DataLake.Analytics  -Version 3.0.0
Install-Package -Id Microsoft.Azure.Management.DataLake.Store  -Version 2.2.0
Install-Package -Id Microsoft.Azure.Management.ResourceManager  -Version 1.6.0-preview
Install-Package -Id Microsoft.Azure.Graph.RBAC -Version 3.4.0-preview
```

## Common variables

```
string subid = "<Subscription ID>"; // Subscription ID (a GUID)
string tenantid = "<Tenant ID>"; // AAD tenant ID or domain. For example, "contoso.onmicrosoft.com"
string rg == "<value>"; // Resource  group name
string clientid = "1950a258-227b-4e31-a9cf-717495945fc2"; // Sample client ID (this will work, but you should
pick your own)
```

## Authentication

You have multiple options for logging on to Azure Data Lake Analytics. The following snippet shows an example of authentication with interactive user authentication with a pop-up.

```
using System;
using System.IO;
using System.Threading;
using System.Security.Cryptography.X509Certificates;

using Microsoft.Rest;
using Microsoft.Rest.Azure.Authentication;
using Microsoft.Azure.Management.DataLake.Analytics;
using Microsoft.Azure.Management.DataLake.Analytics.Models;
using Microsoft.Azure.Management.DataLake.Store;
using Microsoft.Azure.Management.DataLake.Store.Models;
using Microsoft.IdentityModel.Clients.ActiveDirectory;
using Microsoft.Azure.Graph.RBAC;

public static Program
{
   public static string TENANT = "microsoft.onmicrosoft.com";
   public static string CLIENTID = "1950a258-227b-4e31-a9cf-717495945fc2";
   public static System.Uri ARM_TOKEN_AUDIENCE = new System.Uri( @"https://management.core.windows.net/");
   public static System.Uri ADL_TOKEN_AUDIENCE = new System.Uri( @"https://datalake.azure.net/" );
   public static System.Uri GRAPH_TOKEN_AUDIENCE = new System.Uri( @"https://graph.windows.net/" );

   static void Main(string[] args)
   {
      string MY_DOCUMENTS= System.Environment.GetFolderPath( System.Environment.SpecialFolder.MyDocuments);
      string TOKEN_CACHE_PATH = System.IO.Path.Combine(MY_DOCUMENTS, "my.tokencache");

      var tokenCache = GetTokenCache(TOKEN_CACHE_PATH);
      var armCreds = GetCreds_User_Popup(TENANT, ARM_TOKEN_AUDIENCE, CLIENTID, tokenCache);
      var adlCreds = GetCreds_User_Popup(TENANT, ADL_TOKEN_AUDIENCE, CLIENTID, tokenCache);
      var graphCreds = GetCreds_User_Popup(TENANT, GRAPH_TOKEN_AUDIENCE, CLIENTID, tokenCache);
   }
}
```

The source code for **GetCreds_User_Popup** and the code for other options for authentication are covered in Data Lake Analytics .NET authentication options

# Create the client management objects

```
var resourceManagementClient = new ResourceManagementClient(armCreds) { SubscriptionId = subid };

var adlaAccountClient = new DataLakeAnalyticsAccountManagementClient(armCreds);
adlaAccountClient.SubscriptionId = subid;

var adlsAccountClient = new DataLakeStoreAccountManagementClient(armCreds);
adlsAccountClient.SubscriptionId = subid;

var adlaCatalogClient = new DataLakeAnalyticsCatalogManagementClient(adlCreds);
var adlaJobClient = new DataLakeAnalyticsJobManagementClient(adlCreds);

var adlsFileSystemClient = new DataLakeStoreFileSystemManagementClient(adlCreds);

var  graphClient = new GraphRbacManagementClient(graphCreds);
graphClient.TenantID = domain;
```

# Manage accounts

### Create an Azure Resource Group

If you haven't already created one, you must have an Azure Resource Group to create your Data Lake Analytics components. You need your authentication credentials, subscription ID, and a location. The following code shows how to create a resource group:

```
var resourceGroup = new ResourceGroup { Location = location };
resourceManagementClient.ResourceGroups.CreateOrUpdate(groupName, rg);
```

For more information, see Azure Resource Groups and Data Lake Analytics.

### Create a Data Lake Store account

Ever ADLA account requires an ADLS account. If you don't already have one to use, you can create one with the following code:

```
var new_adls_params = new DataLakeStoreAccount(location: _location);
adlsAccountClient.Account.Create(rg, adls, new_adls_params);
```

### Create a Data Lake Analytics account

The following code creates an ADLS account

```
var new_adla_params = new DataLakeAnalyticsAccount()
{
   DefaultDataLakeStoreAccount = adls,
   Location = location
};

adlaClient.Account.Create(rg, adla, new_adla_params);
```

### List Data Lake Store accounts

```
var adlsAccounts = adlsAccountClient.Account.List().ToList();
foreach (var adls in adlsAccounts)
{
   Console.WriteLine($"ADLS: {0}", adls.Name);
}
```

### List Data Lake Analytics accounts

```
var adlaAccounts = adlaClient.Account.List().ToList();

for (var adla in AdlaAccounts)
{
   Console.WriteLine($"ADLA: {0}, adla.Name");
}
```

### Checking if an account exists

```
bool exists = adlaClient.Account.Exists(rg, adla));
```

### Get information about an account

```
bool exists = adlaClient.Account.Exists(rg, adla));
if (exists)
{
   var adla_accnt = adlaClient.Account.Get(rg, adla);
}
```

### Delete an account

```
if (adlaClient.Account.Exists(rg, adla))
{
   adlaClient.Account.Delete(rg, adla);
}
```

#### Get the default Data Lake Store account

Every Data Lake Analytics account requires a default Data Lake Store account. Use this code to determine the default Store account for an Analytics account.

```
if (adlaClient.Account.Exists(rg, adla))
{
  var adla_accnt = adlaClient.Account.Get(rg, adla);
  string def_adls_account = adla_accnt.DefaultDataLakeStoreAccount;
}
```

## Manage data sources

Data Lake Analytics currently supports the following data sources:

- Azure Data Lake Store
- Azure Storage Account

#### Link to an Azure Storage account

You can create links to Azure Storage accounts.

```
string storage_key = "xxxxxxxxxxxxxxxxxxx";
string storage_account = "mystorageaccount";
var addParams = new AddStorageAccountParameters(storage_key);
adlaClient.StorageAccounts.Add(rg, adla, storage_account, addParams);
```

#### List Azure Storage data sources

```
var stg_accounts = adlaAccountClient.StorageAccounts.ListByAccount(rg, adla);

if (stg_accounts != null)
{
  foreach (var stg_account in stg_accounts)
  {
      Console.WriteLine($"Storage account: {0}", stg_account.Name);
  }
}
```

#### List Data Lake Store data sources

```
var adls_accounts = adlsClient.Account.List();

if (adls_accounts != null)
{
  foreach (var adls_accnt in adls_accounts)
  {
      Console.WriteLine($"ADLS account: {0}", adls_accnt.Name);
  }
}
```

#### Upload and download folders and files

You can use the Data Lake Store file system client management object to upload and download individual files or

folders from Azure to your local computer, using the following methods:

- UploadFolder
- UploadFile
- DownloadFolder
- DownloadFile

The first parameter for these methods is the name of the Data Lake Store Account, followed by parameters for the source path and the destination path.

The following example shows how to download a folder in the Data Lake Store.

```
adlsFileSystemClient.FileSystem.DownloadFolder(adls, sourcePath, destinationPath);
```

**Create a file in a Data Lake Store account**

```
using (var memstream = new MemoryStream())
{
   using (var sw = new StreamWriter(memstream, UTF8Encoding.UTF8))
   {
      sw.WriteLine("Hello World");
      sw.Flush();

      memstream.Position = 0;

      adlsFileSystemClient.FileSystem.Create(adls, "/Samples/Output/randombytes.csv", memstream);
   }
}
```

**Verify Azure Storage account paths**

The following code checks if an Azure Storage account (storageAccntName) exists in a Data Lake Analytics account (analyticsAccountName), and if a container (containerName) exists in the Azure Storage account.

```
string storage_account = "mystorageaccount";
string storage_container = "mycontainer";
bool accountExists = adlaClient.Account.StorageAccountExists(rg, adla, storage_account));
bool containerExists = adlaClient.Account.StorageContainerExists(rg, adla, storage_account,
storage_container));
```

# Manage catalog and jobs

The DataLakeAnalyticsCatalogManagementClient object provides methods for managing the SQL database provided for each Azure Data Lake Analytics account. The DataLakeAnalyticsJobManagementClient provides methods to submit and manage jobs run on the database with U-SQL scripts.

**List databases and schemas**

Among the several things you can list, the most common are databases and their schema. The following code obtains a collection of databases, and then enumerates the schema for each database.

```
var databases = adlaCatalogClient.Catalog.ListDatabases(adla);
foreach (var db in databases)
{
  Console.WriteLine($"Database: {db.Name}");
  Console.WriteLine(" - Schemas:");
  var schemas = adlaCatalogClient.Catalog.ListSchemas(adla, db.Name);
  foreach (var schm in schemas)
  {
      Console.WriteLine($"\t{schm.Name}");
  }
}
```

**List table columns**

The following code shows how to access the database with a Data Lake Analytics Catalog management client to list the columns in a specified table.

```
var tbl = adlaCatalogClient.Catalog.GetTable(adla, "master", "dbo", "MyTableName");
IEnumerable<USqlTableColumn> columns = tbl.ColumnList;

foreach (USqlTableColumn utc in columns)
{
  Console.WriteLine($"\t{utc.Name}");
}
```

**Submit a U-SQL job**

The following code shows how to use a Data Lake Analytics Job management client to submit a job.

```
string scriptPath = "/Samples/Scripts/SearchResults_Wikipedia_Script.txt";
Stream scriptStrm = adlsFileSystemClient.FileSystem.Open(_adlsAccountName, scriptPath);
string scriptTxt = string.Empty;
using (StreamReader sr = new StreamReader(scriptStrm))
{
    scriptTxt = sr.ReadToEnd();
}

var jobName = "SR_Wikipedia";
var jobId = Guid.NewGuid();
var properties = new USqlJobProperties(scriptTxt);
var parameters = new JobInformation(jobName, JobType.USql, properties, priority: 1, degreeOfParallelism: 1,
jobId: jobId);
var jobInfo = adlaJobClient.Job.Create(adla, jobId, parameters);
Console.WriteLine($"Job {jobName} submitted.");
```

**List failed jobs**

The following code lists information about jobs that failed.

```
var odq = new ODataQuery<JobInformation> { Filter = "result eq 'Failed'" };
var jobs = adlaJobClient.Job.List(adla, odq);
foreach (var j in jobs)
{
   Console.WriteLine($"{j.Name}\t{j.JobId}\t{j.Type}\t{j.StartTime}\t{j.EndTime}");
}
```

**List pipelines**

The following code lists information about each pipeline of jobs submitted to the account.

```
var pipelines = adlaJobClient.Pipeline.List(adla);
foreach (var p in pipelines)
{
   Console.WriteLine($"Pipeline: {p.Name}\t{p.PipelineId}\t{p.LastSubmitTime}");
}
```

#### List recurrences

The following code lists information about each recurrence of jobs submitted to the account.

```
var recurrences = adlaJobClient.Recurrence.List(adla);
foreach (var r in recurrences)
{
   Console.WriteLine($"Recurrence: {r.Name}\t{r.RecurrenceId}\t{r.LastSubmitTime}");
}
```

## Common graph scenarios

#### Look up user in the AAD directory

```
var userinfo = graphClient.Users.Get( "bill@contoso.com" );
```

#### Get the ObjectId of a user in the AAD directory

```
var userinfo = graphClient.Users.Get( "bill@contoso.com" );
Console.WriteLine( userinfo.ObjectId )
```

## Manage compute policies

The DataLakeAnalyticsAccountManagementClient object provides methods for managing the compute policies for a Data Lake Analytics account.

#### List compute policies

The following code retrieves a list of compute policies for a Data Lake Analytics account.

```
var policies = adlaAccountClient.ComputePolicies.ListByAccount(rg, adla);
foreach (var p in policies)
{
   Console.WriteLine($"Name: {p.Name}\tType: {p.ObjectType}\tMax AUs / job:
{p.MaxDegreeOfParallelismPerJob}\tMin priority / job: {p.MinPriorityPerJob}");
}
```

#### Create a new compute policy

The following code creates a new compute policy for a Data Lake Analytics account, setting the maximum AUs available to the specified user to 50, and the minimum job priority to 250.

```
var userAadObjectId = "3b097601-4912-4d41-b9d2-78672fc2acde";
var newPolicyParams = new ComputePolicyCreateOrUpdateParameters(userAadObjectId, "User", 50, 250);
adlaAccountClient.ComputePolicies.CreateOrUpdate(rg, adla, "GaryMcDaniel", newPolicyParams);
```

## Next steps

- Overview of Microsoft Azure Data Lake Analytics

- Manage Azure Data Lake Analytics using Azure portal
- Monitor and troubleshoot Azure Data Lake Analytics jobs using Azure portal

# Manage Azure Data Lake Analytics using Python

8/27/2018 • 4 minutes to read • Edit Online

This article describes how to manage Azure Data Lake Analytics accounts, data sources, users, and jobs by using Python.

## Supported Python versions

- Use a 64-bit version of Python.
- You can use the standard Python distribution found at **Python.org downloads**.
- Many developers find it convenient to use the **Anaconda Python distribution**.
- This article was written using Python version 3.6 from the standard Python distribution

## Install Azure Python SDK

Install the following modules:

- The **azure-mgmt-resource** module includes other Azure modules for Active Directory, etc.
- The **azure-datalake-store** module includes the Azure Data Lake Store filesystem operations.
- The **azure-mgmt-datalake-store** module includes the Azure Data Lake Store account management operations.
- The **azure-mgmt-datalake-analytics** module includes the Azure Data Lake Analytics operations.

First, ensure you have the latest `pip` by running the following command:

```
python -m pip install --upgrade pip
```

This document was written using `pip version 9.0.1`.

Use the following `pip` commands to install the modules from the commandline:

```
pip install azure-mgmt-resource
pip install azure-datalake-store
pip install azure-mgmt-datalake-store
pip install azure-mgmt-datalake-analytics
```

## Create a new Python script

Paste the following code into the script:

```
## Use this only for Azure AD service-to-service authentication
#from azure.common.credentials import ServicePrincipalCredentials

## Use this only for Azure AD end-user authentication
#from azure.common.credentials import UserPassCredentials

## Required for Azure Resource Manager
from azure.mgmt.resource.resources import ResourceManagementClient
from azure.mgmt.resource.resources.models import ResourceGroup

## Required for Azure Data Lake Store account management
from azure.mgmt.datalake.store import DataLakeStoreAccountManagementClient
from azure.mgmt.datalake.store.models import DataLakeStoreAccount

## Required for Azure Data Lake Store filesystem management
from azure.datalake.store import core, lib, multithread

## Required for Azure Data Lake Analytics account management
from azure.mgmt.datalake.analytics.account import DataLakeAnalyticsAccountManagementClient
from azure.mgmt.datalake.analytics.account.models import DataLakeAnalyticsAccount, DataLakeStoreAccountInfo

## Required for Azure Data Lake Analytics job management
from azure.mgmt.datalake.analytics.job import DataLakeAnalyticsJobManagementClient
from azure.mgmt.datalake.analytics.job.models import JobInformation, JobState, USqlJobProperties

## Required for Azure Data Lake Analytics catalog management
from azure.mgmt.datalake.analytics.catalog import DataLakeAnalyticsCatalogManagementClient

## Use these as needed for your application
import logging, getpass, pprint, uuid, time
```

Run this script to verify that the modules can be imported.

# Authentication

### Interactive user authentication with a pop-up

This method is not supported.

### Interactive user authentication with a device code

```
user = input('Enter the user to authenticate with that has permission to subscription: ')
password = getpass.getpass()
credentials = UserPassCredentials(user, password)
```

### Noninteractive authentication with SPI and a secret

```
credentials = ServicePrincipalCredentials(client_id = 'FILL-IN-HERE', secret = 'FILL-IN-HERE', tenant =
'FILL-IN-HERE')
```

### Noninteractive authentication with API and a certificate

This method is not supported.

# Common script variables

These variables are used in the samples.

```
subid= '<Azure Subscription ID>'
rg = '<Azure Resource Group Name>'
location = '<Location>' # i.e. 'eastus2'
adls = '<Azure Data Lake Store Account Name>'
adla = '<Azure Data Lake Analytics Account Name>'
```

## Create the clients

```
resourceClient = ResourceManagementClient(credentials, subid)
adlaAcctClient = DataLakeAnalyticsAccountManagementClient(credentials, subid)
adlaJobClient = DataLakeAnalyticsJobManagementClient( credentials, 'azuredatalakeanalytics.net')
```

## Create an Azure Resource Group

```
armGroupResult = resourceClient.resource_groups.create_or_update( rg, ResourceGroup( location=location ) )
```

## Create Data Lake Analytics account

First create a store account.

```
adlsAcctResult = adlsAcctClient.account.create(
    rg,
    adls,
    DataLakeStoreAccount(
        location=location)
    )
).wait()
```

Then create an ADLA account that uses that store.

```
adlaAcctResult = adlaAcctClient.account.create(
    rg,
    adla,
    DataLakeAnalyticsAccount(
        location=location,
        default_data_lake_store_account=adls,
        data_lake_store_accounts=[DataLakeStoreAccountInfo(name=adls)]
    )
).wait()
```

## Submit a job

```
script = """
@a  =
    SELECT * FROM
        (VALUES
            ("Contoso", 1500.0),
            ("Woodgrove", 2700.0)
        ) AS
              D( customer, amount );
OUTPUT @a
    TO "/data.csv"
    USING Outputters.Csv();
"""

jobId = str(uuid.uuid4())
jobResult = adlaJobClient.job.create(
    adla,
    jobId,
    JobInformation(
        name='Sample Job',
        type='USql',
        properties=USqlJobProperties(script=script)
    )
)
```

## Wait for a job to end

```
jobResult = adlaJobClient.job.get(adla, jobId)
while(jobResult.state != JobState.ended):
    print('Job is not yet done, waiting for 3 seconds. Current state: ' + jobResult.state.value)
    time.sleep(3)
    jobResult = adlaJobClient.job.get(adla, jobId)

print ('Job finished with result: ' + jobResult.result.value)
```

## List pipelines and recurrences

Depending whether your jobs have pipeline or recurrence metadata attached, you can list pipelines and recurrences.

```
pipelines = adlaJobClient.pipeline.list(adla)
for p in pipelines:
    print('Pipeline: ' + p.name + ' ' + p.pipelineId)

recurrences = adlaJobClient.recurrence.list(adla)
for r in recurrences:
    print('Recurrence: ' + r.name + ' ' + r.recurrenceId)
```

## Manage compute policies

The DataLakeAnalyticsAccountManagementClient object provides methods for managing the compute policies for a Data Lake Analytics account.

**List compute policies**

The following code retrieves a list of compute policies for a Data Lake Analytics account.

```
policies = adlaAccountClient.computePolicies.listByAccount(rg, adla)
for p in policies:
    print('Name: ' + p.name + 'Type: ' + p.objectType + 'Max AUs / job: ' + p.maxDegreeOfParallelismPerJob +
'Min priority / job: ' + p.minPriorityPerJob)
```

**Create a new compute policy**

The following code creates a new compute policy for a Data Lake Analytics account, setting the maximum AUs available to the specified user to 50, and the minimum job priority to 250.

```
userAadObjectId = "3b097601-4912-4d41-b9d2-78672fc2acde"
newPolicyParams = ComputePolicyCreateOrUpdateParameters(userAadObjectId, "User", 50, 250)
adlaAccountClient.computePolicies.createOrUpdate(rg, adla, "GaryMcDaniel", newPolicyParams)
```

# Next steps

- To see the same tutorial using other tools, click the tab selectors on the top of the page.
- To learn U-SQL, see Get started with Azure Data Lake Analytics U-SQL language.
- For management tasks, see Manage Azure Data Lake Analytics using Azure portal.

# Manage Azure Data Lake Analytics using a Java app

8/27/2018 • 5 minutes to read • Edit Online

This article describes how to manage Azure Data Lake Analytics accounts, data sources, users, and jobs using an app written using the Azure Java SDK.

## Prerequisites

- **Java Development Kit (JDK) 8** (using Java version 1.8).
- **IntelliJ** or another suitable Java development environment. The instructions in this document use IntelliJ.
- Create an Azure Active Directory (AAD) application and retrieve its **Client ID**, **Tenant ID**, and **Key**. For more information about AAD applications and instructions on how to get a client ID, see Create Active Directory application and service principal using portal. The Reply URI and Key is available from the portal once you have the application created and key generated.

## Authenticating using Azure Active Directory

The code following snippet provides code for **non-interactive** authentication, where the application provides its own credentials.

## Create a Java application

1. Open IntelliJ and create a Java project using the **Command-Line App** template.
2. Right-click on the project on the left-hand side of your screen and click **Add Framework Support**. Choose **Maven** and click **OK**.
3. Open the newly created **"pom.xml"** file and add the following snippet of text between the **</version>** tag and the **</project>** tag:

```
<repositories>
    <repository>
        <id>adx-snapshots</id>
        <name>Azure ADX Snapshots</name>
        <url>http://adxsnapshots.azurewebsites.net/</url>
        <layout>default</layout>
        <snapshots>
            <enabled>true</enabled>
        </snapshots>
    </repository>
    <repository>
        <id>oss-snapshots</id>
        <name>Open Source Snapshots</name>
        <url>https://oss.sonatype.org/content/repositories/snapshots/</url>
        <layout>default</layout>
        <snapshots>
            <enabled>true</enabled>
            <updatePolicy>always</updatePolicy>
        </snapshots>
    </repository>
</repositories>
<dependencies>
    <dependency>
        <groupId>com.microsoft.azure</groupId>
        <artifactId>azure-client-authentication</artifactId>
        <version>1.0.0-20160513.000802-24</version>
    </dependency>
    <dependency>
        <groupId>com.microsoft.azure</groupId>
        <artifactId>azure-client-runtime</artifactId>
        <version>1.0.0-20160513.000812-28</version>
    </dependency>
    <dependency>
        <groupId>com.microsoft.rest</groupId>
        <artifactId>client-runtime</artifactId>
        <version>1.0.0-20160513.000825-29</version>
    </dependency>
    <dependency>
        <groupId>com.microsoft.azure</groupId>
        <artifactId>azure-mgmt-datalake-store</artifactId>
        <version>1.0.0-SNAPSHOT</version>
    </dependency>
    <dependency>
        <groupId>com.microsoft.azure</groupId>
        <artifactId>azure-mgmt-datalake-analytics</artifactId>
        <version>1.0.0-SNAPSHOT</version>
    </dependency>
</dependencies>
```

Go to **File > Settings > Build > Execution > Deployment**. Select **Build Tools > Maven > Importing**. Then check **Import Maven projects automatically**.

Open `Main.java` and replace the existing code block with the following code snippet:

```
package com.company;

import com.microsoft.azure.CloudException;
import com.microsoft.azure.credentials.ApplicationTokenCredentials;
import com.microsoft.azure.management.datalake.store.*;
import com.microsoft.azure.management.datalake.store.models.*;
import com.microsoft.azure.management.datalake.analytics.*;
import com.microsoft.azure.management.datalake.analytics.models.*;
import com.microsoft.rest.credentials.ServiceClientCredentials;
import java.io.*;
import java.nio.charset.Charset;
import java.nio.file.Files;
```

```
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.ArrayList;
import java.util.UUID;
import java.util.List;

public class Main {
    private static String _adlsAccountName;
    private static String _adlaAccountName;
    private static String _resourceGroupName;
    private static String _location;

    private static String _tenantId;
    private static String _subId;
    private static String _clientId;
    private static String _clientSecret;

    private static DataLakeStoreAccountManagementClient _adlsClient;
    private static DataLakeStoreFileSystemManagementClient _adlsFileSystemClient;
    private static DataLakeAnalyticsAccountManagementClient _adlaClient;
    private static DataLakeAnalyticsJobManagementClient _adlaJobClient;
    private static DataLakeAnalyticsCatalogManagementClient _adlaCatalogClient;

    public static void main(String[] args) throws Exception {

        _adlsAccountName = "<DATA-LAKE-STORE-NAME>";
        _adlaAccountName = "<DATA-LAKE-ANALYTICS-NAME>";
        _resourceGroupName = "<RESOURCE-GROUP-NAME>";
        _location = "East US 2";

        _tenantId = "<TENANT-ID>";
        _subId =  "<SUBSCRIPTION-ID>";
        _clientId = "<CLIENT-ID>";

        _clientSecret = "<CLIENT-SECRET>";

        String localFolderPath = "C:\\local_path\\";

        // ----------------------------------------
        // Authenticate
        // ----------------------------------------
        ApplicationTokenCredentials creds = new ApplicationTokenCredentials(_clientId, _tenantId,
_clientSecret, null);
        SetupClients(creds);

        // ----------------------------------------
        // List Data Lake Store and Analytics accounts that this app can access
        // ----------------------------------------
        System.out.println(String.format("All ADL Store accounts that this app can access in subscription
%s:", _subId));
        List<DataLakeStoreAccount> adlsListResult = _adlsClient.getAccountOperations().list().getBody();
        for (DataLakeStoreAccount acct : adlsListResult) {
            System.out.println(acct.getName());
        }

        System.out.println(String.format("All ADL Analytics accounts that this app can access in subscription
%s:", _subId));
        List<DataLakeAnalyticsAccount> adlaListResult = _adlaClient.getAccountOperations().list().getBody();
        for (DataLakeAnalyticsAccount acct : adlaListResult) {
            System.out.println(acct.getName());
        }
        WaitForNewline("Accounts displayed.", "Creating files.");

        // ----------------------------------------
        // Create a file in Data Lake Store: input1.csv
        // ----------------------------------------

        byte[] bytesContents = "123,abc".getBytes();
        _adlsFileSystemClient.getFileSystemOperations().create(_adlsAccountName, "/input1.csv",
bytesContents, true);
```

```
bytesContents, true);

        WaitForNewline("File created.", "Submitting a job.");

        // ----------------------------------------
        // Submit a job to Data Lake Analytics
        // ----------------------------------------

string script = "@input =  EXTRACT Data string FROM \"/input1.csv\" USING Extractors.Csv(); OUTPUT @input TO
@\"/output1.csv\" USING Outputters.Csv();", "testJob";
        UUID jobId = SubmitJobByScript(script);
        WaitForNewline("Job submitted.", "Getting job status.");

        // ----------------------------------------
        // Wait for job completion and output job status
        // ----------------------------------------
        System.out.println(String.format("Job status: %s", GetJobStatus(jobId)));
        System.out.println("Waiting for job completion.");
        WaitForJob(jobId);
        System.out.println(String.format("Job status: %s", GetJobStatus(jobId)));
        WaitForNewline("Job completed.", "Downloading job output.");

        // ----------------------------------------
        // Download job output from Data Lake Store
        // ----------------------------------------
        DownloadFile("/output1.csv", localFolderPath + "output1.csv");
        WaitForNewline("Job output downloaded.", "Deleting file.");

    }
}
```

Provide the values for parameters called out in the code snippet:

- `localFolderPath`
- `_adlaAccountName`
- `_adlsAccountName`
- `_resourceGroupName`

Replace the placeholders for:

- `CLIENT-ID`,
- `CLIENT-SECRET`,
- `TENANT-ID`
- `SUBSCRIPTION-ID`

# Helper functions

### Setup clients

```
public static void SetupClients(ServiceClientCredentials creds)
{
    _adlsClient = new DataLakeStoreAccountManagementClientImpl(creds);
    _adlsFileSystemClient = new DataLakeStoreFileSystemManagementClientImpl(creds);
    _adlaClient = new DataLakeAnalyticsAccountManagementClientImpl(creds);
    _adlaJobClient = new DataLakeAnalyticsJobManagementClientImpl(creds);
    _adlaCatalogClient = new DataLakeAnalyticsCatalogManagementClientImpl(creds);
    _adlsClient.setSubscriptionId(_subId);
    _adlaClient.setSubscriptionId(_subId);
}
```

### Wait for input

```
public static void WaitForNewline(String reason, String nextAction)
{
    if (nextAction == null)
        nextAction = "";

    System.out.println(reason + "\r\nPress ENTER to continue...");
    try{System.in.read();}
    catch(Exception e){}

    if (!nextAction.isEmpty())
    {
        System.out.println(nextAction);
    }
}
```

### Create accounts

```
public static void CreateAccounts() throws InterruptedException, CloudException, IOException
{
    // Create ADLS account
    DataLakeStoreAccount adlsParameters = new DataLakeStoreAccount();
    adlsParameters.setLocation(_location);

    _adlsClient.getAccountOperations().create(_resourceGroupName, _adlsAccountName, adlsParameters);

    // Create ADLA account
    DataLakeStoreAccountInfo adlsInfo = new DataLakeStoreAccountInfo();
    adlsInfo.setName(_adlsAccountName);

    DataLakeStoreAccountInfoProperties adlsInfoProperties = new DataLakeStoreAccountInfoProperties();
    adlsInfo.setProperties(adlsInfoProperties);

    List<DataLakeStoreAccountInfo> adlsInfoList = new ArrayList<DataLakeStoreAccountInfo>();
    adlsInfoList.add(adlsInfo);

    DataLakeAnalyticsAccountProperties adlaProperties = new DataLakeAnalyticsAccountProperties();
    adlaProperties.setDataLakeStoreAccounts(adlsInfoList);
    adlaProperties.setDefaultDataLakeStoreAccount(_adlsAccountName);

    DataLakeAnalyticsAccount adlaParameters = new DataLakeAnalyticsAccount();
    adlaParameters.setLocation(_location);
    adlaParameters.setName(_adlaAccountName);
    adlaParameters.setProperties(adlaProperties);

    _adlaClient.getAccountOperations().create(_resourceGroupName, _adlaAccountName, adlaParameters);
}
```

### Create a file

```
public static void CreateFile(String path, String contents, boolean force) throws IOException, CloudException
{
    byte[] bytesContents = contents.getBytes();

    _adlsFileSystemClient.getFileSystemOperations().create(_adlsAccountName, path, bytesContents, force);
}
```

### Delete a file

```
public static void DeleteFile(String filePath) throws IOException, CloudException
{
    _adlsFileSystemClient.getFileSystemOperations().delete(filePath, _adlsAccountName);
}
```

### Download a file

```
public static void DownloadFile(String srcPath, String destPath) throws IOException, CloudException
{
    InputStream stream = _adlsFileSystemClient.getFileSystemOperations().open(srcPath,
_adlsAccountName).getBody();

    PrintWriter pWriter = new PrintWriter(destPath, Charset.defaultCharset().name());

    String fileContents = "";
    if (stream != null) {
        Writer writer = new StringWriter();

        char[] buffer = new char[1024];
        try {
            Reader reader = new BufferedReader(
                    new InputStreamReader(stream, "UTF-8"));
            int n;
            while ((n = reader.read(buffer)) != -1) {
                writer.write(buffer, 0, n);
            }
        } finally {
            stream.close();
        }
        fileContents =  writer.toString();
    }

    pWriter.println(fileContents);
    pWriter.close();
}
```

### Submit a U-SQL job

```
public static UUID SubmitJobByScript(String script, String jobName) throws IOException, CloudException
{
    UUID jobId = java.util.UUID.randomUUID();
    USqlJobProperties properties = new USqlJobProperties();
    properties.setScript(script);
    JobInformation parameters = new JobInformation();
    parameters.setName(jobName);
    parameters.setJobId(jobId);
    parameters.setType(JobType.USQL);
    parameters.setProperties(properties);

    JobInformation jobInfo = _adlaJobClient.getJobOperations().create(_adlaAccountName, jobId,
parameters).getBody();

    return jobId;
}

// Wait for job completion
public static JobResult WaitForJob(UUID jobId) throws IOException, CloudException
{
    JobInformation jobInfo = _adlaJobClient.getJobOperations().get(_adlaAccountName, jobId).getBody();
    while (jobInfo.getState() != JobState.ENDED)
    {
        jobInfo = _adlaJobClient.getJobOperations().get(_adlaAccountName,jobId).getBody();
    }
    return jobInfo.getResult();
}
```

**Retrieve job status**

```
public static String GetJobStatus(UUID jobId) throws IOException, CloudException
{
    JobInformation jobInfo = _adlaJobClient.getJobOperations().get(_adlaAccountName, jobId).getBody();
    return jobInfo.getState().toValue();
}
```

# Next steps

- To learn U-SQL, see Get started with Azure Data Lake Analytics U-SQL language, and U-SQL language reference.
- For management tasks, see Manage Azure Data Lake Analytics using Azure portal.
- To get an overview of Data Lake Analytics, see Azure Data Lake Analytics overview.

# Manage Azure Data Lake Analytics using Azure SDK for Node.js

8/27/2018 • 2 minutes to read • Edit Online

This article describes how to manage Azure Data Lake Analytics accounts, data sources, users, and jobs using an app written using the Azure SDK for Node.js.

The following versions are supported:

- **Node.js version: 0.10.0 or higher**
- **REST API version for Account: 2015-10-01-preview**
- **REST API version for Catalog: 2015-10-01-preview**
- **REST API version for Job: 2016-03-20-preview**

## Features

- Account management: create, get, list, update, and delete.
- Job management: submit, get, list, and cancel.
- Catalog management: get and list.

## How to Install

```
npm install azure-arm-datalake-analytics
```

## Authenticate using Azure Active Directory

```
var msrestAzure = require('ms-rest-azure');
//user authentication
var credentials = new msRestAzure.UserTokenCredentials('your-client-id', 'your-domain', 'your-username',
'your-password', 'your-redirect-uri');
//service principal authentication
var credentials = new msRestAzure.ApplicationTokenCredentials('your-client-id', 'your-domain', 'your-
secret');
```

## Create the Data Lake Analytics client

```
var adlaManagement = require("azure-arm-datalake-analytics");
var acccountClient = new adlaManagement.DataLakeAnalyticsAccountClient(credentials, 'your-subscription-id');
var jobClient = new adlaManagement.DataLakeAnalyticsJobClient(credentials, 'azuredatalakeanalytics.net');
var catalogClient = new adlaManagement.DataLakeAnalyticsCatalogClient(credentials,
'azuredatalakeanalytics.net');
```

## Create a Data Lake Analytics account

```
var util = require('util');
var resourceGroupName = 'testrg';
var accountName = 'testadlaacct';
var location = 'eastus2';

// A Data Lake Store account must already have been created to create
// a Data Lake Analytics account. See the Data Lake Store readme for
// information on doing so. For now, we assume one exists already.
var datalakeStoreAccountName = 'existingadlsaccount';

// account object to create
var accountToCreate = {
  tags: {
    testtag1: 'testvalue1',
    testtag2: 'testvalue2'
  },
  name: accountName,
  location: location,
  properties: {
    defaultDataLakeStoreAccount: datalakeStoreAccountName,
    dataLakeStoreAccounts: [
      {
        name: datalakeStoreAccountName
      }
    ]
  }
};

client.account.create(resourceGroupName, accountName, accountToCreate, function (err, result, request,
response) {
  if (err) {
    console.log(err);
    /*err has reference to the actual request and response, so you can see what was sent and received on the
wire.
      The structure of err looks like this:
      err: {
        code: 'Error Code',
        message: 'Error Message',
        body: 'The response body if any',
        request: reference to a stripped version of http request
        response: reference to a stripped version of the response
      }
    */
  } else {
    console.log('result is: ' + util.inspect(result, {depth: null}));
  }
});
```

## Get a list of jobs

```
var util = require('util');
var accountName = 'testadlaacct';
jobClient.job.list(accountName, function (err, result, request, response) {
  if (err) {
    console.log(err);
  } else {
    console.log('result is: ' + util.inspect(result, {depth: null}));
  }
});
```

## Get a list of databases in the Data Lake Analytics Catalog

```
var util = require('util');
var accountName = 'testadlaacct';
catalogClient.catalog.listDatabases(accountName, function (err, result, request, response) {
  if (err) {
    console.log(err);
  } else {
    console.log('result is: ' + util.inspect(result, {depth: null}));
  }
});
```

## See also

- Microsoft Azure SDK for Node.js
- Microsoft Azure SDK for Node.js - Data Lake Store Management

# Adding a user in the Azure portal

9/13/2018 • 2 minutes to read • Edit Online

## Start the Add User Wizard

1. Open your Azure Data Lake Analytics via https://portal.azure.com.
2. Click **Add User Wizard**.
3. In the **Select user** step, find the user you want to add. Click **Select**.
4. the **Select role** step, pick **Data Lake Analytics Developer**. This role has the minimum set of permissions required to submit/monitor/manage U-SQL jobs. Assign to this role if the group is not intended for managing Azure services.
5. In the **Select catalog permissions** step, select any additional databases that user will need access to. Read and Write Access to the master database is required to submit jobs. When you are done, click **OK**.
6. In the final step called **Assign selected permissions** review the changes the wizard will make. Click **OK**.

## Configure ACLs for data folders

Grant "R-X" or "RWX", as needed, on folders containing input data and output data.

## Optionally, add the user to the Azure Data Lake Storage Gen1 role **Reader** role.

1. Find your Azure Data Lake Storage Gen1 account.
2. Click on **Users**.
3. Click **Add**.
4. Select an Azure RBAC Role to assign this group.
5. Assign to Reader role. This role has the minimum set of permissions required to browse/manage data stored in ADLSGen1. Assign to this role if the Group is not intended for managing Azure services.
6. Type in the name of the Group.
7. Click **OK**.

## Adding a user using PowerShell

1. Follow the instructions in this guide: How to install and configure Azure PowerShell.
2. Download the Add-AdlaJobUser.ps1 PowerShell script.
3. Run the PowerShell script.

The sample command to give user access to submit jobs, view new job metadata, and view old metadata is:

```
Add-AdlaJobUser.ps1 -Account myadlsaccount -EntityToAdd 546e153e-0ecf-417b-ab7f-aa01ce4a7bff -EntityType User -
FullReplication
```

## Next steps

- Overview of Azure Data Lake Analytics
- Get started with Data Lake Analytics by using the Azure portal
- Manage Azure Data Lake Analytics by using Azure PowerShell

# Manage Azure Data Lake Analytics using policies

8/27/2018 • 4 minutes to read • Edit Online

Using account policies, you can control how resources an Azure Data Lake Analytics account are used. These policies allow you to control the cost of using Azure Data Lake Analytics. For example, with these policies you can prevent unexpected cost spikes by limiting how many AUs the account can simultaneously use.

## Account-level policies

These policies apply to all jobs in a Data Lake Analytics account.

**Maximum number of AUs in a Data Lake Analytics account**

A policy controls the total number of Analytics Units (AUs) your Data Lake Analytics account can use. By default, the value is set to 250. For example, if this value is set to 250 AUs, you can have one job running with 250 AUs assigned to it, or 10 jobs running with 25 AUs each. Additional jobs that are submitted are queued until the running jobs are finished. When running jobs are finished, AUs are freed up for the queued jobs to run.

To change the number of AUs for your Data Lake Analytics account:

1. In the Azure portal, go to your Data Lake Analytics account.
2. Click **Properties**.
3. Under **Maximum AUs**, move the slider to select a value, or enter the value in the text box.
4. Click **Save**.

> **NOTE**
> If you need more than the default (250) AUs, in the portal, click **Help+Support** to submit a support request. The number of AUs available in your Data Lake Analytics account can be increased.

**Maximum number of jobs that can run simultaneously**

A policy controls how many jobs can run at the same time. By default, this value is set to 20. If your Data Lake Analytics has AUs available, new jobs are scheduled to run immediately until the total number of running jobs reaches the value of this policy. When you reach the maximum number of jobs that can run simultaneously, subsequent jobs are queued in priority order until one or more running jobs complete (depending on AU availability).

To change the number of jobs that can run simultaneously:

1. In the Azure portal, go to your Data Lake Analytics account.
2. Click **Properties**.
3. Under **Maximum Number of Running Jobs**, move the slider to select a value, or enter the value in the text box.
4. Click **Save**.

> **NOTE**
> If you need to run more than the default (20) number of jobs, in the portal, click **Help+Support** to submit a support request. The number of jobs that can run simultaneously in your Data Lake Analytics account can be increased.

**How long to keep job metadata and resources**

When your users run U-SQL jobs, the Data Lake Analytics service retains all related files. Related files include the U-SQL script, the DLL files referenced in the U-SQL script, compiled resources, and statistics. The files are in the /system/ folder of the default Azure Data Lake Storage account. This policy controls how long these resources are stored before they are automatically deleted (the default is 30 days). You can use these files for debugging, and for performance-tuning of jobs that you'll rerun in the future.

To change how long to keep job metadata and resources:

1. In the Azure portal, go to your Data Lake Analytics account.
2. Click **Properties**.
3. Under **Days to Retain Job Queries**, move the slider to select a value, or enter the value in the text box.
4. Click **Save**.

## Job-level policies

With job-level policies, you can control the maximum AUs and the maximum priority that individual users (or members of specific security groups) can set on jobs that they submit. This policy lets you control the costs incurred by users. It also lets you control the effect that scheduled jobs might have on high-priority production jobs that are running in the same Data Lake Analytics account.

Data Lake Analytics has two policies that you can set at the job level:

- **AU limit per job**: Users can only submit jobs that have up to this number of AUs. By default, this limit is the same as the maximum AU limit for the account.
- **Priority**: Users can only submit jobs that have a priority lower than or equal to this value. A higher number indicates a lower priority. By default, this limit is set to 1, which is the highest possible priority.

There is a default policy set on every account. The default policy applies to all users of the account. You can set additional policies for specific users and groups.

> **NOTE**
> Account-level policies and job-level policies apply simultaneously.

**Add a policy for a specific user or group**

1. In the Azure portal, go to your Data Lake Analytics account.
2. Click **Properties**.
3. Under **Job Submission Limits**, click the **Add Policy** button. Then, select or enter the following settings:
   a. **Compute Policy Name**: Enter a policy name, to remind you of the purpose of the policy.
   b. **Select User or Group**: Select the user or group this policy applies to.
   c. **Set the Job AU Limit**: Set the AU limit that applies to the selected user or group.
   d. **Set the Priority Limit**: Set the priority limit that applies to the selected user or group.
4. Click **Ok**.
5. The new policy is listed in the **Default** policy table, under **Job Submission Limits**.

**Delete or edit an existing policy**

1. In the Azure portal, go to your Data Lake Analytics account.
2. Click **Properties**.
3. Under **Job Submission Limits**, find the policy you want to edit.
4. To see the **Delete** and **Edit** options, in the rightmost column of the table, click `...`.

## Additional resources for job policies

- Policy overview blog post
- Account-level policies blog post
- Job-level policies blog post

## Next steps

- Overview of Azure Data Lake Analytics
- Get started with Data Lake Analytics by using the Azure portal
- Manage Azure Data Lake Analytics by using Azure PowerShell

# Configure user access to job information to job information in Azure Data Lake Analytics

8/27/2018 • 2 minutes to read • Edit Online

In Azure Data Lake Analytics, you can use multiple user accounts or service principals to run jobs.

In order for those same users to see the detailed job information, the users need to be able to read the contents of the job folders. The job folders are located in `/system/` directory.

If the necessary permissions are not configured, the user may see an error:

```
Graph data not available - You don't have permissions to access the graph data.
```

## Configure user access to job information

You can use the **Add User Wizard** to configure the ACLs on the folders. For more information, see Add a new user.

If you need more granular control, or need to script the permissions, then secure the folders as follows:

1. Grant **execute** permissions (via an access ACL) on the root folder:
   - /
2. Grant **execute** and **read** permissions (via an access ACL and a default ACL) on the folders that contain the job folders. For example, for a specific job that ran on May 25, 2018, these folders need to be accessed:
   - /system
   - /system/jobservice
   - /system/jobservice/jobs
   - /system/jobservice/jobs/Usql
   - /system/jobservice/jobs/Usql/2018
   - /system/jobservice/jobs/Usql/2018/05
   - /system/jobservice/jobs/Usql/2018/05/25
   - /system/jobservice/jobs/Usql/2018/05/25/11
   - /system/jobservice/jobs/Usql/2018/05/25/11/01
   - /system/jobservice/jobs/Usql/2018/05/25/11/01/b074bd7a-1448-d879-9d75-f562b101bd3d

## Next steps

Add a new user

# Accessing diagnostic logs for Azure Data Lake Analytics

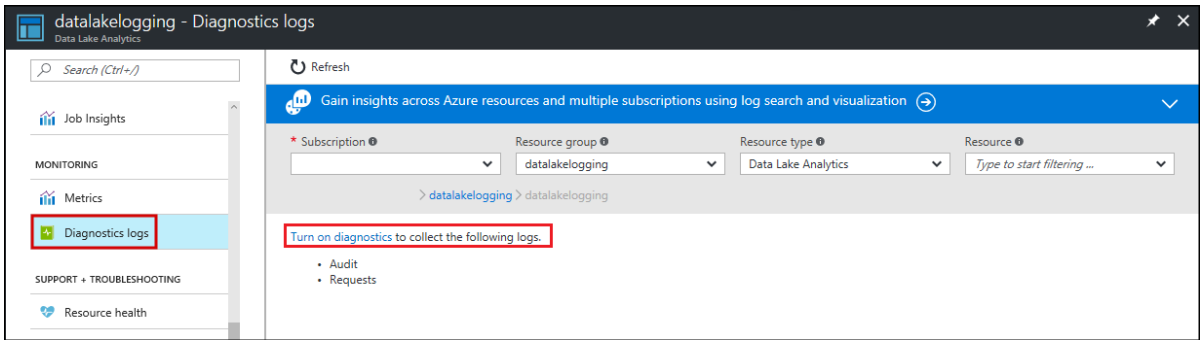
8/27/2018 • 4 minutes to read • Edit Online

Diagnostic logging allows you to collect data access audit trails. These logs provide information such as:

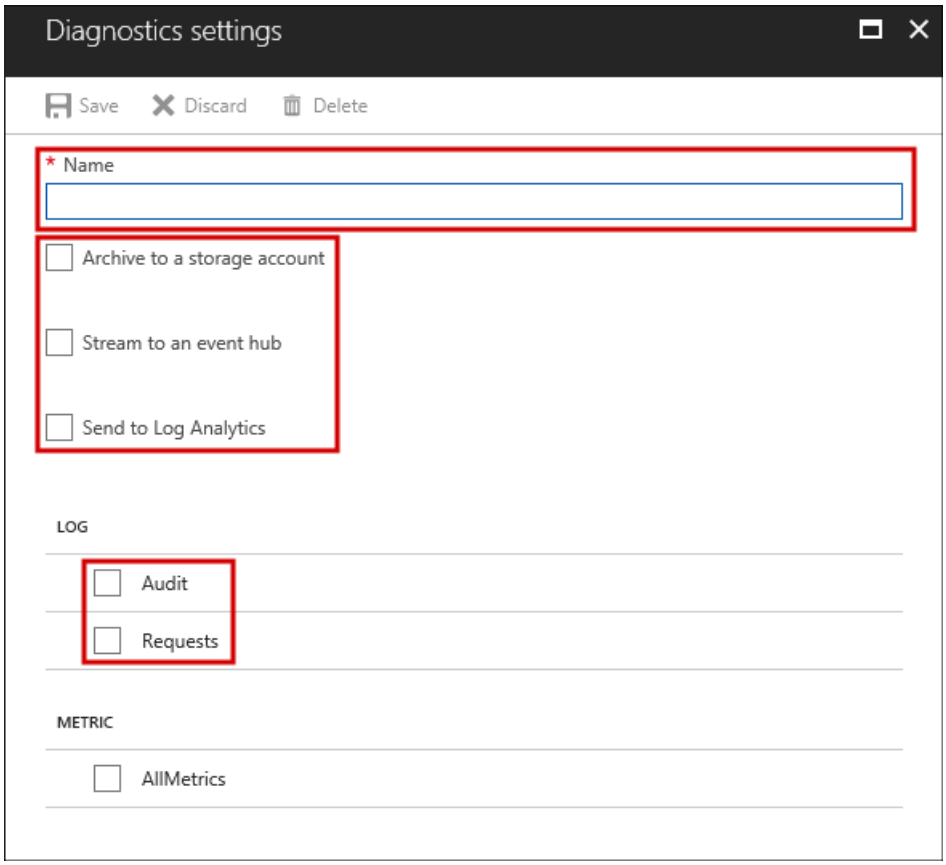
- A list of users that accessed the data.
- How frequently the data is accessed.
- How much data is stored in the account.

## Enable logging

1. Sign on to the Azure portal.
2. Open your Data Lake Analytics account and select **Diagnostic logs** from the **Monitor** section. Next, select **Turn on diagnostics**.




3. From **Diagnostics settings**, enter a **Name** for this logging configuration and then select logging options.



- You can choose to store/process the data in three different ways.
    - Select **Archive to a storage account** to store logs in an Azure storage account. Use this option if you want to archive the data. If you select this option, you must provide an Azure storage account to save the logs to.
    - Select **Stream to an Event Hub** to stream log data to an Azure Event Hub. Use this option if you have a downstream processing pipeline that is analyzing incoming logs in real time. If you select this option, you must provide the details for the Azure Event Hub you want to use.
    - Select **Send to Log Analytics** to send the data to the Log Analytics service. Use this option if you want to use Log Analytics to gather and analyze logs.
- Specify whether you want to get audit logs or request logs or both. A request log captures every API request. An audit log records all operations that are triggered by that API request.
- For **Archive to a storage account**, specify the number of days to retain the data.
- Click **Save**.

> **NOTE**
> You must select either **Archive to a storage account**, **Stream to an Event Hub** or **Send to Log Analytics** before clicking the **Save** button.

**Use the Azure Storage account that contains log data**

1. To display the blob containers that hold logging data, open the Azure Storage account used for Data Lake Analytics for logging, and then click **Blobs**.
    - The container **insights-logs-audit** contains the audit logs.
    - The container **insights-logs-requests** contains the request logs.
2. Within the containers, the logs are stored under the following file structure:

```
resourceId=/
  SUBSCRIPTIONS/
    <<SUBSCRIPTION_ID>>/
      RESOURCEGROUPS/
        <<RESOURCE_GRP_NAME>>/
          PROVIDERS/
            MICROSOFT.DATALAKEANALYTICS/
              ACCOUNTS/
                <DATA_LAKE_ANALYTICS_NAME>>/
                  y=####/
                    m=##/
                      d=##/
                        h=##/
                          m=00/
                            PT1H.json
```

> **NOTE**
> The `##` entries in the path contain the year, month, day, and hour in which the log was created. Data Lake Analytics creates one file every hour, so `m=` always contains a value of `00`.

As an example, the complete path to an audit log could be:

```
 https://adllogs.blob.core.windows.net/insights-logs-audit/resourceId=/SUBSCRIPTIONS/<sub-
id>/RESOURCEGROUPS/myresourcegroup/PROVIDERS/MICROSOFT.DATALAKEANALYTICS/ACCOUNTS/mydatalakeanalytics/y
=2016/m=07/d=18/h=04/m=00/PT1H.json
```

Similarly, the complete path to a request log could be:

```
 https://adllogs.blob.core.windows.net/insights-logs-requests/resourceId=/SUBSCRIPTIONS/<sub-
id>/RESOURCEGROUPS/myresourcegroup/PROVIDERS/MICROSOFT.DATALAKEANALYTICS/ACCOUNTS/mydatalakeanalytics/y
=2016/m=07/d=18/h=14/m=00/PT1H.json
```

# Log structure

The audit and request logs are in a structured JSON format.

### Request logs

Here's a sample entry in the JSON-formatted request log. Each blob has one root object called **records** that contains an array of log objects.

```
{
"records":
  [
    . . . .
    ,
    {
         "time": "2016-07-07T21:02:53.456Z",
         "resourceId":
"/SUBSCRIPTIONS/<subscription_id>/RESOURCEGROUPS/<resource_group_name>/PROVIDERS/MICROSOFT.DATALAKEANALYTICS/A
CCOUNTS/<data_lake_analytics_account_name>",
         "category": "Requests",
         "operationName": "GetAggregatedJobHistory",
         "resultType": "200",
         "callerIpAddress": "::ffff:1.1.1.1",
         "correlationId": "4a11c709-05f5-417c-a98d-6e81b3e29c58",
         "identity": "1808bd5f-62af-45f4-89d8-03c5e81bac30",
         "properties": {
             "HttpMethod":"POST",
             "Path":"/JobAggregatedHistory",
             "RequestContentLength":122,
             "ClientRequestId":"3b7adbd9-3519-4f28-a61c-bd89506163b8",
             "StartTime":"2016-07-07T21:02:52.472Z",
             "EndTime":"2016-07-07T21:02:53.456Z"
             }
    }
    ,
    . . . .
  ]
}
```

**Request log schema**

| NAME | TYPE | DESCRIPTION |
|---|---|---|
| time | String | The timestamp (in UTC) of the log |
| resourceId | String | The identifier of the resource that operation took place on |

| NAME | TYPE | DESCRIPTION |
| --- | --- | --- |
| category | String | The log category. For example, **Requests**. |
| operationName | String | Name of the operation that is logged. For example, GetAggregatedJobHistory. |
| resultType | String | The status of the operation, For example, 200. |
| callerIpAddress | String | The IP address of the client making the request |
| correlationId | String | The identifier of the log. This value can be used to group a set of related log entries. |
| identity | Object | The identity that generated the log |
| properties | JSON | See the next section (Request log properties schema) for details |

**Request log properties schema**

| NAME | TYPE | DESCRIPTION |
| --- | --- | --- |
| HttpMethod | String | The HTTP Method used for the operation. For example, GET. |
| Path | String | The path the operation was performed on |
| RequestContentLength | int | The content length of the HTTP request |
| ClientRequestId | String | The identifier that uniquely identifies this request |
| StartTime | String | The time at which the server received the request |
| EndTime | String | The time at which the server sent a response |

### Audit logs

Here's a sample entry in the JSON-formatted audit log. Each blob has one root object called **records** that contains an array of log objects.

```
{
"records":
  [
    . . . .
    ,
    {
         "time": "2016-07-28T19:15:16.245Z",
         "resourceId":
"/SUBSCRIPTIONS/<subscription_id>/RESOURCEGROUPS/<resource_group_name>/PROVIDERS/MICROSOFT.DATALAKEANALYTICS/A
CCOUNTS/<data_lake_ANALYTICS_account_name>",
         "category": "Audit",
         "operationName": "JobSubmitted",
         "identity": "user@somewhere.com",
         "properties": {
             "JobId":"D74B928F-5194-4E6C-971F-C27026C290E6",
             "JobName": "New Job",
             "JobRuntimeName": "default",
             "SubmitTime": "7/28/2016 7:14:57 PM"
             }
    }
    ,
    . . . .
  ]
}
```

**Audit log schema**

| NAME | TYPE | DESCRIPTION |
| --- | --- | --- |
| time | String | The timestamp (in UTC) of the log |
| resourceId | String | The identifier of the resource that operation took place on |
| category | String | The log category. For example, **Audit**. |
| operationName | String | Name of the operation that is logged. For example, JobSubmitted. |
| resultType | String | A substatus for the job status (operationName). |
| resultSignature | String | Additional details on the job status (operationName). |
| identity | String | The user that requested the operation. For example, susan@contoso.com. |
| properties | JSON | See the next section (Audit log properties schema) for details |

> **NOTE**
>
> **resultType** and **resultSignature** provide information on the result of an operation, and only contain a value if an operation has completed. For example, they only contain a value when **operationName** contains a value of **JobStarted** or **JobEnded**.

**Audit log properties schema**

| NAME | TYPE | DESCRIPTION |
| --- | --- | --- |
| JobId | String | The ID assigned to the job |
| JobName | String | The name that was provided for the job |
| JobRunTime | String | The runtime used to process the job |
| SubmitTime | String | The time (in UTC) that the job was submitted |
| StartTime | String | The time the job started running after submission (in UTC) |
| EndTime | String | The time the job ended |
| Parallelism | String | The number of Data Lake Analytics units requested for this job during submission |

> **NOTE**
> **SubmitTime**, **StartTime**, **EndTime**, and **Parallelism** provide information on an operation. These entries only contain a value if that operation has started or completed. For example, **SubmitTime** only contains a value after **operationName** has the value **JobSubmitted**.

## Process the log data

Azure Data Lake Analytics provides a sample on how to process and analyze the log data. You can find the sample at https://github.com/Azure/AzureDataLake/tree/master/Samples/AzureDiagnosticsSample.

## Next steps

- Overview of Azure Data Lake Analytics

# Adjust quotas and limits in Azure Data Lake Analytics

8/27/2018 • 2 minutes to read • Edit Online

Learn how to adjust and increase the quota and limits in Azure Data Lake Analytics (ADLA) accounts. Knowing these limits may help you understand your U-SQL job behavior. All quota limits are soft, so you can increase the maximum limits by contacting Azure support.

## Azure subscriptions limits

**Maximum number of ADLA accounts per subscription per region:** 5

If you try to create a sixth ADLA account, you will get an error "You have reached the maximum number of Data Lake Analytics accounts allowed (5) in region under subscription name".

If you want to go beyond this limit, you can try these options:

- choose another region if suitable
- contact Azure support by opening a support ticket to request a quota increase.

## Default ADLA account limits

**Maximum number of Analytics Units (AUs) per account:** 32

This is the maximum number of AUs that can run concurrently in your account. If your total number of running AUs across all jobs exceeds this limit, newer jobs are queued automatically. For example:

- If you have only one job running with 32 AUs, when you submit a second job it will wait in the job queue until the first job completes.
- If you already have four jobs running and each is using 8 AUs, when you submit a fifth job that needs 8 AUs it waits in the job queue until there are 8 AUs available.

**Maximum number of Analytics Units (AUs) per job:** 32

This is the default maximum number of AUs that each individual job can be assigned in your account. Jobs that are assigned more than this limit will be rejected, unless the submitter is affected by a compute policy (job submission limit) that gives them more AUs per job. The upper bound of this value is the AU limit for the account.

**Maximum number of concurrent U-SQL jobs per account:** 20

This is the maximum number of jobs that can run concurrently in your account. Above this value, newer jobs are queued automatically.

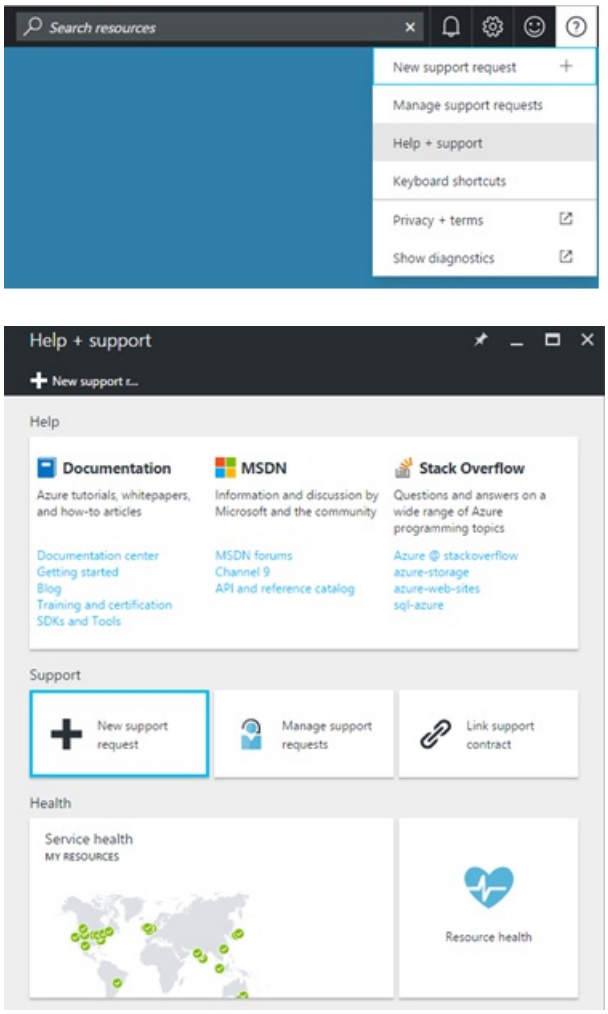
## Adjust ADLA account limits

1. Sign on to the Azure portal.
2. Choose an existing ADLA account.
3. Click **Properties**.
4. Adjust the values for **Maximum AUs**, **Maximum number of running jobs**, and **Job submission limits** to suit your needs.

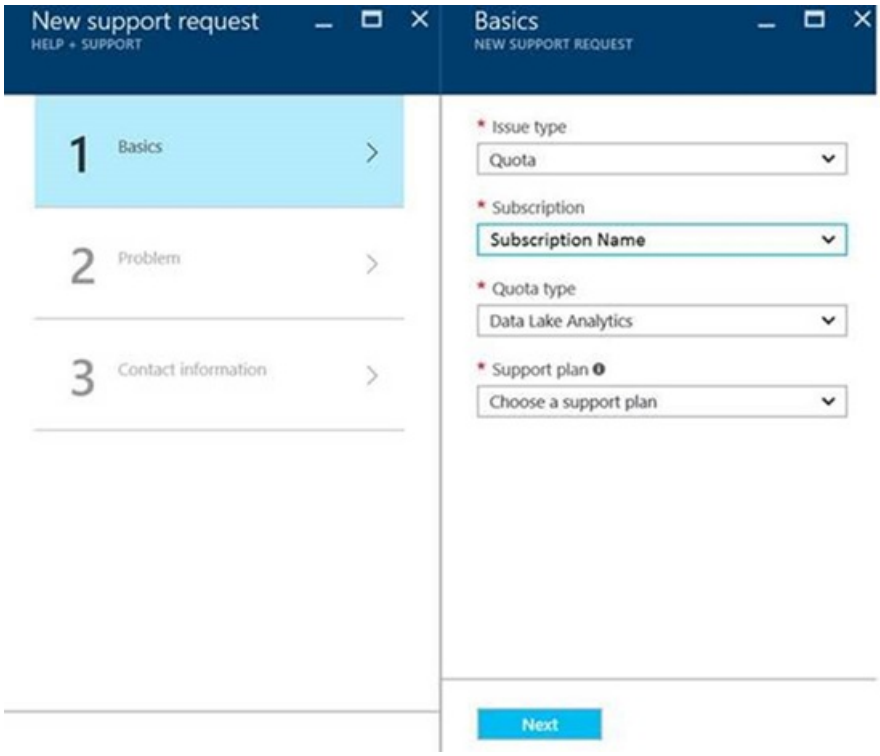
## Increase maximum quota limits

You can find more information about Azure limits in the Azure service-specific limits documentation.

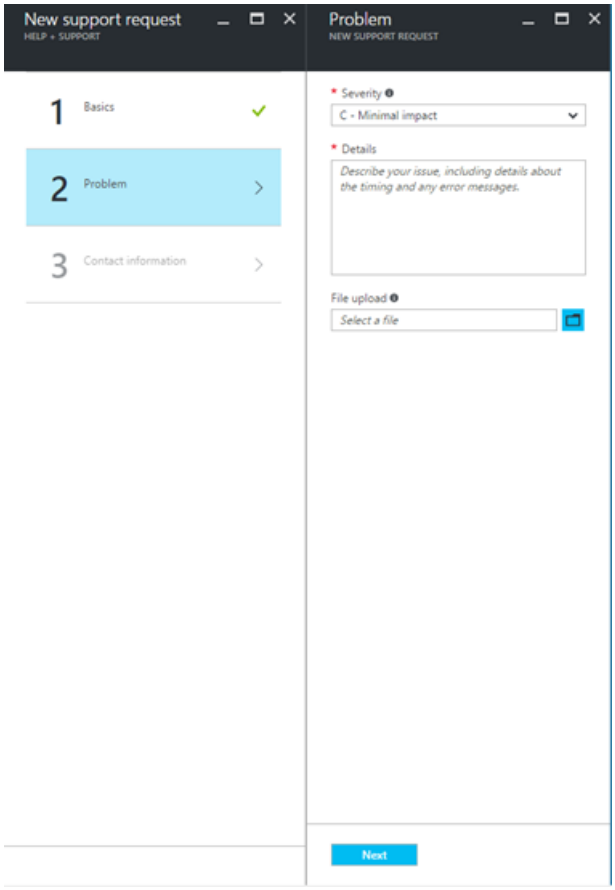
1. Open a support request in Azure portal.




2. Select the issue type **Quota**.
3. Select your **Subscription** (make sure it is not a "trial" subscription).
4. Select quota type **Data Lake Analytics**.



5. In the problem page, explain your requested increase limit with **Details** of why you need this extra capacity.




6. Verify your contact information and create the support request.

Microsoft reviews your request and tries to accommodate your business needs as soon as possible.

## Next steps

- Overview of Microsoft Azure Data Lake Analytics
- Manage Azure Data Lake Analytics using Azure PowerShell
- Monitor and troubleshoot Azure Data Lake Analytics jobs using Azure portal

# Get started with U-SQL in Azure Data Lake Analytics

9/26/2018 • 4 minutes to read • Edit Online

U-SQL is a language that combines declarative SQL with imperative C# to let you process data at any scale. Through the scalable, distributed-query capability of U-SQL, you can efficiently analyze data across relational stores such as Azure SQL Database. With U-SQL, you can process unstructured data by applying schema on read and inserting custom logic and UDFs. Additionally, U-SQL includes extensibility that gives you fine-grained control over how to execute at scale.

## Learning resources

- The U-SQL Tutorial provides a guided walkthrough of most of the U-SQL language. This document is recommended reading for all developers wanting to learn U-SQL.
- For detailed information about the **U-SQL language syntax**, see the U-SQL Language Reference.
- To understand the **U-SQL design philosophy**, see the Visual Studio blog post Introducing U-SQL – A Language that makes Big Data Processing Easy.

## Prerequisites

Before you go through the U-SQL samples in this document, read and complete Tutorial: Develop U-SQL scripts using Data Lake Tools for Visual Studio. That tutorial explains the mechanics of using U-SQL with Azure Data Lake Tools for Visual Studio.

## Your first U-SQL script

The following U-SQL script is simple and lets us explore many aspects the U-SQL language.

```
@searchlog =
    EXTRACT UserId          int,
            Start           DateTime,
            Region          string,
            Query           string,
            Duration        int?,
            Urls            string,
            ClickedUrls     string
    FROM "/Samples/Data/SearchLog.tsv"
    USING Extractors.Tsv();

OUTPUT @searchlog
    TO "/output/SearchLog-first-u-sql.csv"
    USING Outputters.Csv();
```

This script doesn't have any transformation steps. It reads from the source file called `SearchLog.tsv`, schematizes it, and writes the rowset back into a file called SearchLog-first-u-sql.csv.

Notice the question mark next to the data type in the `Duration` field. It means that the `Duration` field could be null.

**Key concepts**

- **Rowset variables**: Each query expression that produces a rowset can be assigned to a variable. U-SQL follows the T-SQL variable naming pattern (`@searchlog`, for example) in the script.
- The **EXTRACT** keyword reads data from a file and defines the schema on read. `Extractors.Tsv` is a built-in

U-SQL extractor for tab-separated-value files. You can develop custom extractors.

- The **OUTPUT** writes data from a rowset to a file. `Outputters.Csv()` is a built-in U-SQL outputter to create a comma-separated-value file. You can develop custom outputters.

**File paths**

The EXTRACT and OUTPUT statements use file paths. File paths can be absolute or relative:

This following absolute file path refers to a file in a Data Lake Store named `mystore`:

```
adl://mystore.azuredatalakestore.net/Samples/Data/SearchLog.tsv
```

This following file path starts with `"/"`. It refers to a file in the default Data Lake Store account:

```
/output/SearchLog-first-u-sql.csv
```

## Use scalar variables

You can use scalar variables as well to make your script maintenance easier. The previous U-SQL script can also be written as:

```
DECLARE @in  string = "/Samples/Data/SearchLog.tsv";
DECLARE @out string = "/output/SearchLog-scalar-variables.csv";

@searchlog =
    EXTRACT UserId          int,
            Start           DateTime,
            Region          string,
            Query           string,
            Duration        int?,
            Urls            string,
            ClickedUrls     string
    FROM @in
    USING Extractors.Tsv();

OUTPUT @searchlog
    TO @out
    USING Outputters.Csv();
```

## Transform rowsets

Use **SELECT** to transform rowsets:

```
@searchlog =
    EXTRACT UserId          int,
            Start           DateTime,
            Region          string,
            Query           string,
            Duration        int?,
            Urls            string,
            ClickedUrls     string
    FROM "/Samples/Data/SearchLog.tsv"
    USING Extractors.Tsv();

@rs1 =
    SELECT Start, Region, Duration
    FROM @searchlog
WHERE Region == "en-gb";

OUTPUT @rs1
    TO "/output/SearchLog-transform-rowsets.csv"
    USING Outputters.Csv();
```

The WHERE clause uses a C# Boolean expression. You can use the C# expression language to do your own expressions and functions. You can even perform more complex filtering by combining them with logical conjunctions (ANDs) and disjunctions (ORs).

The following script uses the DateTime.Parse() method and a conjunction.

```
@searchlog =
    EXTRACT UserId          int,
            Start           DateTime,
            Region          string,
            Query           string,
            Duration        int?,
            Urls            string,
            ClickedUrls     string
    FROM "/Samples/Data/SearchLog.tsv"
    USING Extractors.Tsv();

@rs1 =
    SELECT Start, Region, Duration
    FROM @searchlog
WHERE Region == "en-gb";

@rs1 =
    SELECT Start, Region, Duration
    FROM @rs1
    WHERE Start >= DateTime.Parse("2012/02/16") AND Start <= DateTime.Parse("2012/02/17");

OUTPUT @rs1
    TO "/output/SearchLog-transform-datetime.csv"
    USING Outputters.Csv();
```

**NOTE**

The second query is operating on the result of the first rowset, which creates a composite of the two filters. You can also reuse a variable name, and the names are scoped lexically.

## Aggregate rowsets

U-SQL gives you the familiar ORDER BY, GROUP BY, and aggregations.

The following query finds the total duration per region, and then displays the top five durations in order.

U-SQL rowsets do not preserve their order for the next query. Thus, to order an output, you need to add ORDER BY to the OUTPUT statement:

```
DECLARE @outpref string = "/output/Searchlog-aggregation";
DECLARE @out1    string = @outpref+"_agg.csv";
DECLARE @out2    string = @outpref+"_top5agg.csv";

@searchlog =
    EXTRACT UserId          int,
            Start           DateTime,
            Region          string,
            Query           string,
            Duration        int?,
            Urls            string,
            ClickedUrls     string
    FROM "/Samples/Data/SearchLog.tsv"
    USING Extractors.Tsv();

@rs1 =
    SELECT
        Region,
        SUM(Duration) AS TotalDuration
    FROM @searchlog
GROUP BY Region;

@res =
    SELECT *
    FROM @rs1
    ORDER BY TotalDuration DESC
    FETCH 5 ROWS;

OUTPUT @rs1
    TO @out1
    ORDER BY TotalDuration DESC
    USING Outputters.Csv();

OUTPUT @res
    TO @out2
    ORDER BY TotalDuration DESC
    USING Outputters.Csv();
```

The U-SQL ORDER BY clause requires using the FETCH clause in a SELECT expression.

The U-SQL HAVING clause can be used to restrict the output to groups that satisfy the HAVING condition:

```
@searchlog =
    EXTRACT UserId          int,
            Start           DateTime,
            Region          string,
            Query           string,
            Duration        int?,
            Urls            string,
            ClickedUrls     string
    FROM "/Samples/Data/SearchLog.tsv"
    USING Extractors.Tsv();

@res =
    SELECT
        Region,
        SUM(Duration) AS TotalDuration
    FROM @searchlog
    GROUP BY Region
    HAVING SUM(Duration) > 200;

OUTPUT @res
    TO "/output/Searchlog-having.csv"
    ORDER BY TotalDuration DESC
    USING Outputters.Csv();
```

For advanced aggregation scenarios, see the U-SQL reference documentation for aggregate, analytic, and reference functions

## Next steps

- Overview of Microsoft Azure Data Lake Analytics
- Develop U-SQL scripts by using Data Lake Tools for Visual Studio

# Get started with the U-SQL Catalog in Azure Data Lake Analytics

8/27/2018 • 2 minutes to read • Edit Online

## Create a TVF

In the previous U-SQL script, you repeated the use of EXTRACT to read from the same source file. With the U-SQL table-valued function (TVF), you can encapsulate the data for future reuse.

The following script creates a TVF called `Searchlog()` in the default database and schema:

```
DROP FUNCTION IF EXISTS Searchlog;

CREATE FUNCTION Searchlog()
RETURNS @searchlog TABLE
(
            UserId          int,
            Start           DateTime,
            Region          string,
            Query           string,
            Duration        int?,
            Urls            string,
            ClickedUrls     string
)
AS BEGIN
@searchlog =
    EXTRACT UserId          int,
            Start           DateTime,
            Region          string,
            Query           string,
            Duration        int?,
            Urls            string,
            ClickedUrls     string
    FROM "/Samples/Data/SearchLog.tsv"
USING Extractors.Tsv();
RETURN;
END;
```

The following script shows you how to use the TVF that was defined in the previous script:

```
@res =
    SELECT
        Region,
        SUM(Duration) AS TotalDuration
    FROM Searchlog() AS S
GROUP BY Region
HAVING SUM(Duration) > 200;

OUTPUT @res
    TO "/output/SerachLog-use-tvf.csv"
    ORDER BY TotalDuration DESC
    USING Outputters.Csv();
```

## Create views

If you have a single query expression, instead of a TVF you can use a U-SQL VIEW to encapsulate that expression.

The following script creates a view called `SearchlogView` in the default database and schema:

```
DROP VIEW IF EXISTS SearchlogView;

CREATE VIEW SearchlogView AS
    EXTRACT UserId          int,
            Start           DateTime,
            Region          string,
            Query           string,
            Duration        int?,
            Urls            string,
            ClickedUrls     string
    FROM "/Samples/Data/SearchLog.tsv"
USING Extractors.Tsv();
```

The following script demonstrates the use of the defined view:

```
@res =
    SELECT
        Region,
        SUM(Duration) AS TotalDuration
    FROM SearchlogView
GROUP BY Region
HAVING SUM(Duration) > 200;

OUTPUT @res
    TO "/output/Searchlog-use-view.csv"
    ORDER BY TotalDuration DESC
    USING Outputters.Csv();
```

## Create tables

As with relational database tables, with U-SQL you can create a table with a predefined schema or create a table that infers the schema from the query that populates the table (also known as CREATE TABLE AS SELECT or CTAS).

Create a database and two tables by using the following script:

```
DROP DATABASE IF EXISTS SearchLogDb;
CREATE DATABASE SearchLogDb;
USE DATABASE SearchLogDb;

DROP TABLE IF EXISTS SearchLog1;
DROP TABLE IF EXISTS SearchLog2;

CREATE TABLE SearchLog1 (
            UserId          int,
            Start           DateTime,
            Region          string,
            Query           string,
            Duration        int?,
            Urls            string,
            ClickedUrls     string,

            INDEX sl_idx CLUSTERED (UserId ASC)
                DISTRIBUTED BY HASH (UserId)
);

INSERT INTO SearchLog1 SELECT * FROM master.dbo.Searchlog() AS s;

CREATE TABLE SearchLog2(
    INDEX sl_idx CLUSTERED (UserId ASC)
            DISTRIBUTED BY HASH (UserId)
) AS SELECT * FROM master.dbo.Searchlog() AS S; // You can use EXTRACT or SELECT here
```

## Query tables

You can query tables, such as those created in the previous script, in the same way that you query the data files. Instead of creating a rowset by using EXTRACT, you now can refer to the table name.

To read from the tables, modify the transform script that you used previously:

```
@rs1 =
    SELECT
        Region,
        SUM(Duration) AS TotalDuration
    FROM SearchLogDb.dbo.SearchLog2
GROUP BY Region;

@res =
    SELECT *
    FROM @rs1
    ORDER BY TotalDuration DESC
    FETCH 5 ROWS;

OUTPUT @res
    TO "/output/Searchlog-query-table.csv"
    ORDER BY TotalDuration DESC
    USING Outputters.Csv();
```

> **NOTE**
>
> Currently, you cannot run a SELECT on a table in the same script as the one where you created the table.

## Next Steps

- Overview of Microsoft Azure Data Lake Analytics
- Develop U-SQL scripts using Data Lake Tools for Visual Studio

- Monitor and troubleshoot Azure Data Lake Analytics jobs using Azure portal

# Develop U-SQL user-defined operators (UDOs)

8/27/2018 • 2 minutes to read • Edit Online

This article describes how to develop user-defined operators to process data in a U-SQL job.

## Define and use a user-defined operator in U-SQL

**To create and submit a U-SQL job**

1. From the Visual Studio select **File > New > Project > U-SQL Project**.
2. Click **OK**. Visual Studio creates a solution with a Script.usql file.
3. From **Solution Explorer**, expand Script.usql, and then double-click **Script.usql.cs**.
4. Paste the following code into the file:

```
using Microsoft.Analytics.Interfaces;
using System.Collections.Generic;

namespace USQL_UDO
{
    public class CountryName : IProcessor
    {
        private static IDictionary<string, string> CountryTranslation = new Dictionary<string, string>
        {
            {
                "Deutschland", "Germany"
            },
            {
                "Suisse", "Switzerland"
            },
            {
                "UK", "United Kingdom"
            },
            {
                "USA", "United States of America"
            },
            {
                "中国", "PR China"
            }
        };

        public override IRow Process(IRow input, IUpdatableRow output)
        {

            string UserID = input.Get<string>("UserID");
            string Name = input.Get<string>("Name");
            string Address = input.Get<string>("Address");
            string City = input.Get<string>("City");
            string State = input.Get<string>("State");
            string PostalCode = input.Get<string>("PostalCode");
            string Country = input.Get<string>("Country");
            string Phone = input.Get<string>("Phone");

            if (CountryTranslation.Keys.Contains(Country))
            {
                Country = CountryTranslation[Country];
            }
            output.Set<string>(0, UserID);
            output.Set<string>(1, Name);
            output.Set<string>(2, Address);
            output.Set<string>(3, City);
            output.Set<string>(4, State);
            output.Set<string>(5, PostalCode);
            output.Set<string>(6, Country);
            output.Set<string>(7, Phone);

            return output.AsReadOnly();
        }
    }
}
```

5. Open **Script.usql**, and paste the following U-SQL script:

```
@drivers =
    EXTRACT UserID      string,
            Name        string,
            Address     string,
            City        string,
            State       string,
            PostalCode  string,
            Country     string,
            Phone       string
    FROM "/Samples/Data/AmbulanceData/Drivers.txt"
    USING Extractors.Tsv(Encoding.Unicode);

@drivers_CountryName =
    PROCESS @drivers
    PRODUCE UserID string,
            Name string,
            Address string,
            City string,
            State string,
            PostalCode string,
            Country string,
            Phone string
    USING new USQL_UDO.CountryName();

OUTPUT @drivers_CountryName
    TO "/Samples/Outputs/Drivers.csv"
    USING Outputters.Csv(Encoding.Unicode);
```

6. Specify the Data Lake Analytics account, Database, and Schema.
7. From **Solution Explorer**, right-click **Script.usql**, and then click **Build Script**.
8. From **Solution Explorer**, right-click **Script.usql**, and then click **Submit Script**.
9. If you haven't connected to your Azure subscription, you will be prompted to enter your Azure account credentials.
10. Click **Submit**. Submission results and job link are available in the Results window when the submission is completed.
11. Click the **Refresh** button to see the latest job status and refresh the screen.

**To see the output**

1. From **Server Explorer**, expand **Azure**, expand **Data Lake Analytics**, expand your Data Lake Analytics account, expand **Storage Accounts**, right-click the Default Storage, and then click **Explorer**.
2. Expand Samples, expand Outputs, and then double-click **Drivers.csv**.

## See also

- Extending U-SQL Expressions with User-Code
- Use Data Lake Tools for Visual Studio for developing U-SQL applications

# Extend U-SQL scripts with Python code in Azure Data Lake Analytics

8/27/2018 • 2 minutes to read • Edit Online

## Prerequisites

Before you begin, ensure the Python extensions are installed in your Azure Data Lake Analytics account.

- Navigate to you Data Lake Analytics Account in the Azure portal
- In the left menu, under **GETTING STARTED** click on **Sample Scripts**
- Click **Install U-SQL Extensions** then **OK**

## Overview

Python Extensions for U-SQL enable developers to perform massively parallel execution of Python code. The following example illustrates the basic steps:

- Use the `REFERENCE ASSEMBLY` statement to enable Python extensions for the U-SQL Script
- Using the `REDUCE` operation to partition the input data on a key
- The Python extensions for U-SQL include a built-in reducer (`Extension.Python.Reducer`) that runs Python code on each vertex assigned to the reducer
- The U-SQL script contains the embedded Python code that has a function called `usqlml_main` that accepts a pandas DataFrame as input and returns a pandas DataFrame as output.

--

```
REFERENCE ASSEMBLY [ExtPython];

DECLARE @myScript = @"
def get_mentions(tweet):
    return ';'.join( ( w[1:] for w in tweet.split() if w[0]=='@' ) )

def usqlml_main(df):
    del df['time']
    del df['author']
    df['mentions'] = df.tweet.apply(get_mentions)
    del df['tweet']
    return df
";

@t  =
    SELECT * FROM
       (VALUES
           ("D1","T1","A1","@foo Hello World @bar"),
           ("D2","T2","A2","@baz Hello World @beer")
       ) AS
           D( date, time, author, tweet );

@m  =
    REDUCE @t ON date
    PRODUCE date string, mentions string
    USING new Extension.Python.Reducer(pyScript:@myScript);

OUTPUT @m
    TO "/tweetmentions.csv"
    USING Outputters.Csv();
```

# How Python Integrates with U-SQL

### Datatypes

- String and numeric columns from U-SQL are converted as-is between Pandas and U-SQL
- U-SQL Nulls are converted to and from Pandas `NA` values

### Schemas

- Index vectors in Pandas are not supported in U-SQL. All input data frames in the Python function always have a 64-bit numerical index from 0 through the number of rows minus 1.
- U-SQL datasets cannot have duplicate column names
- U-SQL datasets column names that are not strings.

### Python Versions

Only Python 3.5.1 (compiled for Windows) is supported.

### Standard Python modules

All the standard Python modules are included.

### Additional Python modules

Besides the standard Python libraries, several commonly used python libraries are included:

```
pandas
numpy
numexpr
```

### Exception Messages

Currently, an exception in Python code shows up as generic vertex failure. In the future, the U-SQL Job error

messages will display the Python exception message.

**Input and Output size limitations**

Every vertex has a limited amount of memory assigned to it. Currently, that limit is 6 GB for an AU. Because the input and output DataFrames must exist in memory in the Python code, the total size for the input and output cannot exceed 6 GB.

## See also

- Overview of Microsoft Azure Data Lake Analytics
- Develop U-SQL scripts using Data Lake Tools for Visual Studio
- Using U-SQL window functions for Azure Data Lake Analytics jobs
- Use Azure Data Lake Tools for Visual Studio Code

# Extend U-SQL scripts with R code in Azure Data Lake Analytics

8/27/2018 • 4 minutes to read • Edit Online

The following example illustrates the basic steps for deploying R code:

- Use the `REFERENCE ASSEMBLY` statement to enable R extensions for the U-SQL Script.
- Use the `REDUCE` operation to partition the input data on a key.
- The R extensions for U-SQL include a built-in reducer ( `Extension.R.Reducer` ) that runs R code on each vertex assigned to the reducer.
- Usage of dedicated named data frames called `inputFromUSQL` and `outputToUSQL` respectively to pass data between U-SQL and R. Input and output DataFrame identifier names are fixed (that is, users cannot change these predefined names of input and output DataFrame identifiers).

## Embedding R code in the U-SQL script

You can inline the R code your U-SQL script by using the command parameter of the `Extension.R.Reducer` . For example, you can declare the R script as a string variable and pass it as a parameter to the Reducer.

```
REFERENCE ASSEMBLY [ExtR];

DECLARE @myRScript = @"
inputFromUSQL$Species = as.factor(inputFromUSQL$Species)
lm.fit=lm(unclass(Species)~.-Par, data=inputFromUSQL)
#do not return readonly columns and make sure that the column names are the same in usql and r scripts,
outputToUSQL=data.frame(summary(lm.fit)$coefficients)
colnames(outputToUSQL) <- c(""Estimate"", ""StdError"", ""tValue"", ""Pr"")
outputToUSQL
";

@RScriptOutput = REDUCE … USING new Extension.R.Reducer(command:@myRScript, rReturnType:"dataframe");
```

## Keep the R code in a separate file and reference it the U-SQL script

The following example illustrates a more complex usage. In this case, the R code is deployed as a RESOURCE that is the U-SQL script.

Save this R code as a separate file.

```
load("my_model_LM_Iris.rda")
outputToUSQL=data.frame(predict(lm.fit, inputFromUSQL, interval="confidence"))
```

Use a U-SQL script to deploy that R script with the DEPLOY RESOURCE statement.

```
REFERENCE ASSEMBLY [ExtR];

DEPLOY RESOURCE @"/usqlext/samples/R/RinUSQL_PredictUsingLinearModelasDF.R";
DEPLOY RESOURCE @"/usqlext/samples/R/my_model_LM_Iris.rda";
DECLARE @IrisData string = @"/usqlext/samples/R/iris.csv";
DECLARE @OutputFilePredictions string = @"/my/R/Output/LMPredictionsIris.txt";
DECLARE @PartitionCount int = 10;

@InputData =
    EXTRACT
        SepalLength double,
        SepalWidth double,
        PetalLength double,
        PetalWidth double,
        Species string
    FROM @IrisData
    USING Extractors.Csv();

@ExtendedData =
    SELECT
        Extension.R.RandomNumberGenerator.GetRandomNumber(@PartitionCount) AS Par,
        SepalLength,
        SepalWidth,
        PetalLength,
        PetalWidth
    FROM @InputData;

// Predict Species

@RScriptOutput = REDUCE @ExtendedData ON Par
    PRODUCE Par, fit double, lwr double, upr double
    READONLY Par
    USING new Extension.R.Reducer(scriptFile:"RinUSQL_PredictUsingLinearModelasDF.R", rReturnType:"dataframe",
stringsAsFactors:false);
    OUTPUT @RScriptOutput TO @OutputFilePredictions USING Outputters.Tsv();
```

# How R Integrates with U-SQL

### Datatypes

- String and numeric columns from U-SQL are converted as-is between R DataFrame and U-SQL [supported types: `double`, `string`, `bool`, `integer`, `byte`].
- The `Factor` datatype is not supported in U-SQL.
- `byte[]` must be serialized as a base64-encoded `string`.
- U-SQL strings can be converted to factors in R code, once U-SQL create R input dataframe or by setting the reducer parameter `stringsAsFactors: true`.

### Schemas

- U-SQL datasets cannot have duplicate column names.
- U-SQL datasets column names must be strings.
- Column names must be the same in U-SQL and R scripts.
- Readonly column cannot be part of the output dataframe. Because readonly columns are automatically injected back in the U-SQL table if it is a part of output schema of UDO.

### Functional limitations

- The R Engine can't be instantiated twice in the same process.
- Currently, U-SQL does not support Combiner UDOs for prediction using partitioned models generated using Reducer UDOs. Users can declare the partitioned models as resource and use them in their R Script (see sample code `ExtR_PredictUsingLMRawStringReducer.usql`)

### R Versions

Only R 3.2.2 is supported.

### Standard R modules

```
base
boot
Class
Cluster
codetools
compiler
datasets
doParallel
doRSR
foreach
foreign
Graphics
grDevices
grid
iterators
KernSmooth
lattice
MASS
Matrix
Methods
mgcv
nlme
Nnet
Parallel
pkgXMLBuilder
RevoIOQ
revoIpe
RevoMods
RevoPemaR
RevoRpeConnector
RevoRsrConnector
RevoScaleR
RevoTreeView
RevoUtils
RevoUtilsMath
Rpart
RUnit
spatial
splines
Stats
stats4
survival
Tcltk
Tools
translations
utils
XML
```

### Input and Output size limitations

Every vertex has a limited amount of memory assigned to it. Because the input and output DataFrames must exist in memory in the R code, the total size for the input and output cannot exceed 500 MB.

### Sample Code

More sample code is available in your Data Lake Store account after you install the U-SQL Advanced Analytics extensions. The path for more sample code is: `<your_account_address>/usqlext/samples/R`.

## Deploying Custom R modules with U-SQL

First, create an R custom module and zip it and then upload the zipped R custom module file to your ADL store. In the example, we will upload magittr_1.5.zip to the root of the default ADLS account for the ADLA account we are using. Once you upload the module to ADL store, declare it as use DEPLOY RESOURCE to make it available in your U-SQL script and call `install.packages` to install it.

```
REFERENCE ASSEMBLY [ExtR];
DEPLOY RESOURCE @"/magrittr_1.5.zip";

DECLARE @IrisData string =  @"/usqlext/samples/R/iris.csv";
DECLARE @OutputFileModelSummary string = @"/R/Output/CustomePackages.txt";

// R script to run
DECLARE @myRScript = @"
# install the magrittr package,
install.packages('magrittr_1.5.zip', repos = NULL),
# load the magrittr package,
require(magrittr),
# demonstrate use of the magrittr package,
2 %>% sqrt
";

@InputData =
EXTRACT SepalLength double,
SepalWidth double,
PetalLength double,
PetalWidth double,
Species string
FROM @IrisData
USING Extractors.Csv();

@ExtendedData =
SELECT 0 AS Par,
*
FROM @InputData;

@RScriptOutput = REDUCE @ExtendedData ON Par
PRODUCE Par, RowId int, ROutput string
READONLY Par
USING new Extension.R.Reducer(command:@myRScript, rReturnType:"charactermatrix");

OUTPUT @RScriptOutput TO @OutputFileModelSummary USING Outputters.Tsv();
```

## Next Steps

- Overview of Microsoft Azure Data Lake Analytics
- Develop U-SQL scripts using Data Lake Tools for Visual Studio
- Using U-SQL window functions for Azure Data Lake Analytics jobs

# Get started with the Cognitive capabilities of U-SQL

9/14/2018 • 2 minutes to read • Edit Online

## Overview

Cognitive capabilities for U-SQL enable developers to use put intelligence in their big data programs.

The following samples using cognitive capabilities are available:

- Imaging: Detect faces
- Imaging: Detect emotion
- Imaging: Detect objects (tagging)
- Imaging: OCR (optical character recognition)
- Text: Key Phrase Extraction & Sentiment Analysis

## Registering Cognitive Extensions in U-SQL

Before you begin, follow the steps in this article to register Cognitive Extensions in U-SQL: Registering Cognitive Extensions in U-SQL.

## Next steps

- U-SQL/Cognitive Samples
- Develop U-SQL scripts using Data Lake Tools for Visual Studio
- Using U-SQL window functions for Azure Data Lake Analytics jobs

# U-SQL programmability guide

8/27/2018 • 41 minutes to read • Edit Online

U-SQL is a query language that's designed for big data-type of workloads. One of the unique features of U-SQL is the combination of the SQL-like declarative language with the extensibility and programmability that's provided by C#. In this guide, we concentrate on the extensibility and programmability of the U-SQL language that's enabled by C#.

## Requirements

Download and install Azure Data Lake Tools for Visual Studio.

## Get started with U-SQL

Look at the following U-SQL script:

```
@a  =
  SELECT * FROM
    (VALUES
       ("Contoso",   1500.0, "2017-03-39"),
       ("Woodgrove", 2700.0, "2017-04-10")
    ) AS D( customer, amount, date );

@results =
  SELECT
    customer,
    amount,
    date
  FROM @a;
```

This script defines two RowSets: `@a` and `@results`. RowSet `@results` is defined from `@a`.

## C# types and expressions in U-SQL script

A U-SQL Expression is a C# expression combined with U-SQL logical operations such `AND`, `OR`, and `NOT`. U-SQL Expressions can be used with SELECT, EXTRACT, WHERE, HAVING, GROUP BY and DECLARE. For example, the following script parses a string as a DateTime value.

```
@results =
  SELECT
    customer,
    amount,
    DateTime.Parse(date) AS date
  FROM @a;
```

The following snippet parses a string as DateTime value in a DECLARE statement.

```
DECLARE @d = DateTime.Parse("2016/01/01");
```

**Use C# expressions for data type conversions**

The following example demonstrates how you can do a datetime data conversion by using C# expressions. In this particular scenario, string datetime data is converted to standard datetime with midnight 00:00:00 time notation.

```
DECLARE @dt = "2016-07-06 10:23:15";

@rs1 =
  SELECT
    Convert.ToDateTime(Convert.ToDateTime(@dt).ToString("yyyy-MM-dd")) AS dt,
    dt AS olddt
  FROM @rs0;

OUTPUT @rs1
  TO @output_file
  USING Outputters.Text();
```

**Use C# expressions for today's date**

To pull today's date, we can use the following C# expression: `DateTime.Now.ToString("M/d/yyyy")`

Here's an example of how to use this expression in a script:

```
@rs1 =
  SELECT
    MAX(guid) AS start_id,
    MIN(dt) AS start_time,
    MIN(Convert.ToDateTime(Convert.ToDateTime(dt<@default_dt?@default_dt:dt).ToString("yyyy-MM-dd"))) AS
start_zero_time,
    MIN(USQL_Programmability.CustomFunctions.GetFiscalPeriod(dt)) AS start_fiscalperiod,
    DateTime.Now.ToString("M/d/yyyy") AS Nowdate,
    user,
    des
  FROM @rs0
  GROUP BY user, des;
```

# Using .NET assemblies

U-SQL's extensibility model relies heavily on the ability to add custom code from .NET assemblies.

**Register a .NET assembly**

Use the `CREATE ASSEMBLY` statement to place a .NET assembly into a U-SQL Database. Afterwards, U-SQL scripts can use those assemblies by using the `REFERENCE ASSEMBLY` statement.

The following code shows how to register an assembly:

```
CREATE ASSEMBLY MyDB.[MyAssembly]
   FROM "/myassembly.dll";
```

The following code shows how to reference an assembly:

```
REFERENCE ASSEMBLY MyDB.[MyAssembly];
```

Consult the assembly registration instructions that covers this topic in greater detail.

**Use assembly versioning**

Currently, U-SQL uses the .NET Framework version 4.5. So ensure that your own assemblies are compatible with that version of the runtime.

As mentioned earlier, U-SQL runs code in a 64-bit (x64) format. So make sure that your code is compiled to run on x64. Otherwise you get the incorrect format error shown earlier.

Each uploaded assembly DLL and resource file, such as a different runtime, a native assembly, or a config file, can be at most 400 MB. The total size of deployed resources, either via DEPLOY RESOURCE or via references to assemblies and their additional files, cannot exceed 3 GB.

Finally, note that each U-SQL database can only contain one version of any given assembly. For example, if you need both version 7 and version 8 of the NewtonSoft Json.Net library, you need to register them in two different databases. Furthermore, each script can only refer to one version of a given assembly DLL. In this respect, U-SQL follows the C# assembly management and versioning semantics.

## Use user-defined functions: UDF

U-SQL user-defined functions, or UDF, are programming routines that accept parameters, perform an action (such as a complex calculation), and return the result of that action as a value. The return value of UDF can only be a single scalar. U-SQL UDF can be called in U-SQL base script like any other C# scalar function.

We recommend that you initialize U-SQL user-defined functions as **public** and **static**.

```
public static string MyFunction(string param1)
{
    return "my result";
}
```

First let's look at the simple example of creating a UDF.

In this use-case scenario, we need to determine the fiscal period, including the fiscal quarter and fiscal month of the first sign-in for the specific user. The first fiscal month of the year in our scenario is June.

To calculate fiscal period, we introduce the following C# function:

```
public static string GetFiscalPeriod(DateTime dt)
{
    int FiscalMonth=0;
    if (dt.Month < 7)
    {
    FiscalMonth = dt.Month + 6;
    }
    else
    {
    FiscalMonth = dt.Month - 6;
    }

    int FiscalQuarter=0;
    if (FiscalMonth >=1 && FiscalMonth<=3)
    {
    FiscalQuarter = 1;
    }
    if (FiscalMonth >= 4 && FiscalMonth <= 6)
    {
    FiscalQuarter = 2;
    }
    if (FiscalMonth >= 7 && FiscalMonth <= 9)
    {
    FiscalQuarter = 3;
    }
    if (FiscalMonth >= 10 && FiscalMonth <= 12)
    {
    FiscalQuarter = 4;
    }

    return "Q" + FiscalQuarter.ToString() + ":P" + FiscalMonth.ToString();
}
```

It simply calculates fiscal month and quarter and returns a string value. For June, the first month of the first fiscal quarter, we use "Q1:P1". For July, we use "Q1:P2", and so on.

This is a regular C# function that we are going to use in our U-SQL project.

Here is how the code-behind section looks in this scenario:

```
using Microsoft.Analytics.Interfaces;
using Microsoft.Analytics.Types.Sql;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace USQL_Programmability
{
    public class CustomFunctions
    {
        public static string GetFiscalPeriod(DateTime dt)
        {
            int FiscalMonth=0;
            if (dt.Month < 7)
            {
                FiscalMonth = dt.Month + 6;
            }
            else
            {
                FiscalMonth = dt.Month - 6;
            }

            int FiscalQuarter=0;
            if (FiscalMonth >=1 && FiscalMonth<=3)
            {
                FiscalQuarter = 1;
            }
            if (FiscalMonth >= 4 && FiscalMonth <= 6)
            {
                FiscalQuarter = 2;
            }
            if (FiscalMonth >= 7 && FiscalMonth <= 9)
            {
                FiscalQuarter = 3;
            }
            if (FiscalMonth >= 10 && FiscalMonth <= 12)
            {
                FiscalQuarter = 4;
            }

            return "Q" + FiscalQuarter.ToString() + ":" + FiscalMonth.ToString();
        }
    }
}
```

Now we are going to call this function from the base U-SQL script. To do this, we have to provide a fully qualified name for the function, including the namespace, which in this case is NameSpace.Class.Function(parameter).

```
USQL_Programmability.CustomFunctions.GetFiscalPeriod(dt)
```

Following is the actual U-SQL base script:

```
DECLARE @input_file string = @"\usql-programmability\input_file.tsv";
DECLARE @output_file string = @"\usql-programmability\output_file.tsv";

@rs0 =
    EXTRACT
            guid Guid,
        dt DateTime,
            user String,
            des String
    FROM @input_file USING Extractors.Tsv();

DECLARE @default_dt DateTime = Convert.ToDateTime("06/01/2016");

@rs1 =
    SELECT
        MAX(guid) AS start_id,
    MIN(dt) AS start_time,
        MIN(Convert.ToDateTime(Convert.ToDateTime(dt<@default_dt?@default_dt:dt).ToString("yyyy-MM-dd"))) AS
start_zero_time,
        MIN(USQL_Programmability.CustomFunctions.GetFiscalPeriod(dt)) AS start_fiscalperiod,
        user,
        des
    FROM @rs0
    GROUP BY user, des;

OUTPUT @rs1
    TO @output_file
    USING Outputters.Text();
```

Following is the output file of the script execution:

```
0d8b9630-d5ca-11e5-8329-251efa3a2941,2016-02-11T07:04:17.2630000-08:00,2016-06-
01T00:00:00.0000000,"Q3:8","User1",""

20843640-d771-11e5-b87b-8b7265c75a44,2016-02-11T07:04:17.2630000-08:00,2016-06-
01T00:00:00.0000000,"Q3:8","User2",""

301f23d2-d690-11e5-9a98-4b4f60a1836f,2016-02-11T09:01:33.9720000-08:00,2016-06-
01T00:00:00.0000000,"Q3:8","User3",""
```

This example demonstrates a simple usage of inline UDF in U-SQL.

**Keep state between UDF invocations**

U-SQL C# programmability objects can be more sophisticated, utilizing interactivity through the code-behind global variables. Let's look at the following business use-case scenario.

In large organizations, users can switch between varieties of internal applications. These can include Microsoft Dynamics CRM, PowerBI, and so on. Customers might want to apply a telemetry analysis of how users switch between different applications, what the usage trends are, and so on. The goal for the business is to optimize application usage. They also might want to combine different applications or specific sign-on routines.

To achieve this goal, we have to determine session IDs and lag time between the last session that occurred.

We need to find a previous sign-in and then assign this sign-in to all sessions that are being generated to the same application. The first challenge is that U-SQL base script doesn't allow us to apply calculations over already-calculated columns with LAG function. The second challenge is that we have to keep the specific session for all sessions within the same time period.

To solve this problem, we use a global variable inside a code-behind section: `static public string globalSession;`.

This global variable is applied to the entire rowset during our script execution.

Here is the code-behind section of our U-SQL program:

```
using Microsoft.Analytics.Interfaces;
using Microsoft.Analytics.Types.Sql;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace USQLApplication21
{
    public class UserSession
    {
        static public string globalSession;
        static public string StampUserSession(string eventTime, string PreviousRow, string Session)
        {

            if (!string.IsNullOrEmpty(PreviousRow))
            {
                double timeGap =
Convert.ToDateTime(eventTime).Subtract(Convert.ToDateTime(PreviousRow)).TotalMinutes;
                if (timeGap <= 60) {return Session;}
                else {return Guid.NewGuid().ToString();}
            }
            else {return Guid.NewGuid().ToString();}

        }

        static public string getStampUserSession(string Session)
        {
            if (Session != globalSession && !string.IsNullOrEmpty(Session)) { globalSession = Session; }
            return globalSession;
        }

    }
}
```

This example shows the global variable `static public string globalSession;` used inside the `getStampUserSession` function and getting reinitialized each time the Session parameter is changed.

The U-SQL base script is as follows:

```
DECLARE @in string = @"\UserSession\test1.tsv";
DECLARE @out1 string = @"\UserSession\Out1.csv";
DECLARE @out2 string = @"\UserSession\Out2.csv";
DECLARE @out3 string = @"\UserSession\Out3.csv";

@records =
    EXTRACT DataId string,
            EventDateTime string,
            UserName string,
            UserSessionTimestamp string

    FROM @in
    USING Extractors.Tsv();

@rs1 =
    SELECT
        EventDateTime,
        UserName,
    LAG(EventDateTime, 1)
        OVER(PARTITION BY UserName ORDER BY EventDateTime ASC) AS prevDateTime,
        string.IsNullOrEmpty(LAG(EventDateTime, 1)
        OVER(PARTITION BY UserName ORDER BY EventDateTime ASC)) AS Flag,
        USQLApplication21.UserSession.StampUserSession
           (
            EventDateTime,
            LAG(EventDateTime, 1) OVER(PARTITION BY UserName ORDER BY EventDateTime ASC),
            LAG(UserSessionTimestamp, 1) OVER(PARTITION BY UserName ORDER BY EventDateTime ASC)
           ) AS UserSessionTimestamp
    FROM @records;

@rs2 =
    SELECT
        EventDateTime,
        UserName,
        LAG(EventDateTime, 1)
        OVER(PARTITION BY UserName ORDER BY EventDateTime ASC) AS prevDateTime,
        string.IsNullOrEmpty( LAG(EventDateTime, 1) OVER(PARTITION BY UserName ORDER BY EventDateTime ASC)) AS
Flag,
        USQLApplication21.UserSession.getStampUserSession(UserSessionTimestamp) AS UserSessionTimestamp
    FROM @rs1
    WHERE UserName != "UserName";

OUTPUT @rs2
    TO @out2
    ORDER BY UserName, EventDateTime ASC
    USING Outputters.Csv();
```

Function `USQLApplication21.UserSession.getStampUserSession(UserSessionTimestamp)` is called here during the second memory rowset calculation. It passes the `UserSessionTimestamp` column and returns the value until `UserSessionTimestamp` has changed.

The output file is as follows:

```
"2016-02-19T07:32:36.8420000-08:00","User1",,True,"72a0660e-22df-428e-b672-e0977007177f"
"2016-02-17T11:52:43.6350000-08:00","User2",,True,"4a0cd19a-6e67-4d95-a119-4eda590226ba"
"2016-02-17T11:59:08.8320000-08:00","User2","2016-02-17T11:52:43.6350000-08:00",False,"4a0cd19a-6e67-4d95-a119-
4eda590226ba"
"2016-02-11T07:04:17.2630000-08:00","User3",,True,"51860a7a-1610-4f74-a9ea-69d5eef7cd9c"
"2016-02-11T07:10:33.9720000-08:00","User3","2016-02-11T07:04:17.2630000-08:00",False,"51860a7a-1610-4f74-a9ea-
69d5eef7cd9c"
"2016-02-15T21:27:41.8210000-08:00","User3","2016-02-11T07:10:33.9720000-08:00",False,"4d2bc48d-bdf3-4591-a9c1-
7b15ceb8e074"
"2016-02-16T05:48:49.6360000-08:00","User3","2016-02-15T21:27:41.8210000-08:00",False,"dd3006d0-2dcd-42d0-b3a2-
bc03dd77c8b9"
"2016-02-16T06:22:43.6390000-08:00","User3","2016-02-16T05:48:49.6360000-08:00",False,"dd3006d0-2dcd-42d0-b3a2-
bc03dd77c8b9"
"2016-02-17T16:29:53.2280000-08:00","User3","2016-02-16T06:22:43.6390000-08:00",False,"2fa899c7-eecf-4b1b-a8cd-
30c5357b4f3a"
"2016-02-17T16:39:07.2430000-08:00","User3","2016-02-17T16:29:53.2280000-08:00",False,"2fa899c7-eecf-4b1b-a8cd-
30c5357b4f3a"
"2016-02-17T17:20:39.3220000-08:00","User3","2016-02-17T16:39:07.2430000-08:00",False,"2fa899c7-eecf-4b1b-a8cd-
30c5357b4f3a"
"2016-02-19T05:23:54.5710000-08:00","User3","2016-02-17T17:20:39.3220000-08:00",False,"6ca7ed80-c149-4c22-b24b-
94ff5b0d824d"
"2016-02-19T05:48:37.7510000-08:00","User3","2016-02-19T05:23:54.5710000-08:00",False,"6ca7ed80-c149-4c22-b24b-
94ff5b0d824d"
"2016-02-19T06:40:27.4830000-08:00","User3","2016-02-19T05:48:37.7510000-08:00",False,"6ca7ed80-c149-4c22-b24b-
94ff5b0d824d"
"2016-02-19T07:27:37.7550000-08:00","User3","2016-02-19T06:40:27.4830000-08:00",False,"6ca7ed80-c149-4c22-b24b-
94ff5b0d824d"
"2016-02-19T19:35:40.9450000-08:00","User3","2016-02-19T07:27:37.7550000-08:00",False,"3f385f0b-3e68-4456-ac74-
ff6cef093674"
"2016-02-20T00:07:37.8250000-08:00","User3","2016-02-19T19:35:40.9450000-08:00",False,"685f76d5-ca48-4c58-b77d-
bd3a9ddb33da"
"2016-02-11T09:01:33.9720000-08:00","User4",,True,"9f0cf696-c8ba-449a-8d5f-1ca6ed8f2ee8"
"2016-02-17T06:30:38.6210000-08:00","User4","2016-02-11T09:01:33.9720000-08:00",False,"8b11fd2a-01bf-4a5e-a9af-
3c92c4e4382a"
"2016-02-17T22:15:26.4020000-08:00","User4","2016-02-17T06:30:38.6210000-08:00",False,"4e1cb707-3b5f-49c1-90c7-
9b33b86ca1f4"
"2016-02-18T14:37:27.6560000-08:00","User4","2016-02-17T22:15:26.4020000-08:00",False,"f4e44400-e837-40ed-8dfd-
2ea264d4e338"
"2016-02-19T01:20:31.4800000-08:00","User4","2016-02-18T14:37:27.6560000-08:00",False,"2136f4cf-7c7d-43c1-8ae2-
08f4ad6a6e08"
```

This example demonstrates a more complicated use-case scenario in which we use a global variable inside a code-behind section that's applied to the entire memory rowset.

## Use user-defined types: UDT

User-defined types, or UDT, is another programmability feature of U-SQL. U-SQL UDT acts like a regular C# user-defined type. C# is a strongly typed language that allows the use of built-in and custom user-defined types.

U-SQL cannot implicitly serialize or de-serialize arbitrary UDTs when the UDT is passed between vertices in rowsets. This means that the user has to provide an explicit formatter by using the IFormatter interface. This provides U-SQL with the serialize and de-serialize methods for the UDT.

> **NOTE**
>
> U-SQL's built-in extractors and outputters currently cannot serialize or de-serialize UDT data to or from files even with the IFormatter set. So when you're writing UDT data to a file with the OUTPUT statement, or reading it with an extractor, you have to pass it as a string or byte array. Then you call the serialization and deserialization code (that is, the UDT's ToString() method) explicitly. User-defined extractors and outputters, on the other hand, can read and write UDTs.

If we try to use UDT in EXTRACTOR or OUTPUTTER (out of previous SELECT), as shown here:

```
@rs1 =
    SELECT
        MyNameSpace.Myfunction_Returning_UDT(filed1) AS myfield
    FROM @rs0;

OUTPUT @rs1
    TO @output_file
    USING Outputters.Text();
```

We receive the following error:

```
Error   1   E_CSC_USER_INVALIDTYPEINOUTPUTTER: Outputters.Text was used to output column myfield of type
MyNameSpace.Myfunction_Returning_UDT.

Description:

Outputters.Text only supports built-in types.

Resolution:

Implement a custom outputter that knows how to serialize this type, or call a serialization method on the type
in
the preceding SELECT.   C:\Users\sergeypu\Documents\Visual Studio 2013\Projects\USQL-Programmability\
USQL-Programmability\Types.usql 52  1   USQL-Programmability
```

To work with UDT in outputter, we either have to serialize it to string with the ToString() method or create a custom outputter.

UDTs currently cannot be used in GROUP BY. If UDT is used in GROUP BY, the following error is thrown:

```
Error   1   E_CSC_USER_INVALIDTYPEINCLAUSE: GROUP BY doesn't support type MyNameSpace.Myfunction_Returning_UDT
for column myfield

Description:

GROUP BY doesn't support UDT or Complex types.

Resolution:

Add a SELECT statement where you can project a scalar column that you want to use with GROUP BY.
C:\Users\sergeypu\Documents\Visual Studio 2013\Projects\USQL-Programmability\USQL-Programmability\Types.usql
62  5   USQL-Programmability
```

To define a UDT, we have to:

- Add the following namespaces:

```
using Microsoft.Analytics.Interfaces
using System.IO;
```

- Add `Microsoft.Analytics.Interfaces`, which is required for the UDT interfaces. In addition, `System.IO` might be needed to define the IFormatter interface.
- Define a used-defined type with SqlUserDefinedType attribute.

**SqlUserDefinedType** is used to mark a type definition in an assembly as a user-defined type (UDT) in U-SQL. The properties on the attribute reflect the physical characteristics of the UDT. This class cannot be inherited.

SqlUserDefinedType is a required attribute for UDT definition.

The constructor of the class:

- SqlUserDefinedTypeAttribute (type formatter)
- Type formatter: Required parameter to define an UDT formatter--specifically, the type of the `IFormatter` interface must be passed here.

```
[SqlUserDefinedType(typeof(MyTypeFormatter))]
public class MyType
{ … }
```

- Typical UDT also requires definition of the IFormatter interface, as shown in the following example:

```
public class MyTypeFormatter : IFormatter<MyType>
{
    public void Serialize(MyType instance, IColumnWriter writer, ISerializationContext context)
    { … }

    public MyType Deserialize(IColumnReader reader, ISerializationContext context)
    { … }
}
```

The `IFormatter` interface serializes and de-serializes an object graph with the root type of <typeparamref name="T">.

<typeparam name="T">The root type for the object graph to serialize and de-serialize.

- **Deserialize**: De-serializes the data on the provided stream and reconstitutes the graph of objects.
- **Serialize**: Serializes an object, or graph of objects, with the given root to the provided stream.

`MyType` instance: Instance of the type.
`IColumnWriter` writer / `IColumnReader` reader: The underlying column stream.
`ISerializationContext` context: Enum that defines a set of flags that specifies the source or destination context for the stream during serialization.

- **Intermediate**: Specifies that the source or destination context is not a persisted store.
- **Persistence**: Specifies that the source or destination context is a persisted store.

As a regular C# type, a U-SQL UDT definition can include overrides for operators such as +/==/!=. It can also include static methods. For example, if we are going to use this UDT as a parameter to a U-SQL MIN aggregate function, we have to define < operator override.

Earlier in this guide, we demonstrated an example for fiscal period identification from the specific date in the format `Qn:Pn (Q1:P10)`. The following example shows how to define a custom type for fiscal period values.

Following is an example of a code-behind section with custom UDT and IFormatter interface:

```
[SqlUserDefinedType(typeof(FiscalPeriodFormatter))]
public struct FiscalPeriod
{
    public int Quarter { get; private set; }

    public int Month { get; private set; }

    public FiscalPeriod(int quarter, int month):this()
    {
    this.Quarter = quarter;
    this.Month = month;
```

```
    }

    public override bool Equals(object obj)
    {
    if (ReferenceEquals(null, obj))
    {
        return false;
    }

    return obj is FiscalPeriod && Equals((FiscalPeriod)obj);
    }

    public bool Equals(FiscalPeriod other)
    {
return this.Quarter.Equals(other.Quarter) && this.Month.Equals(other.Month);
    }

    public bool GreaterThan(FiscalPeriod other)
    {
return this.Quarter.CompareTo(other.Quarter) > 0 || this.Month.CompareTo(other.Month) > 0;
    }

    public bool LessThan(FiscalPeriod other)
    {
return this.Quarter.CompareTo(other.Quarter) < 0 || this.Month.CompareTo(other.Month) < 0;
    }

    public override int GetHashCode()
    {
    unchecked
    {
        return (this.Quarter.GetHashCode() * 397) ^ this.Month.GetHashCode();
    }
    }

    public static FiscalPeriod operator +(FiscalPeriod c1, FiscalPeriod c2)
    {
return new FiscalPeriod((c1.Quarter + c2.Quarter) > 4 ? (c1.Quarter + c2.Quarter)-4 : (c1.Quarter +
c2.Quarter), (c1.Month + c2.Month) > 12 ? (c1.Month + c2.Month) - 12 : (c1.Month + c2.Month));
    }

    public static bool operator ==(FiscalPeriod c1, FiscalPeriod c2)
    {
    return c1.Equals(c2);
    }

    public static bool operator !=(FiscalPeriod c1, FiscalPeriod c2)
    {
    return !c1.Equals(c2);
    }
    public static bool operator >(FiscalPeriod c1, FiscalPeriod c2)
    {
    return c1.GreaterThan(c2);
    }
    public static bool operator <(FiscalPeriod c1, FiscalPeriod c2)
    {
    return c1.LessThan(c2);
    }
    public override string ToString()
    {
    return (String.Format("Q{0}:P{1}", this.Quarter, this.Month));
    }

}

public class FiscalPeriodFormatter : IFormatter<FiscalPeriod>
{
    public void Serialize(FiscalPeriod instance, IColumnWriter writer, ISerializationContext context)
    {
```

```
    using (var binaryWriter = new BinaryWriter(writer.BaseStream))
    {
        binaryWriter.Write(instance.Quarter);
        binaryWriter.Write(instance.Month);
        binaryWriter.Flush();
    }
    }

    public FiscalPeriod Deserialize(IColumnReader reader, ISerializationContext context)
    {
    using (var binaryReader = new BinaryReader(reader.BaseStream))
    {
var result = new FiscalPeriod(binaryReader.ReadInt16(), binaryReader.ReadInt16());
        return result;
    }
    }
}
```

The defined type includes two numbers: quarter and month. Operators `==/!=/>/<` and static method `ToString()` are defined here.

As mentioned earlier, UDT can be used in SELECT expressions, but cannot be used in OUTPUTTER/EXTRACTOR without custom serialization. It either has to be serialized as a string with `ToString()` or used with a custom OUTPUTTER/EXTRACTOR.

Now let's discuss usage of UDT. In a code-behind section, we changed our GetFiscalPeriod function to the following:

```
public static FiscalPeriod GetFiscalPeriodWithCustomType(DateTime dt)
{
    int FiscalMonth = 0;
    if (dt.Month < 7)
    {
    FiscalMonth = dt.Month + 6;
    }
    else
    {
    FiscalMonth = dt.Month - 6;
    }

    int FiscalQuarter = 0;
    if (FiscalMonth >= 1 && FiscalMonth <= 3)
    {
    FiscalQuarter = 1;
    }
    if (FiscalMonth >= 4 && FiscalMonth <= 6)
    {
    FiscalQuarter = 2;
    }
    if (FiscalMonth >= 7 && FiscalMonth <= 9)
    {
    FiscalQuarter = 3;
    }
    if (FiscalMonth >= 10 && FiscalMonth <= 12)
    {
    FiscalQuarter = 4;
    }

    return new FiscalPeriod(FiscalQuarter, FiscalMonth);
}
```

As you can see, it returns the value of our FiscalPeriod type.

Here we provide an example of how to use it further in U-SQL base script. This example demonstrates different

forms of UDT invocation from U-SQL script.

```
DECLARE @input_file string = @"c:\work\cosmos\usql-programmability\input_file.tsv";
DECLARE @output_file string = @"c:\work\cosmos\usql-programmability\output_file.tsv";

@rs0 =
    EXTRACT
        guid string,
        dt DateTime,
        user String,
        des String
    FROM @input_file USING Extractors.Tsv();

@rs1 =
    SELECT
        guid AS start_id,
        dt,
        DateTime.Now.ToString("M/d/yyyy") AS Nowdate,
        USQL_Programmability.CustomFunctions.GetFiscalPeriodWithCustomType(dt).Quarter AS fiscalquarter,
        USQL_Programmability.CustomFunctions.GetFiscalPeriodWithCustomType(dt).Month AS fiscalmonth,
        USQL_Programmability.CustomFunctions.GetFiscalPeriodWithCustomType(dt) + new
USQL_Programmability.CustomFunctions.FiscalPeriod(1,7) AS fiscalperiod_adjusted,
        user,
        des
    FROM @rs0;

@rs2 =
    SELECT
           start_id,
           dt,
           DateTime.Now.ToString("M/d/yyyy") AS Nowdate,
           fiscalquarter,
           fiscalmonth,
           USQL_Programmability.CustomFunctions.GetFiscalPeriodWithCustomType(dt).ToString() AS fiscalperiod,

       // This user-defined type was created in the prior SELECT.  Passing the UDT to this subsequent SELECT
would have failed if the UDT was not annotated with an IFormatter.
           fiscalperiod_adjusted.ToString() AS fiscalperiod_adjusted,
           user,
           des
    FROM @rs1;

OUTPUT @rs2
    TO @output_file
    USING Outputters.Text();
```

Here's an example of a full code-behind section:

```
using Microsoft.Analytics.Interfaces;
using Microsoft.Analytics.Types.Sql;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;

namespace USQL_Programmability
{
    public class CustomFunctions
    {
        static public DateTime? ToDateTime(string dt)
        {
            DateTime dtValue;

            if (!DateTime.TryParse(dt, out dtValue))
                return Convert.ToDateTime(dt);
```

```
            else
                return null;
        }

        public static FiscalPeriod GetFiscalPeriodWithCustomType(DateTime dt)
        {
            int FiscalMonth = 0;
            if (dt.Month < 7)
            {
                FiscalMonth = dt.Month + 6;
            }
            else
            {
                FiscalMonth = dt.Month - 6;
            }

            int FiscalQuarter = 0;
            if (FiscalMonth >= 1 && FiscalMonth <= 3)
            {
                FiscalQuarter = 1;
            }
            if (FiscalMonth >= 4 && FiscalMonth <= 6)
            {
                FiscalQuarter = 2;
            }
            if (FiscalMonth >= 7 && FiscalMonth <= 9)
            {
                FiscalQuarter = 3;
            }
            if (FiscalMonth >= 10 && FiscalMonth <= 12)
            {
                FiscalQuarter = 4;
            }

            return new FiscalPeriod(FiscalQuarter, FiscalMonth);
        }



        [SqlUserDefinedType(typeof(FiscalPeriodFormatter))]
        public struct FiscalPeriod
        {
            public int Quarter { get; private set; }

            public int Month { get; private set; }

            public FiscalPeriod(int quarter, int month):this()
            {
                this.Quarter = quarter;
                this.Month = month;
            }

            public override bool Equals(object obj)
            {
                if (ReferenceEquals(null, obj))
                {
                    return false;
                }

                return obj is FiscalPeriod && Equals((FiscalPeriod)obj);
            }

            public bool Equals(FiscalPeriod other)
            {
return this.Quarter.Equals(other.Quarter) &&    this.Month.Equals(other.Month);
            }

            public bool GreaterThan(FiscalPeriod other)
            {
```

```
return this.Quarter.CompareTo(other.Quarter) > 0 || this.Month.CompareTo(other.Month) > 0;
            }

            public bool LessThan(FiscalPeriod other)
            {
return this.Quarter.CompareTo(other.Quarter) < 0 || this.Month.CompareTo(other.Month) < 0;
            }

            public override int GetHashCode()
            {
                unchecked
                {
                    return (this.Quarter.GetHashCode() * 397) ^ this.Month.GetHashCode();
                }
            }

            public static FiscalPeriod operator +(FiscalPeriod c1, FiscalPeriod c2)
            {
return new FiscalPeriod((c1.Quarter + c2.Quarter) > 4 ? (c1.Quarter + c2.Quarter)-4 : (c1.Quarter +
c2.Quarter), (c1.Month + c2.Month) > 12 ? (c1.Month + c2.Month) - 12 : (c1.Month + c2.Month));
            }

            public static bool operator ==(FiscalPeriod c1, FiscalPeriod c2)
            {
                return c1.Equals(c2);
            }

            public static bool operator !=(FiscalPeriod c1, FiscalPeriod c2)
            {
                return !c1.Equals(c2);
            }
            public static bool operator >(FiscalPeriod c1, FiscalPeriod c2)
            {
                return c1.GreaterThan(c2);
            }
            public static bool operator <(FiscalPeriod c1, FiscalPeriod c2)
            {
                return c1.LessThan(c2);
            }
            public override string ToString()
            {
                return (String.Format("Q{0}:P{1}", this.Quarter, this.Month));
            }

        }

        public class FiscalPeriodFormatter : IFormatter<FiscalPeriod>
        {
public void Serialize(FiscalPeriod instance, IColumnWriter writer, ISerializationContext context)
            {
                using (var binaryWriter = new BinaryWriter(writer.BaseStream))
                {
                    binaryWriter.Write(instance.Quarter);
                    binaryWriter.Write(instance.Month);
                    binaryWriter.Flush();
                }
            }

public FiscalPeriod Deserialize(IColumnReader reader, ISerializationContext context)
            {
                using (var binaryReader = new BinaryReader(reader.BaseStream))
                {
var result = new FiscalPeriod(binaryReader.ReadInt16(), binaryReader.ReadInt16());
                    return result;
                }
            }
        }
    }
}
```

## Use user-defined aggregates: UDAGG

User-defined aggregates are any aggregation-related functions that are not shipped out-of-the-box with U-SQL. The example can be an aggregate to perform custom math calculations, string concatenations, manipulations with strings, and so on.

The user-defined aggregate base class definition is as follows:

```
    [SqlUserDefinedAggregate]
    public abstract class IAggregate<T1, T2, TResult> : IAggregate
    {
        protected IAggregate();

        public abstract void Accumulate(T1 t1, T2 t2);
        public abstract void Init();
        public abstract TResult Terminate();
    }
```

**SqlUserDefinedAggregate** indicates that the type should be registered as a user-defined aggregate. This class cannot be inherited.

SqlUserDefinedType attribute is **optional** for UDAGG definition.

The base class allows you to pass three abstract parameters: two as input parameters and one as the result. The data types are variable and should be defined during class inheritance.

```
public class GuidAggregate : IAggregate<string, string, string>
{
    string guid_agg;

    public override void Init()
    { … }

    public override void Accumulate(string guid, string user)
    { … }

    public override string Terminate()
    { … }
}
```

- **Init** invokes once for each group during computation. It provides an initialization routine for each aggregation group.
- **Accumulate** is executed once for each value. It provides the main functionality for the aggregation algorithm. It can be used to aggregate values with various data types that are defined during class inheritance. It can accept two parameters of variable data types.
- **Terminate** is executed once per aggregation group at the end of processing to output the result for each group.

To declare correct input and output data types, use the class definition as follows:

```
public abstract class IAggregate<T1, T2, TResult> : IAggregate
```

- T1: First parameter to accumulate
- T2: First parameter to accumulate
- TResult: Return type of terminate

For example:

```
public class GuidAggregate : IAggregate<string, int, int>
```

or

```
public class GuidAggregate : IAggregate<string, string, string>
```

**Use UDAGG in U-SQL**

To use UDAGG, first define it in code-behind or reference it from the existent programmability DLL as discussed earlier.

Then use the following syntax:

```
AGG<UDAGG_functionname>(param1,param2)
```

Here is an example of UDAGG:

```
public class GuidAggregate : IAggregate<string, string, string>
{
    string guid_agg;

    public override void Init()
    {
        guid_agg = "";
    }

    public override void Accumulate(string guid, string user)
    {
        if (user.ToUpper()== "USER1")
        {
        guid_agg += "{" + guid + "}";
        }
    }

    public override string Terminate()
    {
        return guid_agg;
    }

}
```

And base U-SQL script:

```
DECLARE @input_file string = @"\usql-programmability\input_file.tsv";
DECLARE @output_file string = @" \usql-programmability\output_file.tsv";

@rs0 =
    EXTRACT
            guid string,
        dt DateTime,
            user String,
            des String
    FROM @input_file
    USING Extractors.Tsv();

@rs1 =
    SELECT
        user,
        AGG<USQL_Programmability.GuidAggregate>(guid,user) AS guid_list
    FROM @rs0
    GROUP BY user;

OUTPUT @rs1 TO @output_file USING Outputters.Text();
```

In this use-case scenario, we concatenate class GUIDs for the specific users.

## Use user-defined objects: UDO

U-SQL enables you to define custom programmability objects, which are called user-defined objects or UDO.

The following is a list of UDO in U-SQL:

- User-defined extractors
  - Extract row by row
  - Used to implement data extraction from custom structured files
- User-defined outputters
  - Output row by row
  - Used to output custom data types or custom file formats
- User-defined processors
  - Take one row and produce one row
  - Used to reduce the number of columns or produce new columns with values that are derived from an existing column set
- User-defined appliers
  - Take one row and produce 0 to n rows
  - Used with OUTER/CROSS APPLY
- User-defined combiners
  - Combines rowsets--user-defined JOINs
- User-defined reducers
  - Take n rows and produce one row
  - Used to reduce the number of rows

UDO is typically called explicitly in U-SQL script as part of the following U-SQL statements:

- EXTRACT
- OUTPUT
- PROCESS

- COMBINE
- REDUCE

> **NOTE**
> UDO's are limited to consume 0.5Gb memory. This memory limitation does not apply to local executions.

## Use user-defined extractors

U-SQL allows you to import external data by using an EXTRACT statement. An EXTRACT statement can use built-in UDO extractors:

- *Extractors.Text()*: Provides extraction from delimited text files of different encodings.
- *Extractors.Csv()*: Provides extraction from comma-separated value (CSV) files of different encodings.
- *Extractors.Tsv()*: Provides extraction from tab-separated value (TSV) files of different encodings.

It can be useful to develop a custom extractor. This can be helpful during data import if we want to do any of the following tasks:

- Modify input data by splitting columns and modifying individual values. The PROCESSOR functionality is better for combining columns.
- Parse unstructured data such as Web pages and emails, or semi-unstructured data such as XML/JSON.
- Parse data in unsupported encoding.

To define a user-defined extractor, or UDE, we need to create an `IExtractor` interface. All input parameters to the extractor, such as column/row delimiters, and encoding, need to be defined in the constructor of the class. The `IExtractor` interface should also contain a definition for the `IEnumerable<IRow>` override as follows:

```
[SqlUserDefinedExtractor]
public class SampleExtractor : IExtractor
{
     public SampleExtractor(string row_delimiter, char col_delimiter)
     { … }

     public override IEnumerable<IRow> Extract(IUnstructuredReader input, IUpdatableRow output)
     { … }
}
```

The **SqlUserDefinedExtractor** attribute indicates that the type should be registered as a user-defined extractor. This class cannot be inherited.

SqlUserDefinedExtractor is an optional attribute for UDE definition. It used to define AtomicFileProcessing property for the UDE object.

- bool AtomicFileProcessing
- **true** = Indicates that this extractor requires atomic input files (JSON, XML, ...)
- **false** = Indicates that this extractor can deal with split / distributed files (CSV, SEQ, ...)

The main UDE programmability objects are **input** and **output**. The input object is used to enumerate input data as `IUnstructuredReader`. The output object is used to set output data as a result of the extractor activity.

The input data is accessed through `System.IO.Stream` and `System.IO.StreamReader`.

For input columns enumeration, we first split the input stream by using a row delimiter.

```
foreach (Stream current in input.Split(my_row_delimiter))
{
…
}
```

Then, further split input row into column parts.

```
foreach (Stream current in input.Split(my_row_delimiter))
{
…
    string[] parts = line.Split(my_column_delimiter);
    foreach (string part in parts)
    { … }
}
```

To set output data, we use the `output.Set` method.

It's important to understand that the custom extractor only outputs columns and values that are defined with the output. Set method call.

```
output.Set<string>(count, part);
```

The actual extractor output is triggered by calling `yield return output.AsReadOnly();`.

Following is the extractor example:

```
[SqlUserDefinedExtractor(AtomicFileProcessing = true)]
public class FullDescriptionExtractor : IExtractor
{
     private Encoding _encoding;
     private byte[] _row_delim;
     private char _col_delim;

    public FullDescriptionExtractor(Encoding encoding, string row_delim = "\r\n", char col_delim = '\t')
    {
         this._encoding = ((encoding == null) ? Encoding.UTF8 : encoding);
         this._row_delim = this._encoding.GetBytes(row_delim);
         this._col_delim = col_delim;

    }

    public override IEnumerable<IRow> Extract(IUnstructuredReader input, IUpdatableRow output)
    {
         string line;
         //Read the input line by line
         foreach (Stream current in input.Split(_encoding.GetBytes("\r\n")))
         {
        using (System.IO.StreamReader streamReader = new StreamReader(current, this._encoding))
         {
             line = streamReader.ReadToEnd().Trim();
             //Split the input by the column delimiter
             string[] parts = line.Split(this._col_delim);
             int count = 0; // start with first column
             foreach (string part in parts)
             {
    if (count == 0)
             {  // for column “guid”, re-generated guid
                 Guid new_guid = Guid.NewGuid();
                 output.Set<Guid>(count, new_guid);
             }
             else if (count == 2)
             {
                 // for column “user”, convert to UPPER case
                 output.Set<string>(count, part.ToUpper());

             }
             else
             {
                 // keep the rest of the columns as-is
                 output.Set<string>(count, part);
             }
             count += 1;
             }

         }
         yield return output.AsReadOnly();
         }
         yield break;
     }
}
```

In this use-case scenario, the extractor regenerates the GUID for “guid” column and converts the values of “user” column to upper case. Custom extractors can produce more complicated results by parsing input data and manipulating it.

Following is base U-SQL script that uses a custom extractor:

```
DECLARE @input_file string = @"\usql-programmability\input_file.tsv";
DECLARE @output_file string = @"\usql-programmability\output_file.tsv";

@rs0 =
    EXTRACT
            guid Guid,
        dt String,
            user String,
            des String
    FROM @input_file
        USING new USQL_Programmability.FullDescriptionExtractor(Encoding.UTF8);

OUTPUT @rs0 TO @output_file USING Outputters.Text();
```

## Use user-defined outputters

User-defined outputter is another U-SQL UDO that allows you to extend built-in U-SQL functionality. Similar to the extractor, there are several built-in outputters.

- *Outputters.Text()*: Writes data to delimited text files of different encodings.
- *Outputters.Csv()*: Writes data to comma-separated value (CSV) files of different encodings.
- *Outputters.Tsv()*: Writes data to tab-separated value (TSV) files of different encodings.

Custom outputter allows you to write data in a custom defined format. This can be useful for the following tasks:

- Writing data to semi-structured or unstructured files.
- Writing data not supported encodings.
- Modifying output data or adding custom attributes.

To define user-defined outputter, we need to create the `IOutputter` interface.

Following is the base `IOutputter` class implementation:

```
public abstract class IOutputter : IUserDefinedOperator
{
    protected IOutputter();

    public virtual void Close();
    public abstract void Output(IRow input, IUnstructuredWriter output);
}
```

All input parameters to the outputter, such as column/row delimiters, encoding, and so on, need to be defined in the constructor of the class. The `IOutputter` interface should also contain a definition for `void Output` override. The attribute `[SqlUserDefinedOutputter(AtomicFileProcessing = true)` can optionally be set for atomic file processing. For more information, see the following details.

```
[SqlUserDefinedOutputter(AtomicFileProcessing = true)]
public class MyOutputter : IOutputter
{

    public MyOutputter(myparam1, myparam2)
    {
      …
    }

    public override void Close()
    {
      …
    }

    public override void Output(IRow row, IUnstructuredWriter output)
    {
      …
    }
}
```

- `Output` is called for each input row. It returns the `IUnstructuredWriter output` rowset.
- The Constructor class is used to pass parameters to the user-defined outputter.
- `Close` is used to optionally override to release expensive state or determine when the last row was written.

**SqlUserDefinedOutputter** attribute indicates that the type should be registered as a user-defined outputter. This class cannot be inherited.

SqlUserDefinedOutputter is an optional attribute for a user-defined outputter definition. It's used to define the AtomicFileProcessing property.

- bool AtomicFileProcessing
- **true** = Indicates that this outputter requires atomic output files (JSON, XML, ...)
- **false** = Indicates that this outputter can deal with split / distributed files (CSV, SEQ, ...)

The main programmability objects are **row** and **output**. The **row** object is used to enumerate output data as `IRow` interface. **Output** is used to set output data to the target file.

The output data is accessed through the `IRow` interface. Output data is passed a row at a time.

The individual values are enumerated by calling the Get method of the IRow interface:

```
row.Get<string>("column_name")
```

Individual column names can be determined by calling `row.Schema`:

```
ISchema schema = row.Schema;
var col = schema[i];
string val = row.Get<string>(col.Name)
```

This approach enables you to build a flexible outputter for any metadata schema.

The output data is written to file by using `System.IO.StreamWriter`. The stream parameter is set to `output.BaseStrea` as part of `IUnstructuredWriter output`.

Note that it's important to flush the data buffer to the file after each row iteration. In addition, the `StreamWriter` object must be used with the Disposable attribute enabled (default) and with the **using** keyword:

```
using (StreamWriter streamWriter = new StreamWriter(output.BaseStream, this._encoding))
{
…
}
```

Otherwise, call Flush() method explicitly after each iteration. We show this in the following example.

**Set headers and footers for user-defined outputter**

To set a header, use single iteration execution flow.

```
public override void Output(IRow row, IUnstructuredWriter output)
{
 …
if (isHeaderRow)
{
    …
}

 …
if (isHeaderRow)
{
    isHeaderRow = false;
}
 …
}
}
```

The code in the first `if (isHeaderRow)` block is executed only once.

For the footer, use the reference to the instance of `System.IO.Stream` object ( `output.BaseStream` ). Write the footer in the Close() method of the `IOutputter` interface. (For more information, see the following example.)

Following is an example of a user-defined outputter:

```
[SqlUserDefinedOutputter(AtomicFileProcessing = true)]
public class HTMLOutputter : IOutputter
{
    // Local variables initialization
    private string row_delimiter;
    private char col_delimiter;
    private bool isHeaderRow;
    private Encoding encoding;
    private bool IsTableHeader = true;
    private Stream g_writer;

    // Parameters definition
    public HTMLOutputter(bool isHeader = false, Encoding encoding = null)
    {
    this.isHeaderRow = isHeader;
    this.encoding = ((encoding == null) ? Encoding.UTF8 : encoding);
    }

    // The Close method is used to write the footer to the file. It's executed only once, after all rows
    public override void Close()
    {
    //Reference to IO.Stream object - g_writer
    StreamWriter streamWriter = new StreamWriter(g_writer, this.encoding);
    streamWriter.Write("</table>");
    streamWriter.Flush();
    streamWriter.Close();
    }

    public override void Output(IRow row, IUnstructuredWriter output)
```

```
    {
    System.IO.StreamWriter streamWriter = new StreamWriter(output.BaseStream, this.encoding);

    // Metadata schema initialization to enumerate column names
    ISchema schema = row.Schema;

    // This is a data-independent header--HTML table definition
    if (IsTableHeader)
    {
        streamWriter.Write("<table border=1>");
        IsTableHeader = false;
    }

    // HTML table attributes
    string header_wrapper_on = "<th>";
    string header_wrapper_off = "</th>";
    string data_wrapper_on = "<td>";
    string data_wrapper_off = "</td>";

    // Header row output--runs only once
    if (isHeaderRow)
    {
        streamWriter.Write("<tr>");
        for (int i = 0; i < schema.Count(); i++)
        {
        var col = schema[i];
        streamWriter.Write(header_wrapper_on + col.Name + header_wrapper_off);
        }
        streamWriter.Write("</tr>");
    }

    // Data row output
    streamWriter.Write("<tr>");
    for (int i = 0; i < schema.Count(); i++)
    {
        var col = schema[i];
        string val = "";
        try
        {
        // Data type enumeration--required to match the distinct list of types from OUTPUT statement
        switch (col.Type.Name.ToString().ToLower())
        {
            case "string": val = row.Get<string>(col.Name).ToString(); break;
            case "guid": val = row.Get<Guid>(col.Name).ToString(); break;
            default: break;
        }
        }
        // Handling NULL values--keeping them empty
        catch (System.NullReferenceException)
        {
        }
        streamWriter.Write(data_wrapper_on + val + data_wrapper_off);
    }
    streamWriter.Write("</tr>");

    if (isHeaderRow)
    {
        isHeaderRow = false;
    }
    // Reference to the instance of the IO.Stream object for footer generation
    g_writer = output.BaseStream;
    streamWriter.Flush();
    }
}

// Define the factory classes
public static class Factory
{
    public static HTMLOutputter HTMLOutputter(bool isHeader = false, Encoding encoding = null)
```

```
    {
    return new HTMLOutputter(isHeader, encoding);
    }
}
```

And U-SQL base script:

```
DECLARE @input_file string = @"\usql-programmability\input_file.tsv";
DECLARE @output_file string = @"\usql-programmability\output_file.html";

@rs0 =
    EXTRACT
            guid Guid,
        dt String,
            user String,
            des String
         FROM @input_file
         USING new USQL_Programmability.FullDescriptionExtractor(Encoding.UTF8);

OUTPUT @rs0
    TO @output_file
    USING new USQL_Programmability.HTMLOutputter(isHeader: true);
```

This is an HTML outputter, which creates an HTML file with table data.

**Call outputter from U-SQL base script**

To call a custom outputter from the base U-SQL script, the new instance of the outputter object has to be created.

```
OUTPUT @rs0 TO @output_file USING new USQL_Programmability.HTMLOutputter(isHeader: true);
```

To avoid creating an instance of the object in base script, we can create a function wrapper, as shown in our earlier example:

```
        // Define the factory classes
        public static class Factory
        {
            public static HTMLOutputter HTMLOutputter(bool isHeader = false, Encoding encoding = null)
            {
                return new HTMLOutputter(isHeader, encoding);
            }
        }
```

In this case, the original call looks like the following:

```
OUTPUT @rs0
TO @output_file
USING USQL_Programmability.Factory.HTMLOutputter(isHeader: true);
```

# Use user-defined processors

User-defined processor, or UDP, is a type of U-SQL UDO that enables you to process the incoming rows by applying programmability features. UDP enables you to combine columns, modify values, and add new columns if necessary. Basically, it helps to process a rowset to produce required data elements.

To define a UDP, we need to create an `IProcessor` interface with the `SqlUserDefinedProcessor` attribute, which is optional for UDP.

This interface should contain the definition for the `IRow` interface rowset override, as shown in the following example:

```
[SqlUserDefinedProcessor]
public class MyProcessor: IProcessor
{
public override IRow Process(IRow input, IUpdatableRow output)
 {
        …
 }
}
```

**SqlUserDefinedProcessor** indicates that the type should be registered as a user-defined processor. This class cannot be inherited.

The SqlUserDefinedProcessor attribute is **optional** for UDP definition.

The main programmability objects are **input** and **output**. The input object is used to enumerate input columns and output, and to set output data as a result of the processor activity.

For input columns enumeration, we use the `input.Get` method.

```
string column_name = input.Get<string>("column_name");
```

The parameter for `input.Get` method is a column that's passed as part of the `PRODUCE` clause of the `PROCESS` statement of the U-SQL base script. We need to use the correct data type here.

For output, use the `output.Set` method.

It's important to note that custom producer only outputs columns and values that are defined with the `output.Set` method call.

```
output.Set<string>("mycolumn", mycolumn);
```

The actual processor output is triggered by calling `return output.AsReadOnly();`.

Following is a processor example:

```
[SqlUserDefinedProcessor]
public class FullDescriptionProcessor : IProcessor
{
public override IRow Process(IRow input, IUpdatableRow output)
 {
     string user = input.Get<string>("user");
     string des = input.Get<string>("des");
     string full_description = user.ToUpper() + "=>" + des;
     output.Set<string>("dt", input.Get<string>("dt"));
     output.Set<string>("full_description", full_description);
     output.Set<Guid>("new_guid", Guid.NewGuid());
     output.Set<Guid>("guid", input.Get<Guid>("guid"));
     return output.AsReadOnly();
 }
}
```

In this use-case scenario, the processor is generating a new column called “full_description” by combining the existing columns--in this case, “user” in upper case, and “des”. It also regenerates a GUID and returns the original and new GUID values.

As you can see from the previous example, you can call C# methods during `output.Set` method call.

Following is an example of base U-SQL script that uses a custom processor:

```
DECLARE @input_file string = @"\usql-programmability\input_file.tsv";
DECLARE @output_file string = @"\usql-programmability\output_file.tsv";

@rs0 =
    EXTRACT
            guid Guid,
        dt String,
            user String,
            des String
    FROM @input_file USING Extractors.Tsv();

@rs1 =
     PROCESS @rs0
     PRODUCE dt String,
             full_description String,
             guid Guid,
             new_guid Guid
     USING new USQL_Programmability.FullDescriptionProcessor();

OUTPUT @rs1 TO @output_file USING Outputters.Text();
```

## Use user-defined appliers

A U-SQL user-defined applier enables you to invoke a custom C# function for each row that's returned by the outer table expression of a query. The right input is evaluated for each row from the left input, and the rows that are produced are combined for the final output. The list of columns that are produced by the APPLY operator are the combination of the set of columns in the left and the right input.

User-defined applier is being invoked as part of the USQL SELECT expression.

The typical call to the user-defined applier looks like the following:

```
SELECT …
FROM …
CROSS APPLYis used to pass parameters
new MyScript.MyApplier(param1, param2) AS alias(output_param1 string, …);
```

For more information about using appliers in a SELECT expression, see U-SQL SELECT Selecting from CROSS APPLY and OUTER APPLY.

The user-defined applier base class definition is as follows:

```
public abstract class IApplier : IUserDefinedOperator
{
protected IApplier();

public abstract IEnumerable<IRow> Apply(IRow input, IUpdatableRow output);
}
```

To define a user-defined applier, we need to create the `IApplier` interface with the [`SqlUserDefinedApplier`] attribute, which is optional for a user-defined applier definition.

```
[SqlUserDefinedApplier]
public class ParserApplier : IApplier
{
    public ParserApplier()
    {
        …
    }

    public override IEnumerable<IRow> Apply(IRow input, IUpdatableRow output)
    {
        …
    }
}
```

- Apply is called for each row of the outer table. It returns the `IUpdatableRow` output rowset.
- The Constructor class is used to pass parameters to the user-defined applier.

**SqlUserDefinedApplier** indicates that the type should be registered as a user-defined applier. This class cannot be inherited.

**SqlUserDefinedApplier** is **optional** for a user-defined applier definition.

The main programmability objects are as follows:

```
public override IEnumerable<IRow> Apply(IRow input, IUpdatableRow output)
```

Input rowsets are passed as `IRow` input. The output rows are generated as `IUpdatableRow` output interface.

Individual column names can be determined by calling the `IRow` Schema method.

```
ISchema schema = row.Schema;
var col = schema[i];
string val = row.Get<string>(col.Name)
```

To get the actual data values from the incoming `IRow`, we use the Get() method of `IRow` interface.

```
mycolumn = row.Get<int>("mycolumn")
```

Or we use the schema column name:

```
row.Get<int>(row.Schema[0].Name)
```

The output values must be set with `IUpdatableRow` output:

```
output.Set<int>("mycolumn", mycolumn)
```

It is important to understand that custom appliers only output columns and values that are defined with `output.Set` method call.

The actual output is triggered by calling `yield return output.AsReadOnly();`.

The user-defined applier parameters can be passed to the constructor. Applier can return a variable number of columns that need to be defined during the applier call in base U-SQL Script.

```
new USQL_Programmability.ParserApplier ("all") AS properties(make string, model string, year string, type
string, millage int);
```

Here is the user-defined applier example:

```
[SqlUserDefinedApplier]
public class ParserApplier : IApplier
{
private string parsingPart;

public ParserApplier(string parsingPart)
{
    if (parsingPart.ToUpper().Contains("ALL")
    || parsingPart.ToUpper().Contains("MAKE")
    || parsingPart.ToUpper().Contains("MODEL")
    || parsingPart.ToUpper().Contains("YEAR")
    || parsingPart.ToUpper().Contains("TYPE")
    || parsingPart.ToUpper().Contains("MILLAGE")
    )
    {
    this.parsingPart = parsingPart;
    }
    else
    {
    throw new ArgumentException("Incorrect parameter. Please use: 'ALL[MAKE|MODEL|TYPE|MILLAGE]'");
    }
}

public override IEnumerable<IRow> Apply(IRow input, IUpdatableRow output)
{

    string[] properties = input.Get<string>("properties").Split(',');

    //  only process with correct number of properties
    if (properties.Count() == 5)
    {

    string make = properties[0];
    string model = properties[1];
    string year = properties[2];
    string type = properties[3];
    int millage = -1;

    // Only return millage if it is number, otherwise, -1
    if (!int.TryParse(properties[4], out millage))
    {
        millage = -1;
    }

    if (parsingPart.ToUpper().Contains("MAKE") || parsingPart.ToUpper().Contains("ALL")) output.Set<string>
("make", make);
    if (parsingPart.ToUpper().Contains("MODEL") || parsingPart.ToUpper().Contains("ALL")) output.Set<string>
("model", model);
    if (parsingPart.ToUpper().Contains("YEAR") || parsingPart.ToUpper().Contains("ALL")) output.Set<string>
("year", year);
    if (parsingPart.ToUpper().Contains("TYPE") || parsingPart.ToUpper().Contains("ALL")) output.Set<string>
("type", type);
    if (parsingPart.ToUpper().Contains("MILLAGE") || parsingPart.ToUpper().Contains("ALL")) output.Set<int>
("millage", millage);
    }
    yield return output.AsReadOnly();
}
}
```

Following is the base U-SQL script for this user-defined applier:

```
DECLARE @input_file string = @"c:\usql-programmability\car_fleet.tsv";
DECLARE @output_file string = @"c:\usql-programmability\output_file.tsv";

@rs0 =
    EXTRACT
        stocknumber int,
        vin String,
        properties String
    FROM @input_file USING Extractors.Tsv();

@rs1 =
    SELECT
        r.stocknumber,
        r.vin,
        properties.make,
        properties.model,
        properties.year,
        properties.type,
        properties.millage
    FROM @rs0 AS r
    CROSS APPLY
    new USQL_Programmability.ParserApplier ("all") AS properties(make string, model string, year string, type
string, millage int);

OUTPUT @rs1 TO @output_file USING Outputters.Text();
```

In this use case scenario, user-defined applier acts as a comma-delimited value parser for the car fleet properties. The input file rows look like the following:

```
103 Z1AB2CD123XY45889   Ford,Explorer,2005,SUV,152345
303 Y0AB2CD34XY458890   Shevrolet,Cruise,2010,4Dr,32455
210 X5AB2CD45XY458893   Nissan,Altima,2011,4Dr,74000
```

It is a typical tab-delimited TSV file with a properties column that contains car properties such as make and model. Those properties must be parsed to the table columns. The applier that's provided also enables you to generate a dynamic number of properties in the result rowset, based on the parameter that's passed. You can generate either all properties or a specific set of properties only.

```
…USQL_Programmability.ParserApplier ("all")
…USQL_Programmability.ParserApplier ("make")
…USQL_Programmability.ParserApplier ("make&model")
```

The user-defined applier can be called as a new instance of applier object:

```
CROSS APPLY new MyNameSpace.MyApplier (parameter: “value”) AS alias([columns types]…);
```

Or with the invocation of a wrapper factory method:

```
    CROSS APPLY MyNameSpace.MyApplier (parameter: “value”) AS alias([columns types]…);
```

## Use user-defined combiners

User-defined combiner, or UDC, enables you to combine rows from left and right rowsets, based on custom logic. User-defined combiner is used with COMBINE expression.

A combiner is being invoked with the COMBINE expression that provides the necessary information about both

the input rowsets, the grouping columns, the expected result schema, and additional information.

To call a combiner in a base U-SQL script, we use the following syntax:

```
Combine_Expression :=
    'COMBINE' Combine_Input
    'WITH' Combine_Input
    Join_On_Clause
    Produce_Clause
    [Readonly_Clause]
    [Required_Clause]
    USING_Clause.
```

For more information, see COMBINE Expression (U-SQL).

To define a user-defined combiner, we need to create the `ICombiner` interface with the [`SqlUserDefinedCombiner`] attribute, which is optional for a user-defined Combiner definition.

Base `ICombiner` class definition:

```
public abstract class ICombiner : IUserDefinedOperator
{
protected ICombiner();
public virtual IEnumerable<IRow> Combine(List<IRowset> inputs,
       IUpdatableRow output);
public abstract IEnumerable<IRow> Combine(IRowset left, IRowset right,
       IUpdatableRow output);
}
```

The custom implementation of an `ICombiner` interface should contain the definition for an `IEnumerable<IRow>` Combine override.

```
[SqlUserDefinedCombiner]
public class MyCombiner : ICombiner
{

public override IEnumerable<IRow> Combine(IRowset left, IRowset right,
    IUpdatableRow output)
{
    …
}
}
```

The **SqlUserDefinedCombiner** attribute indicates that the type should be registered as a user-defined combiner. This class cannot be inherited.

**SqlUserDefinedCombiner** is used to define the Combiner mode property. It is an optional attribute for a user-defined combiner definition.

CombinerMode Mode

CombinerMode enum can take the following values:

- Full (0) Every output row potentially depends on all the input rows from left and right with the same key value.
- Left (1) Every output row depends on a single input row from the left (and potentially all rows from the right with the same key value).
- Right (2) Every output row depends on a single input row from the right (and potentially all rows from the

left with the same key value).

- Inner (3) Every output row depends on a single input row from left and right with the same value.

Example: [ `SqlUserDefinedCombiner(Mode=CombinerMode.Left)` ]

The main programmability objects are:

```
    public override IEnumerable<IRow> Combine(IRowset left, IRowset right,
        IUpdatableRow output
```

Input rowsets are passed as **left** and **right** `IRowset` type of interface. Both rowsets must be enumerated for processing. You can only enumerate each interface once, so we have to enumerate and cache it if necessary.

For caching purposes, we can create a List<T> type of memory structure as a result of a LINQ query execution, specifically List< `IRow` >. The anonymous data type can be used during enumeration as well.

See Introduction to LINQ Queries (C#) for more information about LINQ queries, and IEnumerable<T> Interface for more information about IEnumerable<T> interface.

To get the actual data values from the incoming `IRowset`, we use the Get() method of `IRow` interface.

```
mycolumn = row.Get<int>("mycolumn")
```

Individual column names can be determined by calling the `IRow` Schema method.

```
ISchema schema = row.Schema;
var col = schema[i];
string val = row.Get<string>(col.Name)
```

Or by using the schema column name:

```
c# row.Get<int>(row.Schema[0].Name)
```

The general enumeration with LINQ looks like the following:

```
var myRowset =
            (from row in left.Rows
                        select new
                        {
                            Mycolumn = row.Get<int>("mycolumn"),
                        }).ToList();
```

After enumerating both rowsets, we are going to loop through all rows. For each row in the left rowset, we are going to find all rows that satisfy the condition of our combiner.

The output values must be set with `IUpdatableRow` output.

```
output.Set<int>("mycolumn", mycolumn)
```

The actual output is triggered by calling to `yield return output.AsReadOnly();`.

Following is a combiner example:

```
[SqlUserDefinedCombiner]
public class CombineSales : ICombiner
{

public override IEnumerable<IRow> Combine(IRowset left, IRowset right,
    IUpdatableRow output)
{
    var internetSales =
    (from row in left.Rows
          select new
          {
              ProductKey = row.Get<int>("ProductKey"),
              OrderDateKey = row.Get<int>("OrderDateKey"),
              SalesAmount = row.Get<decimal>("SalesAmount"),
              TaxAmt = row.Get<decimal>("TaxAmt")
          }).ToList();

    var resellerSales =
    (from row in right.Rows
     select new
     {
     ProductKey = row.Get<int>("ProductKey"),
     OrderDateKey = row.Get<int>("OrderDateKey"),
     SalesAmount = row.Get<decimal>("SalesAmount"),
     TaxAmt = row.Get<decimal>("TaxAmt")
     }).ToList();

    foreach (var row_i in internetSales)
    {
    foreach (var row_r in resellerSales)
    {

        if (
        row_i.OrderDateKey > 0
        && row_i.OrderDateKey < row_r.OrderDateKey
        && row_i.OrderDateKey == 20010701
        && (row_r.SalesAmount + row_r.TaxAmt) > 20000)
        {
        output.Set<int>("OrderDateKey", row_i.OrderDateKey);
        output.Set<int>("ProductKey", row_i.ProductKey);
        output.Set<decimal>("Internet_Sales_Amount", row_i.SalesAmount + row_i.TaxAmt);
        output.Set<decimal>("Reseller_Sales_Amount", row_r.SalesAmount + row_r.TaxAmt);
        }

    }
    }
    yield return output.AsReadOnly();
}
}
```

In this use-case scenario, we are building an analytics report for the retailer. The goal is to find all products that cost more than $20,000 and that sell through the website faster than through the regular retailer within a certain time frame.

Here is the base U-SQL script. You can compare the logic between a regular JOIN and a combiner:

```
DECLARE @LocalURI string = @"\usql-programmability\";

DECLARE @input_file_internet_sales string = @LocalURI+"FactInternetSales.txt";
DECLARE @input_file_reseller_sales string = @LocalURI+"FactResellerSales.txt";
DECLARE @output_file1 string = @LocalURI+"output_file1.tsv";
DECLARE @output_file2 string = @LocalURI+"output_file2.tsv";

@fact_internet_sales =
EXTRACT
```

```
    ProductKey int ,
    OrderDateKey int ,
    DueDateKey int ,
    ShipDateKey int ,
    CustomerKey int ,
    PromotionKey int ,
    CurrencyKey int ,
    SalesTerritoryKey int ,
    SalesOrderNumber String ,
    SalesOrderLineNumber  int ,
    RevisionNumber int ,
    OrderQuantity int ,
    UnitPrice decimal ,
    ExtendedAmount decimal,
    UnitPriceDiscountPct float ,
    DiscountAmount float ,
    ProductStandardCost decimal ,
    TotalProductCost decimal ,
    SalesAmount decimal ,
    TaxAmt decimal ,
    Freight decimal ,
    CarrierTrackingNumber String,
    CustomerPONumber String
FROM @input_file_internet_sales
USING Extractors.Text(delimiter:'|', encoding: Encoding.Unicode);

@fact_reseller_sales =
EXTRACT
    ProductKey int ,
    OrderDateKey int ,
    DueDateKey int ,
    ShipDateKey int ,
    ResellerKey int ,
    EmployeeKey int ,
    PromotionKey int ,
    CurrencyKey int ,
    SalesTerritoryKey int ,
    SalesOrderNumber String ,
    SalesOrderLineNumber  int ,
    RevisionNumber int ,
    OrderQuantity int ,
    UnitPrice decimal ,
    ExtendedAmount decimal,
    UnitPriceDiscountPct float ,
    DiscountAmount float ,
    ProductStandardCost decimal ,
    TotalProductCost decimal ,
    SalesAmount decimal ,
    TaxAmt decimal ,
    Freight decimal ,
    CarrierTrackingNumber String,
    CustomerPONumber String
FROM @input_file_reseller_sales
USING Extractors.Text(delimiter:'|', encoding: Encoding.Unicode);

@rs1 =
SELECT
    fis.OrderDateKey,
    fis.ProductKey,
    fis.SalesAmount+fis.TaxAmt AS Internet_Sales_Amount,
    frs.SalesAmount+frs.TaxAmt AS Reseller_Sales_Amount
FROM @fact_internet_sales AS fis
     INNER JOIN @fact_reseller_sales AS frs
     ON fis.ProductKey == frs.ProductKey
WHERE
    fis.OrderDateKey < frs.OrderDateKey
    AND fis.OrderDateKey == 20010701
    AND frs.SalesAmount+frs.TaxAmt > 20000;
```

```
@rs2 =
COMBINE @fact_internet_sales AS fis
WITH @fact_reseller_sales AS frs
ON fis.ProductKey == frs.ProductKey
PRODUCE OrderDateKey int,
        ProductKey int,
        Internet_Sales_Amount decimal,
        Reseller_Sales_Amount decimal
USING new USQL_Programmability.CombineSales();

OUTPUT @rs1 TO @output_file1 USING Outputters.Tsv();
OUTPUT @rs2 TO @output_file2 USING Outputters.Tsv();
```

A user-defined combiner can be called as a new instance of the applier object:

```
USING new MyNameSpace.MyCombiner();
```

Or with the invocation of a wrapper factory method:

```
USING MyNameSpace.MyCombiner();
```

## Use user-defined reducers

U-SQL enables you to write custom rowset reducers in C# by using the user-defined operator extensibility framework and implementing an IReducer interface.

User-defined reducer, or UDR, can be used to eliminate unnecessary rows during data extraction (import). It also can be used to manipulate and evaluate rows and columns. Based on programmability logic, it can also define which rows need to be extracted.

To define a UDR class, we need to create an `IReducer` interface with an optional `SqlUserDefinedReducer` attribute.

This class interface should contain a definition for the `IEnumerable` interface rowset override.

```
[SqlUserDefinedReducer]
public class EmptyUserReducer : IReducer
{

    public override IEnumerable<IRow> Reduce(IRowset input, IUpdatableRow output)
    {
        …
    }

}
```

The **SqlUserDefinedReducer** attribute indicates that the type should be registered as a user-defined reducer. This class cannot be inherited. **SqlUserDefinedReducer** is an optional attribute for a user-defined reducer definition. It's used to define IsRecursive property.

- bool IsRecursive
- **true** = Indicates whether this Reducer is associative and commutative

The main programmability objects are **input** and **output**. The input object is used to enumerate input rows. Output is used to set output rows as a result of reducing activity.

For input rows enumeration, we use the `Row.Get` method.

```
foreach (IRow row in input.Rows)
{
    row.Get<string>("mycolumn");
}
```

The parameter for the `Row.Get` method is a column that's passed as part of the `PRODUCE` class of the `REDUCE` statement of the U-SQL base script. We need to use the correct data type here as well.

For output, use the `output.Set` method.

It is important to understand that custom reducer only outputs values that are defined with the `output.Set` method call.

```
output.Set<string>("mycolumn", guid);
```

The actual reducer output is triggered by calling `yield return output.AsReadOnly();`.

Following is a reducer example:

```
[SqlUserDefinedReducer]
public class EmptyUserReducer : IReducer
{

    public override IEnumerable<IRow> Reduce(IRowset input, IUpdatableRow output)
    {
    string guid;
    DateTime dt;
    string user;
    string des;

    foreach (IRow row in input.Rows)
    {
        guid = row.Get<string>("guid");
        dt = row.Get<DateTime>("dt");
        user = row.Get<string>("user");
        des = row.Get<string>("des");

        if (user.Length > 0)
        {
        output.Set<string>("guid", guid);
        output.Set<DateTime>("dt", dt);
        output.Set<string>("user", user);
        output.Set<string>("des", des);

        yield return output.AsReadOnly();
        }
    }
    }

}
```

In this use-case scenario, the reducer is skipping rows with an empty user name. For each row in rowset, it reads each required column, then evaluates the length of the user name. It outputs the actual row only if user name value length is more than 0.

Following is base U-SQL script that uses a custom reducer:

```
DECLARE @input_file string = @"\usql-programmability\input_file_reducer.tsv";
DECLARE @output_file string = @"\usql-programmability\output_file.tsv";

@rs0 =
    EXTRACT
            guid string,
        dt DateTime,
            user String,
            des String
    FROM @input_file
    USING Extractors.Tsv();

@rs1 =
    REDUCE @rs0 PRESORT guid
    ON guid
    PRODUCE guid string, dt DateTime, user String, des String
    USING new USQL_Programmability.EmptyUserReducer();

@rs2 =
    SELECT guid AS start_id,
           dt AS start_time,
           DateTime.Now.ToString("M/d/yyyy") AS Nowdate,
           USQL_Programmability.CustomFunctions.GetFiscalPeriodWithCustomType(dt).ToString() AS
start_fiscalperiod,
           user,
           des
    FROM @rs1;

OUTPUT @rs2
    TO @output_file
    USING Outputters.Text();
```

# Install Data Lake Tools for Visual Studio

8/27/2018 • 2 minutes to read • Edit Online

Learn how to use Visual Studio to create Azure Data Lake Analytics accounts, define jobs in U-SQL, and submit jobs to the Data Lake Analytics service. For more information about Data Lake Analytics, see Azure Data Lake Analytics overview.

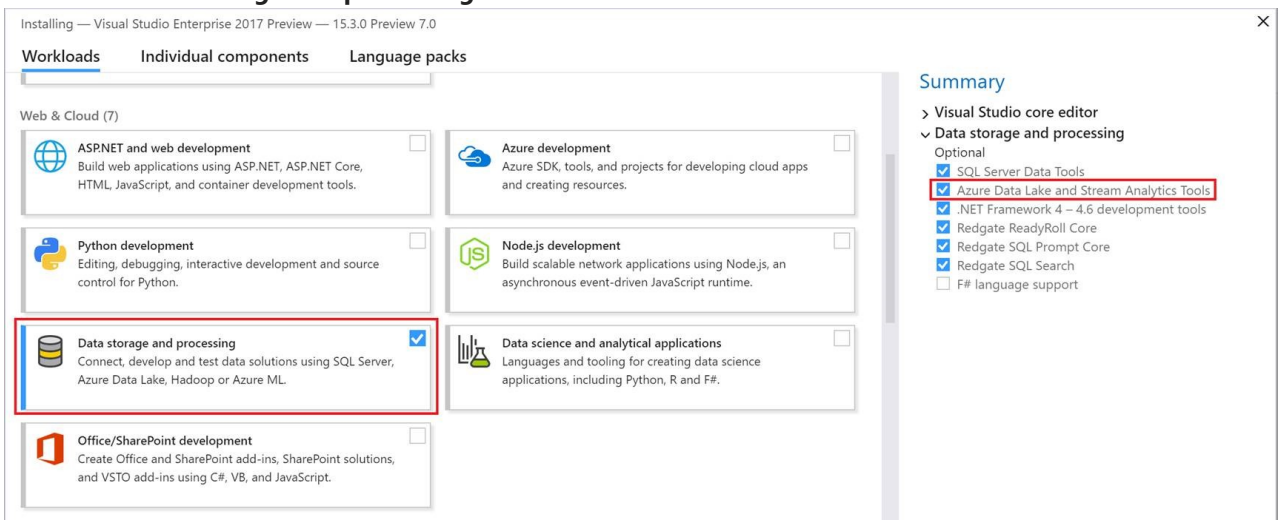
## Prerequisites

- **Visual Studio**: All editions except Express are supported.
  - Visual Studio 2017
  - Visual Studio 2015
  - Visual Studio 2013
- **Microsoft Azure SDK for .NET** version 2.7.1 or later. Install it by using the Web platform installer.
- A **Data Lake Analytics** account. To create an account, see Get Started with Azure Data Lake Analytics using Azure portal.

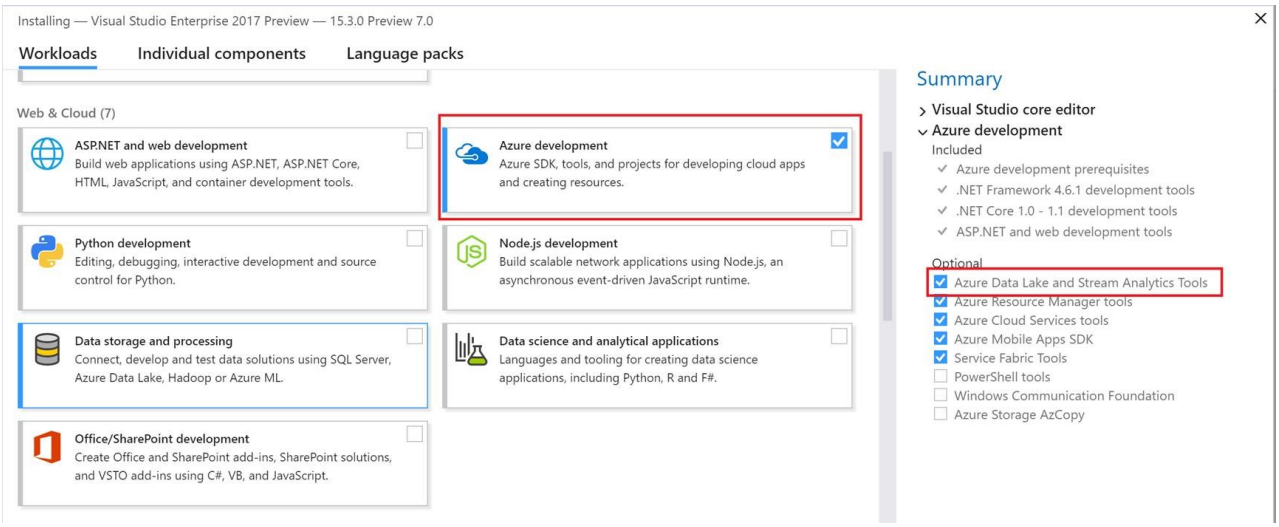
## Install Azure Data Lake Tools for Visual Studio 2017

Azure Data Lake Tools for Visual Studio is supported in Visual Studio 2017 15.3 or above. The tool is part of the **Data storage and processing** and **Azure Development** workloads in Visual Studio Installer. Enable either one of these two workloads as part of your Visual Studio installation.

Enable the **Data storage and processing** workload as shown:




Enable the **Azure development** workload as shown:




## Install Azure Data Lake Tools for Visual Studio 2013 and 2015

Download and install Azure Data Lake Tools for Visual Studio from the Download Center. After installation, note that:

- The **Server Explorer** > **Azure** node contains a **Data Lake Analytics** node.
- The **Tools** menu has a **Data Lake** item.

## Next Steps

- To log diagnostics information, see Accessing diagnostics logs for Azure Data Lake Analytics
- To see a more complex query, see Analyze Website logs using Azure Data Lake Analytics.
- To use the vertex execution view, see Use the Vertex Execution View in Data Lake Tools for Visual Studio

# Run U-SQL scripts on your local machine

8/27/2018 • 6 minutes to read • Edit Online

When you develop U-SQL scripts, you can save time and expense by running the scripts locally. Azure Data Lake Tools for Visual Studio supports running U-SQL scripts on your local machine.

## Basic concepts for local runs

The following chart shows the components for local run and how these components map to cloud run.

| COMPONENT | LOCAL RUN | CLOUD RUN |
|---|---|---|
| Storage | Local data root folder | Default Azure Data Lake Store account |
| Compute | U-SQL local run engine | Azure Data Lake Analytics service |
| Run environment | Working directory on local machine | Azure Data Lake Analytics cluster |

The sections that follow provide more information about local run components.

### Local data root folders

A local data root folder is a **local store** for the local compute account. Any folder in the local file system on your local machine can be a local data root folder. It's the same as the default Azure Data Lake Store account of a Data Lake Analytics account. Switching to a different data root folder is just like switching to a different default store account.

The data root folder is used as follows:

- Store metadata. Examples are databases, tables, table-valued functions, and assemblies.
- Look up the input and output paths that are defined as relative paths in U-SQL scripts. By using relative paths, it's easier to deploy your U-SQL scripts to Azure.

### U-SQL local run engines

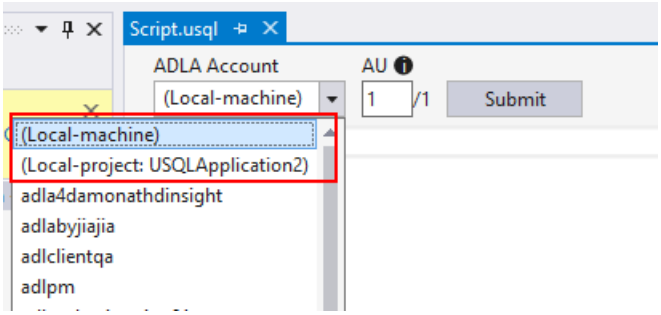
A U-SQL local run engine is a **local compute account** for U-SQL jobs. Users can run U-SQL jobs locally through Azure Data Lake Tools for Visual Studio. Local runs are also supported through the Azure Data Lake U-SQL SDK command-line and programming interfaces. Learn more about the Azure Data Lake U-SQL SDK.

### Working directories

When you run a U-SQL script, a working directory folder is needed to cache compilation results, run logs, and perform other functions. In Azure Data Lake Tools for Visual Studio, the working directory is the U-SQL project's working directory. It's located under `<U-SQL project root path>/bin/debug>`. The working directory is cleaned every time a new run is triggered.

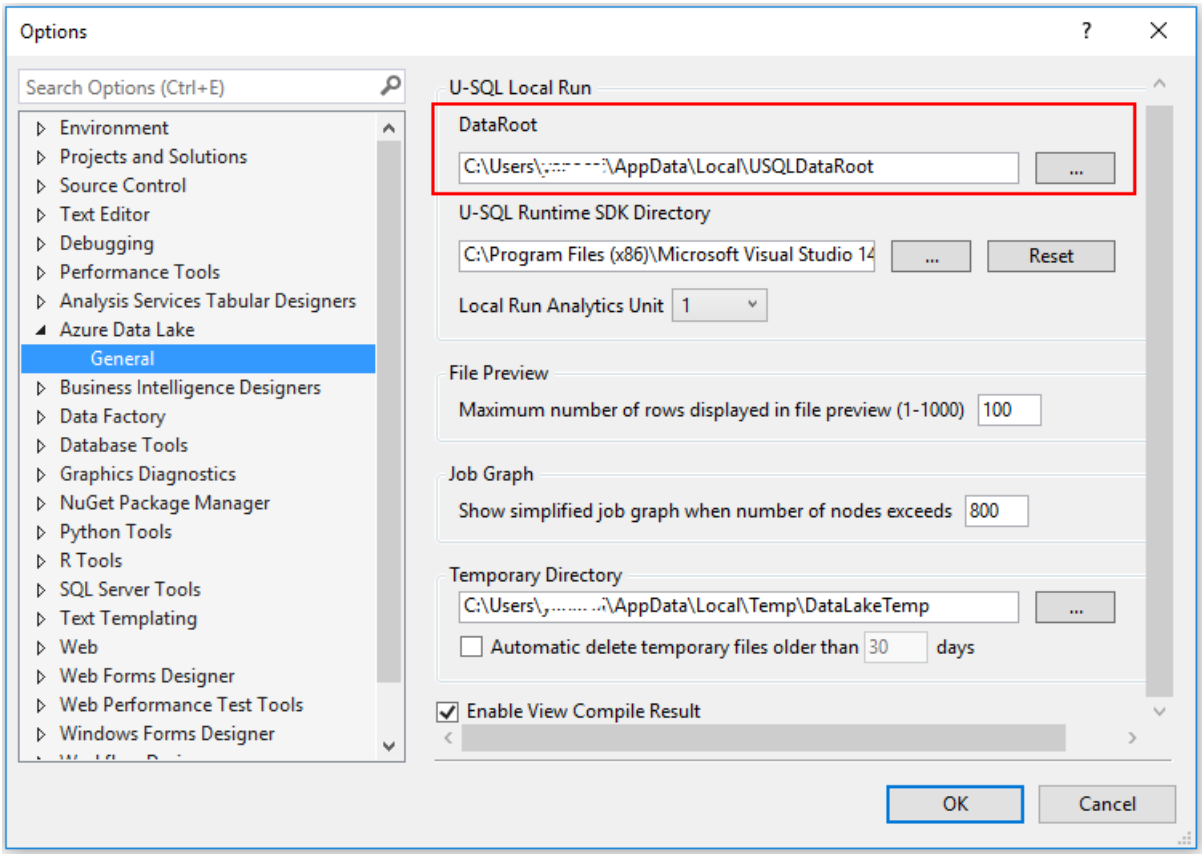
## Local runs in Microsoft Visual Studio

Azure Data Lake Tools for Visual Studio have a built-in local run engine. The tools surface the engine as a local compute account. To run a U-SQL script locally, select the **Local-machine** or **Local-project** account in the script's editor margin drop-down menu. Then select **Submit**.




## Local runs with a Local-machine account

A **Local-machine** account is a shared local compute account with a single local data root folder as the local store account. By default, the data root folder is located at **C:\Users<username>\AppData\Local\USQLDataRoot**. It's also configurable through **Tools** > **Data Lake** > **Options and Settings**.




A U-SQL project is required for a local run. The U-SQL project's working directory is used for the U-SQL local run working directory. Compilation results, run logs, and other job run-related files are generated and stored under the working directory folder during the local run. Every time you rerun the script, all the files in the working directory are cleaned and regenerated.

## Local runs with a Local-project account

A **Local-project** account is a project-isolated local compute account for each project with an isolated local data root folder. Every active U-SQL project that opens in Solution Explorer in Visual Studio has a corresponding `(Local-project: <project name>)` account. The accounts are listed in both Server Explorer in Visual Studio and the U-SQL script editor margin.

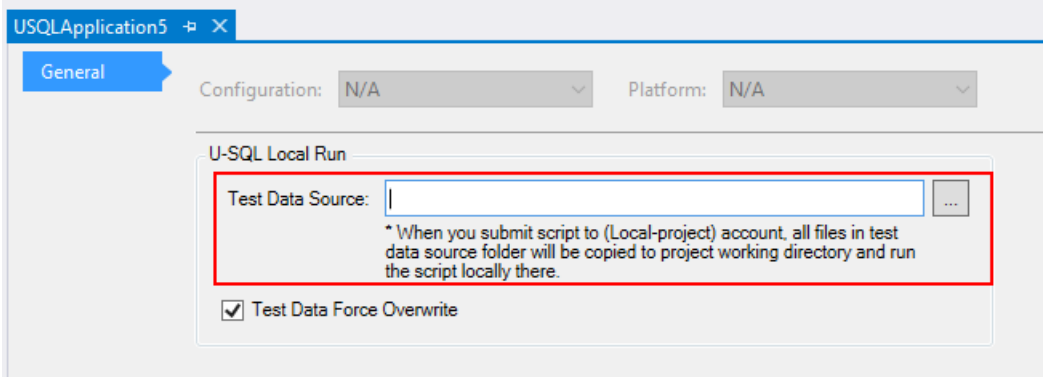
The **Local-project** account provides a clean and isolated development environment. A **Local-machine** account has a shared local data root folder that stores metadata and input and output data for all local jobs. But a **Local-project** account creates a temporary local data root folder under a U-SQL project working directory every time a U-SQL script is run. This temporary data root folder is cleaned when a rebuild or rerun happens.

A U-SQL project manages the isolated local run environment through a project reference and property. You can configure the input data sources for U-SQL scripts in both the project and the referenced database environments.

**Manage the input data source for a Local-project account**

A U-SQL project creates a local data root folder and sets up data for a **Local-project** account. A temporary data root folder is cleaned and recreated under the U-SQL project working directory every time a rebuild and local run happens. All data sources that are configured by the U-SQL project are copied to this temporary local data root folder before the local job runs.

You can configure the root folder of your data sources. Right-click **U-SQL project** > **Property** > **Test Data Source**. When you run a U-SQL script on a **Local-project** account, all files and subfolders in the **Test Data Source** folder are copied to the temporary local data root folder. Files under subfolders are included. After a local job runs, output results can also be found under the temporary local data root folder in the project working directory. All this output is deleted and cleaned when the project gets rebuilt and cleaned.




**Manage a referenced database environment for a Local-project account**

If a U-SQL query uses or queries with U-SQL database objects, you must make the database environments ready locally before you run the U-SQL script locally. For a **Local-project** account, U-SQL database dependencies can be managed by U-SQL project references. You can add U-SQL database project references to your U-SQL project. Before running U-SQL scripts on a **Local-project** account, all referenced databases are deployed to the temporary local data root folder. And for each run, the temporary data root folder is cleaned as a fresh isolated environment.

See this related article:

- Learn how to manage U-SQL database definitions and references in U-SQL database projects.

## The difference between **Local-machine** and **Local-project** accounts

A **Local-machine** account simulates an Azure Data Lake Analytics account on users' local machines. It shares the same experience with an Azure Data Lake Analytics account. A **Local-project** account provides a user-friendly local development environment. This environment helps users deploy database references and input data before they run scripts locally. A **Local-machine** account provides a shared permanent environment that can be accessed through all projects. A **Local-project** account provides an isolated development environment for each project. It's refreshed for each run. A **Local-project** account offers a faster development experience by quickly applying new changes.

More differences between **Local-machine** and **Local-project** accounts are shown in the following table:

| DIFFERENCE ANGLE | LOCAL-MACHINE | LOCAL-PROJECT |
|---|---|---|
| Local access | Can be accessed by all projects. | Only the corresponding project can access this account. |

| DIFFERENCE ANGLE | LOCAL-MACHINE | LOCAL-PROJECT |
|---|---|---|
| Local data root folder | A permanent local folder. Configured through **Tools** > **Data Lake** > **Options and Settings**. | A temporary folder created for each local run under the U-SQL project working directory. The folder gets cleaned when a rebuild or rerun happens. |
| Input data for a U-SQL script | The relative path under the permanent local data root folder. | Set through **U-SQL project property** > **Test Data Source**. All files and subfolders are copied to the temporary data root folder before a local run. |
| Output data for a U-SQL script | Relative path under the permanent local data root folder. | Output to the temporary data root folder. The results are cleaned when a rebuild or rerun happens. |
| Referenced database deployment | Referenced databases aren't deployed automatically when running against a **Local-machine** account. It's the same for submitting to an Azure Data Lake Analytics account. | Referenced databases are deployed to the **Local-project** account automatically before a local run. All database environments are cleaned and redeployed when a rebuild or rerun happens. |

## A local run with the U-SQL SDK

You can run U-SQL scripts locally in Visual Studio and also use the Azure Data Lake U-SQL SDK to run U-SQL scripts locally with command-line and programming interfaces. Through these interfaces, you can automate U-SQL local runs and tests.

Learn more about the Azure Data Lake U-SQL SDK.

## Next steps

- How to set up a CI/CD pipeline for Azure Data Lake Analytics.
- How to test your Azure Data Lake Analytics code.

# Debug Azure Data Lake Analytics code locally

8/27/2018 • 2 minutes to read • Edit Online

You can use Azure Data Lake Tools for Visual Studio to run and debug Azure Data Lake Analytics code on your local workstation, just as you can in the Azure Data Lake Analytics service.

Learn how to run U-SQL script on your local machine.

## Debug scripts and C# assemblies locally

You can debug C# assemblies without submitting and registering them to the Azure Data Lake Analytics service. You can set breakpoints in both the code-behind file and in a referenced C# project.

**Debug local code in a code-behind file**

1. Set breakpoints in the code-behind file.
2. Select **F5** to debug the script locally.

> **NOTE**
> The following procedure works only in Visual Studio 2015. In older Visual Studio versions, you might need to manually add the **PDB** files.

**Debug local code in a referenced C# project**

1. Create a C# assembly project, and build it to generate the output **DLL** file.
2. Register the **DLL** file by using a U-SQL statement:

```
CREATE ASSEMBLY assemblyname FROM @"..\..\path\to\output\.dll";
```

3. Set breakpoints in the C# code.
4. Select **F5** to debug the script by referencing the C# **DLL** file locally.

## Next steps

- For an example of a more complex query, see Analyze website logs using Azure Data Lake Analytics.
- To view job details, see Use Job Browser and Job View for Azure Data Lake Analytics jobs.
- To use the vertex execution view, see Use the Vertex Execution View in Data Lake Tools for Visual Studio.

# Use a U-SQL database project to develop a U-SQL database for Azure Data Lake

9/17/2018 • 4 minutes to read • Edit Online

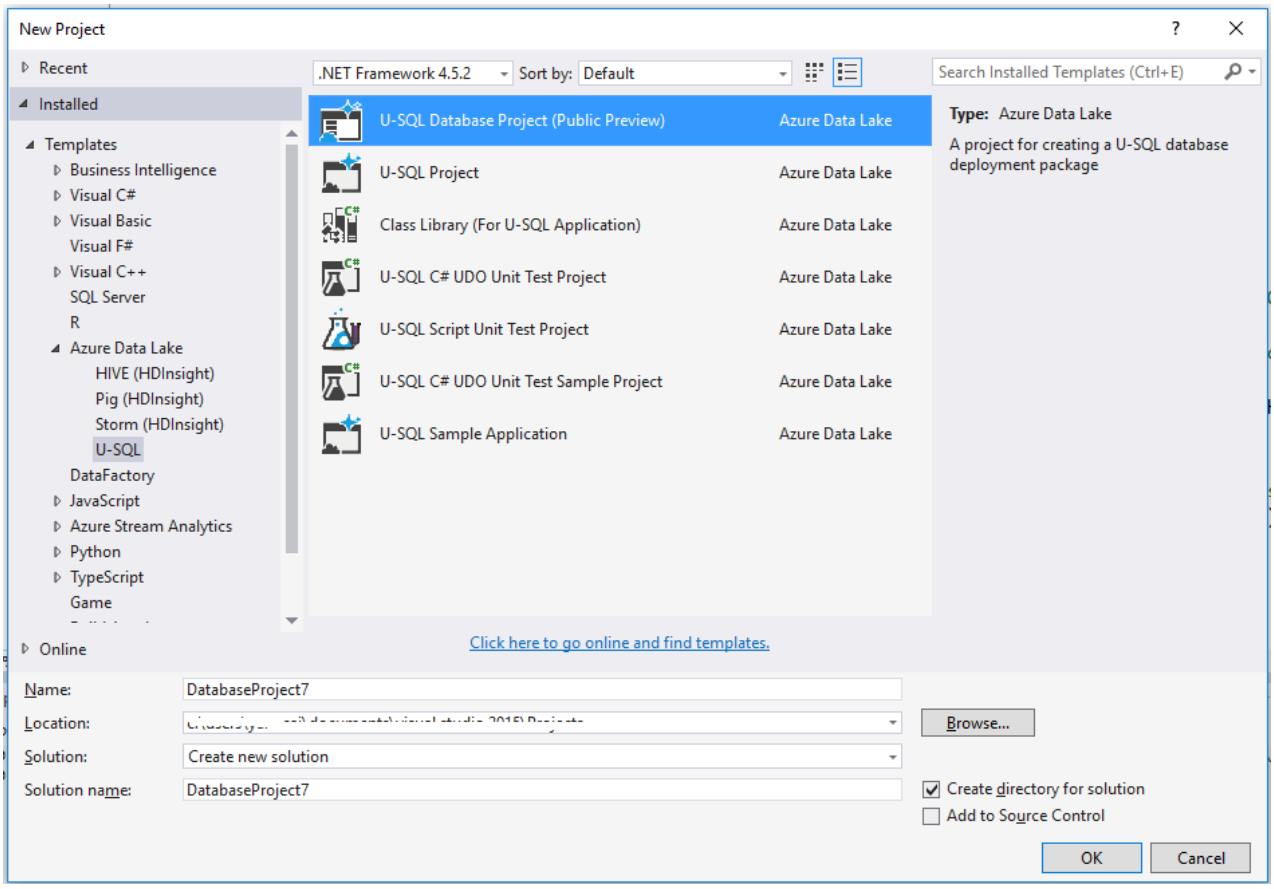
U-SQL database provides structured views over unstructured data and managed structured data in tables. It also provides a general metadata catalog system for organizing your structured data and custom code. The database is the concept that groups these related objects together.

Learn more about U-SQL database and Data Definition Language (DDL).

The U-SQL database project is a project type in Visual Studio that helps developers develop, manage, and deploy their U-SQL databases quickly and easily.

## Create a U-SQL database project

Azure Data Lake Tools for Visual Studio added a new project template called U-SQL database project after version 2.3.3000.0. To create a U-SQL project, select **File > New > Project**. The U-SQL Database Project can be found under **Azure Data Lake > U-SQL node**.



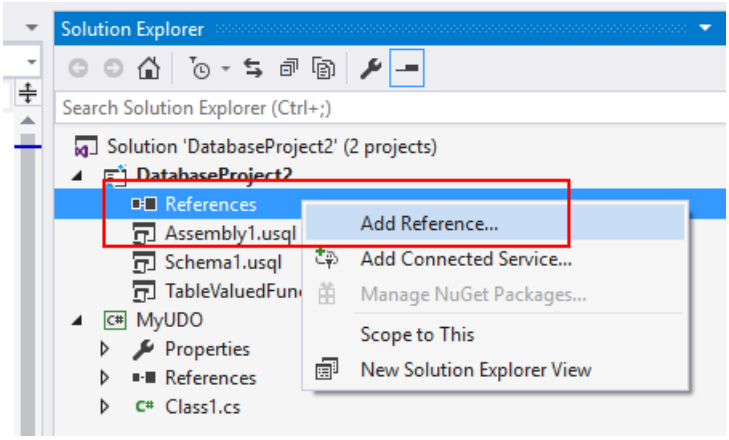

## Develop U-SQL database objects by using a database project

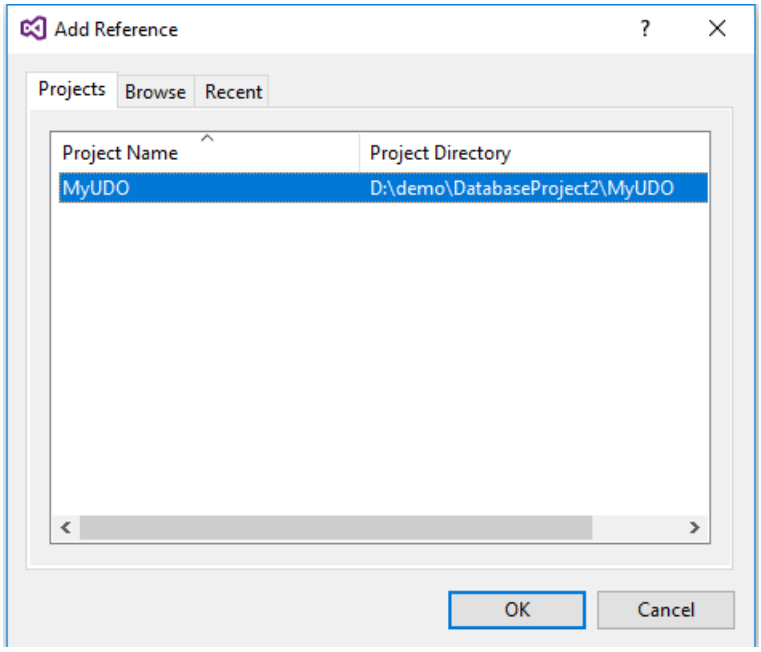
Right-click the U-SQL database project. The select **Add > New item**. You can find all new supported object types in the **Add New Item** Wizard.

For a non-assembly object (for example, a table-valued function), a new U-SQL script is created after you add a new item. You can start to develop the DDL statement for that object in the editor.

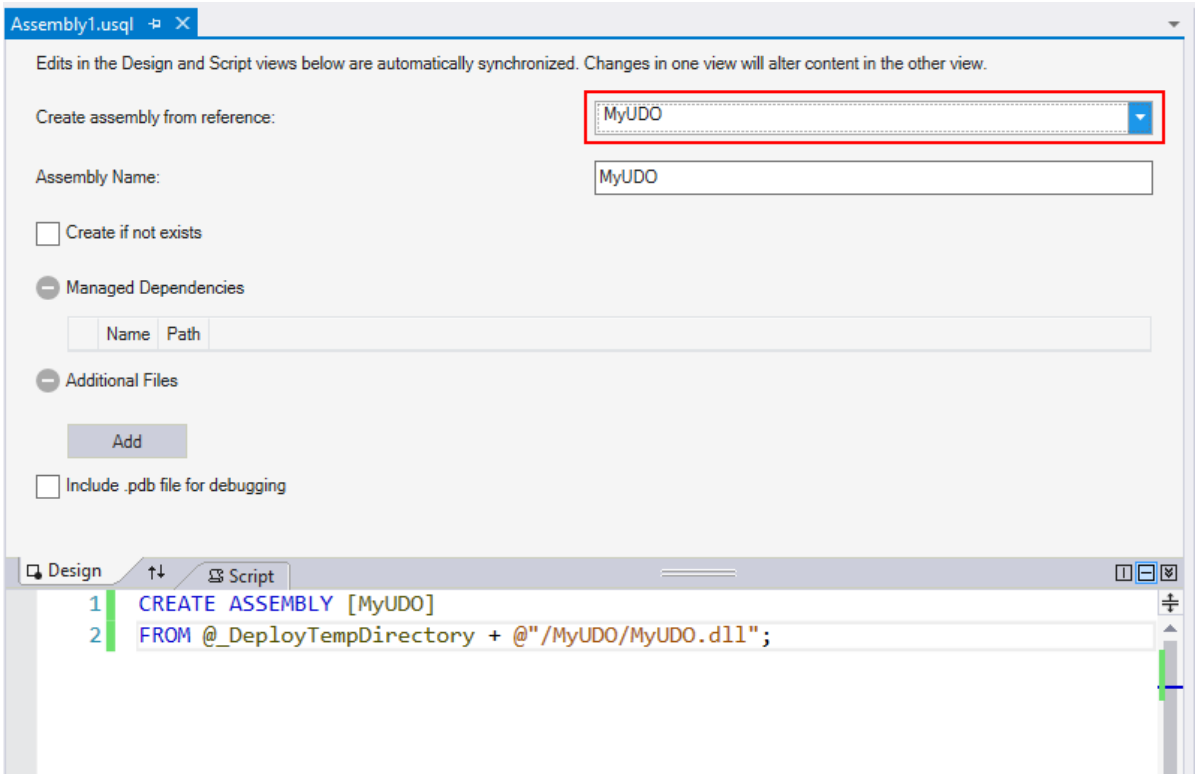
For an assembly object, the tool provides a user-friendly UI editor that helps you register the assembly and deploy DLL files and other additional files. The following steps show you how to add an assembly object definition to the U-SQL database project:

1. Add references to the C# project that include the UDO/UDAG/UDF for the U-SQL database project.




2. In the assembly design view, choose the referenced assembly from **Create assembly from reference** drop-down menu.




3. Add **Managed Dependencies** and **Additional Files** if there are any. When you add additional files, the tool uses the relative path to make sure it can find the assemblies both on your local machine and on the build machine later.

@_DeployTempDirectory is a predefined variable that points the tool to the build output folder. Under the build output folder, every assembly has a subfolder named with the assembly name. All DLLs and additional files are in that subfolder.

# Build a U-SQL database project

The build output for a U-SQL database project is a U-SQL database deployment package, named with the suffix `.usqldbpack`. The `.usqldbpack` package is a .zip file that includes all DDL statements in a single U-SQL script in the **DDL** folder, and all DLLs and additional files for assemblies in the **Temp** folder.

Learn more about how to build a U-SQL database project with the MSBuild command line and a Azure DevOps Services build task.

# Deploy a U-SQL database

The .usqldbpack package can be deployed to either a local account or an Azure Data Lake Analytics account by using Visual Studio or the deployment SDK.

### Deploy a U-SQL database in Visual Studio

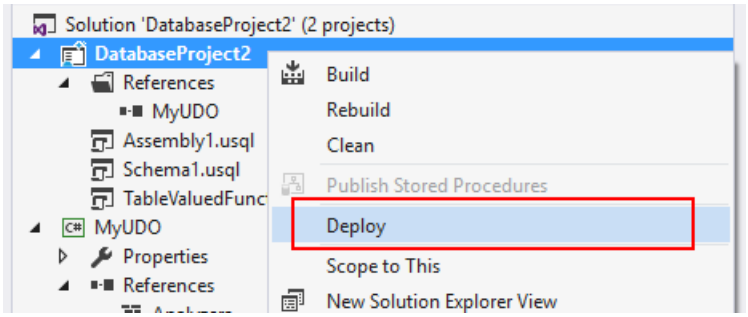
You can deploy a U-SQL database through a U-SQL database project or a .usqldbpack package in Visual Studio.

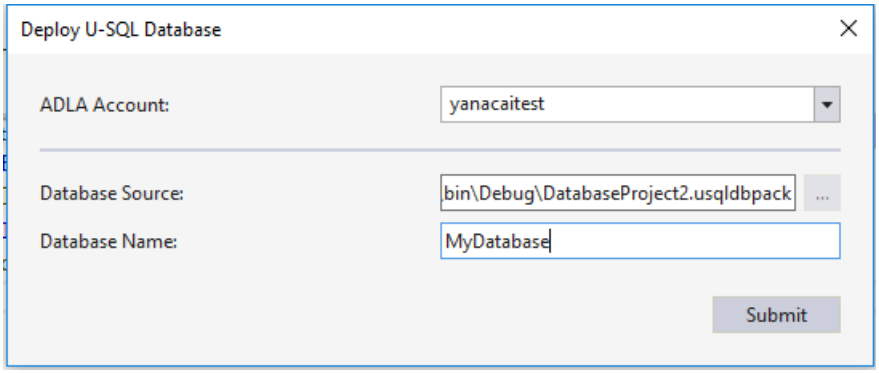
#### Deploy through a U-SQL database project

1. Right-click the U-SQL database project, and then select **Deploy**.
2. In the **Deploy U-SQL Database Wizard**, select the **ADLA account** to which you want to deploy the database. Both local accounts and ADLA accounts are supported.
3. **Database Source** is filled in automatically, and points to the .usqldbpack package in the project's build output folder.
4. Enter a name in **Database Name** to create a database. If a database with that same name already exists in the target Azure Data Lake Analytics account, all objects that are defined in the database project are created

without recreating the database.

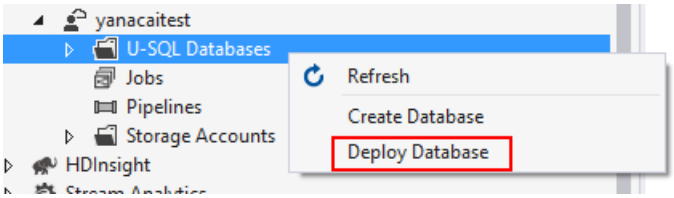
5. To deploy the U-SQL database, select **Submit**. All resources (assemblies and additional files) are uploaded, and a U-SQL job that includes all DDL statements is submitted.

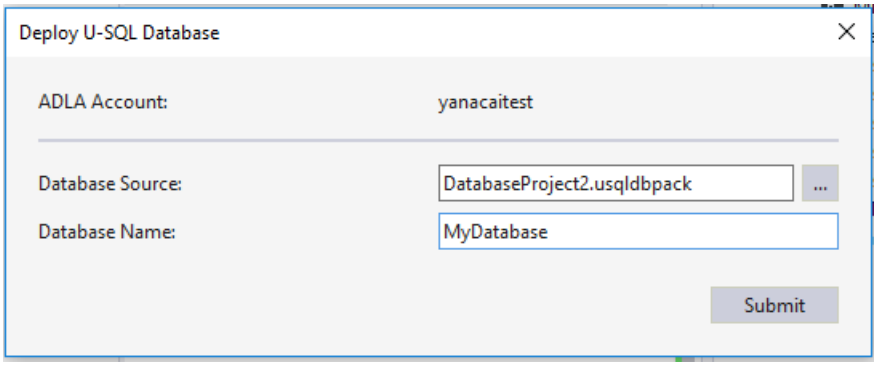



**Deploy through a U-SQL database deployment package**

1. Open **Server Explorer**. Then expand the **Azure Data Lake Analytics account** to which you want to deploy the database.
2. Right click **U-SQL Databases**, and then choose **Deploy Database**.
3. Set **Database Source** to the U-SQL database deployment package (.usqldbpack file) path.
4. Enter the **Database Name** to create a database. If there is a database with the same name that already exists in the target Azure Data Lake Analytics account, all objects that are defined in the database project are created without recreating the database.




### Deploy U-SQL database by using the SDK

`PackageDeploymentTool.exe` provides the programming and command-line interfaces that help to deploy U-SQL databases. The SDK is included in the U-SQL SDK Nuget package, located at `build/runtime/PackageDeploymentTool.exe`.

Learn more about the SDK and how to set up CI/CD pipeline for U-SQL database deployment.

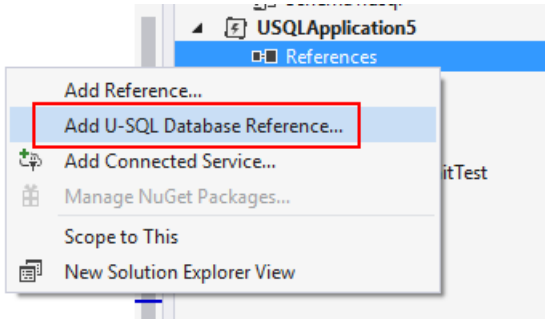
# Reference a U-SQL database project

A U-SQL project can reference a U-SQL database project. The reference affects two workloads:

- *Project build*: Set up the referenced database environments before building the U-SQL scripts.
- *Local run against (a local-project) account*: The referenced database environments are deployed to (a local-project) account before U-SQL script execution. Learn more about local runs and the difference between (the local-machine) and (a local-project) account here.

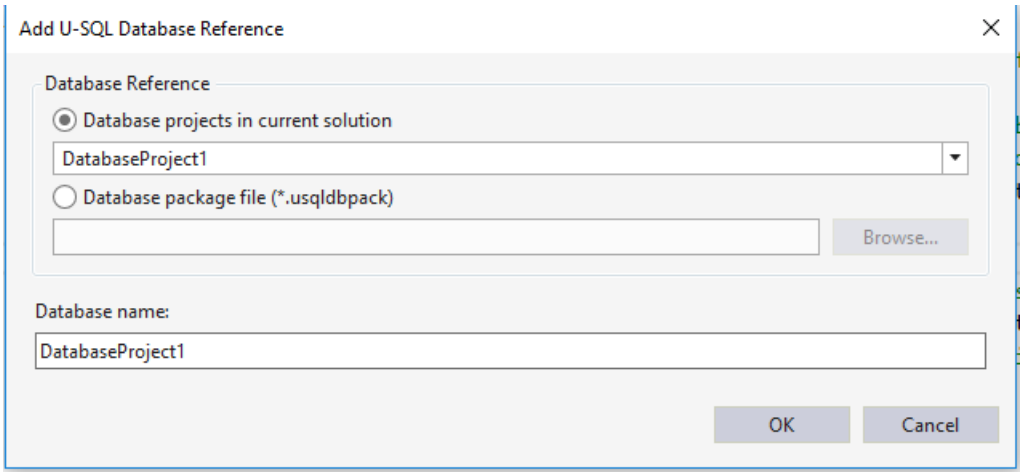
**How to add a U-SQL database reference**

1. Right-click the U-SQL project in **Solution Explorer**, and then choose **Add U-SQL Database Reference...**.




2. Configure a database reference from a U-SQL database project in the current solution or in a U-SQL database package file.

3. Provide the name for the database.




# Next steps

- How to set up a CI/CD pipeline for Azure Data Lake Analytics
- How to test your Azure Data Lake Analytics code
- Run U-SQL script on your local machine

# Use Job Browser and Job View for Azure Data Lake Analytics

8/27/2018 • 10 minutes to read • Edit Online

The Azure Data Lake Analytics service archives submitted jobs in a query store. In this article, you learn how to use Job Browser and Job View in Azure Data Lake Tools for Visual Studio to find the historical job information.

By default, the Data Lake Analytics service archives the jobs for 30 days. The expiration period can be configured from the Azure portal by configuring the customized expiration policy. You will not be able to access the job information after expiration.

## Prerequisites

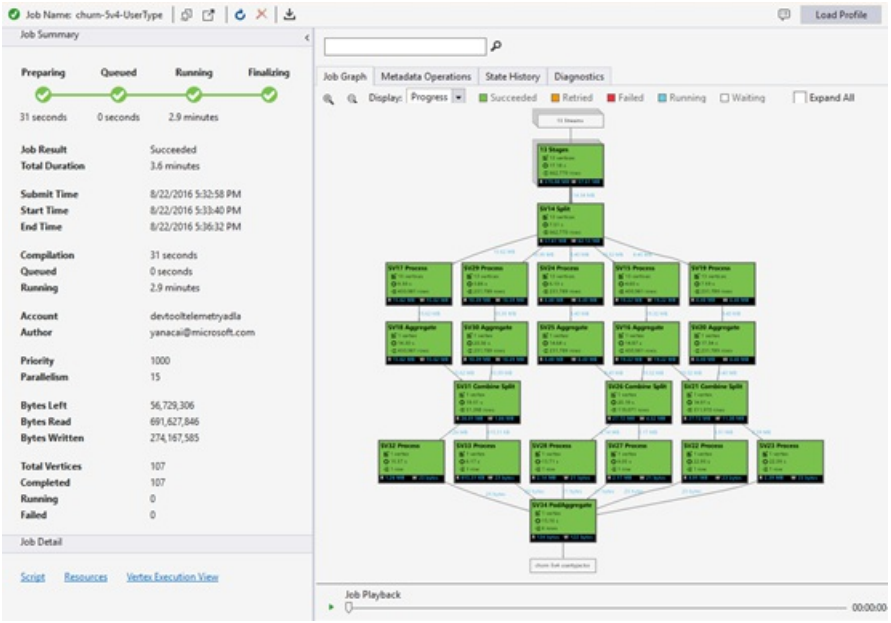
See Data Lake Tools for Visual Studio prerequisites.

## Open the Job Browser

Access the Job Browser via **Server Explorer>Azure>Data Lake Analytics>Jobs** in Visual Studio. Using the Job Browser, you can access the query store of a Data Lake Analytics account. Job Browser displays Query Store on the left, showing basic job information, and Job View on the right showing detailed job information.

## Job View

Job View shows the detailed information of a job. To open a job, you can double-click a job in the Job Browser, or open it from the Data Lake menu by clicking Job View. You should see a dialog populated with the job URL.




Job View contains:

- Job Summary

  Refresh the Job View to see the more recent information of running jobs.

  - Job Status (graph):

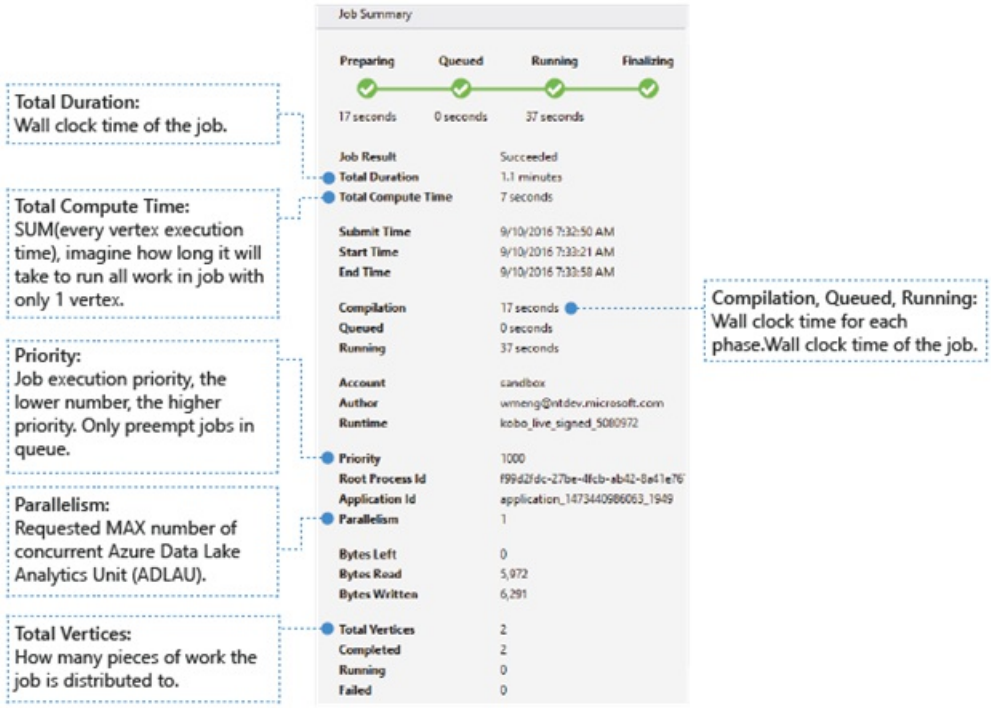
    Job Status outlines the job phases:


- Preparing: Upload your script to the cloud, compiling and optimizing the script using the compile service.
- Queued: Jobs are queued when they are waiting for enough resources, or the jobs exceed the max concurrent jobs per account limitation. The priority setting determines the sequence of queued jobs - the lower the number, the higher the priority.
- Running: The job is actually running in your Data Lake Analytics account.
- Finalizing: The job is completing (for example, finalizing the file).

  The job can fail in every phase. For example, compilation errors in the Preparing phase, timeout errors in the Queued phase, and execution errors in the Running phase, etc.

- Basic Information

The basic job information shows in the lower part of the Job Summary panel.



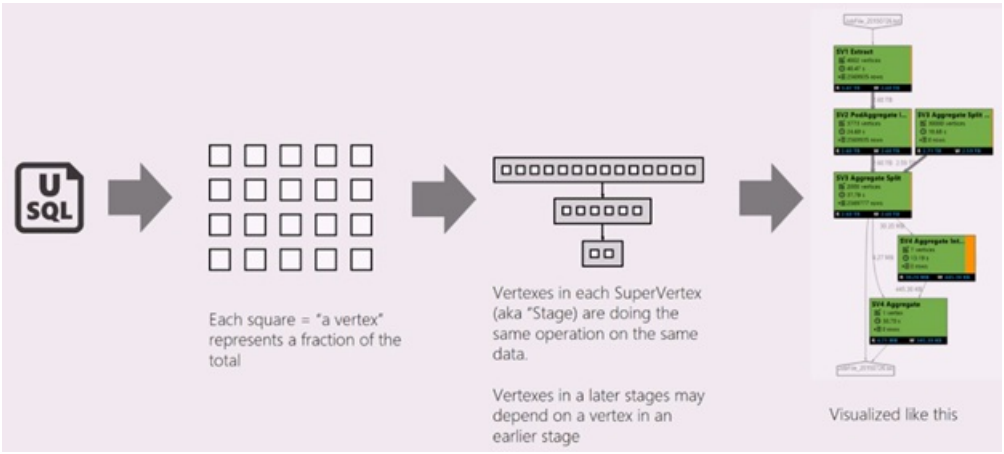

- Job Result: Succeeded or failed. The job may fail in every phase.
- Total Duration: Wall clock time (duration) between submitting time and ending time.
- Total Compute Time: The sum of every vertex execution time, you can consider it as the time that the job is executed in only one vertex. Refer to Total Vertices to find more information about vertex.
- Submit/Start/End Time: The time when the Data Lake Analytics service receives job submission/starts to run the job/ends the job successfully or not.
- Compilation/Queued/Running: Wall clock time spent during the Preparing/Queued/Running phase.
- Account: The Data Lake Analytics account used for running the job.
- Author: The user who submitted the job, it can be a real person's account or a system account.
- Priority: The priority of the job. The lower the number, the higher the priority. It only affects the sequence of the jobs in the queue. Setting a higher priority does not preempt running jobs.
- Parallelism: The requested maximum number of concurrent Azure Data Lake Analytics Units (ADLAUs), aka vertices. Currently, one vertex is equal to one VM with two virtual core and six GB

RAM, though this could be upgraded in future Data Lake Analytics updates.

- Bytes Left: Bytes that need to be processed until the job completes.
- Bytes read/written: Bytes that have been read/written since the job started running.
- Total vertices: The job is broken up into many pieces of work, each piece of work is called a vertex. This value describes how many pieces of work the job consists of. You can consider a vertex as a basic process unit, aka Azure Data Lake Analytics Unit (ADLAU), and vertices can be run in parallelism.
- Completed/Running/Failed: The count of completed/running/failed vertices. Vertices can fail due to both user code and system failures, but the system retries failed vertices automatically a few times. If the vertex is still failed after retrying, the whole job will fail.

- Job Graph

A U-SQL script represents the logic of transforming input data to output data. The script is compiled and optimized to a physical execution plan at the Preparing phase. Job Graph is to show the physical execution plan. The following diagram illustrates the process:

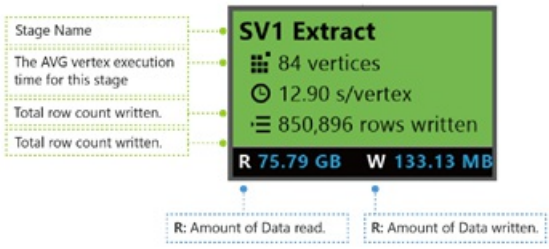



A job is broken up into many pieces of work. Each piece of work is called a Vertex. The vertices are grouped as Super Vertex (aka stage), and visualized as Job Graph. The green stage placards in the job graph show the stages.

Every vertex in a stage is doing the same kind of work with different pieces of the same data. For example, if you have a file with one TB data, and there are hundreds of vertices reading from it, each of them is reading a chunk. Those vertices are grouped in the same stage and doing same work on different pieces of same input file.

- Stage information

  In a particular stage, some numbers are shown in the placard.

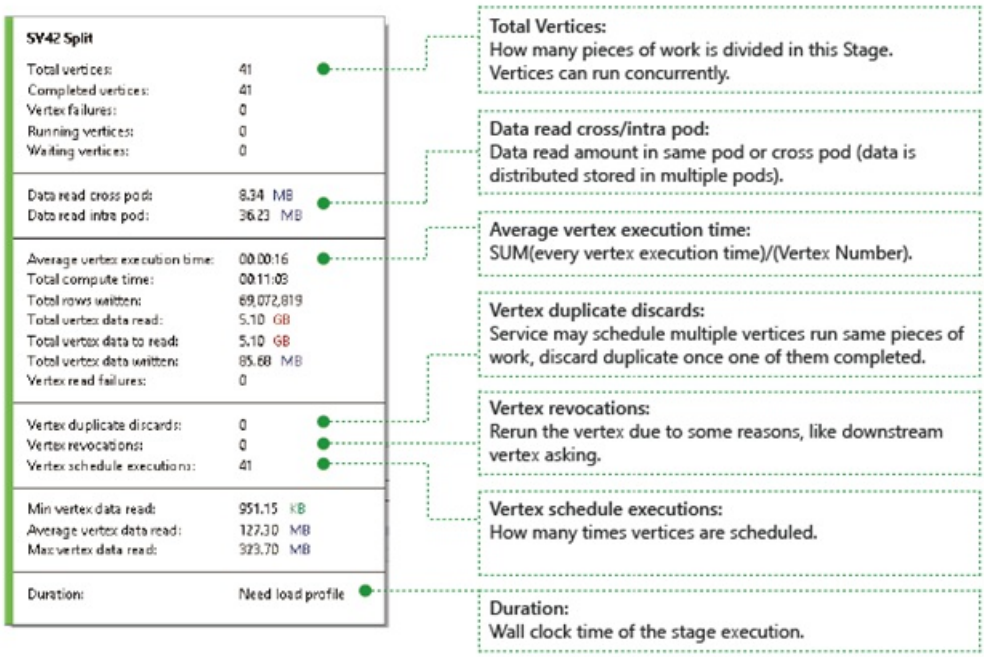



- SV1 Extract: The name of a stage, named by a number and the operation method.
- 84 vertices: The total count of vertices in this stage. The figure indicates how many pieces of work is divided in this stage.
- 12.90 s/vertex: The average vertex execution time for this stage. This figure is calculated by SUM (every vertex execution time) / (total Vertex count). Which means if you could assign all the

vertices executed in parallelism, the whole stage is completed in 12.90 s. It also means if all the work in this stage is done serially, the cost would be #vertices * AVG time.

- 850,895 rows written: Total row count written in this stage.
- R/W: Amount of data read/Written in this stage in bytes.
- Colors: Colors are used in the stage to indicate different vertex status.
  - Green indicates the vertex is succeeded.
  - Orange indicates the vertex is retried. The retried vertex was failed but is retried automatically and successfully by the system, and the overall stage is completed successfully. If the vertex retried but still failed, the color turns red and the whole job failed.
  - Red indicates failed, which means a certain vertex had been retried a few times by the system but still failed. This scenario causes the whole job to fail.
  - Blue means a certain vertex is running.
  - White indicates the vertex is Waiting. The vertex may be waiting to be scheduled once an ADLAU becomes available, or it may be waiting for input since its input data might not be ready.

You can find more details for the stage by hovering your mouse cursor by one state:




- Vertices: Describes the vertices details, for example, how many vertices in total, how many vertices have been completed, are they failed or still running/waiting, etc.
- Data read cross/intra pod: Files and data are stored in multiple pods in distributed file system. The value here describes how much data has been read in the same pod or cross pod.
- Total compute time: The sum of every vertex execution time in the stage, you can consider it as the time it would take if all work in the stage is executed in only one vertex.
- Data and rows written/read: Indicates how much data or rows have been read/written, or need to be read.
- Vertex read failures: Describes how many vertices are failed while read data.
- Vertex duplicate discards: If a vertex runs too slow, the system may schedule multiple vertices to run the same piece of work. Reductant vertices will be discarded once one of the vertices complete successfully. Vertex duplicate discards records the number of vertices that are discarded as duplications in the stage.
- Vertex revocations: The vertex was succeeded, but get rerun later due to some reasons. For example, if downstream vertex loses intermediate input data, it will ask the upstream vertex to rerun.
- Vertex schedule executions: The total time that the vertices have been scheduled.

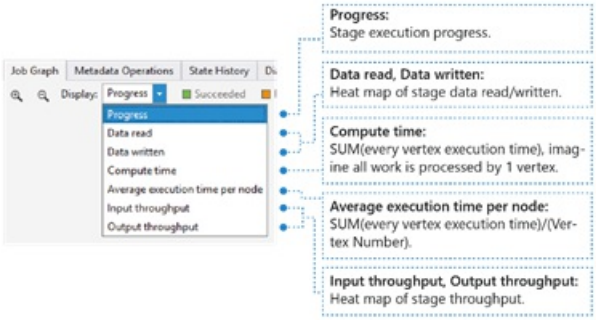
- Min/Average/Max Vertex data read: The minimum/average/maximum of every vertex read data.
- Duration: The wall clock time a stage takes, you need to load profile to see this value.
- Job Playback

  Data Lake Analytics runs jobs and archives the vertices running information of the jobs, such as when the vertices are started, stopped, failed and how they are retried, etc. All of the information is automatically logged in the query store and stored in its Job Profile. You can download the Job Profile through "Load Profile" in Job View, and you can view the Job Playback after downloading the Job Profile.

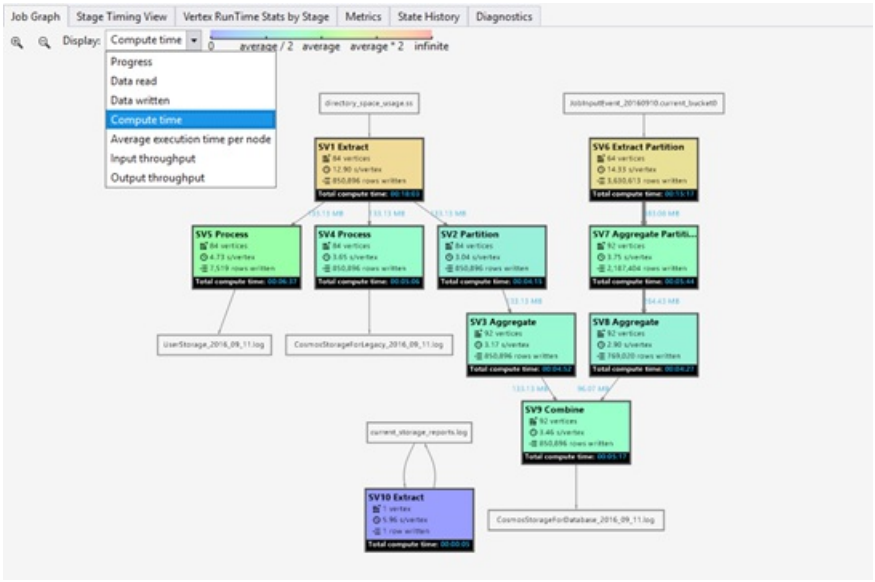
  Job Playback is an epitome visualization of what happened in the cluster. It helps you watch job execution progress and visually detect out performance anomalies and bottlenecks in a very short time (less than 30s usually).

- Job Heat Map Display

  Job Heat Map can be selected through the Display dropdown in Job Graph.




  It shows the I/O, time and throughput heat map of a job, through which you can find where the job spends most of the time, or whether your job is an I/O boundary job, and so on.

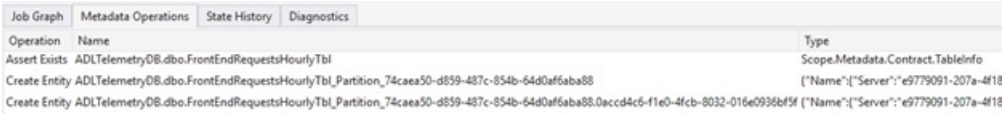



  - Progress: The job execution progress, see Information in stage information.
  - Data read/written: The heat map of total data read/written in each stage.
  - Compute time: The heat map of SUM (every vertex execution time), you can consider this as how long it would take if all work in the stage is executed with only 1 vertex.
  - Average execution time per node: The heat map of SUM (every vertex execution time) / (Vertex Number). Which means if you could assign all the vertices executed in parallelism, the whole stage will be done in this time frame.
  - Input/Output throughput: The heat map of input/output throughput of each stage, you can

confirm if your job is an I/O bound job through this.

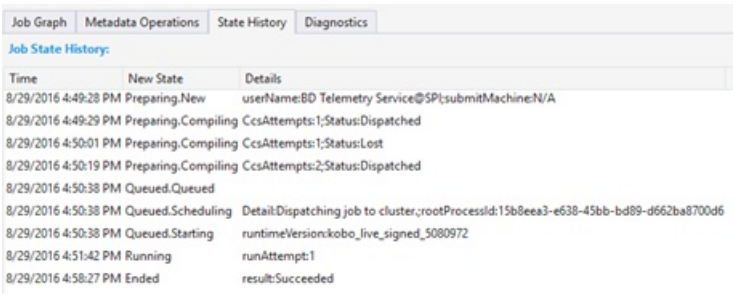
- Metadata Operations

  You can perform some metadata operations in your U-SQL script, such as create a database, drop a table, etc. These operations are shown in Metadata Operation after compilation. You may find assertions, create entities, drop entities here.



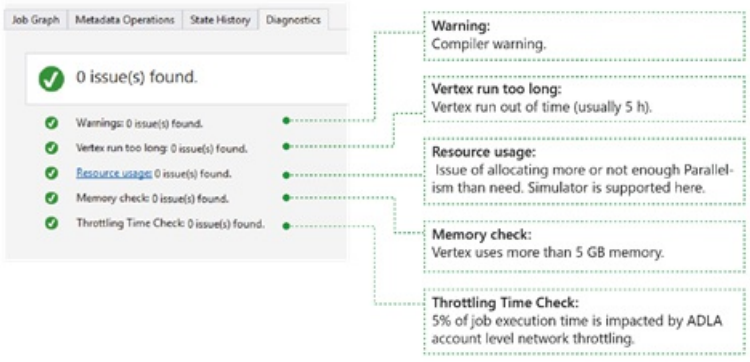

- State History

  The State History is also visualized in Job Summary, but you can get more details here. You can find the detailed information such as when the job is prepared, queued, started running, ended. Also you can find how many times the job has been compiled (the CcsAttempts: 1), when is the job dispatched to the cluster actually (the Detail: Dispatching job to cluster), etc.




- Diagnostics

  The tool diagnoses job execution automatically. You will receive alerts when there are some errors or performance issues in your jobs. Please note that you need to download Profile to get full information here.




  - Warnings: An alert shows up here with compiler warning. You can click “x issue(s)” link to have more details once the alert appears.
  - Vertex run too long: If any vertex runs out of time (say 5 hours), issues will be found here.
  - Resource usage: If you allocated more or not enough Parallelism than need, issues will be found here. Also you can click Resource usage to see more details and perform what-if scenarios to find a better resource allocation (for more details, see this guide).
  - Memory check: If any vertex uses more than 5 GB of memory, issues will be found here. Job execution may get killed by system if it uses more memory than system limitation.

# Job Detail

Job Detail shows the detailed information of the job, including Script, Resources and Vertex Execution View.


- Script

  The U-SQL script of the job is stored in the query store. You can view the original U-SQL script and re-submit it if needed.

- Resources

  You can find the job compilation outputs stored in the query store through Resources. For instance, you can find "algebra.xml" which is used to show the Job Graph, the assemblies you registered, etc. here.

- Vertex execution view

  It shows vertices execution details. The Job Profile archives every vertex execution log, such as total data read/written, runtime, state, etc. Through this view, you can get more details on how a job ran. For more information, see Use the Vertex Execution View in Data Lake Tools for Visual Studio.

## Next Steps

- To log diagnostics information, see Accessing diagnostics logs for Azure Data Lake Analytics
- To see a more complex query, see Analyze Website logs using Azure Data Lake Analytics.
- To use vertex execution view, see Use the Vertex Execution View in Data Lake Tools for Visual Studio

# Debug user-defined C# code for failed U-SQL jobs

8/27/2018 • 3 minutes to read • Edit Online

U-SQL provides an extensibility model using C#. In U-SQL scripts, it is easy to call C# functions and perform analytic functions that SQL-like declarative language does not support. To learn more for U-SQL extensibility, see U-SQL programmability guide.

In practice, any code may need debugging, but it is hard to debug a distributed job with custom code on the cloud with limited log files. Azure Data Lake Tools for Visual Studio provides a feature called **Failed Vertex Debug**, which helps you more easily debug the failures that occur in your custom code. When U-SQL job fails, the service keeps the failure state and the tool helps you to download the cloud failure environment to the local machine for debugging. The local download captures the entire cloud environment, including any input data and user code.

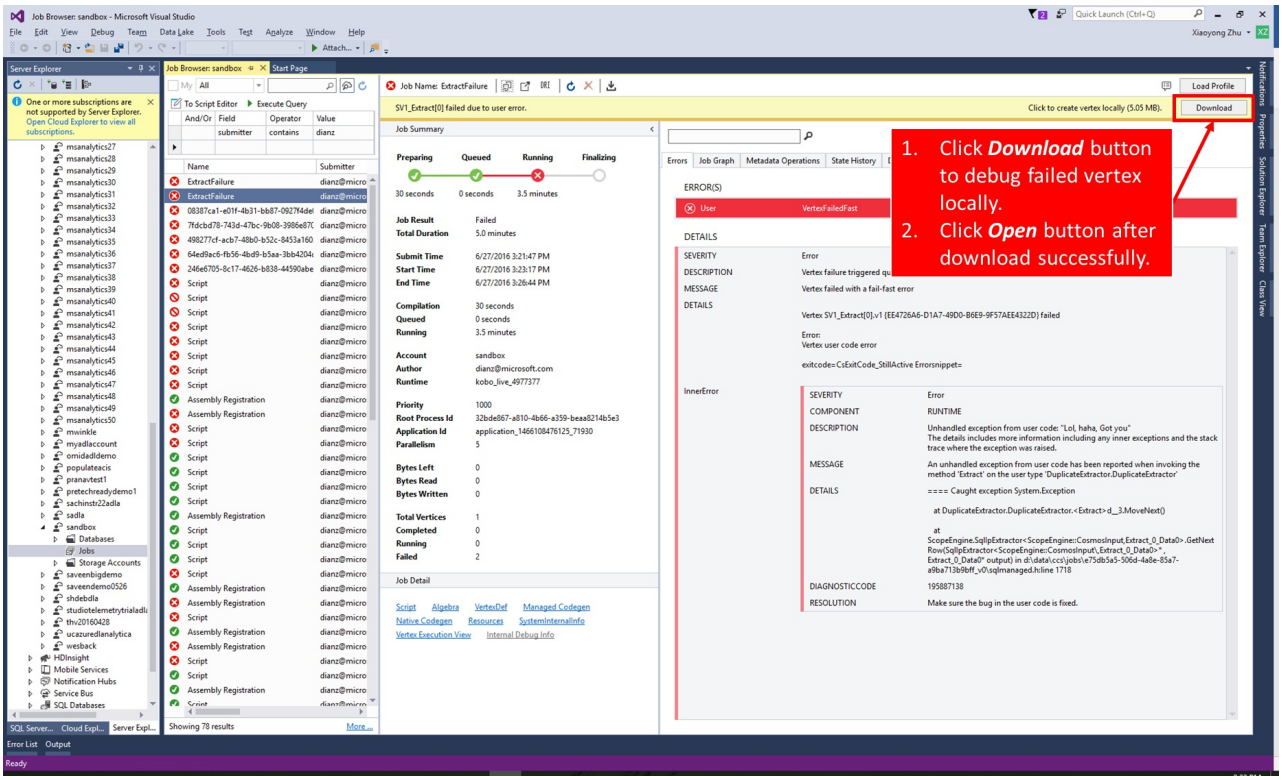
The following video demonstrates Failed Vertex Debug in Azure Data Lake Tools for Visual Studio.

> **IMPORTANT**
> Visual Studio requires the following two updates for using this feature: Microsoft Visual C++ 2015 Redistributable Update 3 and the Universal C Runtime for Windows.

## Download failed vertex to local machine

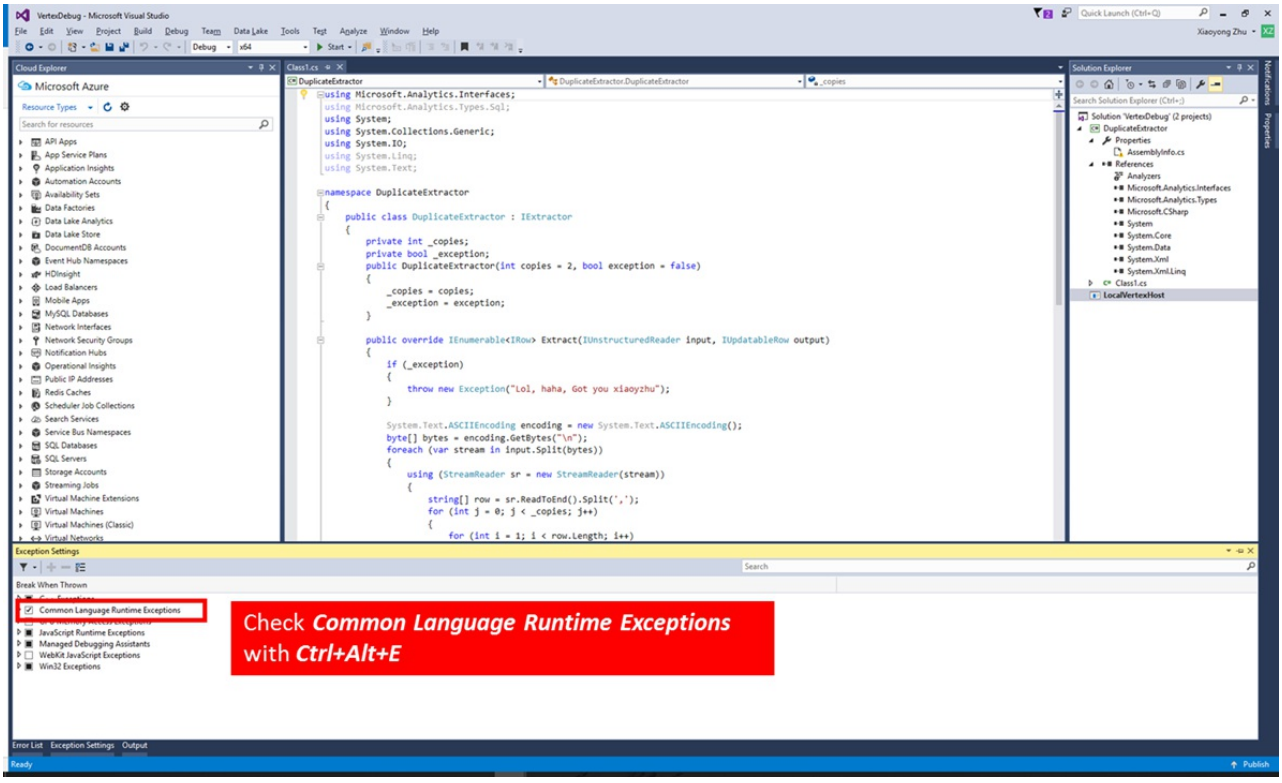
When you open a failed job in Azure Data Lake Tools for Visual Studio, you see a yellow alert bar with detailed error messages in the error tab.

1. Click **Download** to download all the required resources and input streams. If the download doesn't complete, click **Retry**.
2. Click **Open** after the download completes to generate a local debugging environment. A new debugging solution will be opened, and if you have existing solution opened in Visual Studio, please make sure to save and close it before debugging.




# Configure the debugging environment

> **NOTE**
>
> Before debugging, be sure to check **Common Language Runtime Exceptions** in the Exception Settings window (**Ctrl + Alt + E**).




In the new launched Visual Studio instance, you may or may not find the user-defined C# source code:

1. I can find my source code in the solution
2. I cannot find my source code in the solution

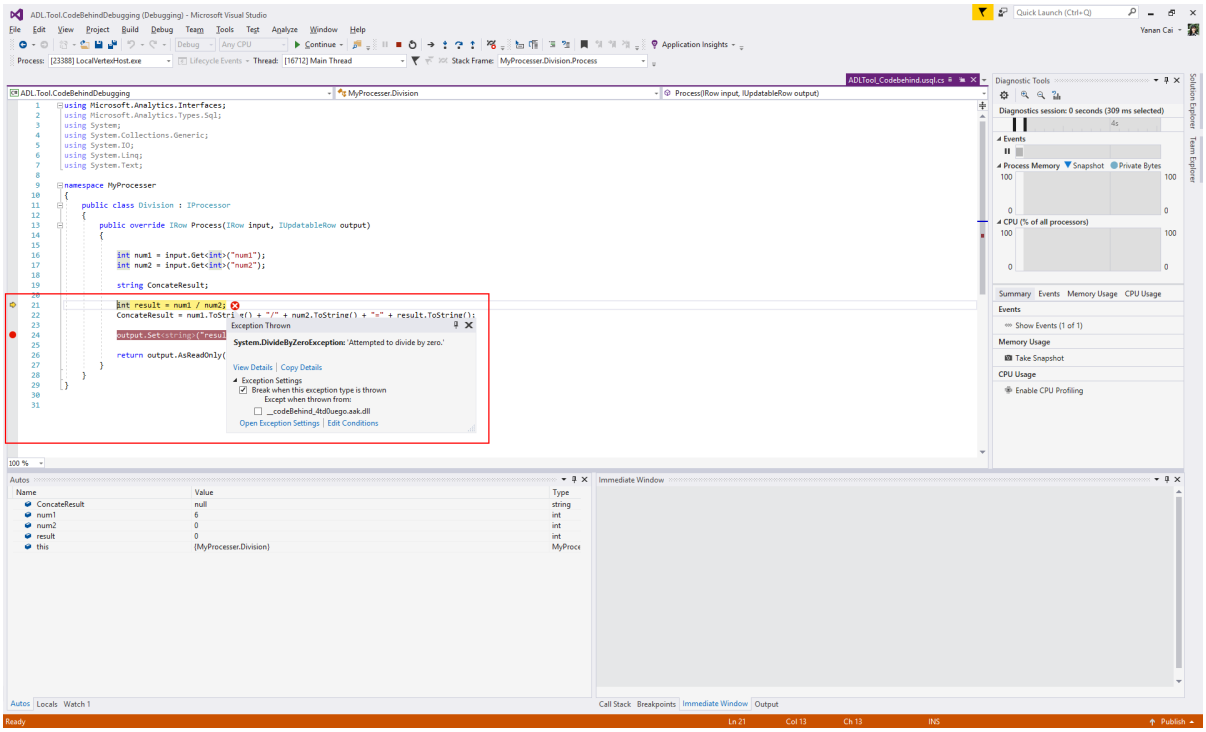
### Source code is included in debugging solution

There are two cases that the C# source code is captured:

1. The user code is defined in code-behind file (typically named `Script.usql.cs` in a U-SQL project).
2. The user code is defined in C# class library project for U-SQL application, and registered as assembly with **debug info**.

If the source code is imported to the solution, you can use the Visual Studio debugging tools (watch, variables, etc.) to troubleshoot the problem:

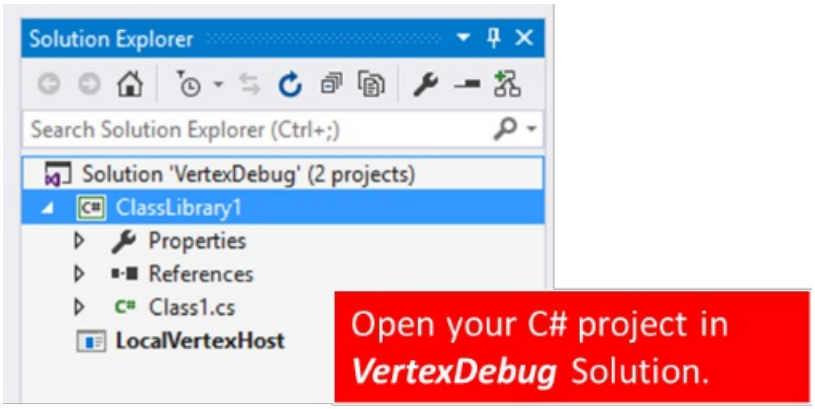
1. Press **F5** to start debugging. The code runs until it is stopped by an exception.
2. Open the source code file and set breakpoints, then press **F5** to debug the code step by step.



### Source code is not included in debugging solution

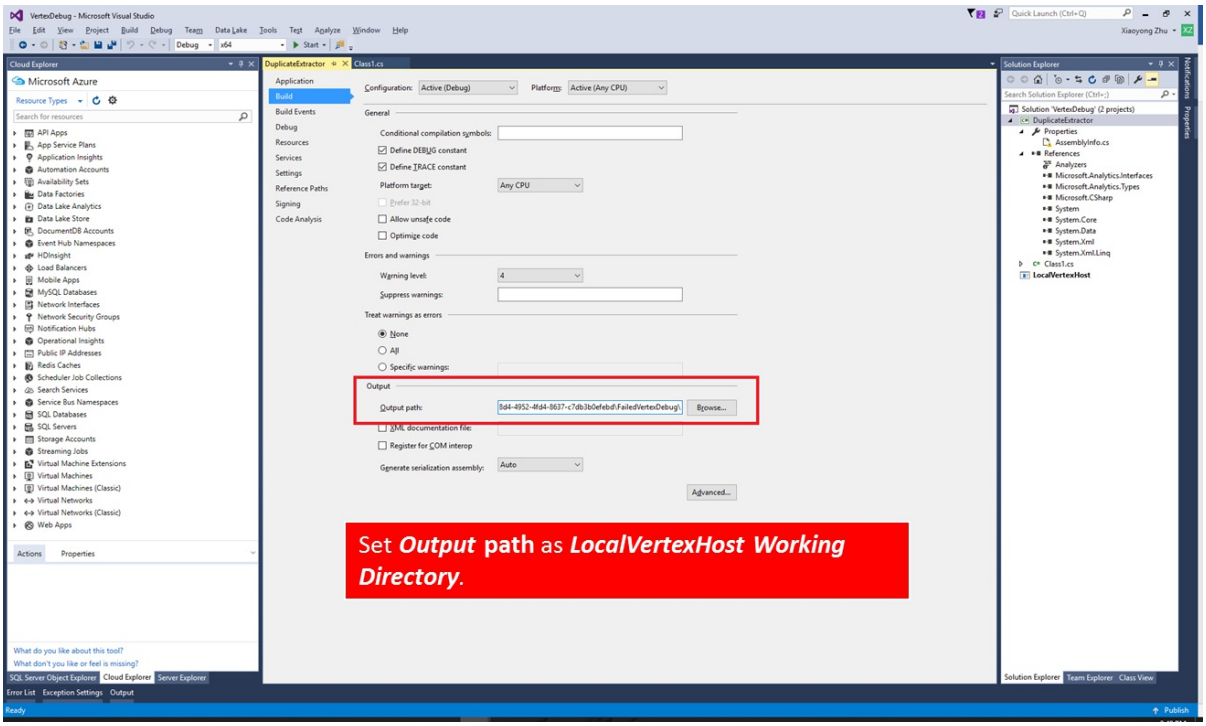
If the user code is not included in code-behind file, or you did not register the assembly with **debug info**, then the source code is not included automatically in the debugging solution. In this case, you need extra steps to add your source code:

1. Right-click **Solution 'VertexDebug' > Add > Existing Project...** to find the assembly source code and add the project to the debugging solution.




2. Get the project folder path for **FailedVertexDebugHost** project.

3. Right-Click **the added assembly source code project > Properties**, select the **Build** tab at left, and paste the copied path ending with \bin\debug as **Output > Output path**. The final output path is like "\fd91dd21-776e-4729-a78b-81ad85a4fba6\loiu0t1y.mfo\FailedVertexDebug\FailedVertexDebugHost\bin\Debug".



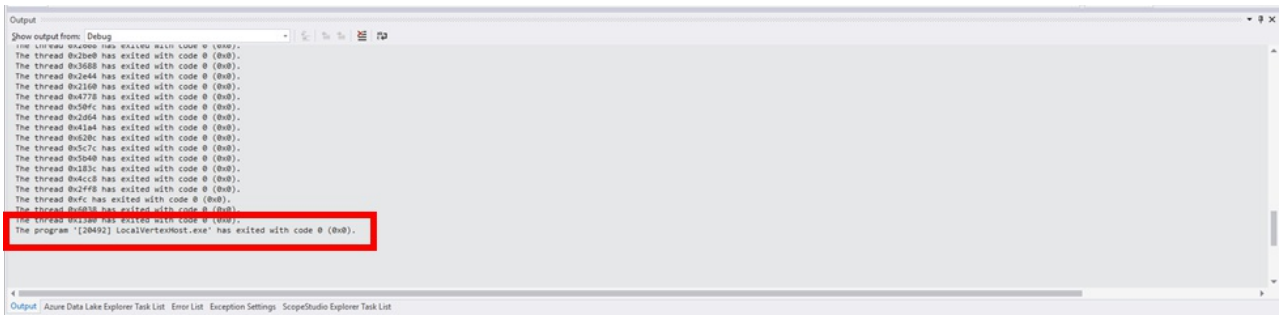

After these settings, start debugging with **F5** and breakpoints. You can also use the Visual Studio debugging tools (watch, variables, etc.) to troubleshoot the problem.

> **NOTE**
>
> Rebuild the assembly source code project each time after you modify the code to generate updated .pdb files.

# Resubmit the job

After debugging, if the project completes successfully the output window shows the following message:

```
The Program 'LocalVertexHost.exe' has exited with code 0 (0x0).
```



To resubmit the failed job:

1. For jobs with code-behind solutions, copy the C# code into the code-behind source file (typically `Script.usql.cs`).

2. For jobs with assemblies, right-click the assembly source code project in debugging solution and register the updated .dll assemblies into your Azure Data Lake catalog.

3. Resubmit the U-SQL job.

## Next steps

- U-SQL programmability guide
- Develop U-SQL User-defined operators for Azure Data Lake Analytics jobs
- Test and debug U-SQL jobs by using local run and the Azure Data Lake U-SQL SDK
- How to troubleshoot an abnormal recurring job

# Troubleshoot an abnormal recurring job

8/27/2018 • 2 minutes to read • Edit Online

This article shows how to use Azure Data Lake Tools for Visual Studio to troubleshoot problems with recurring jobs. Learn more about pipeline and recurring jobs from the Azure Data Lake and Azure HDInsight blog.

Recurring jobs usually share the same query logic and similar input data. For example, imagine that you have a recurring job running every Monday morning at 8 A.M. to count last week's weekly active user. The scripts for these jobs share one script template that contains the query logic. The inputs for these jobs are the usage data for last week. Sharing the same query logic and similar input usually means that performance of these jobs is similar and stable. If one of your recurring jobs suddenly performs abnormally, fails, or slows down a lot, you might want to:

- See the statistics reports for the previous runs of the recurring job to see what happened.
- Compare the abnormal job with a normal one to figure out what has been changed.

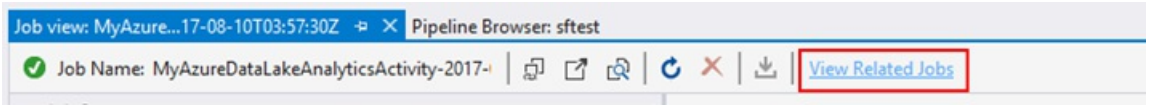
**Related Job View** in Azure Data Lake Tools for Visual Studio helps you accelerate the troubleshooting progress with both cases.

## Step 1: Find recurring jobs and open Related Job View

To use Related Job View to troubleshoot a recurring job problem, you need to first find the recurring job in Visual Studio and then open Related Job View.

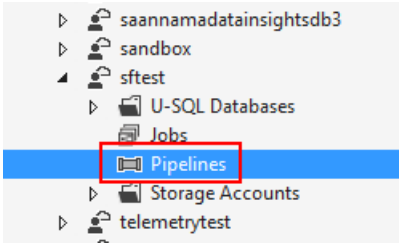
**Case 1: You have the URL for the recurring job**

Through **Tools** > **Data Lake** > **Job View**, you can paste the job URL to open Job View in Visual Studio. Select **View Related Jobs** to open Related Job View.

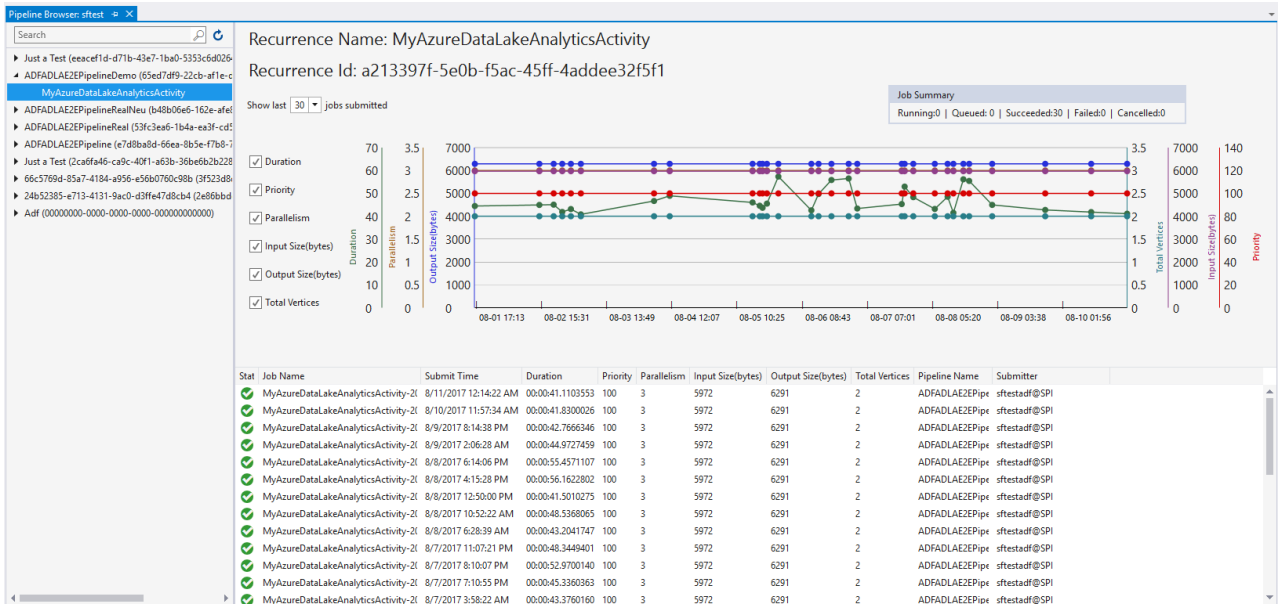



**Case 2: You have the pipeline for the recurring job, but not the URL**

In Visual Studio, you can open Pipeline Browser through Server Explorer > your Azure Data Lake Analytics account > **Pipelines**. (If you can't find this node in Server Explorer, download the latest plug-in.)



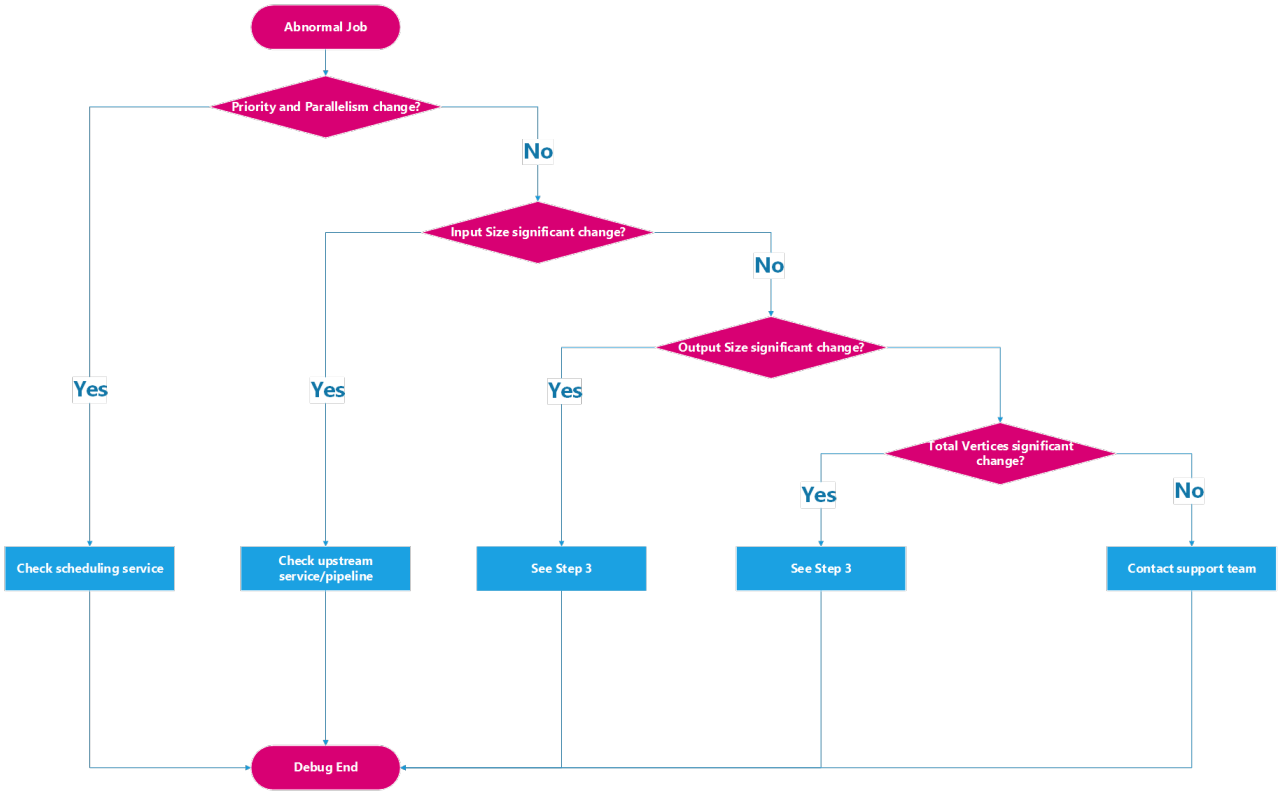

In Pipeline Browser, all pipelines for the Data Lake Analytics account are listed at left. You can expand the pipelines to find all recurring jobs, and then select the one that has problems. Related Job View opens at right.




## Step 2: Analyze a statistics report

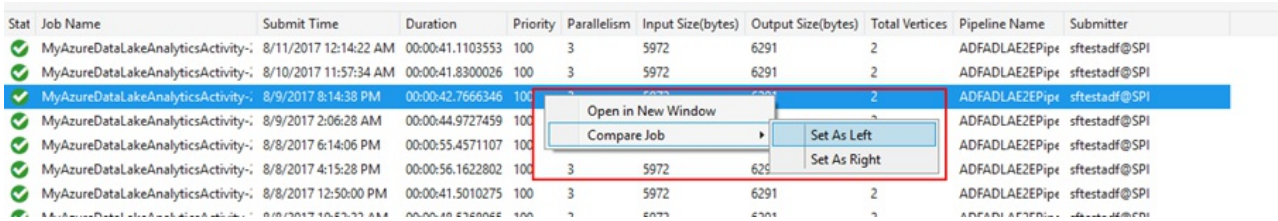
A summary and a statistics report are shown at top of Related Job View. There, you can find the potential root cause of the problem.

1. In the report, the X-axis shows the job submission time. Use it to find the abnormal job.
2. Use the process in the following diagram to check statistics and get insights about the problem and the possible solutions.

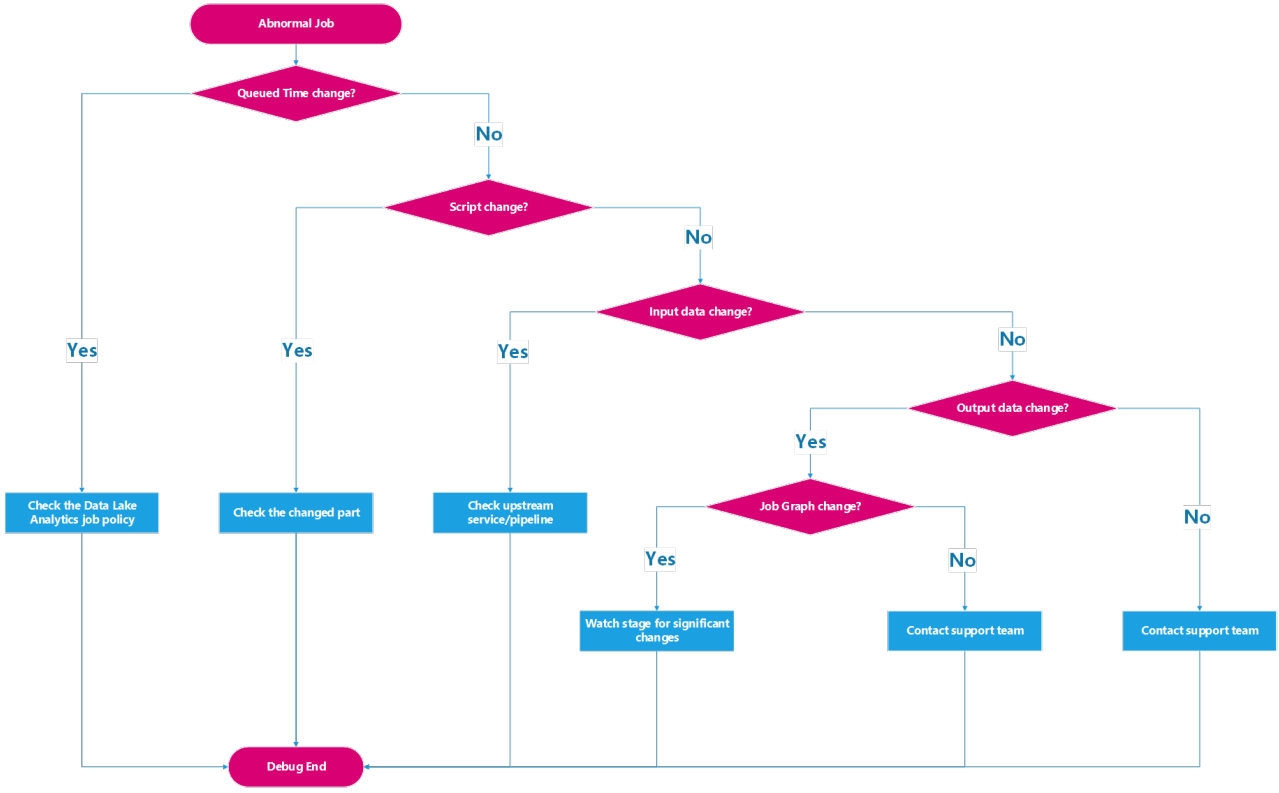



## Step 3: Compare the abnormal job to a normal job

You can find all submitted recurring jobs through the job list at the bottom of Related Job View. To find more insights and potential solutions, right-click the abnormal job. Use the Job Diff view to compare the abnormal job with a previous normal one.




Pay attention to the big differences between these two jobs. Those differences are probably causing the performance problems. To check further, use the steps in the following diagram:




## Next steps

- Resolve data-skew problems
- Debug user-defined C# code for failed U-SQL jobs

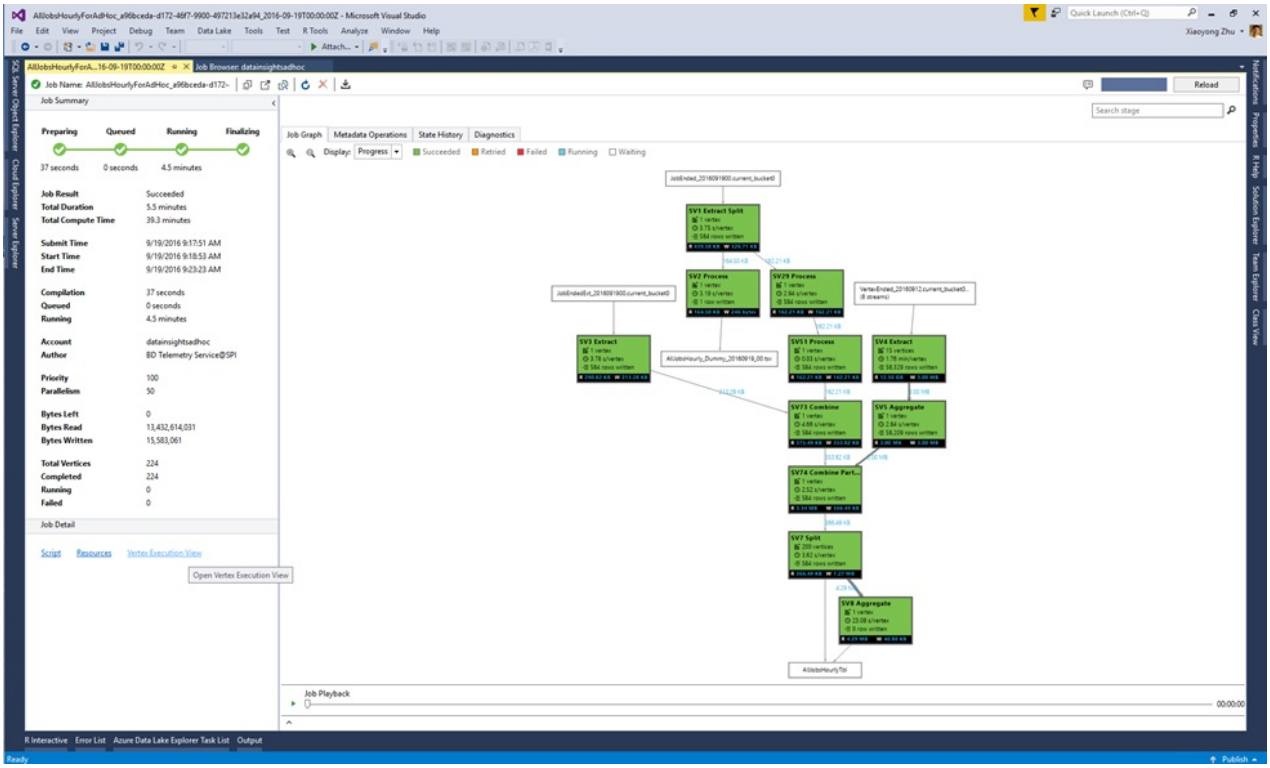
# Use the Vertex Execution View in Data Lake Tools for Visual Studio

8/27/2018 • 2 minutes to read • Edit Online

Learn how to use the Vertex Execution View to exam Data Lake Analytics jobs.

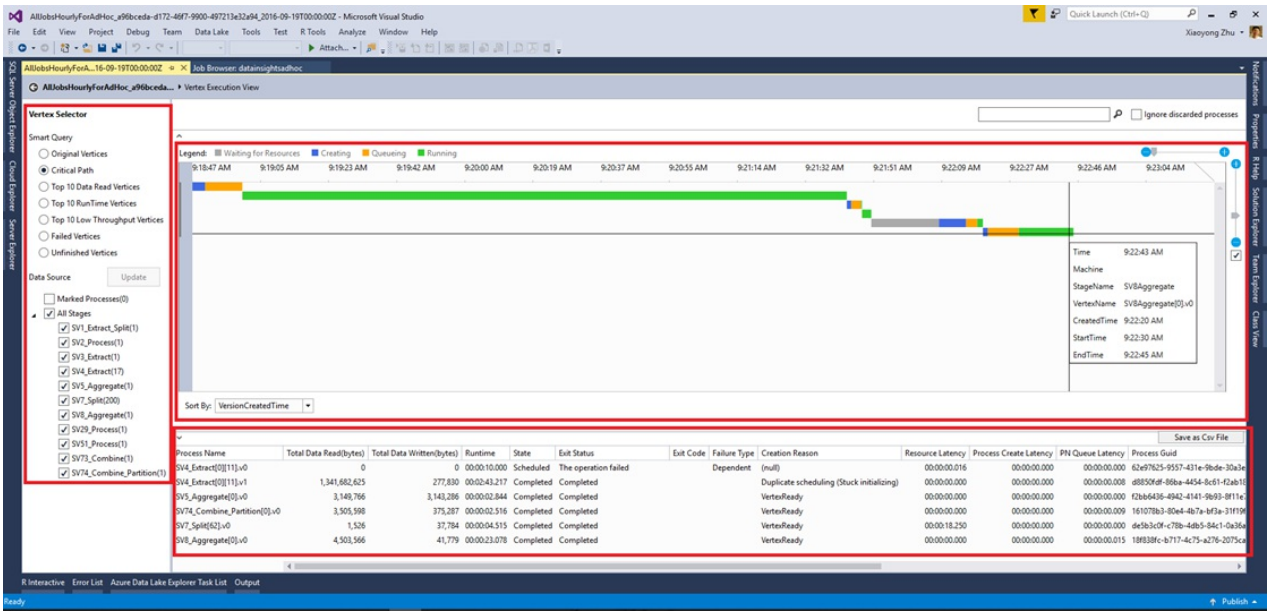
## Open the Vertex Execution View

Open a U-SQL job in Data Lake Tools for Visual Studio. Click **Vertex Execution View** in the bottom left corner. You may be prompted to load profiles first and it can take some time depending on your network connectivity.

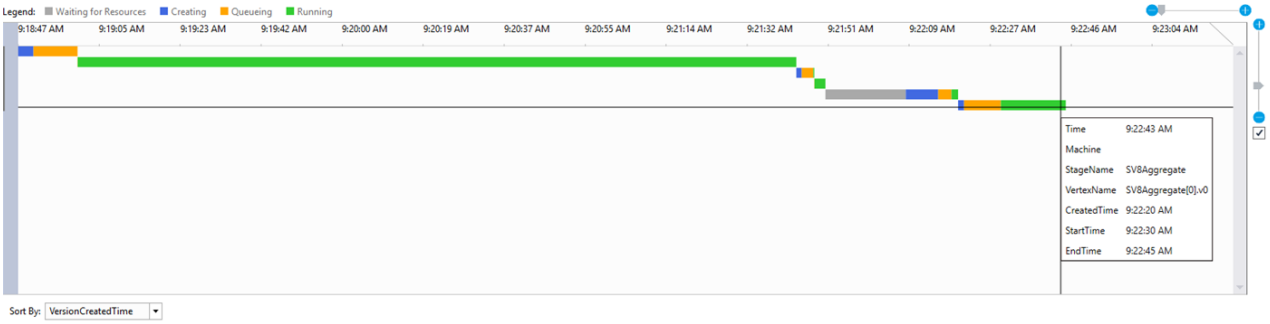



## Understand Vertex Execution View

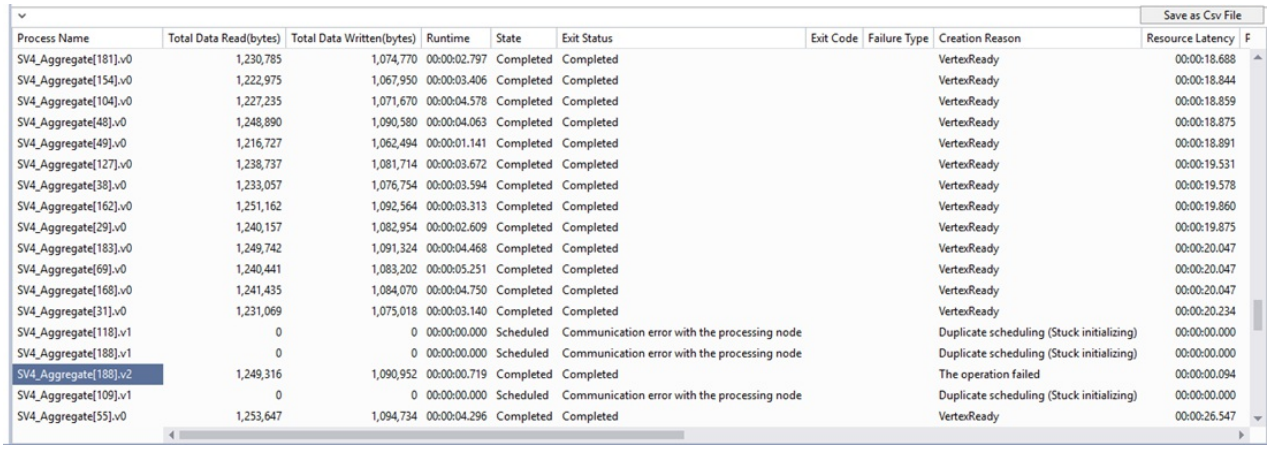
The Vertex Execution View has three parts:



The **Vertex selector** on the left lets you select vertices by features (such as top 10 data read, or choose by stage). One of the most commonly-used filters is to see the **vertices on critical path**. The **Critical path** is the longest chain of vertices of a U-SQL job. Understanding the critical path is useful for optimizing your jobs by checking which vertex takes the longest time.




The top center pane shows the **running status of all the vertices**.



| Process Name | Total Data Read(bytes) | Total Data Written(bytes) | Runtime | State | Exit Status | Exit Code | Failure Type | Creation Reason | Resource Latency | F |
|---|---|---|---|---|---|---|---|---|---|---|
| SV4_Aggregate[181].v0 | 1,230,785 | 1,074,770 | 00:00:02.797 | Completed | Completed | | | VertexReady | 00:00:18.688 | |
| SV4_Aggregate[154].v0 | 1,222,975 | 1,067,950 | 00:00:03.406 | Completed | Completed | | | VertexReady | 00:00:18.844 | |
| SV4_Aggregate[104].v0 | 1,227,235 | 1,071,670 | 00:00:04.578 | Completed | Completed | | | VertexReady | 00:00:18.859 | |
| SV4_Aggregate[48].v0 | 1,248,890 | 1,090,580 | 00:00:04.063 | Completed | Completed | | | VertexReady | 00:00:18.875 | |
| SV4_Aggregate[49].v0 | 1,216,727 | 1,062,494 | 00:00:01.141 | Completed | Completed | | | VertexReady | 00:00:18.891 | |
| SV4_Aggregate[127].v0 | 1,238,737 | 1,081,714 | 00:00:03.672 | Completed | Completed | | | VertexReady | 00:00:19.531 | |
| SV4_Aggregate[38].v0 | 1,233,057 | 1,076,754 | 00:00:03.594 | Completed | Completed | | | VertexReady | 00:00:19.578 | |
| SV4_Aggregate[162].v0 | 1,251,162 | 1,092,564 | 00:00:03.313 | Completed | Completed | | | VertexReady | 00:00:19.860 | |
| SV4_Aggregate[29].v0 | 1,240,157 | 1,082,954 | 00:00:02.609 | Completed | Completed | | | VertexReady | 00:00:19.875 | |
| SV4_Aggregate[183].v0 | 1,249,742 | 1,091,324 | 00:00:04.468 | Completed | Completed | | | VertexReady | 00:00:20.047 | |
| SV4_Aggregate[69].v0 | 1,240,441 | 1,083,202 | 00:00:05.251 | Completed | Completed | | | VertexReady | 00:00:20.047 | |
| SV4_Aggregate[168].v0 | 1,241,435 | 1,084,070 | 00:00:04.750 | Completed | Completed | | | VertexReady | 00:00:20.047 | |
| SV4_Aggregate[31].v0 | 1,231,069 | 1,075,018 | 00:00:03.140 | Completed | Completed | | | VertexReady | 00:00:20.234 | |
| SV4_Aggregate[118].v1 | 0 | 0 | 00:00:00.000 | Scheduled | Communication error with the processing node | | | Duplicate scheduling (Stuck initializing) | 00:00:00.000 | |
| SV4_Aggregate[188].v1 | 0 | 0 | 00:00:00.000 | Scheduled | Communication error with the processing node | | | Duplicate scheduling (Stuck initializing) | 00:00:00.000 | |
| SV4_Aggregate[188].v2 | 1,249,316 | 1,090,952 | 00:00:00.719 | Completed | Completed | | | The operation failed | 00:00:00.094 | |
| SV4_Aggregate[109].v1 | 0 | 0 | 00:00:00.000 | Scheduled | Communication error with the processing node | | | Duplicate scheduling (Stuck initializing) | 00:00:00.000 | |
| SV4_Aggregate[55].v0 | 1,253,647 | 1,094,734 | 00:00:04.296 | Completed | Completed | | | VertexReady | 00:00:26.547 | |

The bottom center pane shows information about each vertex:

- Process Name: The name of the vertex instance. It is composed of different parts in StageName|VertexName|VertexRunInstance. For example, the SV7_Split[62].v1 vertex stands for the second running instance (.v1, index starting from 0) of Vertex number 62 in Stage SV7_Split.
- Total Data Read/Written: The data was read/written by this vertex.
- State/Exit Status: The final status when the vertex is ended.
- Exit Code/Failure Type: The error when the vertex failed.
- Creation Reason: Why the vertex was created.

- Resource Latency/Process Latency/PN Queue Latency: the time taken for the vertex to wait for resources, to process data, and to stay in the queue.
- Process/Creator GUID: GUID for the current running vertex or its creator.
- Version: the N-th instance of the running vertex (the system might schedule new instances of a vertex for many reasons, for example failover, compute redundancy, etc.)
- Version Created Time.
- Process Create Start Time/Process Queued Time/Process Start Time/Process Complete Time: when the vertex process starts creation; when the vertex process starts to queue; when the certain vertex process starts; when the certain vertex is completed.

## Next steps

- To log diagnostics information, see Accessing diagnostics logs for Azure Data Lake Analytics
- To see a more complex query, see Analyze Website logs using Azure Data Lake Analytics.
- To view job details, see Use Job Browser and Job View for Azure Data lake Analytics jobs

# Export a U-SQL database

8/27/2018 • 3 minutes to read • Edit Online

In this article, learn how to use Azure Data Lake Tools for Visual Studio to export a U-SQL database as a single U-SQL script and downloaded resources. You can import the exported database to a local account in the same process.

Customers usually maintain multiple environments for development, test, and production. These environments are hosted on both a local account, on a developer's local computer, and in an Azure Data Lake Analytics account in Azure.

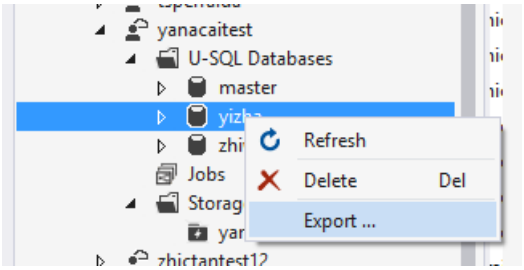
When you develop and tune U-SQL queries in development and test environments, developers often need to re-create their work in a production database. The Database Export Wizard helps accelerate this process. By using the wizard, developers can clone the existing database environment and sample data to other Data Lake Analytics accounts.

## Export steps

**Step 1: Export the database in Server Explorer**

All Data Lake Analytics accounts that you have permissions for are listed in Server Explorer. To export the database:

1. In Server Explorer, expand the account that contains the database that you want to export.
2. Right-click the database, and then select **Export**.

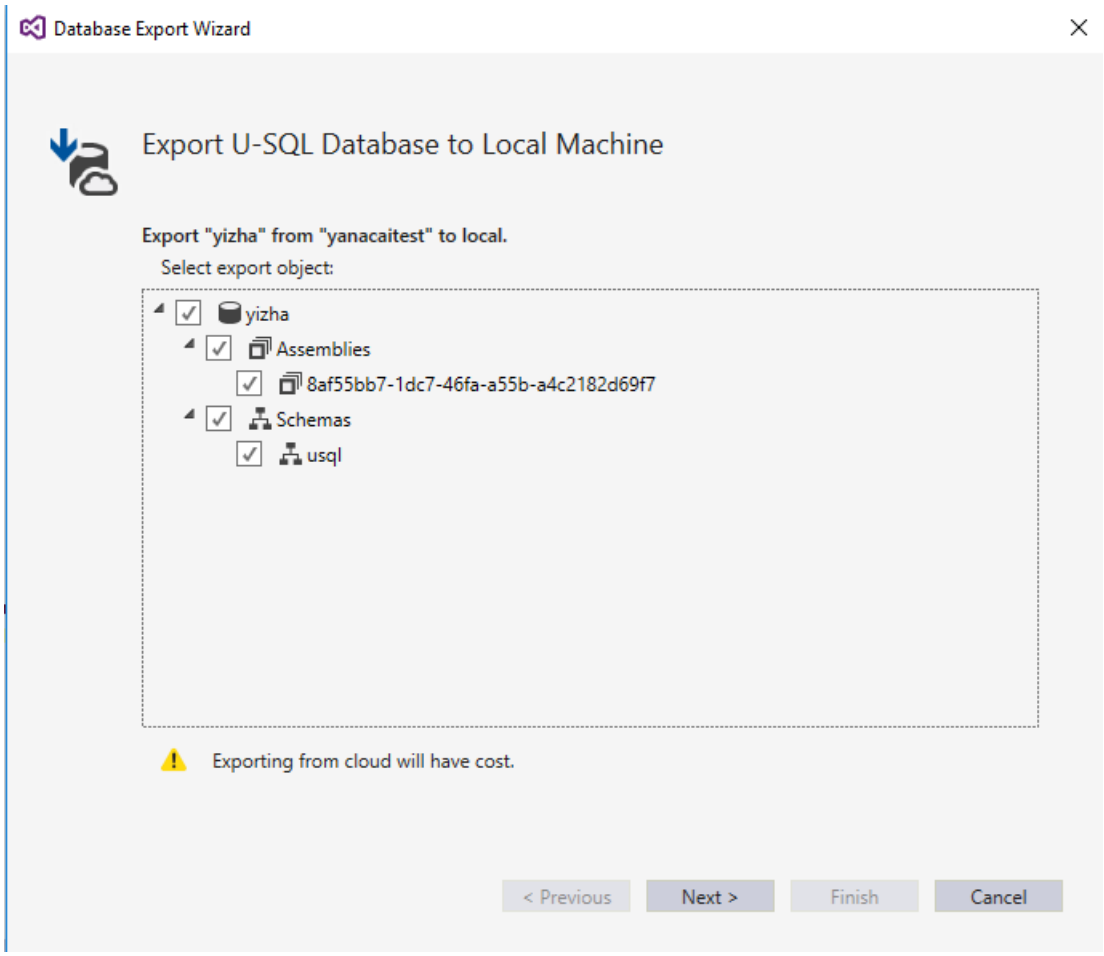



If the **Export** menu option isn't available, you need to update the tool to the lasted release.

**Step 2: Configure the objects that you want to export**

If you need only a small part of a large database, you can configure a subset of objects that you want to export in the export wizard.

The export action is completed by running a U-SQL job. Therefore, exporting from an Azure account incurs some cost.

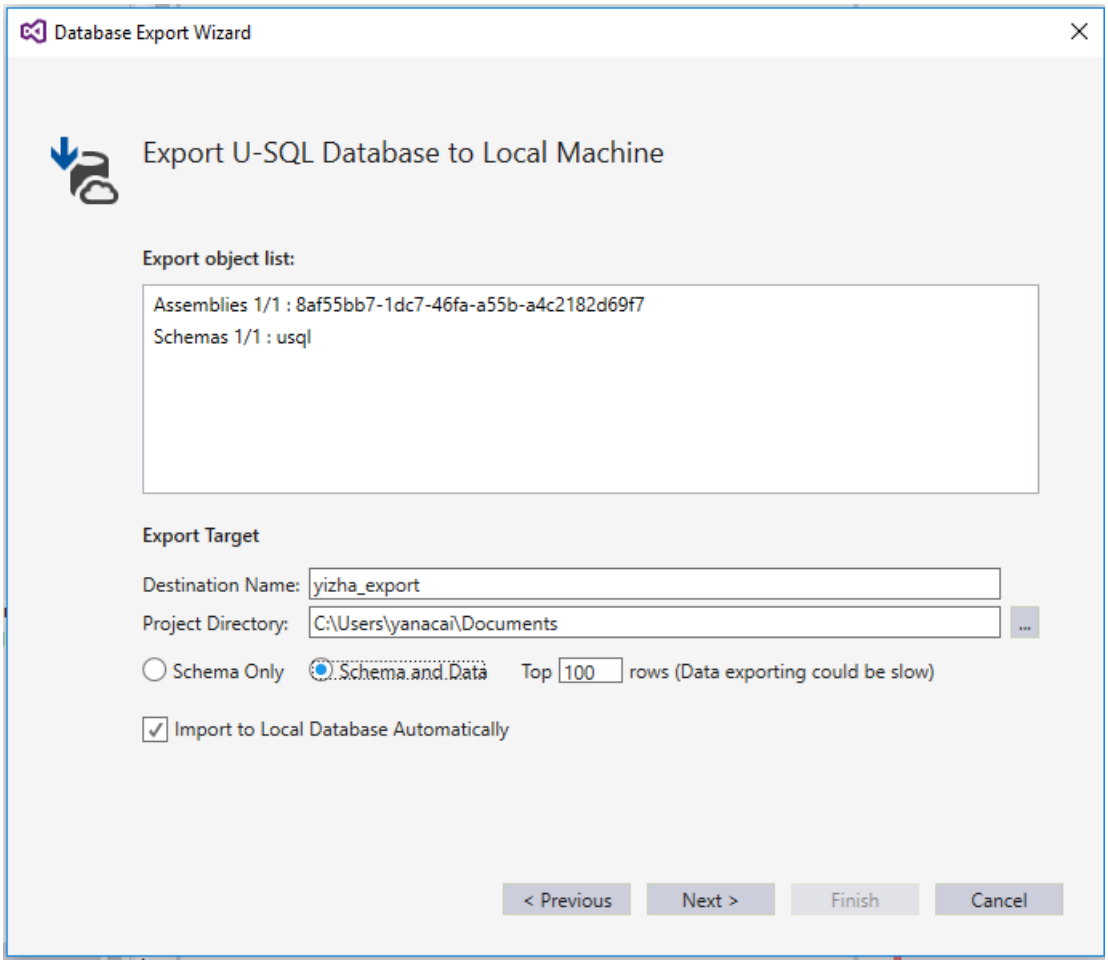



**Step 3: Check the objects list and other configurations**

In this step, you can verify the selected objects in the **Export object list** box. If there are any errors, select **Previous** to go back and correctly configure the objects that you want to export.

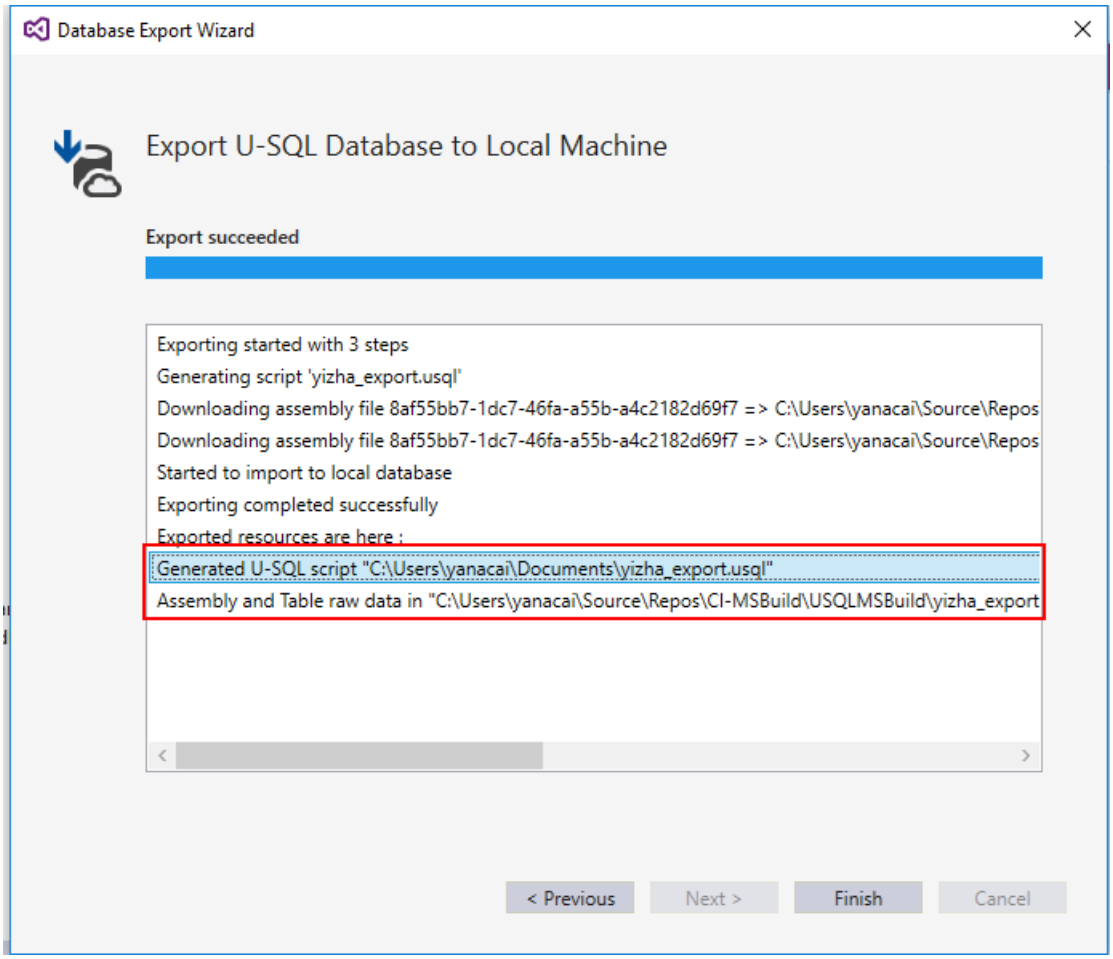
You can also configure other settings for the export target. Configuration descriptions are listed in the following table:

| CONFIGURATION | DESCRIPTION |
|---|---|
| Destination Name | This name indicates where you want to save the exported database resources. Examples are assemblies, additional files, and sample data. A folder with this name is created under your local data root folder. |
| Project Directory | This path defines where you want to save the exported U-SQL script. All database object definitions are saved at this location. |
| Schema Only | If you select this option, only database definitions and resources (like assemblies and additional files) are exported. |
| Schema and Data | If you select this option, database definitions, resources, and data are exported. The top N rows of tables are exported. |
| Import to Local Database Automatically | If you select this option, the exported database is automatically imported to your local database when exporting is finished. |




**Step 4: Check the export results**

When exporting is finished, you can view the exported results in the log window in the wizard. The following example shows how to find exported U-SQL script and database resources, including assemblies, additional files, and sample data:




## Import the exported database to a local account

The most convenient way to import the exported database is to select the **Import to Local Database Automatically** check box during the exporting process in Step 3. If you didn't check this box, first, find the exported U-SQL script in the export log. Then, run the U-SQL script locally to import the database to your local account.

## Import the exported database to a Data Lake Analytics account

To import the database to different Data Lake Analytics account:

1. Upload the exported resources, including assemblies, additional files, and sample data, to the default Azure Data Lake Store account of the Data Lake Analytics account that you want to import to. You can find the exported resource folder under the local data root folder. Upload the entire folder to the root of the default Data Lake Store account.
2. When uploading is finished, submit the exported U-SQL script to the Data Lake Analytics account that you want to import the database to.

## Known limitations

Currently, if you select the **Schema and Data** option in Step 3, the tool runs a U-SQL job to export the data stored in tables. Because of this, the data exporting process might be slow and you might incur costs.

## Next steps

- Learn about U-SQL databases
- Test and debug U-SQL jobs by using local run and the Azure Data Lake U-SQL SDK

# Analyze Website logs using Azure Data Lake Analytics

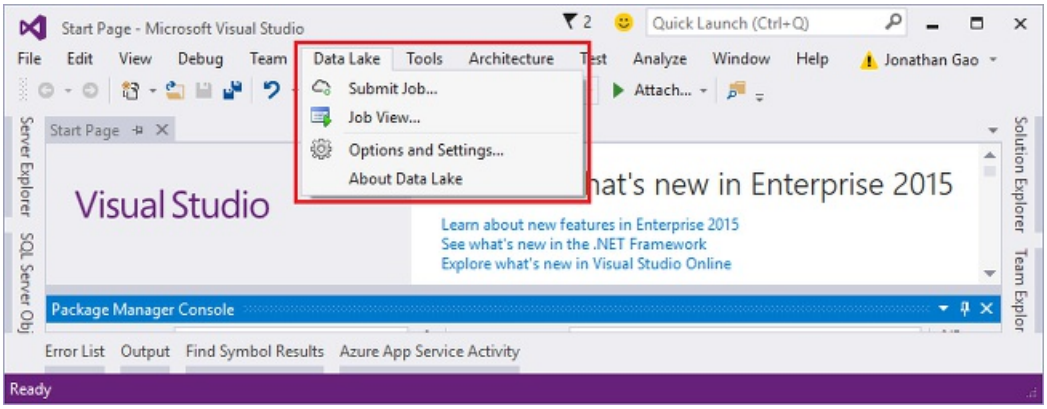
8/27/2018 • 4 minutes to read • Edit Online

Learn how to analyze website logs using Data Lake Analytics, especially on finding out which referrers ran into errors when they tried to visit the website.

## Prerequisites

- **Visual Studio 2015 or Visual Studio 2013**.
- **Data Lake Tools for Visual Studio**.

  Once Data Lake Tools for Visual Studio is installed, you will see a **Data Lake** item in the **Tools** menu in Visual Studio:




- **Basic knowledge of Data Lake Analytics and the Data Lake Tools for Visual Studio**. To get started, see:
  - Develop U-SQL script using Data Lake tools for Visual Studio.
- **A Data Lake Analytics account.** See Create an Azure Data Lake Analytics account.
- **Install the sample data.** In the Azure Portal, open you Data Lake Analytics account and click **Sample Scripts** on the left menu, then click **Copy Sample Data**.

## Connect to Azure

Before you can build and test any U-SQL scripts, you must first connect to Azure.

**To connect to Data Lake Analytics**

1. Open Visual Studio.
2. Click **Data Lake > Options and Settings**.
3. Click **Sign In**, or **Change User** if someone has signed in, and follow the instructions.
4. Click **OK** to close the Options and Settings dialog.

**To browse your Data Lake Analytics accounts**

1. From Visual Studio, open **Server Explorer** by press **CTRL+ALT+S**.
2. From **Server Explorer**, expand **Azure**, and then expand **Data Lake Analytics**. You shall see a list of your Data Lake Analytics accounts if there are any. You cannot create Data Lake Analytics accounts from the studio.

To create an account, see Get Started with Azure Data Lake Analytics using Azure Portal or Get Started with Azure Data Lake Analytics using Azure PowerShell.

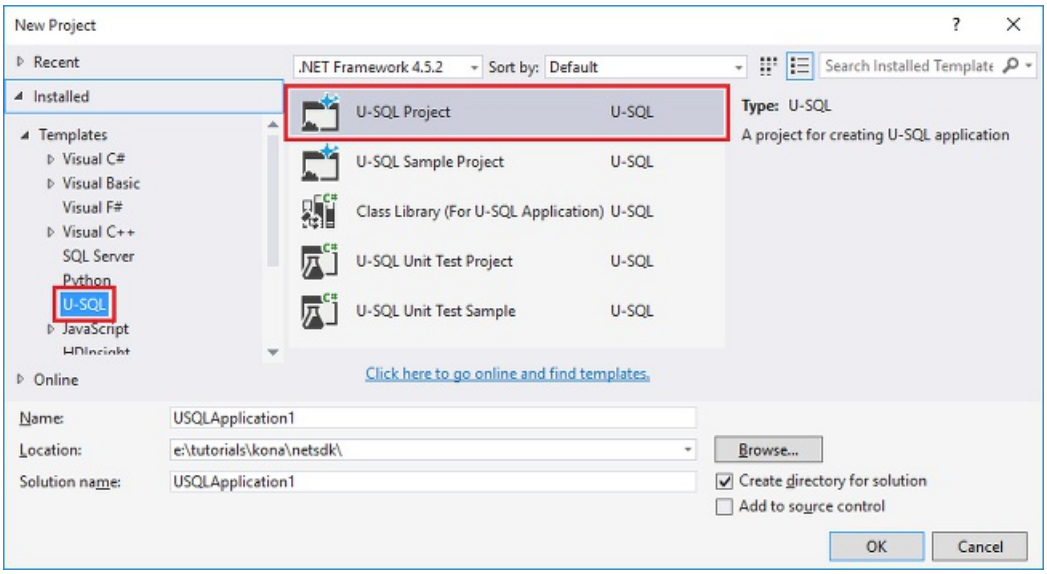
## Develop U-SQL application

A U-SQL application is mostly a U-SQL script. To learn more about U-SQL, see Get started with U-SQL.

You can add addition user-defined operators to the application. For more information, see Develop U-SQL user defined operators for Data Lake Analytics jobs.

**To create and submit a Data Lake Analytics job**

1. Click the **File > New > Project**.
2. Select the U-SQL Project type.




3. Click **OK**. Visual studio creates a solution with a Script.usql file.
4. Enter the following script into the Script.usql file:

```
// Create a database for easy reuse, so you don't need to read from a file every time.
CREATE DATABASE IF NOT EXISTS SampleDBTutorials;

// Create a Table valued function. TVF ensures that your jobs fetch data from the weblog file with the
correct schema.
DROP FUNCTION IF EXISTS SampleDBTutorials.dbo.WeblogsView;
CREATE FUNCTION SampleDBTutorials.dbo.WeblogsView()
RETURNS @result TABLE
(
    s_date DateTime,
    s_time string,
    s_sitename string,
    cs_method string,
    cs_uristem string,
    cs_uriquery string,
    s_port int,
    cs_username string,
    c_ip string,
    cs_useragent string,
    cs_cookie string,
    cs_referer string,
    cs_host string,
    sc_status int,
    sc_substatus int,
    sc_win32status int,
    sc_bytes int,
    cs_bytes int,
```

```
    s_timetaken int
)
AS
BEGIN

    @result = EXTRACT
        s_date DateTime,
        s_time string,
        s_sitename string,
        cs_method string,
        cs_uristem string,
        cs_uriquery string,
        s_port int,
        cs_username string,
        c_ip string,
        cs_useragent string,
        cs_cookie string,
        cs_referer string,
        cs_host string,
        sc_status int,
        sc_substatus int,
        sc_win32status int,
        sc_bytes int,
        cs_bytes int,
        s_timetaken int
    FROM @"/Samples/Data/WebLog.log"
    USING Extractors.Text(delimiter:' ');
    RETURN;
END;

// Create a table for storing referrers and status
DROP TABLE IF EXISTS SampleDBTutorials.dbo.ReferrersPerDay;
@weblog = SampleDBTutorials.dbo.WeblogsView();
CREATE TABLE SampleDBTutorials.dbo.ReferrersPerDay
(
    INDEX idx1
    CLUSTERED(Year ASC)
    DISTRIBUTED BY HASH(Year)
) AS

SELECT s_date.Year AS Year,
    s_date.Month AS Month,
    s_date.Day AS Day,
    cs_referer,
    sc_status,
    COUNT(DISTINCT c_ip) AS cnt
FROM @weblog
GROUP BY s_date,
        cs_referer,
        sc_status;
```

To understand the U-SQL, see Get started with Data Lake Analytics U-SQL language.

5. Add a new U-SQL script to your project and enter the following:

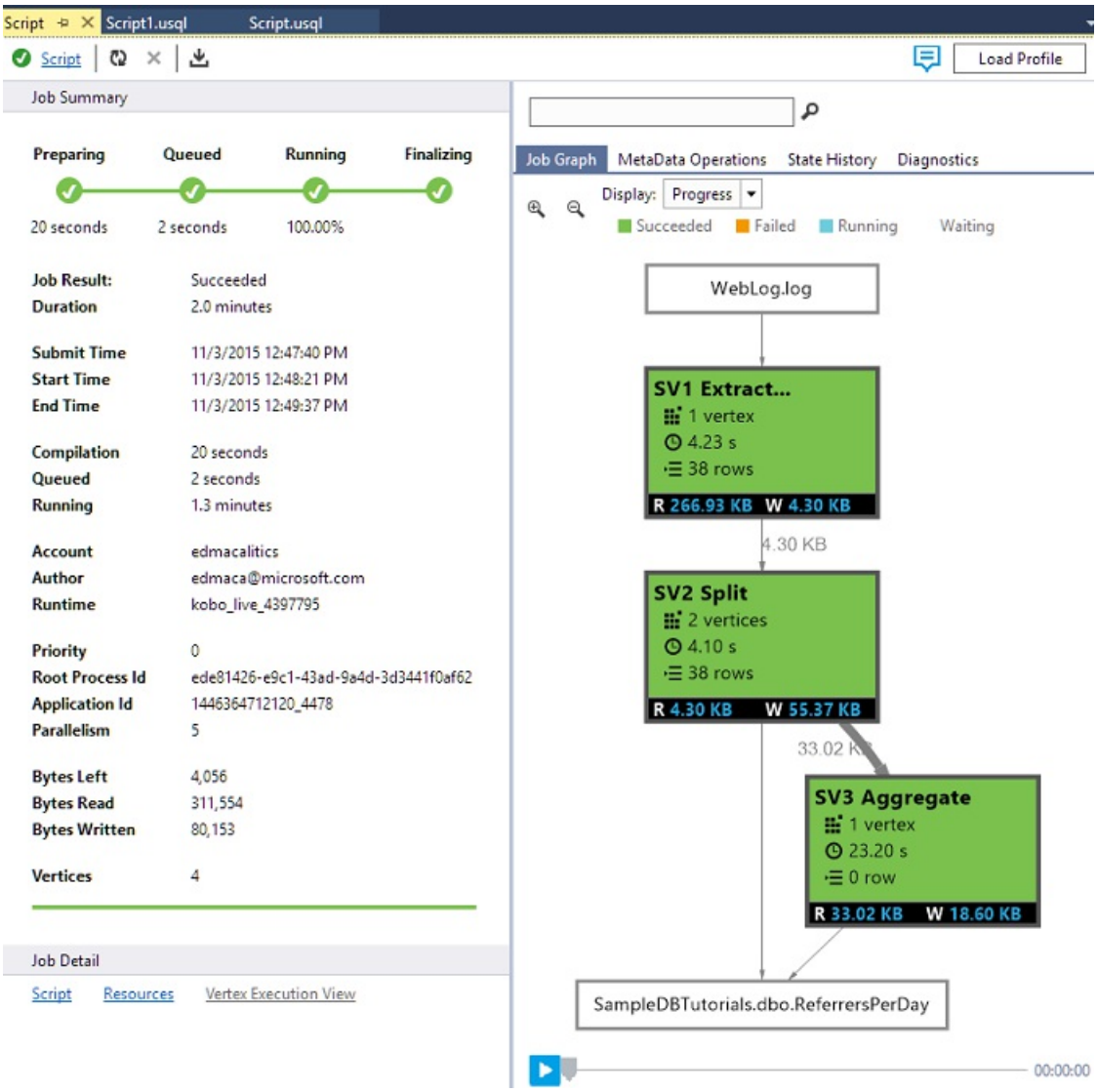
```
// Query the referrers that ran into errors
@content =
    SELECT *
    FROM SampleDBTutorials.dbo.ReferrersPerDay
    WHERE sc_status >=400 AND sc_status < 500;

OUTPUT @content
TO @"/Samples/Outputs/UnsuccessfulResponses.log"
USING Outputters.Tsv();
```

6. Switch back to the first U-SQL script and next to the **Submit** button, specify your Analytics account.

7. From **Solution Explorer**, right click **Script.usql**, and then click **Build Script**. Verify the results in the Output pane.
8. From **Solution Explorer**, right click **Script.usql**, and then click **Submit Script**.
9. Verify the **Analytics Account** is the one where you want to run the job, and then click **Submit**. Submission results and job link are available in the Data Lake Tools for Visual Studio Results window when the submission is completed.
10. Wait until the job is completed successfully. If the job failed, it is most likely missing the source file. Please see the Prerequisite section of this tutorial. For additional troubleshooting information, see Monitor and troubleshoot Azure Data Lake Analytics jobs.

    When the job is completed, you shall see the following screen:




11. Now repeat steps 7- 10 for **Script1.usql**.

**To see the job output**

1. From **Server Explorer**, expand **Azure**, expand **Data Lake Analytics**, expand your Data Lake Analytics account, expand **Storage Accounts**, right-click the default Data Lake Storage account, and then click **Explorer**.
2. Double-click **Samples** to open the folder, and then double-click **Outputs**.
3. Double-click **UnsuccessfulResponsees.log**.
4. You can also double-click the output file inside the graph view of the job in order to navigate directly to the output.

## See also

To get started with Data Lake Analytics using different tools, see:

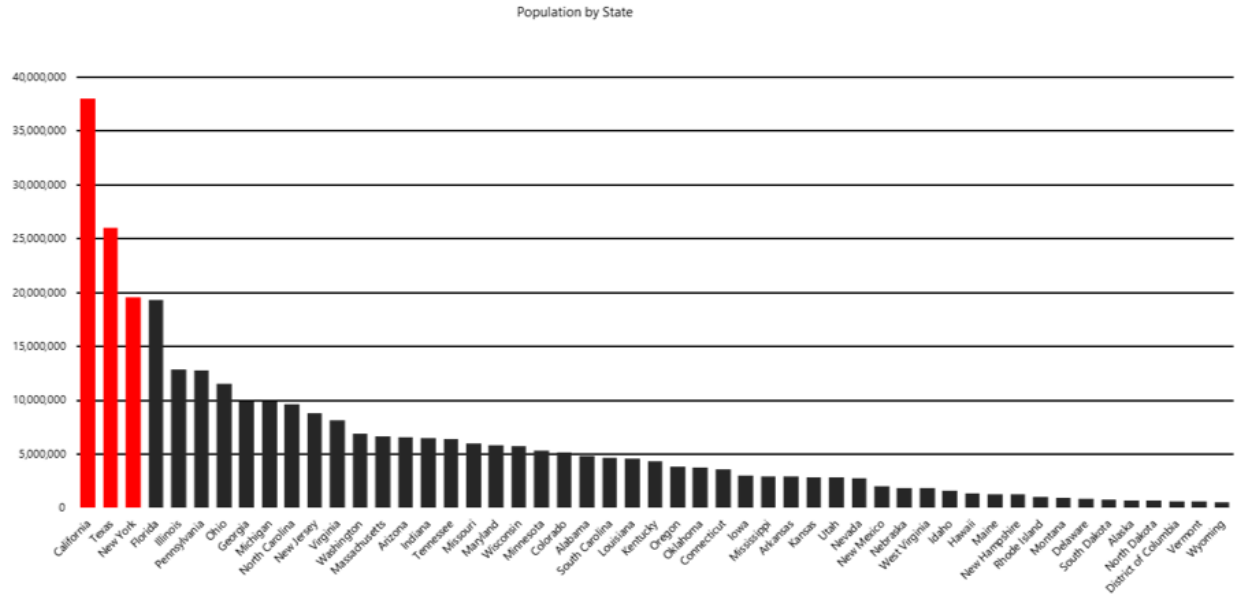
- Get started with Data Lake Analytics using Azure Portal
- Get started with Data Lake Analytics using Azure PowerShell
- Get started with Data Lake Analytics using .NET SDK

# Resolve data-skew problems by using Azure Data Lake Tools for Visual Studio

8/27/2018 • 7 minutes to read • Edit Online

## What is data skew?

Briefly stated, data skew is an over-represented value. Imagine that you have assigned 50 tax examiners to audit tax returns, one examiner for each US state. The Wyoming examiner, because the population there is small, has little to do. In California, however, the examiner is kept very busy because of the state's large population.




In our scenario, the data is unevenly distributed across all tax examiners, which means that some examiners must work more than others. In your own job, you frequently experience situations like the tax-examiner example here. In more technical terms, one vertex gets much more data than its peers, a situation that makes the vertex work more than the others and that eventually slows down an entire job. What's worse, the job might fail, because vertices might have, for example, a 5-hour runtime limitation and a 6-GB memory limitation.

## Resolving data-skew problems

Azure Data Lake Tools for Visual Studio can help detect whether your job has a data-skew problem. If a problem exists, you can resolve it by trying the solutions in this section.

## Solution 1: Improve table partitioning

**Option 1: Filter the skewed key value in advance**

If it does not affect your business logic, you can filter the higher-frequency values in advance. For example, if there are a lot of 000-000-000 in column GUID, you might not want to aggregate that value. Before you aggregate, you can write "WHERE GUID != "000-000-000"" to filter the high-frequency value.

**Option 2: Pick a different partition or distribution key**

In the preceding example, if you want only to check the tax-audit workload all over the country, you can improve the data distribution by selecting the ID number as your key. Picking a different partition or distribution key can sometimes distribute the data more evenly, but you need to make sure that this choice doesn't affect your business logic. For instance, to calculate the tax sum for each state, you might want to designate *State* as the partition key. If

you continue to experience this problem, try using Option 3.

**Option 3: Add more partition or distribution keys**

Instead of using only *State* as a partition key, you can use more than one key for partitioning. For example, consider adding *ZIP Code* as an additional partition key to reduce data-partition sizes and distribute the data more evenly.

**Option 4: Use round-robin distribution**

If you cannot find an appropriate key for partition and distribution, you can try to use round-robin distribution. Round-robin distribution treats all rows equally and randomly puts them into corresponding buckets. The data gets evenly distributed, but it loses locality information, a drawback that can also reduce job performance for some operations. Additionally, if you are doing aggregation for the skewed key anyway, the data-skew problem will persist. To learn more about round-robin distribution, see the U-SQL Table Distributions section in CREATE TABLE (U-SQL): Creating a Table with Schema.

## Solution 2: Improve the query plan

**Option 1: Use the CREATE STATISTICS statement**

U-SQL provides the CREATE STATISTICS statement on tables. This statement gives more information to the query optimizer about the data characteristics, such as value distribution, that are stored in a table. For most queries, the query optimizer already generates the necessary statistics for a high-quality query plan. Occasionally, you might need to improve query performance by creating additional statistics with CREATE STATISTICS or by modifying the query design. For more information, see the CREATE STATISTICS (U-SQL) page.

Code example:

```
CREATE STATISTICS IF NOT EXISTS stats_SampleTable_date ON SampleDB.dbo.SampleTable(date) WITH FULLSCAN;
```

> **NOTE**
> Statistics information is not updated automatically. If you update the data in a table without re-creating the statistics, the query performance might decline.

**Option 2: Use SKEWFACTOR**

If you want to sum the tax for each state, you must use GROUP BY state, an approach that doesn't avoid the data-skew problem. However, you can provide a data hint in your query to identify data skew in keys so that the optimizer can prepare an execution plan for you.

Usually, you can set the parameter as 0.5 and 1, with 0.5 meaning not much skew and 1 meaning heavy skew. Because the hint affects execution-plan optimization for the current statement and all downstream statements, be sure to add the hint before the potential skewed key-wise aggregation.

```
SKEWFACTOR (columns) = x

Provides a hint that the given columns have a skew factor x from 0 (no skew) through 1 (very heavy skew).
```

Code example:

```
//Add a SKEWFACTOR hint.
@Impressions =
    SELECT * FROM
    searchDM.SML.PageView(@start, @end) AS PageView
    OPTION(SKEWFACTOR(Query)=0.5)
    ;

//Query 1 for key: Query, ClientId
@Sessions =
    SELECT
        ClientId,
        Query,
        SUM(PageClicks) AS Clicks
    FROM
        @Impressions
    GROUP BY
        Query, ClientId
    ;

//Query 2 for Key: Query
@Display =
    SELECT * FROM @Sessions
        INNER JOIN @Campaigns
            ON @Sessions.Query == @Campaigns.Query
    ;
```

**Option 3: Use ROWCOUNT**

In addition to SKEWFACTOR, for specific skewed-key join cases, if you know that the other joined row set is small, you can tell the optimizer by adding a ROWCOUNT hint in the U-SQL statement before JOIN. This way, optimizer can choose a broadcast join strategy to help improve performance. Be aware that ROWCOUNT does not resolve the data-skew problem, but it can offer some additional help.

```
OPTION(ROWCOUNT = n)

Identify a small row set before JOIN by providing an estimated integer row count.
```

Code example:

```
//Unstructured (24-hour daily log impressions)
@Huge   = EXTRACT ClientId int, ...
            FROM @"wasb://ads@wcentralus/2015/10/30/{*}.nif"
            ;

//Small subset (that is, ForgetMe opt out)
@Small  = SELECT * FROM @Huge
            WHERE Bing.ForgetMe(x,y,z)
            OPTION(ROWCOUNT=500)
            ;

//Result (not enough information to determine simple broadcast JOIN)
@Remove = SELECT * FROM Bing.Sessions
            INNER JOIN @Small ON Sessions.Client == @Small.Client
            ;
```

# Solution 3: Improve the user-defined reducer and combiner

You can sometimes write a user-defined operator to deal with complicated process logic, and a well-written reducer and combiner might mitigate a data-skew problem in some cases.

**Option 1: Use a recursive reducer, if possible**

By default, a user-defined reducer runs in non-recursive mode, which means that reduce work for a key is distributed into a single vertex. But if your data is skewed, the huge data sets might be processed in a single vertex and run for a long time.

To improve performance, you can add an attribute in your code to define reducer to run in recursive mode. Then, the huge data sets can be distributed to multiple vertices and run in parallel, which speeds up your job.

To change a non-recursive reducer to recursive, you need to make sure that your algorithm is associative. For example, the sum is associative, and the median is not. You also need to make sure that the input and output for reducer keep the same schema.

Attribute of recursive reducer:

```
[SqlUserDefinedReducer(IsRecursive = true)]
```

Code example:

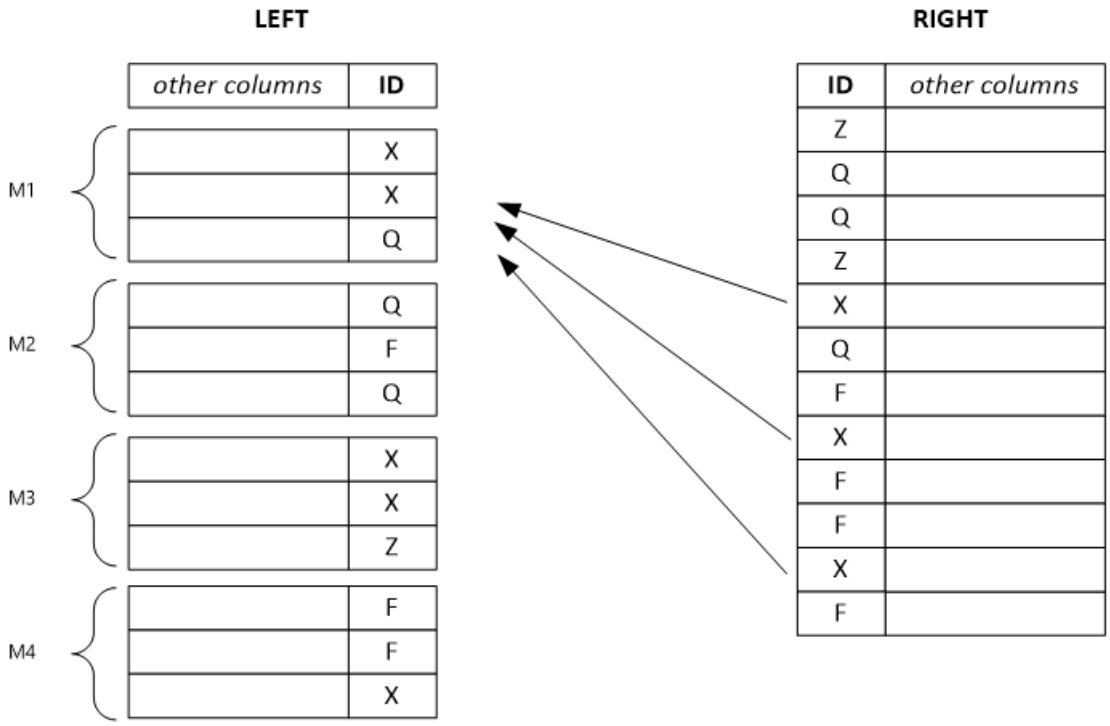
```
[SqlUserDefinedReducer(IsRecursive = true)]
public class TopNReducer : IReducer
{
    public override IEnumerable<IRow>
        Reduce(IRowset input, IUpdatableRow output)
    {
        //Your reducer code goes here.
    }
}
```

**Option 2: Use row-level combiner mode, if possible**

Similar to the ROWCOUNT hint for specific skewed-key join cases, combiner mode tries to distribute huge skewed-key value sets to multiple vertices so that the work can be executed concurrently. Combiner mode can't resolve data-skew issues, but it can offer some additional help for huge skewed-key value sets.

By default, the combiner mode is Full, which means that the left row set and right row set cannot be separated. Setting the mode as Left/Right/Inner enables row-level join. The system separates the corresponding row sets and distributes them into multiple vertices that run in parallel. However, before you configure the combiner mode, be careful to ensure that the corresponding row sets can be separated.

The example that follows shows a separated left row set. Each output row depends on a single input row from the left, and it potentially depends on all rows from the right with the same key value. If you set the combiner mode as left, the system separates the huge left-row set into small ones and assigns them to multiple vertices.




**NOTE**

If you set the wrong combiner mode, the combination is less efficient, and the results might be wrong.

Attributes of combiner mode:

- [SqlUserDefinedCombiner(Mode=CombinerMode.Full)]: Every output row potentially depends on all the input rows from left and right with the same key value.
- SqlUserDefinedCombiner(Mode=CombinerMode.Left): Every output row depends on a single input row from the left (and potentially all rows from the right with the same key value).
- qlUserDefinedCombiner(Mode=CombinerMode.Right): Every output row depends on a single input row from the right (and potentially all rows from the left with the same key value).
- SqlUserDefinedCombiner(Mode=CombinerMode.Inner): Every output row depends on a single input row from the left and the right with the same value.

Code example:

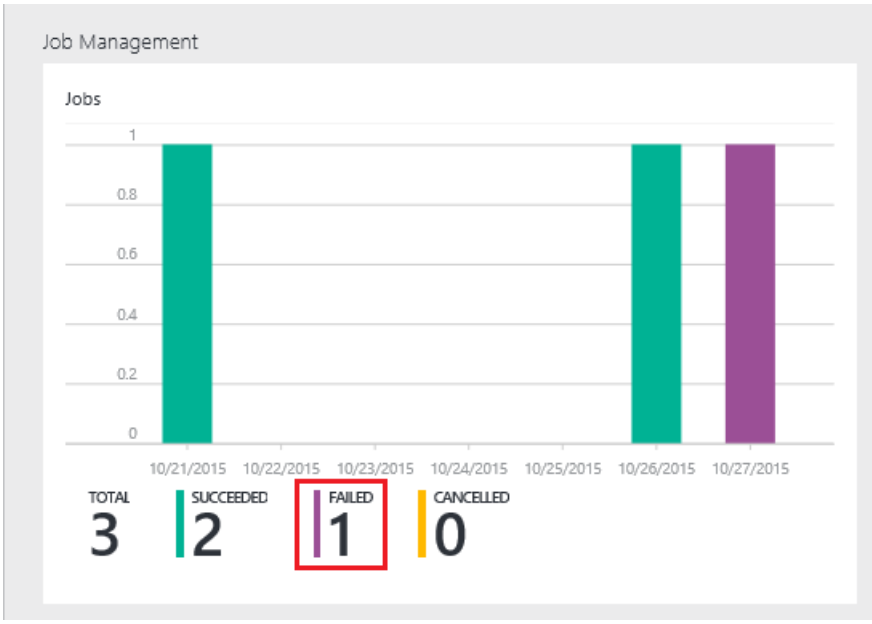
```
[SqlUserDefinedCombiner(Mode = CombinerMode.Right)]
public class WatsonDedupCombiner : ICombiner
{
    public override IEnumerable<IRow>
        Combine(IRowset left, IRowset right, IUpdatableRow output)
    {
    //Your combiner code goes here.
    }
}
```

# Monitor jobs in Azure Data Lake Analytics using the Azure Portal

8/27/2018 • 2 minutes to read • Edit Online

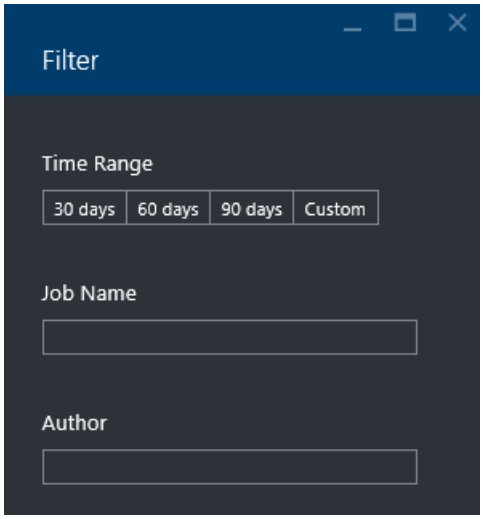
**To see all the jobs**

1. From the Azure portal, click **Microsoft Azure** in the upper left corner.
2. Click the tile with your Data Lake Analytics account name. The job summary is shown on the **Job Management** tile.

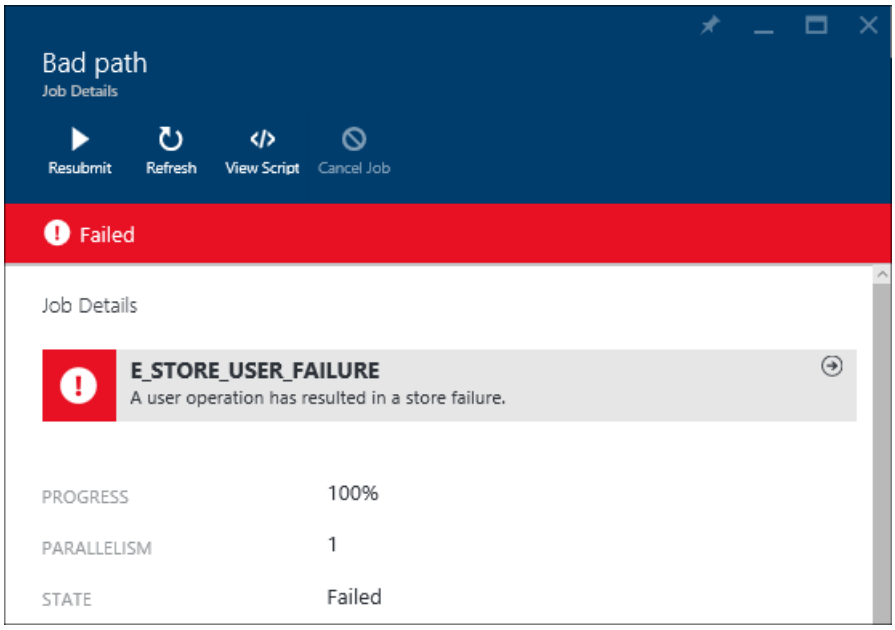



The job Management gives you a glance of the job status. Notice there is a failed job.

3. Click the **Job Management** tile to see the jobs. The jobs are categorized in **Running**, **Queued**, and **Ended**. You shall see your failed job in the **Ended** section. It shall be first one in the list. When you have a lot of jobs, you can click **Filter** to help you to locate jobs.

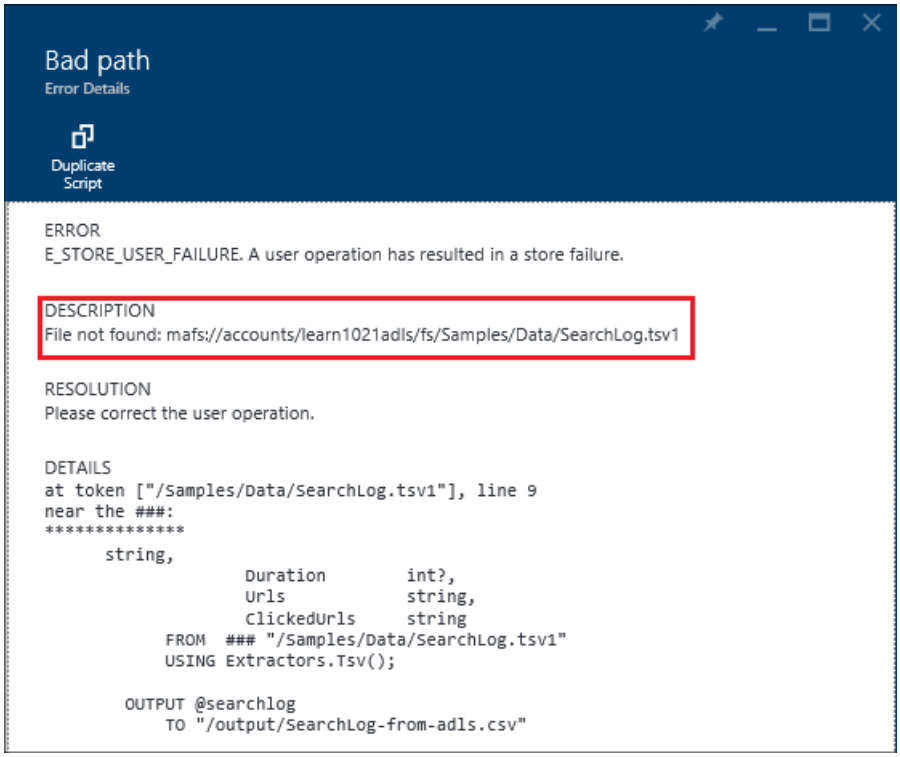



4. Click the failed job from the list to open the job details:




Notice the **Resubmit** button. After you fix the problem, you can resubmit the job.

5. Click highlighted part from the previous screenshot to open the error details. You shall see something like:




It tells you the source folder is not found.

6. Click **Duplicate Script**.
7. Update the **FROM** path to:

   "/Samples/Data/SearchLog.tsv"

8. Click **Submit Job**.

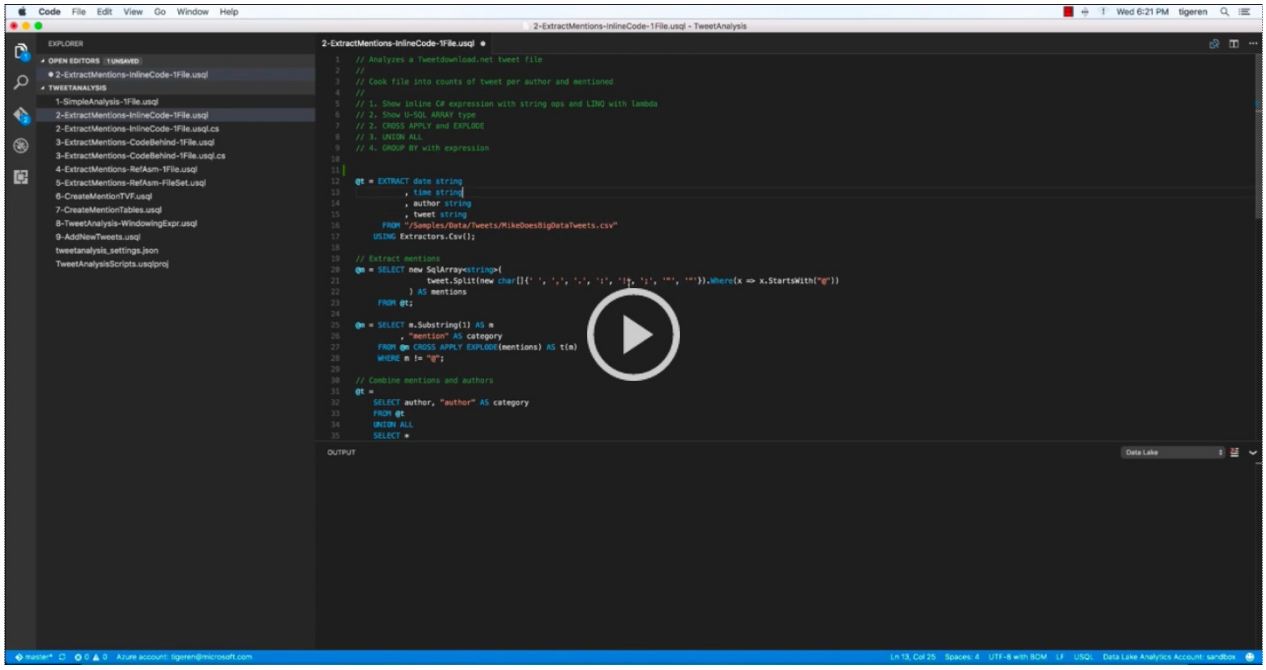
## See also

- Azure Data Lake Analytics overview
- Get started with Azure Data Lake Analytics using Azure PowerShell
- Manage Azure Data Lake Analytics using Azure portal

# Use Azure Data Lake Tools for Visual Studio Code

9/14/2018 • 14 minutes to read • Edit Online

In this article, learn how you can use Azure Data Lake Tools for Visual Studio Code (VS Code) to create, test, and run U-SQL scripts. The information is also covered in the following video:



## Prerequisites

Azure Data Lake Tools for VS Code supports Windows, Linux, and macOS. U-SQL local run and local debug works only in Windows.

- Visual Studio Code

For MacOS and Linux:

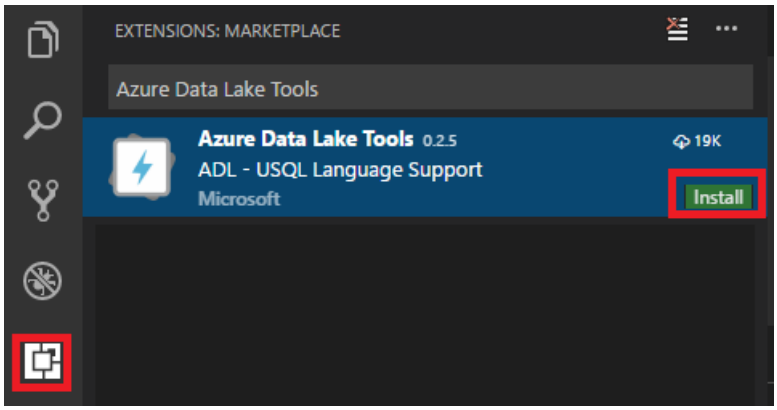
- .NET Core SDK 2.0
- Mono 5.2.x

## Install Azure Data Lake Tools

After you install the prerequisites, you can install Azure Data Lake Tools for VS Code.

**To install Azure Data Lake Tools**

1. Open Visual Studio Code.
2. Select **Extensions** in the left pane. Enter **Azure Data Lake Tools** in the search box.
3. Select **Install** next to **Azure Data Lake Tools**.




After a few seconds, the **Install** button changes to **Reload**.

4. Select **Reload** to activate the **Azure Data Lake Tools** extension.
5. Select **Reload Window** to confirm. You can see **Azure Data Lake Tools** in the **Extensions** pane.

## Activate Azure Data Lake Tools

Create a .usql file or open an existing .usql file to activate the extension.

## Work with U-SQL

To work with U-SQL, you need open either a U-SQL file or a folder.

**To open the sample script**

Open the command palette (Ctrl+Shift+P) and enter **ADL: Open Sample Script**. It opens another instance of this sample. You can also edit, configure, and submit a script on this instance.

**To open a folder for your U-SQL project**

1. From Visual Studio Code, select the **File** menu, and then select **Open Folder**.
2. Specify a folder, and then select **Select Folder**.
3. Select the **File** menu, and then select **New**. An Untitled-1 file is added to the project.
4. Enter the following code in the Untitled-1 file:

```
@departments  =
    SELECT * FROM
        (VALUES
            (31,    "Sales"),
            (33,    "Engineering"),
            (34,    "Clerical"),
            (35,    "Marketing")
        ) AS
              D( DepID, DepName );

OUTPUT @departments
    TO "/Output/departments.csv"
USING Outputters.Csv();
```

The script creates a departments.csv file with some data included in the /output folder.

5. Save the file as **myUSQL.usql** in the opened folder.

**To compile a U-SQL script**

1. Select Ctrl+Shift+P to open the command palette.
2. Enter **ADL: Compile Script**. The compile results appear in the **Output** window. You can also right-click a

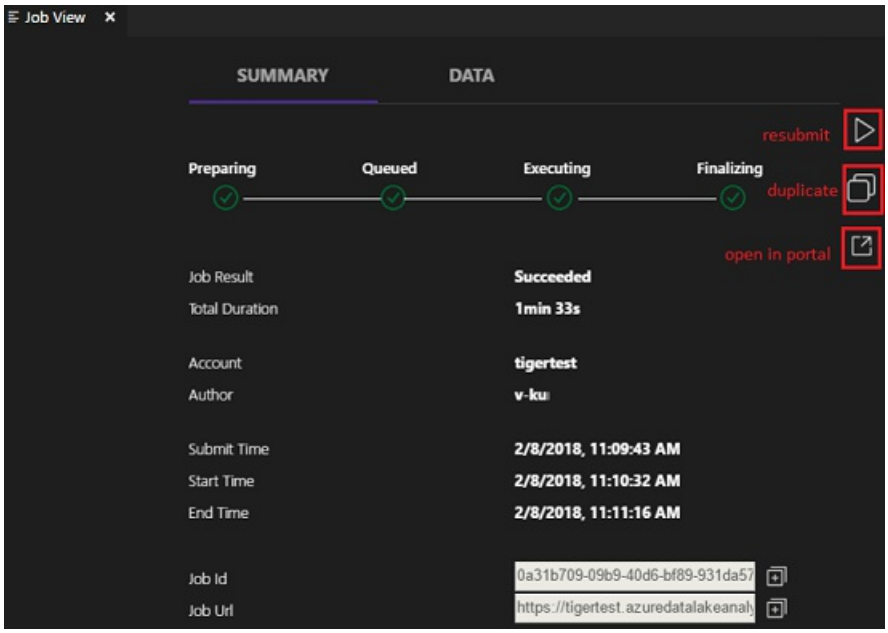
script file, and then select **ADL: Compile Script** to compile a U-SQL job. The compilation result appears in the **Output** pane.

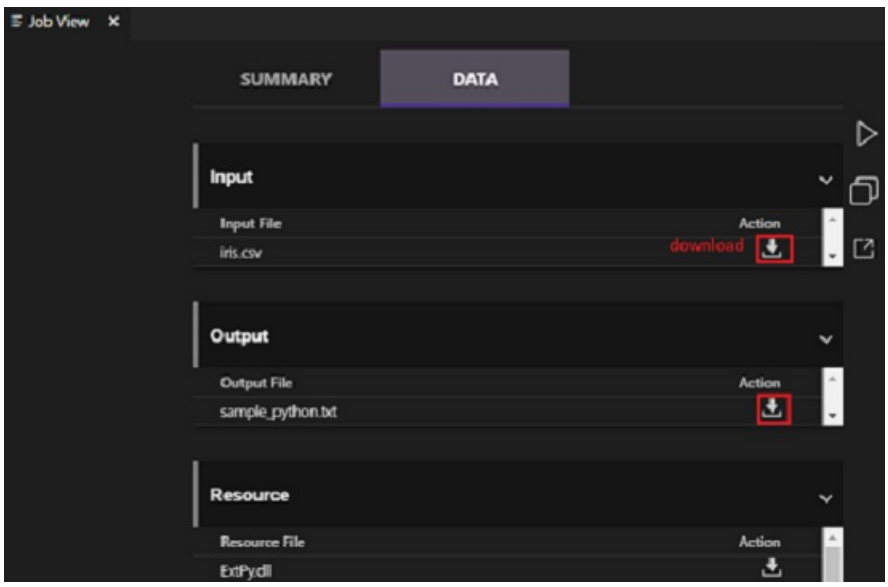
**To submit a U-SQL script**

1. Select Ctrl+Shift+P to open the command palette.
2. Enter **ADL: Submit Job**. You can also right-click a script file, and then select **ADL: Submit Job**.

After you submit a U-SQL job, the submission logs appear in the **Output** window in VS Code. The job view appears in the right pane. If the submission is successful, the job URL appears too. You can open the job URL in a web browser to track the real-time job status.

On the job view's **SUMMARY** tab, you can see the job details. Main functions include resubmit a script, duplicate a script, and open in the portal. On the job view's **DATA** tab, you can refer to the input files, output files, and resource files. Files can be downloaded to the local computer.



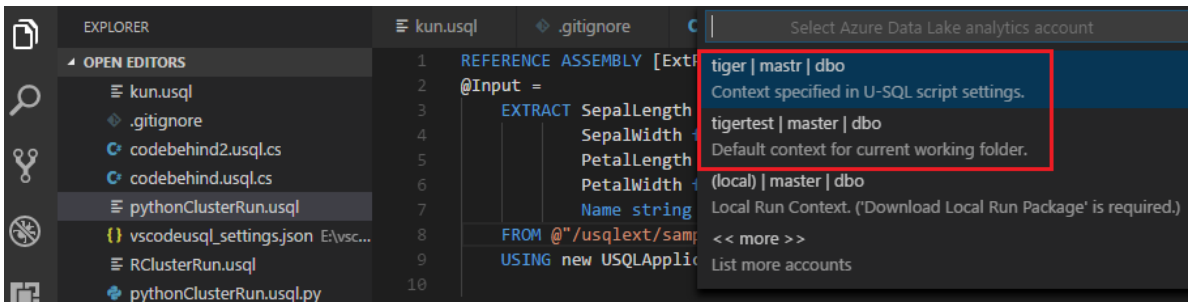




**To set the default context**

You can set the default context to apply this setting to all script files if you have not set parameters for files individually.

1. Select Ctrl+Shift+P to open the command palette.
2. Enter **ADL: Set Default Context**. Or right-click the script editor and select **ADL: Set Default Context**.

3. Choose the account, database, and schema that you want. The setting is saved to the xxx_settings.json configuration file.




**To set script parameters**

1. Select Ctrl+Shift+P to open the command palette.
2. Enter **ADL: Set Script Parameters**.
3. The xxx_settings.json file is opened with the following properties:

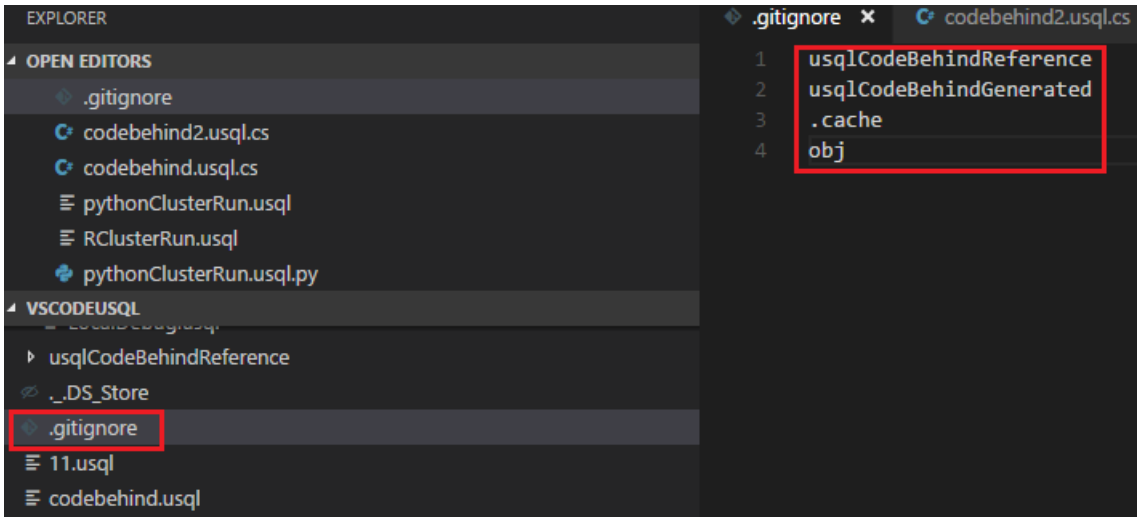
- **account**: An Azure Data Lake Analytics account under your Azure subscription that's needed to compile and run U-SQL jobs. You need configure the computer account before you compile and run U-SQL jobs.
- **database**: A database under your account. The default is **master**.
- **schema**: A schema under your database. The default is **dbo**.
- **optionalSettings**:
  - **priority**: The priority range is from 1 to 1000, with 1 as the highest priority. The default value is **1000**.
  - **degreeOfParallelism**: The parallelism range is from 1 to 150. The default value is the maximum parallelism allowed in your Azure Data Lake Analytics account.

> **NOTE**
> After you save the configuration, the account, database, and schema information appear on the status bar at the lower-left corner of the corresponding .usql file if you don't have a default context set up.

**To set Git ignore**

1. Select Ctrl+Shift+P to open the command palette.
2. Enter **ADL: Set Git Ignore**.

- If you don't have a **.gitIgnore** file in your VS Code working folder, a file named **.gitIgnore** is created in your folder. Four items (**usqlCodeBehindReference**, **usqlCodeBehindGenerated**, **.cache**, **obj**) are added in the file by default. You can make more updates if needed.
- If you already have a **.gitIgnore** file in your VS Code working folder, the tool adds four items (**usqlCodeBehindReference**, **usqlCodeBehindGenerated**, **.cache**, **obj**) in your **.gitIgnore** file if the four items were not included in the file.




## Work with code-behind files: C Sharp, Python, and R

Azure Data Lake Tools supports multiple custom codes. For instructions, see Develop U-SQL with Python, R, and C Sharp for Azure Data Lake Analytics in VS Code.

## Work with assemblies

For information on developing assemblies, see Develop U-SQL assemblies for Azure Data Lake Analytics jobs.

You can use Data Lake Tools to register custom code assemblies in the Data Lake Analytics catalog.

**To register an assembly**

You can register the assembly through the **ADL: Register Assembly** or **ADL: Register Assembly (Advanced)** command.

**To register through the ADL: Register Assembly command**

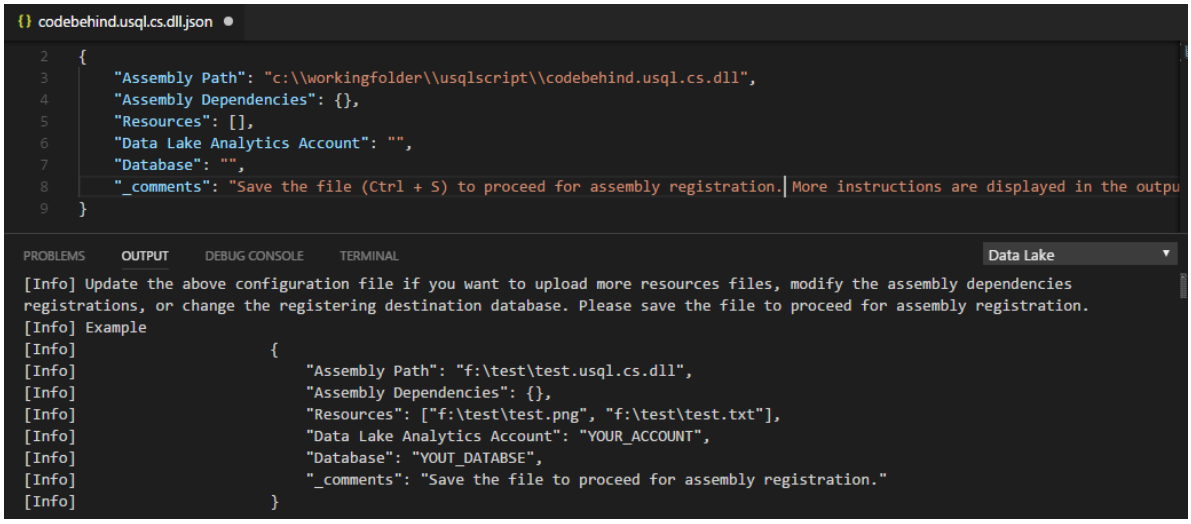
1. Select Ctrl+Shift+P to open the command palette.
2. Enter **ADL: Register Assembly**.
3. Specify the local assembly path.
4. Select a Data Lake Analytics account.
5. Select a database.

The portal is opened in a browser and displays the assembly registration process.

A more convenient way to trigger the **ADL: Register Assembly** command is to right-click the .dll file in File Explorer.

**To register through the ADL: Register Assembly (Advanced) command**

1. Select Ctrl+Shift+P to open the command palette.
2. Enter **ADL: Register Assembly (Advanced)**.
3. Specify the local assembly path.
4. The JSON file is displayed. Review and edit the assembly dependencies and resource parameters, if needed. Instructions are displayed in the **Output** window. To proceed to the assembly registration, save (Ctrl+S) the JSON file.




> **NOTE**
>
> - Azure Data Lake Tools autodetects whether the DLL has any assembly dependencies. The dependencies are displayed in the JSON file after they're detected.
> - You can upload your DLL resources (for example, .txt, .png, and .csv) as part of the assembly registration.

Another way to trigger the **ADL: Register Assembly (Advanced)** command is to right-click the .dll file in File Explorer.

The following U-SQL code demonstrates how to call an assembly. In the sample, the assembly name is *test*.

```
REFERENCE ASSEMBLY [test];

@a =
    EXTRACT
        Iid int,
    Starts DateTime,
    Region string,
    Query string,
    DwellTime int,
    Results string,
    ClickedUrls string
    FROM @"Sample/SearchLog.txt"
    USING Extractors.Tsv();

@d =
    SELECT DISTINCT Region
    FROM @a;

@d1 =
    PROCESS @d
    PRODUCE
        Region string,
    Mkt string
    USING new USQLApplication_codebehind.MyProcessor();

OUTPUT @d1
    TO @"Sample/SearchLogtest.txt"
    USING Outputters.Tsv();
```

## Use U-SQL local run and local debug for Windows users

U-SQL local run tests your local data and validates your script locally before your code is published to Data Lake Analytics. You can use the local debug feature to complete the following tasks before your code is submitted to Data Lake Analytics:

- Debug your C# code-behind.
- Step through the code.
- Validate your script locally.

The local run and local debug feature only works in Windows environments, and is not supported on macOS and Linux-based operating systems.

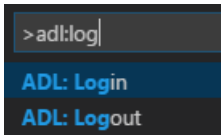
For instructions on local run and local debug, see U-SQL local run and local debug with Visual Studio Code.

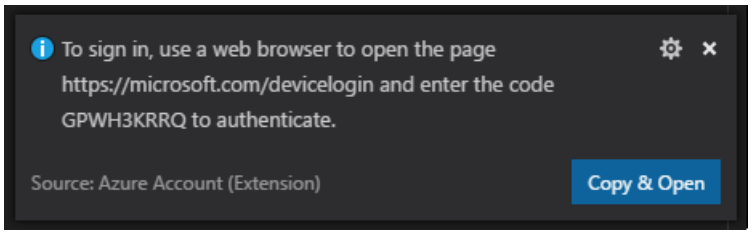
## Connect to Azure

Before you can compile and run U-SQL scripts in Data Lake Analytics, you must connect to your Azure account.

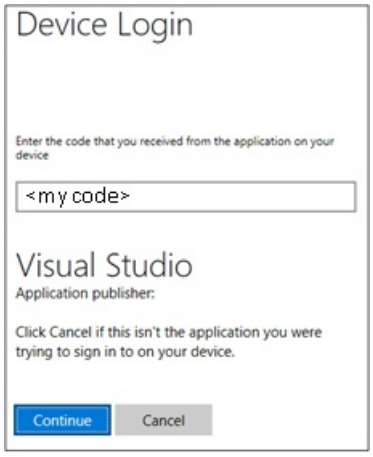
**To connect to Azure by using a command**

1. Select Ctrl+Shift+P to open the command palette.
2. Enter **ADL: Login**. The login information appears on the lower right.






3. Select **Copy & Open** to open the login webpage. Paste the code into the box, and then select **Continue**.



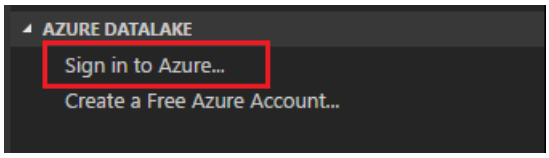

4. Follow the instructions to sign in from the webpage. When you're connected, your Azure account name appears on the status bar in the lower-left corner of the VS Code window.

> **NOTE**
>
> - Data Lake Tools automatically signs you in the next time if you don't sign out.
> - If your account has two factors enabled, we recommend that you use phone authentication rather than using a PIN.

To sign out, enter the command **ADL: Logout**.

**To connect to Azure from the explorer**

Expand **AZURE DATALAKE**, select **Sign in to Azure**, and then follow step 3 and step 4 of To connect to Azure by using a command.



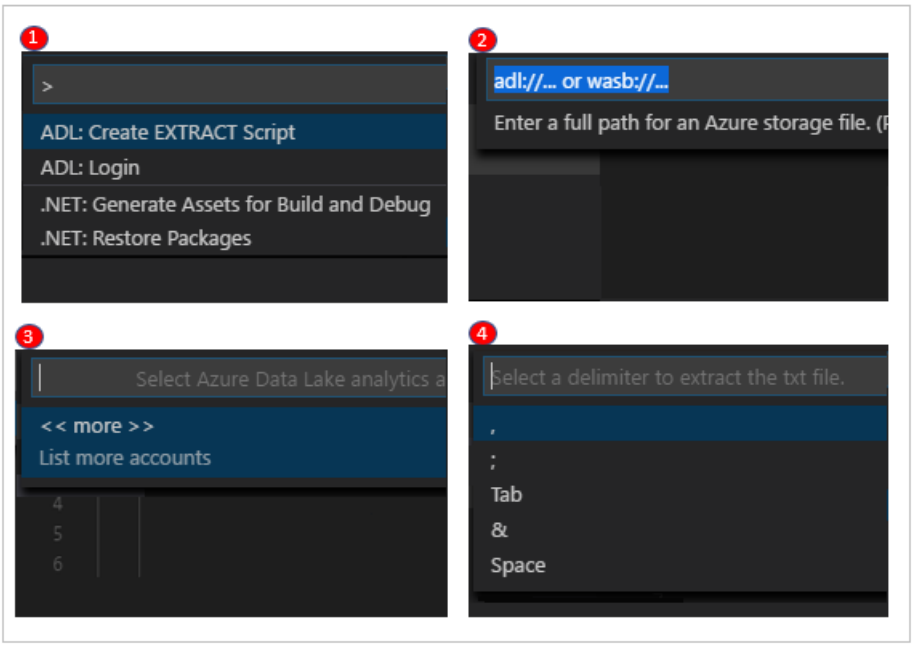

You can't sign out from the explorer. To sign out, seeTo connect to Azure by using a command.

## Create an extraction script

You can create an extraction script for .csv, .tsv, and .txt files by using the command **ADL: Create EXTRACT Script** or from the Azure Data Lake explorer.

**To create an extraction script by using a command**

1. Select Ctrl+Shift+P to open the command palette, and enter **ADL: Create EXTRACT Script**.
2. Specify the full path for an Azure Storage file, and select the Enter key.
3. Select one account.
4. For a .txt file, select a delimiter to extract the file.




The extraction script is generated based on your entries. For a script that cannot detect the columns, choose one from the two options. If not, only one script will be generated.

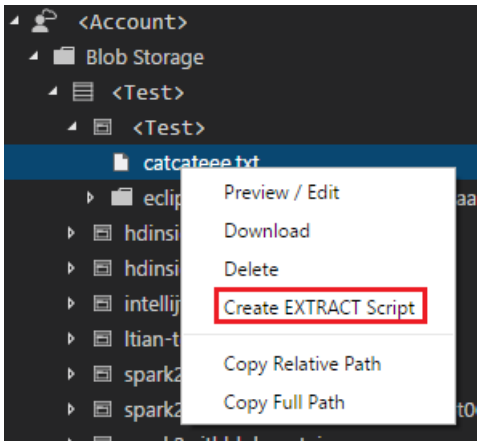
```
/************ File Information **********
        File Name: text.txt
        Modified: Wed May 16 2018 00:04:09 GMT-0700 (Pacific Daylight Time)
        File Size: 51 bytes
        Path: adl://<myaccount>.azuredatalakestore.net/my/text.txt
***************************************/

// Please choose your extract script from the two options below.
// EXTRACT WITH HEADER ROW
@input =
    EXTRACT [1 1 1 1 1 1 1 1] string
    FROM "/my/text.txt"
    USING Extractors.Text(skipFirstNRows:1, delimiter:',', quoting:true);
// For more information on U-SQL extractor parameters, please see https://msdn.mi


//EXTRACT WITHOUT HEADER ROW
@input =
    EXTRACT [Column_0] string
    FROM "/my/text.txt"
    USING Extractors.Text(delimiter:',', quoting:true);
// For more information on U-SQL extractor parameters, please see https://msdn.mi
```

**To create an extraction script from the explorer**

Another way to create the extraction script is through the right-click (shortcut) menu on the .csv, .tsv, or .txt file in Azure Data Lake Store or Azure Blob storage.




## Integrate with Azure Data Lake Analytics through a command

You can access Azure Data Lake Analytics resources to list accounts, access metadata, and view analytics jobs.

**To list the Azure Data Lake Analytics accounts under your Azure subscription**

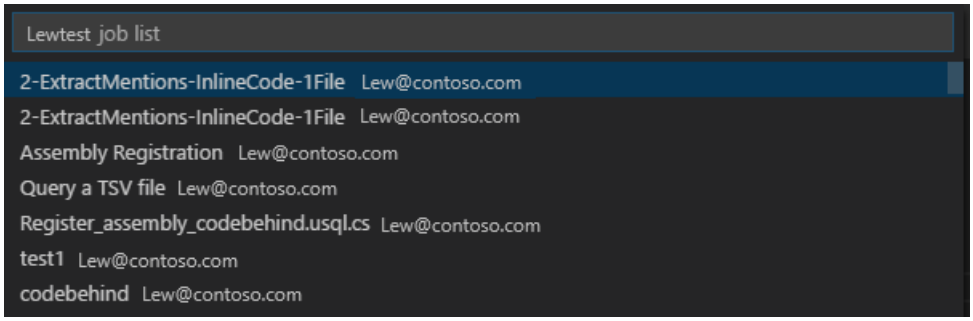
1. Select Ctrl+Shift+P to open the command palette.
2. Enter **ADL: List Accounts**. The accounts appear in the **Output** pane.

**To access Azure Data Lake Analytics metadata**

1. Select Ctrl+Shift+P, and then enter **ADL: List Tables**.
2. Select one of the Data Lake Analytics accounts.
3. Select one of the Data Lake Analytics databases.
4. Select one of the schemas. You can see the list of tables.

**To view Azure Data Lake Analytics jobs**

1. Open the command palette (Ctrl+Shift+P) and select **ADL: Show Jobs**.
2. Select a Data Lake Analytics or local account.
3. Wait for the job list to appear for the account.
4. Select a job from the job list. Data Lake Tools opens the job view in the right pane and displays some information in the VS Code output.




## Integrate with Azure Data Lake Store through a command

You can use Azure Data Lake Store-related commands to:

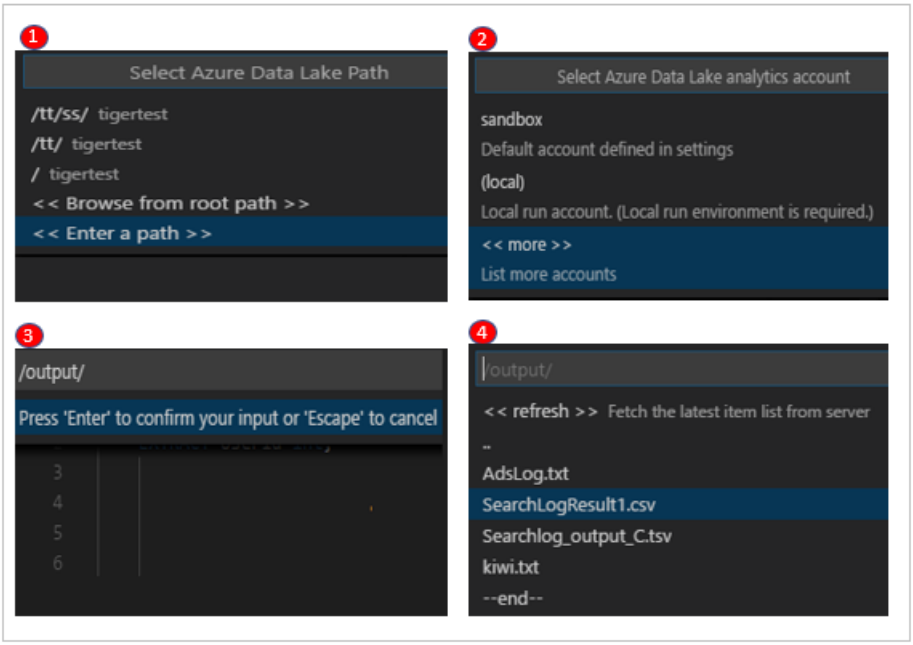
- Browse through the Azure Data Lake Store resources
- Preview the Azure Data Lake Store file
- Upload the file directly to Azure Data Lake Store in VS Code
- Download the file directly from Azure Data Lake Store in VS Code

**List the storage path**

**To list the storage path through the command palette**

1. Right-click the script editor and select **ADL: List Path**.
2. Choose the folder in the list, or select **Enter a path** or **Browse from root path**. (We're using **Enter a path** as an example.)
3. Select your Data Lake Analytics account.
4. Browse to or enter the storage folder path (for example, /output/).

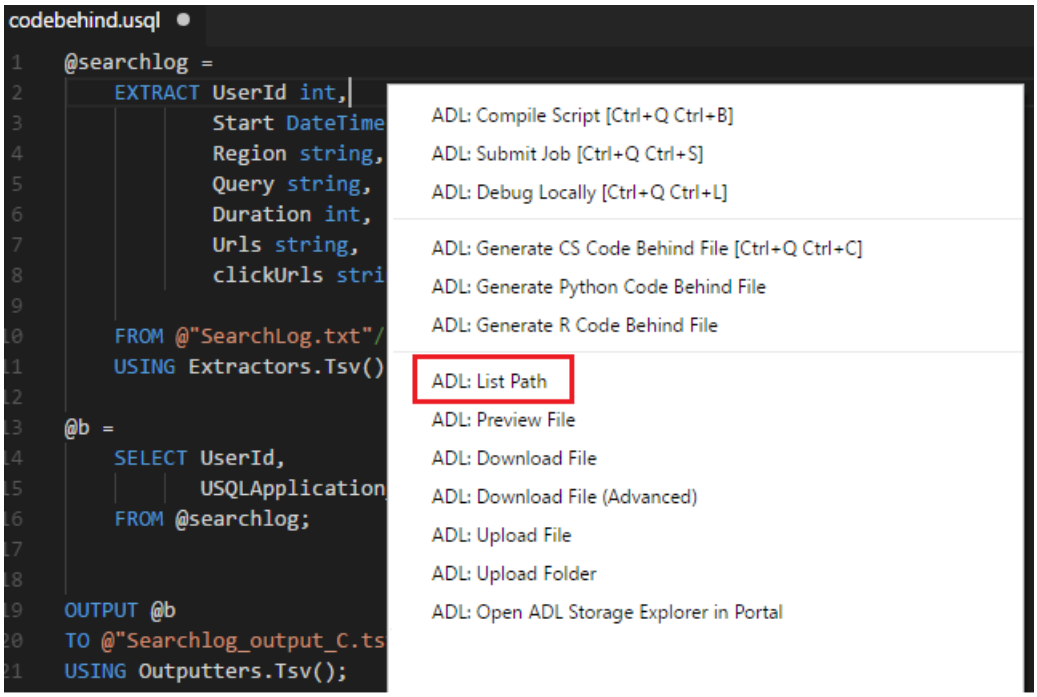
The command palette lists the path information based on your entries.




A more convenient way to list the relative path is through the shortcut menu.

**To list the storage path through the shortcut menu**

Right-click the path string and select **List Path**.

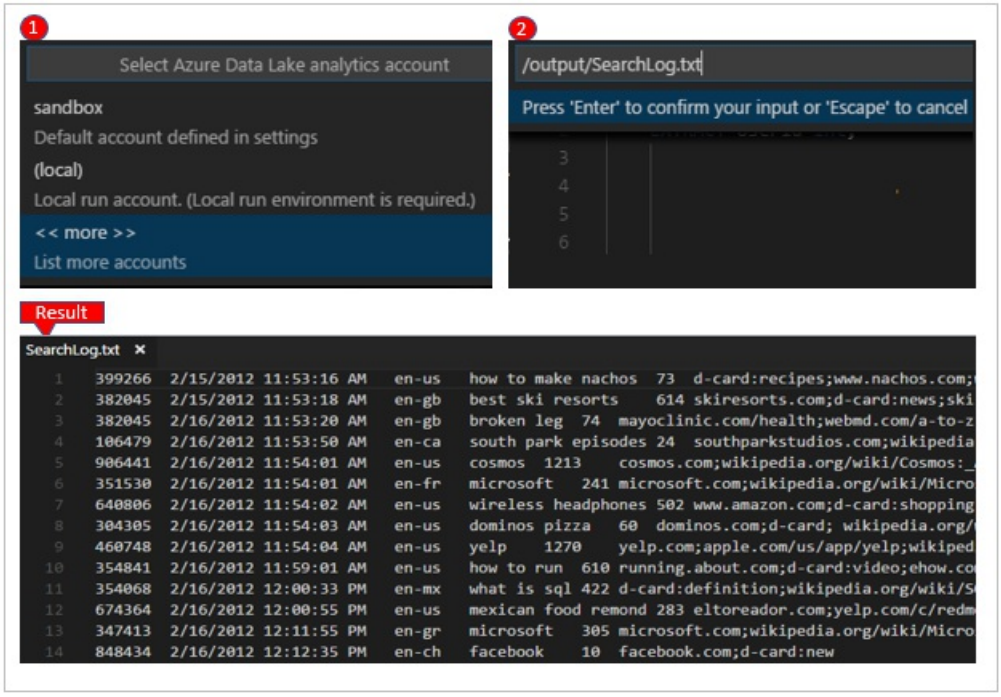



**Preview the storage file**

1. Right-click the script editor and select **ADL: Preview File**.
2. Select your Data Lake Analytics account.

3. Enter an Azure Storage file path (for example, /output/SearchLog.txt).

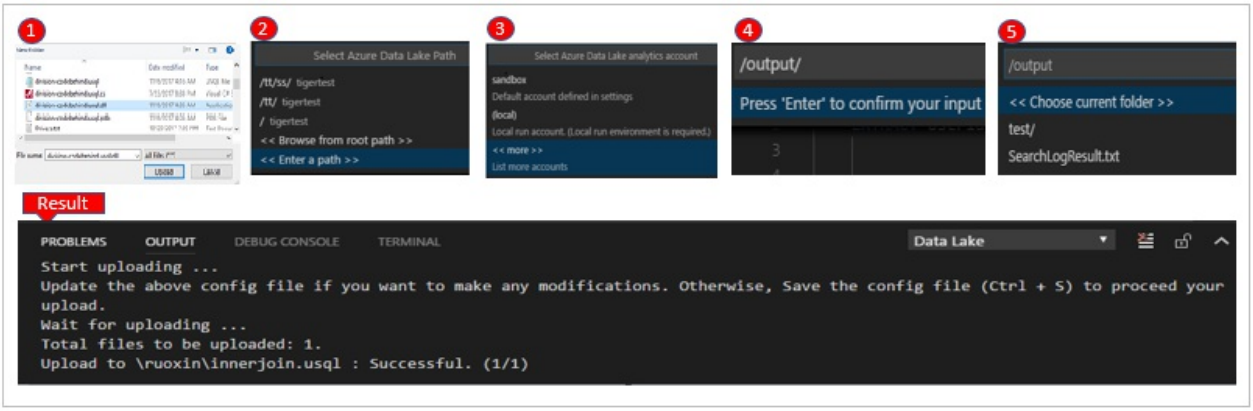
The file opens in VS Code.




Another way to preview the file is through the shortcut menu on the file's full path or the file's relative path in the script editor.

**Upload a file or folder**

1. Right-click the script editor and select **Upload File** or **Upload Folder**.
2. Choose one file or multiple files if you selected **Upload File**, or choose the whole folder if you selected **Upload Folder**. Then select **Upload**.
3. Choose the storage folder in the list, or select **Enter a path** or **Browse from root path**. (We're using **Enter a path** as an example.)
4. Select your Data Lake Analytics account.
5. Browse to or enter the storage folder path (for example, /output/).
6. Select **Choose Current Folder** to specify your upload destination.




Another way to upload files to storage is through the shortcut menu on the file's full path or the file's relative path in the script editor.

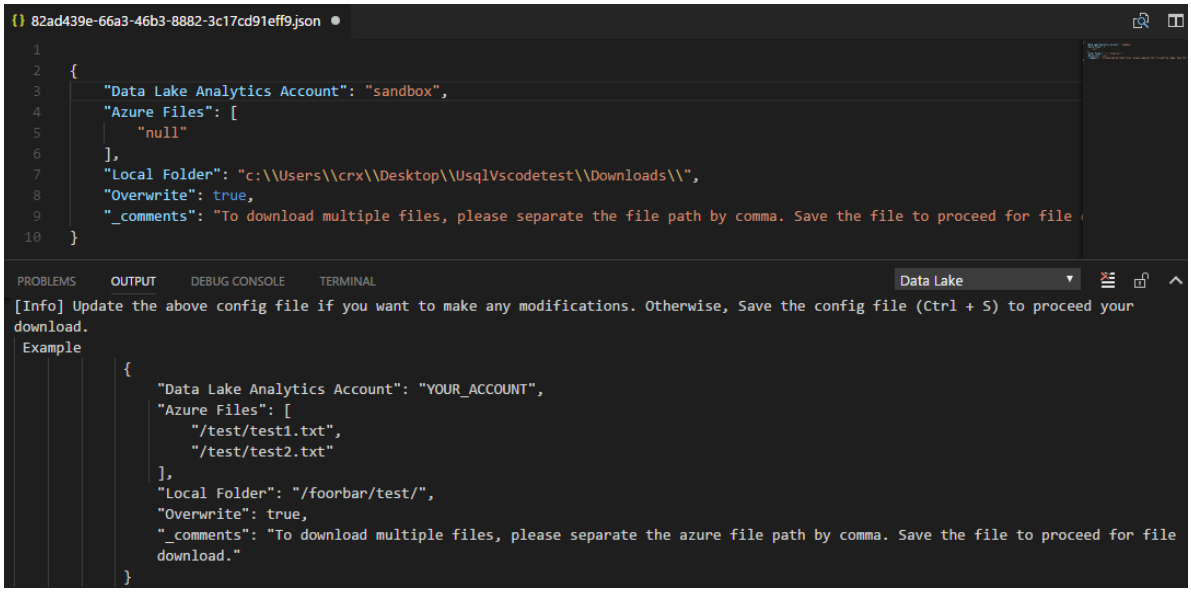
You can monitor the upload status.

**Download a file**

You can download a file by using the command **ADL: Download File** or **ADL: Download File (Advanced)**.

**To download a file through the ADL: Download File (Advanced) command**

1. Right-click the script editor, and then select **Download File (Advanced)**.
2. VS Code displays a JSON file. You can enter file paths and download multiple files at the same time. Instructions are displayed in the **Output** window. To proceed to download the file or files, save (Ctrl+S) the JSON file.



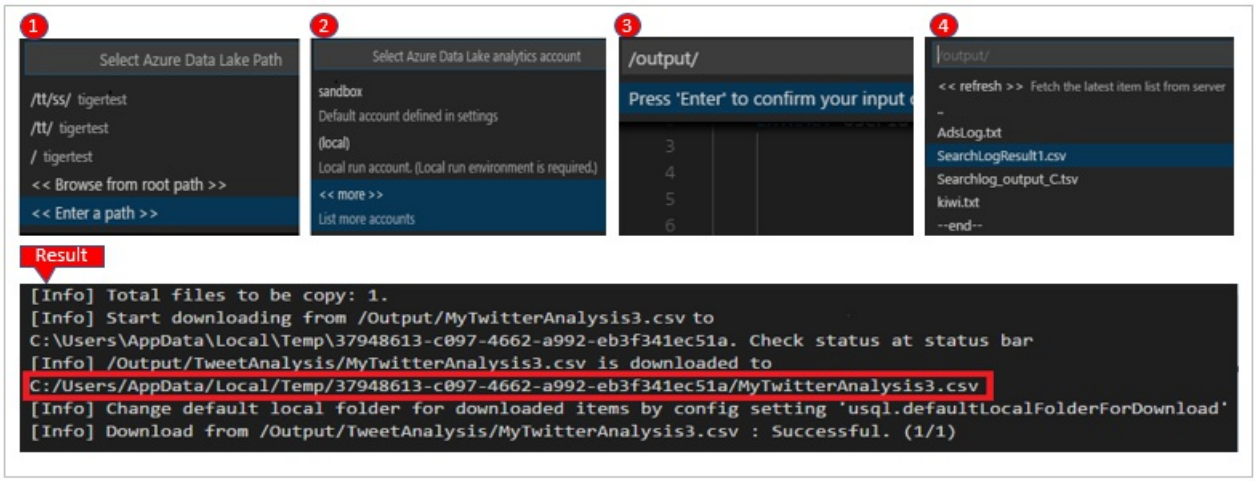

The **Output** window displays the file download status.

```
[Info] Wait for downloading ...
[Info] Total files to be copy: 2.
[Info] /ru/SearchLogResult1.txt is downloaded to
C:/Users/AppData/Local/Temp/8f9d91b6-4750-420a-ba3d-605a7606a33c/SearchLogResult1.txt
[Info] Change default local folder for downloaded items by config setting 'usql.defaultLocalFolderForDownload'
[Info] Download from /ru/SearchLogResult1.txt : Successful. (1/2)
[Info] /ru/SearchLog.txt is downloaded to C:/Users/crx/AppData/Local/Temp/8f9d91b6-4750-420a-ba3d-605a7606a33c/SearchLog.txt
[Info] Change default local folder for downloaded items by config setting 'usql.defaultLocalFolderForDownload'
[Info] Download from /ru/SearchLog.txt : Successful. (2/2)
```

You can monitor the download status.

**To download a file through the ADL: Download File command**

1. Right-click the script editor, select **Download File**, and then select the destination folder from the **Select Folder** dialog box.
2. Choose the folder in the list, or select **Enter a path** or **Browse from root path**. (We're using **Enter a path** as an example.)
3. Select your Data Lake Analytics account.
4. Browse to or enter the storage folder path (for example, /output/), and then choose a file to download.

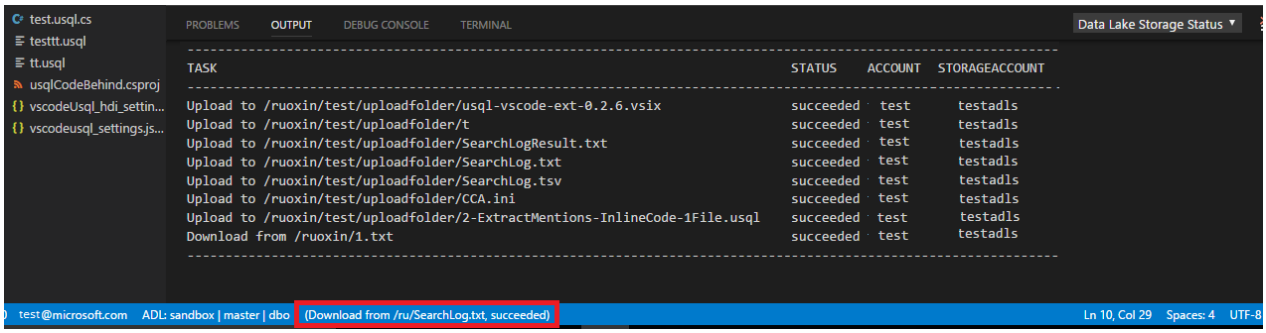


Another way to download storage files is through the shortcut menu on the file's full path or the file's relative path in the script editor.

You can monitor the download status.

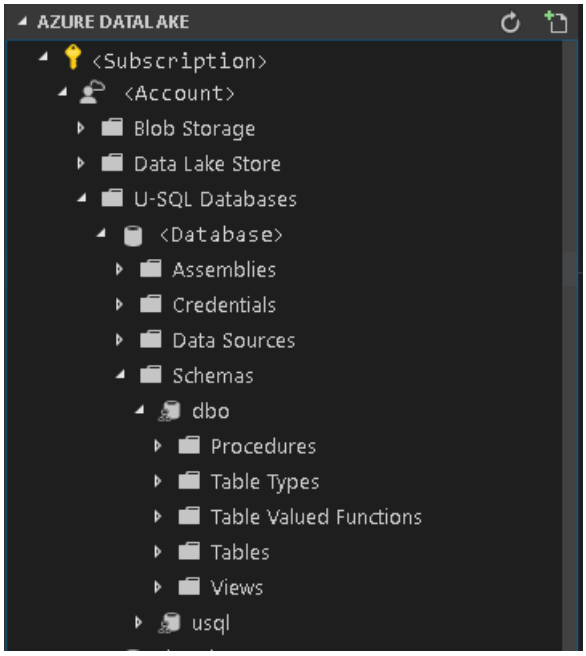
### Check storage tasks' status

The upload and download status appears on the status bar. Select the status bar, and then the status appears on the **OUTPUT** tab.




# Integrate with Azure Data Lake Analytics from the explorer

After you log in, all the subscriptions for your Azure account are listed in the left pane, under **AZURE DATALAKE**.



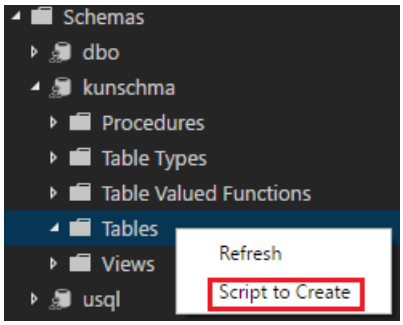

### Data Lake Analytics metadata navigation

Expand your Azure subscription. Under the **U-SQL Databases** node, you can browse through your U-SQL database and view folders like **Schemas**, **Credentials**, **Assemblies**, **Tables**, and **Index**.

### Data Lake Analytics metadata entity management

Expand **U-SQL Databases**. You can create a database, schema, table, table type, index, or statistic by right-clicking the corresponding node, and then selecting **Script to Create** on the shortcut menu. On the opened script page, edit the script according to your needs. Then submit the job by right-clicking it and selecting **ADL: Submit Job**.

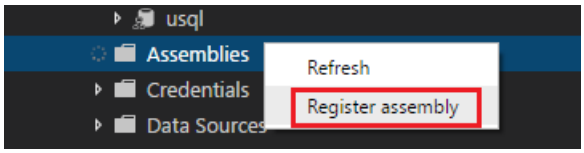
After you finish creating the item, right-click the node and then select **Refresh** to show the item. You can also delete the item by right-clicking it and then selecting **Delete**.





**Data Lake Analytics assembly registration**

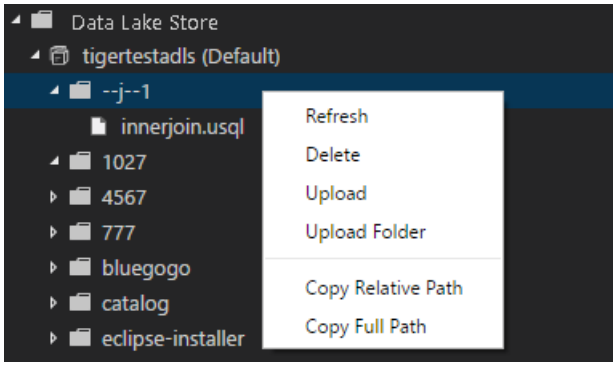
You can register an assembly in the corresponding database by right-clicking the **Assemblies** node, and then selecting **Register assembly**.



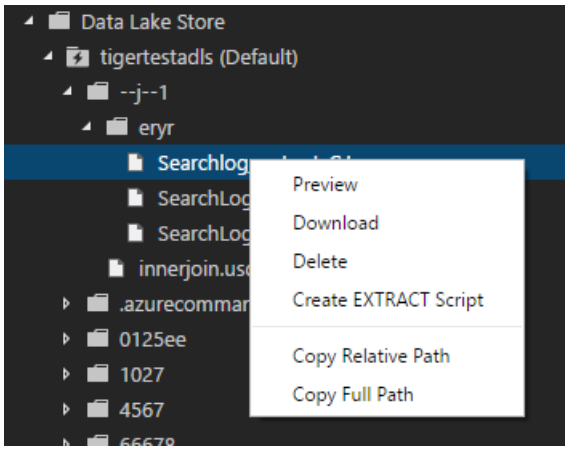

# Integrate with Azure Data Lake Store from the explorer

Browse to **Data Lake Store**:

- You can right-click the folder node and then use the **Refresh**, **Delete**, **Upload**, **Upload Folder**, **Copy Relative Path**, and **Copy Full Path** commands on the shortcut menu.



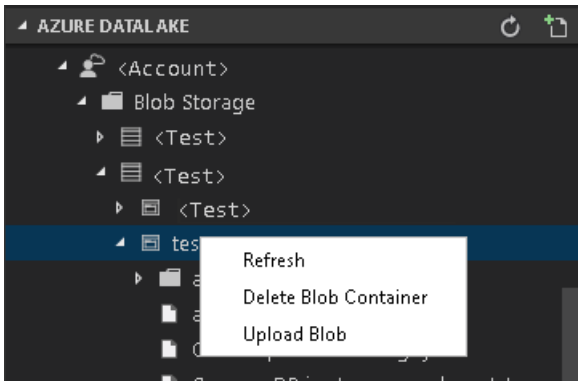

- You can right-click the file node and then use the **Preview**, **Download**, **Delete**, **Create EXTRACT Script** (available only for CSV, TSV, and TXT files), **Copy Relative Path**, and **Copy Full Path** commands on the shortcut menu.



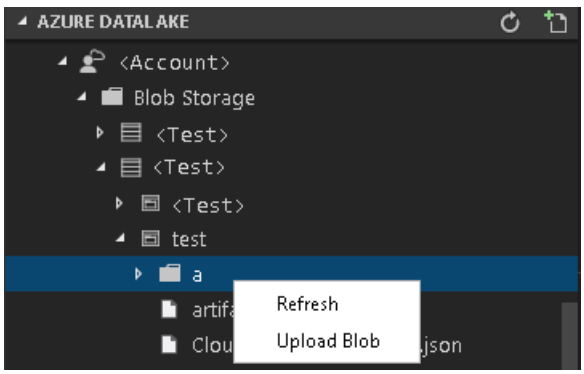

## Integrate with Azure Blob storage from the explorer

Browse to Blob storage:

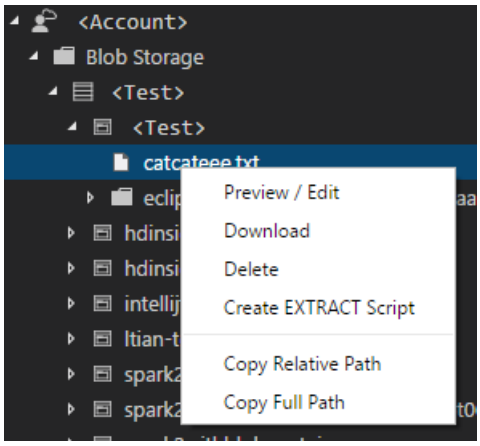
- You can right-click the blob container node and then use the **Refresh**, **Delete Blob Container**, and **Upload Blob** commands on the shortcut menu.




- You can right-click the folder node and then use the **Refresh** and **Upload Blob** commands on the shortcut menu.




- You can right-click the file node and then use the **Preview/Edit**, **Download**, **Delete**, **Create EXTRACT Script** (available only for CSV, TSV, and TXT files), **Copy Relative Path**, and **Copy Full Path** commands on the shortcut menu.




## Open the Data Lake explorer in the portal

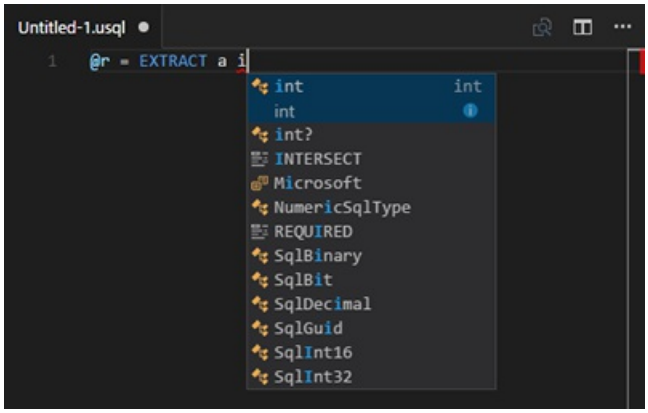
1. Select Ctrl+Shift+P to open the command palette.
2. Enter **Open Web Azure Storage Explorer** or right-click a relative path or the full path in the script editor, and then select **Open Web Azure Storage Explorer**.
3. Select a Data Lake Analytics account.

Data Lake Tools opens the Azure Storage path in the Azure portal. You can find the path and preview the file from the web.

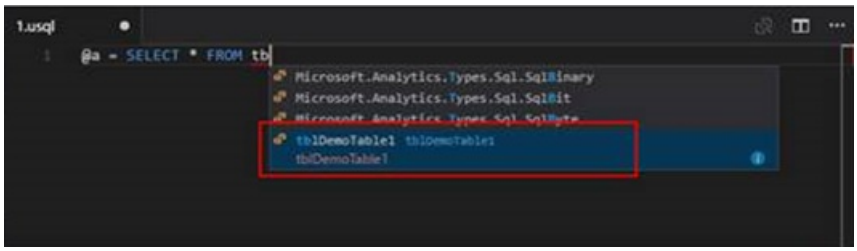
## Additional features

Data Lake Tools for VS Code supports the following features:

- **IntelliSense autocomplete**: Suggestions appear in pop-up windows around items like keywords, methods, and variables. Different icons represent different types of objects:
  - Scala data type
  - Complex data type
  - Built-in UDTs
  - .NET collection and classes
  - C# expressions
  - Built-in C# UDFs, UDOs, and UDAAGs
  - U-SQL functions
  - U-SQL windowing functions




- **IntelliSense autocomplete on Data Lake Analytics metadata**: Data Lake Tools downloads the Data Lake Analytics metadata information locally. The IntelliSense feature automatically populates objects from the Data Lake Analytics metadata. These objects include the database, schema, table, view, table-valued function, procedures, and C# assemblies.




- **IntelliSense error marker**: Data Lake Tools underlines editing errors for U-SQL and C#.
- **Syntax highlights**: Data Lake Tools uses colors to differentiate items like variables, keywords, data types, and functions.


> **NOTE**
> We recommend that you upgrade to Azure Data Lake Tools for Visual Studio version 2.3.3000.4 or later. The previous versions are no longer available for download and are now deprecated.

## Next steps

- Develop U-SQL with Python, R, and C Sharp for Azure Data Lake Analytics in VS Code
- U-SQL local run and local debug with Visual Studio Code
- Tutorial: Get started with Azure Data Lake Analytics
- Tutorial: Develop U-SQL scripts by using Data Lake Tools for Visual Studio

# Develop U-SQL with Python, R, and C# for Azure Data Lake Analytics in Visual Studio Code

8/27/2018 • 3 minutes to read • Edit Online

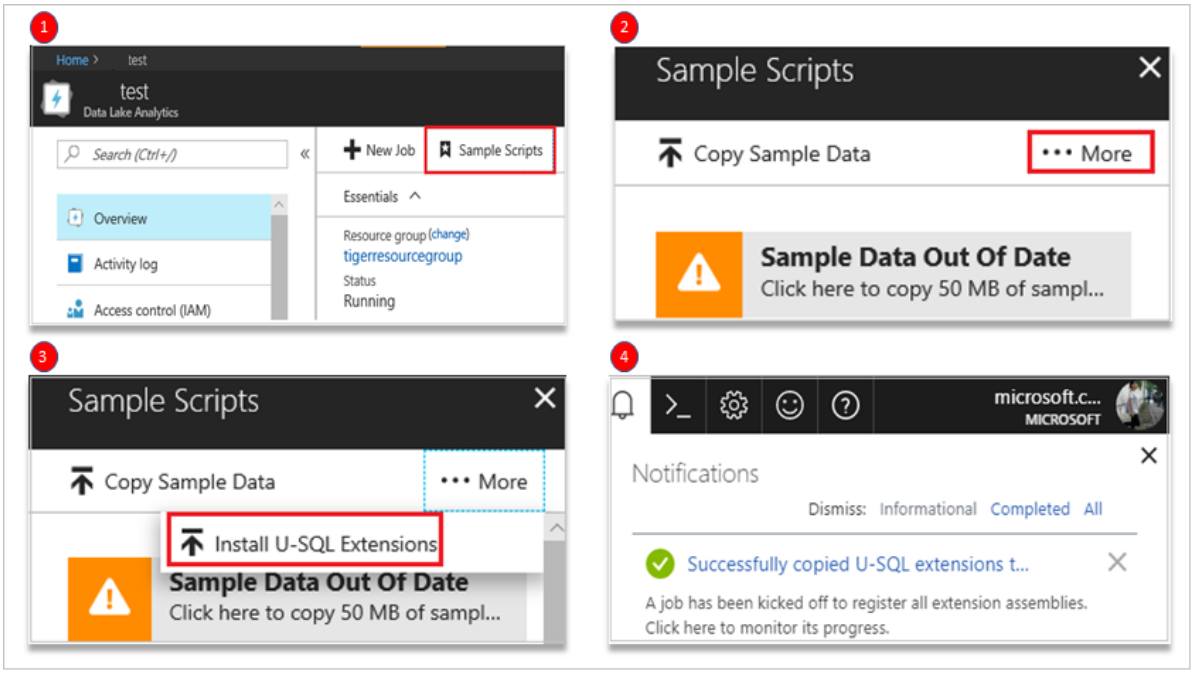
Learn how to use Visual Studio Code (VSCode) to write Python, R and C# code behind with U-SQL and submit jobs to Azure Data Lake service. For more information about Azure Data Lake Tools for VSCode, see Use the Azure Data Lake Tools for Visual Studio Code.

Before writing code-behind custom code, you need to open a folder or a workspace in VSCode.

## Prerequisites for Python and R

Register Python and, R extensions assemblies for your ADL account.

1. Open your account in portal.
   - Select **Overview**.
   - Click **Sample Script**.
2. Click **More**.
3. Select **Install U-SQL Extensions**.
4. Confirmation message is displayed after the U-SQL extensions are installed.




> **NOTE**
> For best experiences on Python and R language service, please install VSCode Python and R extension.

## Develop Python file

1. Click the **New File** in your workspace.
2. Write your code in U-SQL. The following is a code sample.

```
REFERENCE ASSEMBLY [ExtPython];
@t  =
    SELECT * FROM
    (VALUES
        ("D1","T1","A1","@foo Hello World @bar"),
        ("D2","T2","A2","@baz Hello World @beer")
    ) AS
        D( date, time, author, tweet );

@m  =
    REDUCE @t ON date
    PRODUCE date string, mentions string
    USING new Extension.Python.Reducer("pythonSample.usql.py", pyVersion : "3.5.1");

OUTPUT @m
    TO "/tweetmentions.csv"
    USING Outputters.Csv();
```

3. Right-click a script file, and then select **ADL: Generate Python Code Behind File**.
4. The **xxx.usql.py** file is generated in your working folder. Write your code in Python file. The following is a code sample.

```
def get_mentions(tweet):
    return ';'.join( ( w[1:] for w in tweet.split() if w[0]=='@' ) )

def usqlml_main(df):
    del df['time']
    del df['author']
    df['mentions'] = df.tweet.apply(get_mentions)
    del df['tweet']
    return df
```

5. Right-click in **USQL** file, you can click **Compile Script** or **Submit Job** to running job.

## Develop R file

1. Click the **New File** in your workspace.
2. Write your code in U-SQL file. The following is a code sample.

```
DEPLOY RESOURCE @"/usqlext/samples/R/my_model_LM_Iris.rda";
DECLARE @IrisData string = @"/usqlext/samples/R/iris.csv";
DECLARE @OutputFilePredictions string = @"/my/R/Output/LMPredictionsIris.txt";
DECLARE @PartitionCount int = 10;

@InputData =
    EXTRACT SepalLength double,
            SepalWidth double,
            PetalLength double,
            PetalWidth double,
            Species string
    FROM @IrisData
    USING Extractors.Csv();

@ExtendedData =
    SELECT Extension.R.RandomNumberGenerator.GetRandomNumber(@PartitionCount) AS Par,
        SepalLength,
        SepalWidth,
        PetalLength,
        PetalWidth
    FROM @InputData;

// Predict Species

@RScriptOutput =
    REDUCE @ExtendedData
    ON Par
    PRODUCE Par,
            fit double,
            lwr double,
            upr double
    READONLY Par
    USING new Extension.R.Reducer(scriptFile : "RClusterRun.usql.R", rReturnType : "dataframe",
stringsAsFactors : false);
OUTPUT @RScriptOutput
TO @OutputFilePredictions
USING Outputters.Tsv();
```

3. Right-click in **USQL** file, and then select **ADL: Generate R Code Behind File**.
4. The **xxx.usql.r** file is generated in your working folder. Write your code in R file. The following is a code sample.

```
load("my_model_LM_Iris.rda")
outputToUSQL=data.frame(predict(lm.fit, inputFromUSQL, interval="confidence"))
```

5. Right-click in **USQL** file, you can click **Compile Script** or **Submit Job** to running job.

## Develop C# file

A code-behind file is a C# file associated with a single U-SQL script. You can define a script dedicated to UDO, UDA, UDT, and UDF in the code-behind file. The UDO, UDA, UDT, and UDF can be used directly in the script without registering the assembly first. The code-behind file is put in the same folder as its peering U-SQL script file. If the script is named xxx.usql, the code-behind is named as xxx.usql.cs. If you manually delete the code-behind file, the code-behind feature is disabled for its associated U-SQL script. For more information about writing customer code for U-SQL script, see Writing and Using Custom Code in U-SQL: User-Defined Functions.

1. Click the **New File** in your workspace.
2. Write your code in U-SQL file. The following is a code sample.

```
@a =
    EXTRACT
        Iid int,
    Starts DateTime,
    Region string,
    Query string,
    DwellTime int,
    Results string,
    ClickedUrls string
    FROM @"/Samples/Data/SearchLog.tsv"
    USING Extractors.Tsv();

@d =
    SELECT DISTINCT Region
    FROM @a;

@d1 =
    PROCESS @d
    PRODUCE
        Region string,
    Mkt string
    USING new USQLApplication_codebehind.MyProcessor();

OUTPUT @d1
    TO @"/output/SearchLogtest.txt"
    USING Outputters.Tsv();
```

3. Right-click in **USQL** file, and then select **ADL: Generate CS Code Behind File**.
4. The **xxx.usql.cs** file is generated in your working folder. Write your code in CS file. The following is a code sample.

```
namespace USQLApplication_codebehind
{
    [SqlUserDefinedProcessor]

    public class MyProcessor : IProcessor
    {
        public override IRow Process(IRow input, IUpdatableRow output)
        {
            output.Set(0, input.Get<string>(0));
            output.Set(1, input.Get<string>(0));
            return output.AsReadOnly();
        }
    }
}
```

5. Right-click in **USQL** file, you can click **Compile Script** or **Submit Job** to running job.

## Next steps

- Use the Azure Data Lake Tools for Visual Studio Code
- U-SQL local run and local debug with Visual Studio Code
- Get started with Data Lake Analytics using PowerShell
- Get started with Data Lake Analytics using the Azure portal
- Use Data Lake Tools for Visual Studio for developing U-SQL applications
- Use Data Lake Analytics(U-SQL) catalog

# Run U-SQL and debug locally in Visual Studio Code

9/14/2018 • 2 minutes to read • Edit Online

This article describes how to run U-SQL jobs on a local development machine to speed up early coding phases or to debug code locally in Visual Studio Code. For instructions on Azure Data Lake Tool for Visual Studio Code, see Use Azure Data Lake Tools for Visual Studio Code.

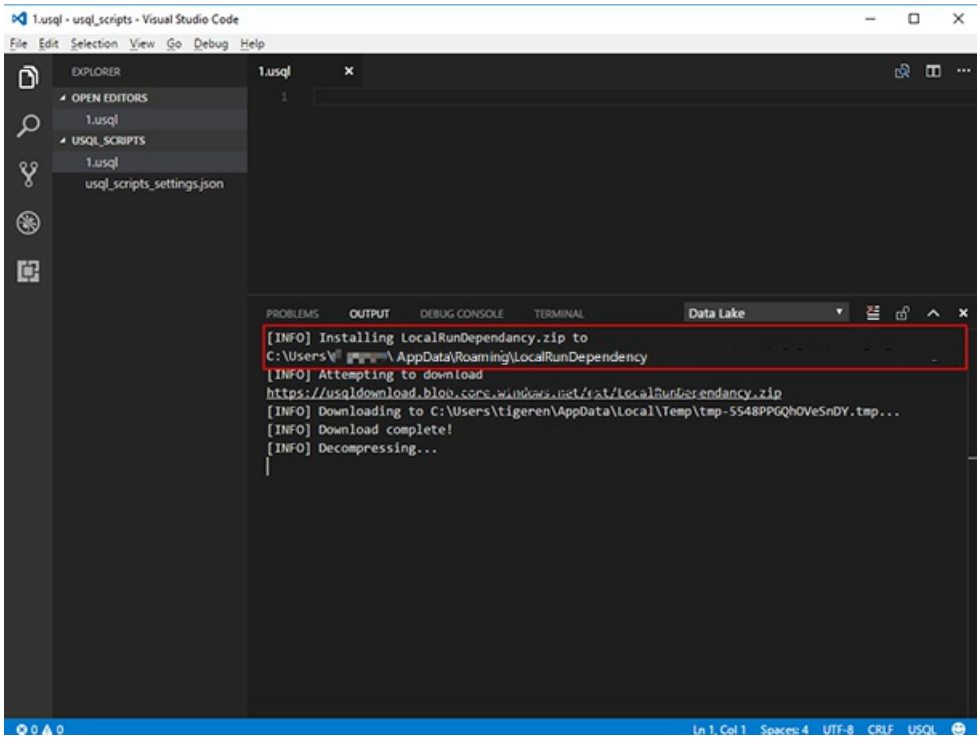
Only Windows installations of the Azure Data Lake Tools for Visual Studio support the action to run U-SQL locally and debug U-SQL locally. Installations on macOS and Linux-based operating systems do not support this feature.

## Set up the U-SQL local run environment

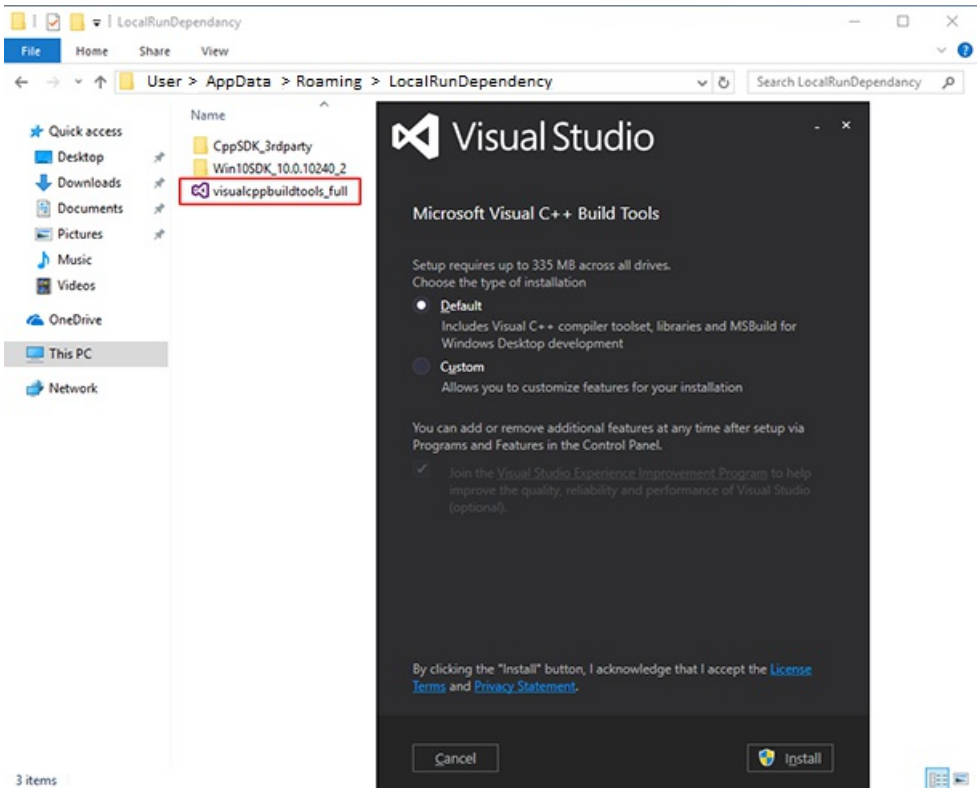
1. Select Ctrl+Shift+P to open the command palette, and then enter **ADL: Download Local Run Package** to download the packages.


2. Locate the dependency packages from the path shown in the **Output** pane, and then install BuildTools and Win10SDK 10240. Here is an example path:

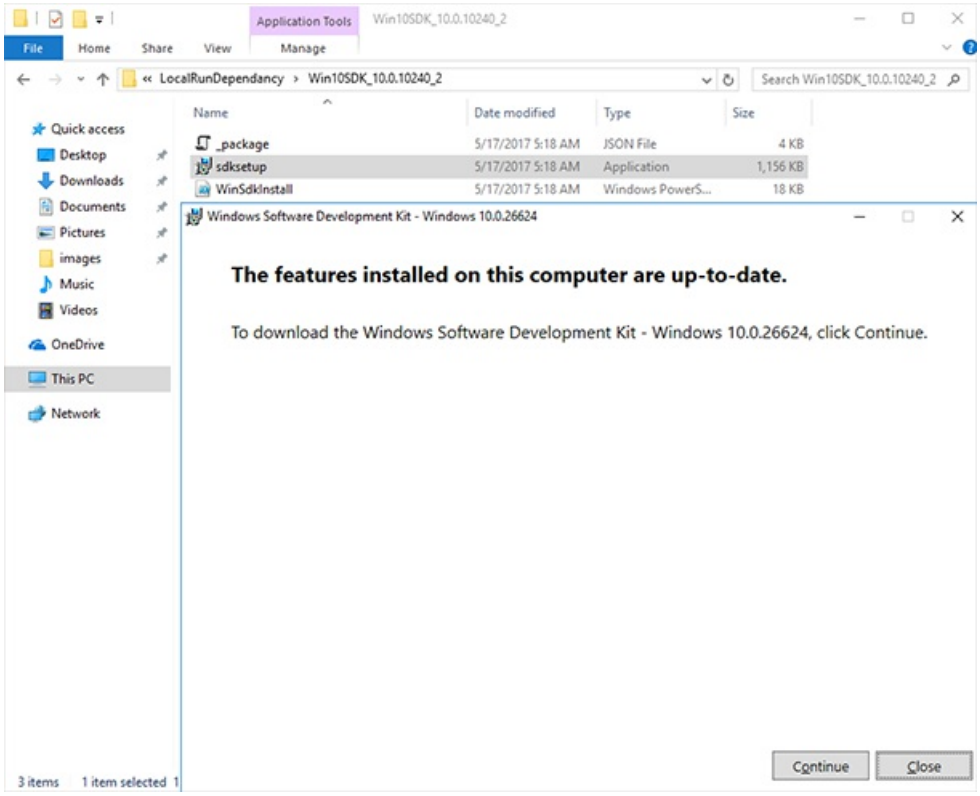
```
C:\Users\xxx\AppData\Roaming\LocalRunDependency
```




2.1 To install **BuildTools**, click visualcppbuildtools_full.exe in the LocalRunDependency folder, then follow the wizard instructions.




2.2 To install **Win10SDK 10240**, click sdksetup.exe in the LocalRunDependency/Win10SDK_10.0.10240_2 folder, then follow the wizard instructions.



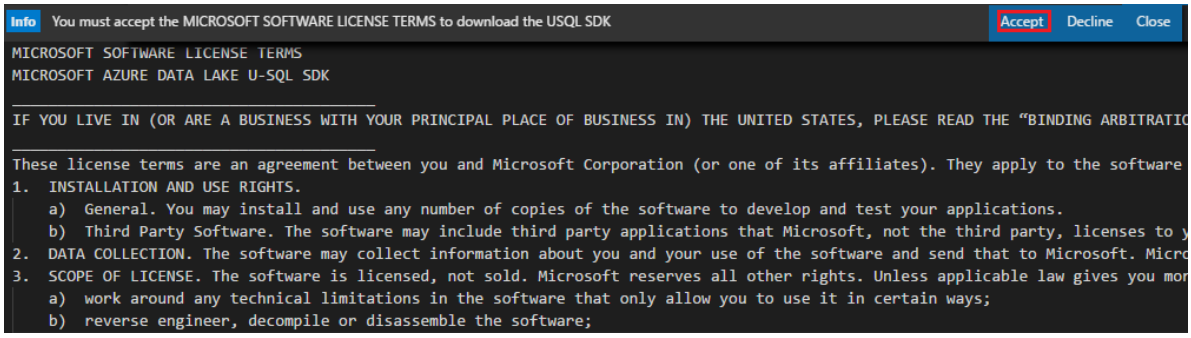

3. Set up the environment variable. Set the **SCOPE_CPP_SDK** environment variable to:
   `C:\Users\XXX\AppData\Roaming\LocalRunDependency\CppSDK_3rdparty`

## Start the local run service and submit the U-SQL job to a local account

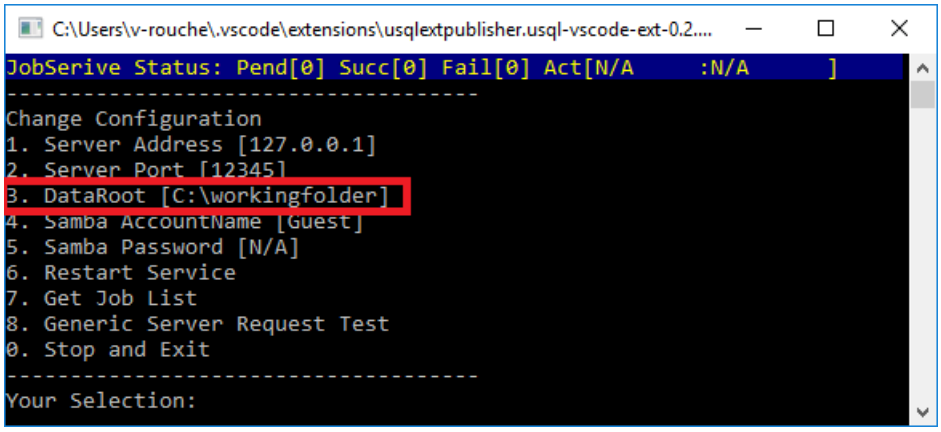
For the first-time user, use **ADL: Download Local Run Package** to download local run packages, if you have not set up U-SQL local run environment.

1. Select Ctrl+Shift+P to open the command palette, and then enter **ADL: Start Local Run Service**.

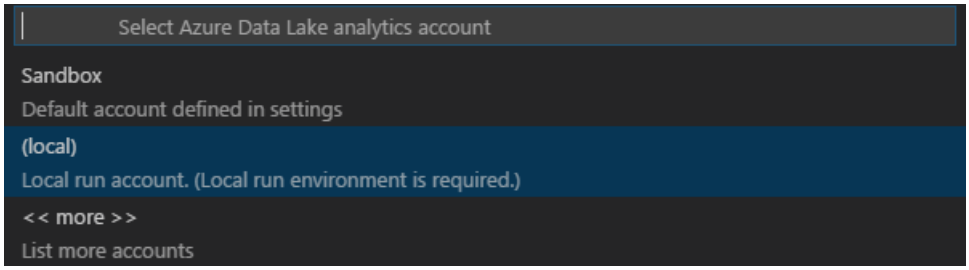
2. Select **Accept** to accept the Microsoft Software License Terms for the first time.



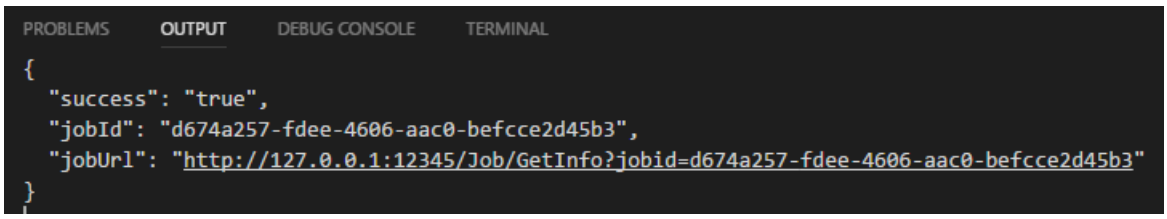

3. The cmd console opens. For first-time users, you need to enter **3**, and then locate the local folder path for your data input and output. For other options, you can use the default values.

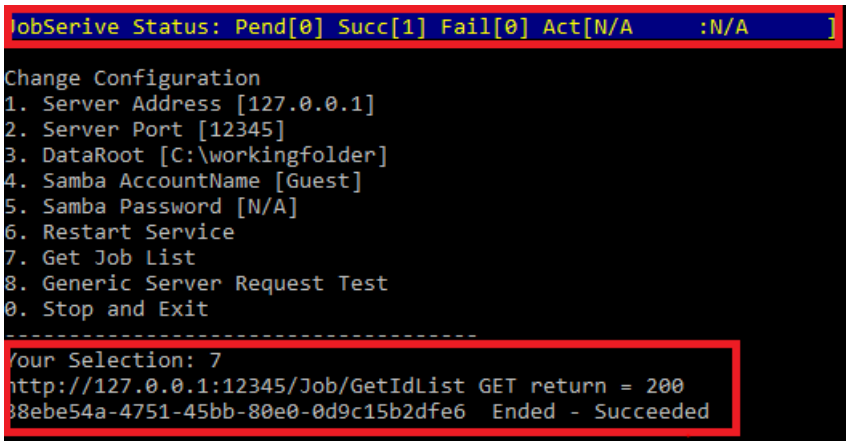



4. Select Ctrl+Shift+P to open the command palette, enter **ADL: Submit Job**, and then select **Local** to submit the job to your local account.




5. After you submit the job, you can view the submission details. To view the submission details, select **jobUrl** in the **Output** window. You can also view the job submission status from the cmd console. Enter **7** in the cmd console if you want to know more job details.






## Start a local debug for the U-SQL job

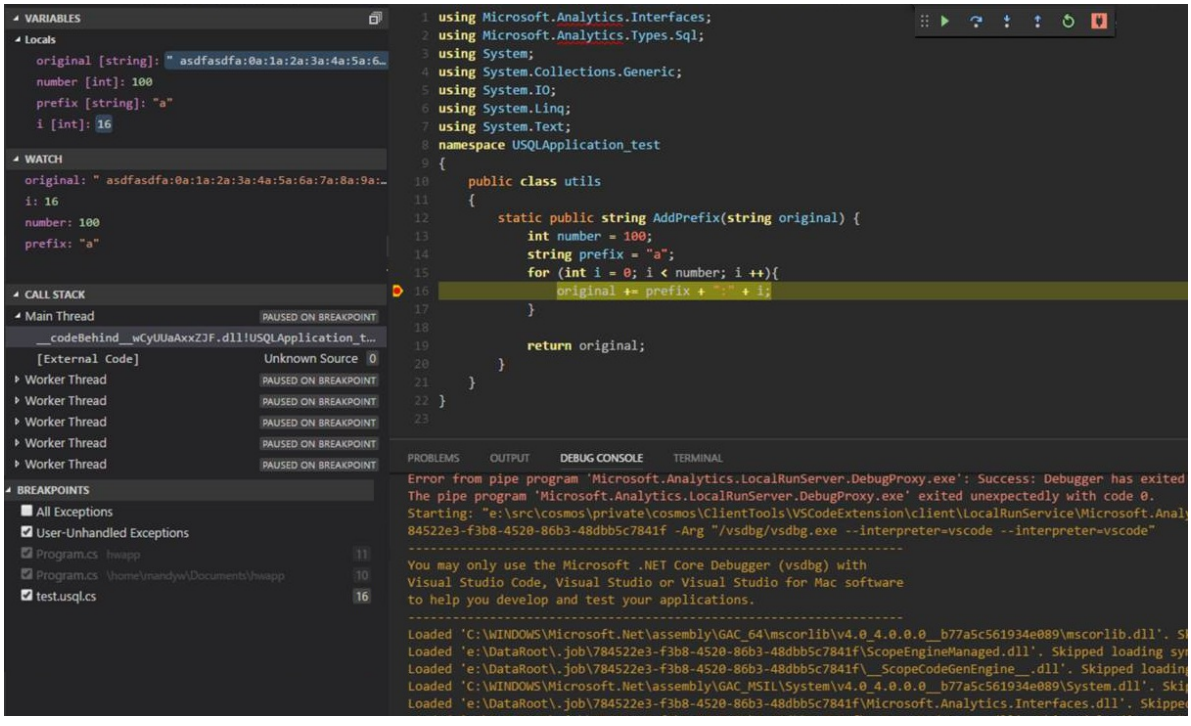
For the first-time user:

1. Use **ADL: Download Local Run Package** to download local run packages, if you have not set up U-SQL local run environment.
2. Install .NET Core SDK 2.0 as suggested in the message box, if not installed.


3. Install C# for Visual Studio Code as suggested in the message box if not installed. Click **Install** to continue, and then restart VSCode.


Follow steps below to perform local debug:

1. Select Ctrl+Shift+P to open the command palette, and then enter **ADL: Start Local Run Service**. The cmd console opens. Make sure that the **DataRoot** is set.
2. Set a breakpoint in your C# code-behind.
3. Back to script editor, right-click and select **ADL: Local Debug**.



## Next steps

- Use the Azure Data Lake Tools for Visual Studio Code
- Develop U-SQL with Python, R, and CSharp for Azure Data Lake Analytics in VSCode
- Get started with Data Lake Analytics using PowerShell
- Get started with Data Lake Analytics using the Azure portal
- Use Data Lake Tools for Visual Studio for developing U-SQL applications
- Use Data Lake Analytics(U-SQL) catalog

# Schedule U-SQL jobs using SQL Server Integration Services (SSIS)

9/13/2018 • 5 minutes to read • Edit Online

In this document, you learn how to orchestrate and create U-SQL jobs using SQL Server Integration Service (SSIS).

## Prerequisites

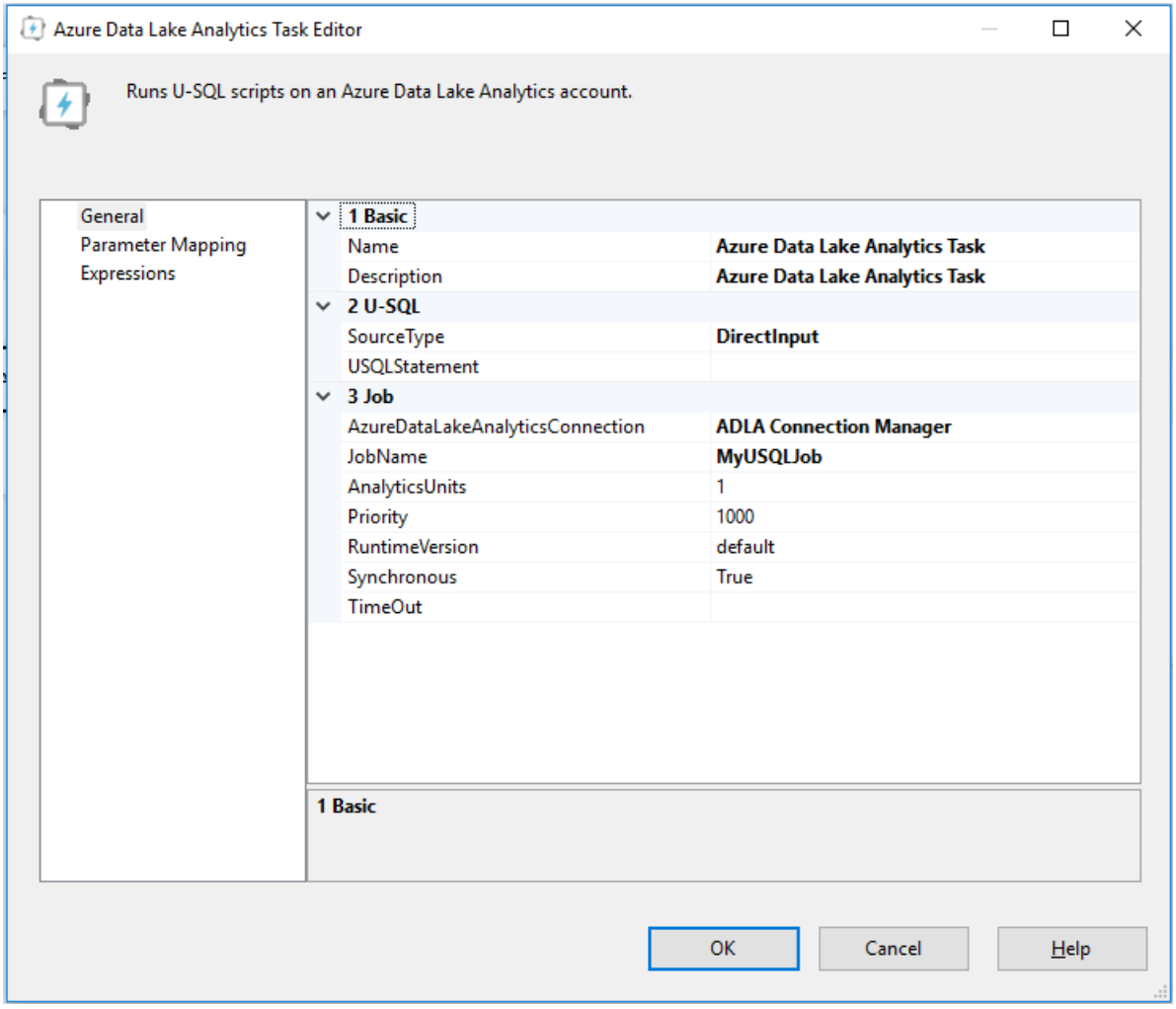
Azure Feature Pack for Integration Services provides the Azure Data Lake Analytics task and the Azure Data Lake Analytics Connection Manager that helps connect to Azure Data Lake Analytics service. To use this task, make sure you install:

- Download and install SQL Server Data Tools (SSDT) for Visual Studio
- Install Azure Feature Pack for Integration Services (SSIS)

## Azure Data Lake Analytics task

The Azure Data Lake Analytics task let users submit U-SQL jobs to the Azure Data Lake Analytics account.

Learn how to configure Azure Data Lake Analytics task.



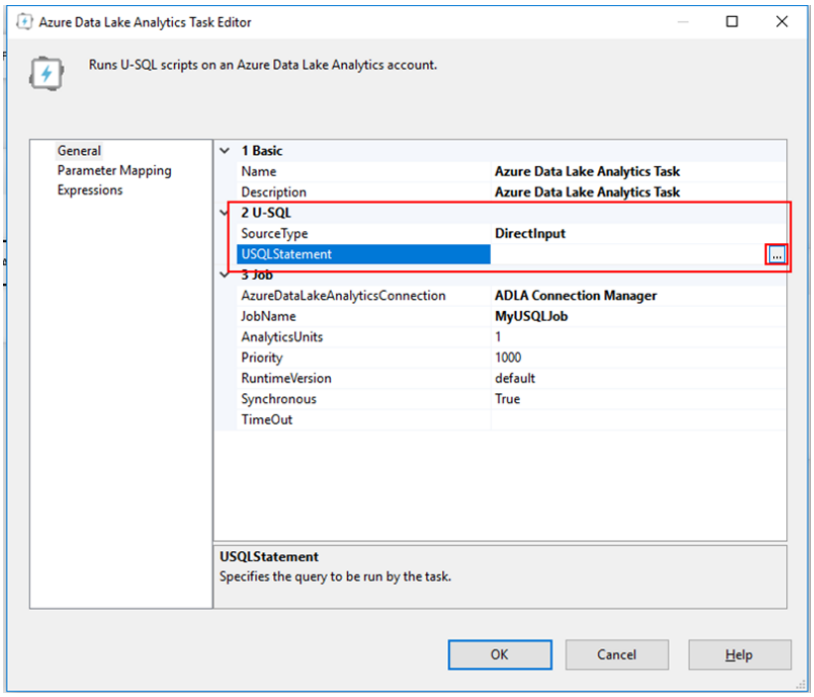

You can get the U-SQL script from different places by using SSIS built-in functions and tasks, below scenarios

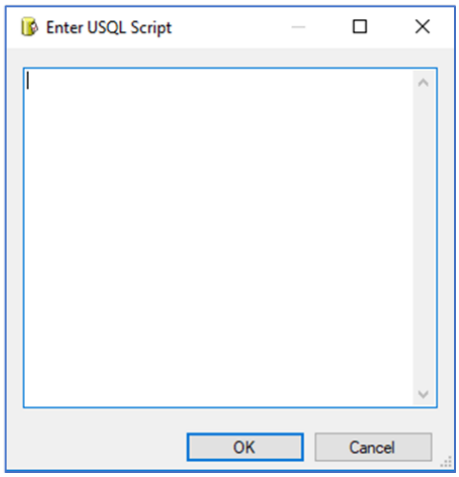
show how can you configure the U-SQL scripts for different user cases.

## Scenario 1-Use inline script call tvfs and stored procs

In Azure Data Lake Analytics Task Editor, configure **SourceType** as **DiretInput**, and put the U-SQL statements into **USQLStatemnt**.

For easy maintainence and code management, only put short U-SQL script as inline scripts, for example, you can call existing table valued functions and stored procedures in your U-SQL databases.




Related article: How to pass parameter to stored procedures

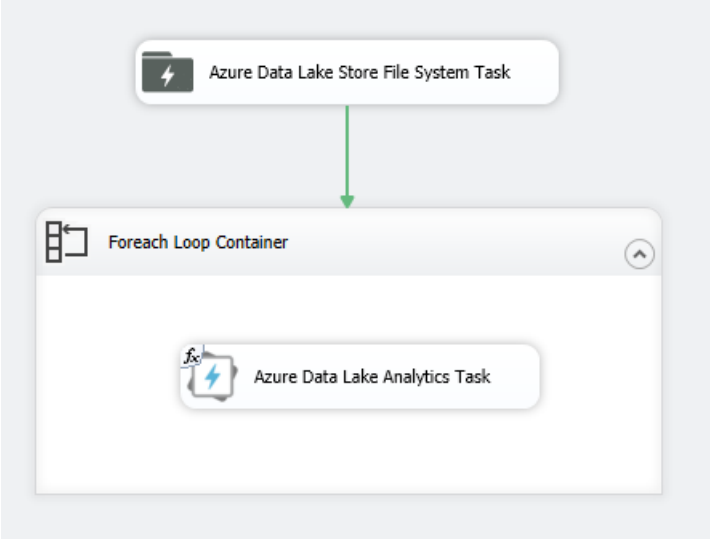
## Scenario 2-Use U-SQL files in Azure Data Lake Store

You can also use U-SQL files in the Azure Data Lake Store by using **Azure Data Lake Store File System Task** in Azure Feature Pack. This approach enables you to use the scripts stored on cloud.

Follow below steps to set up the connection between Azure Data Lake Store File System Task and Azure Data Lake Analytics Task.

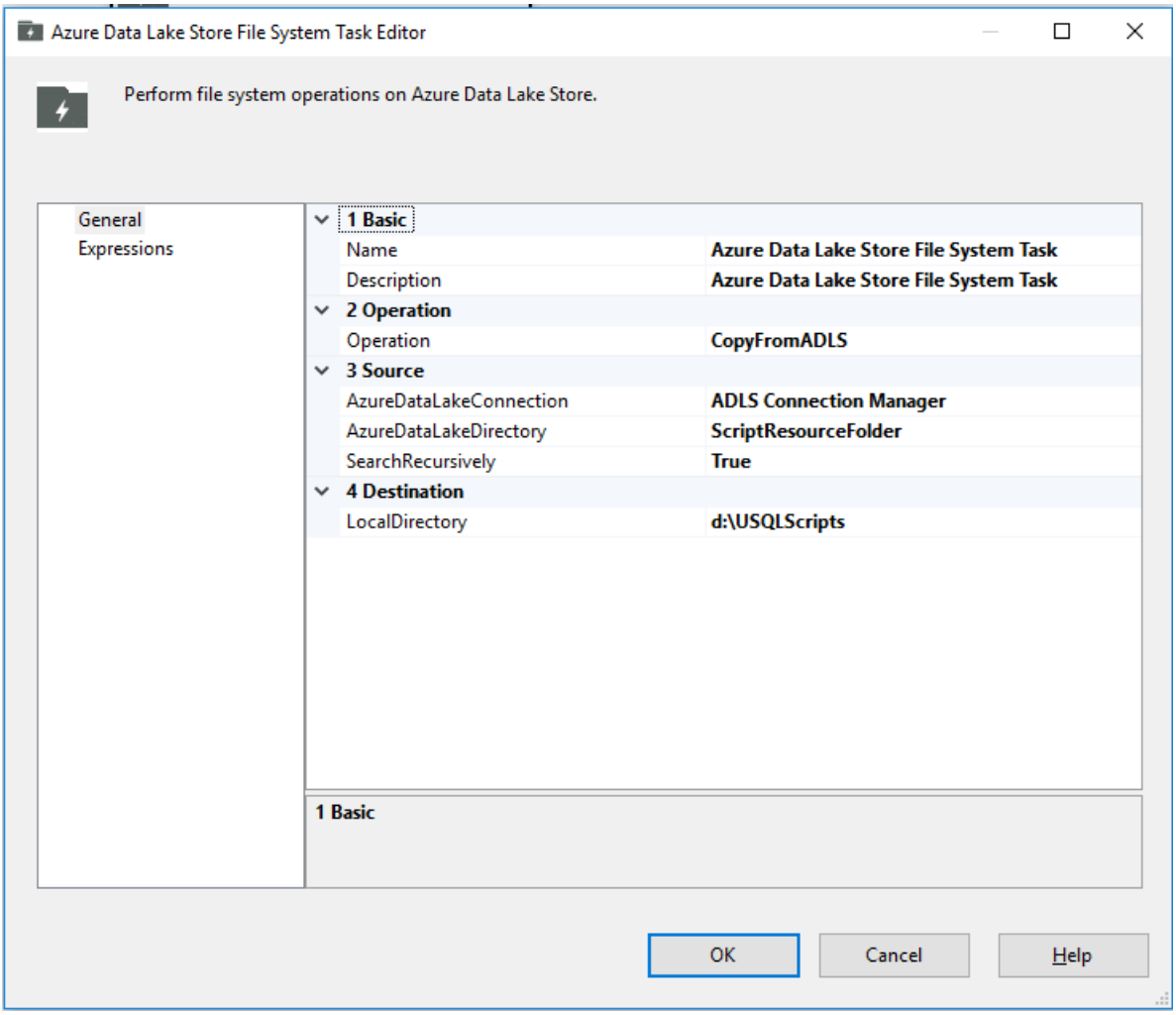
### Set task control flow

In SSIS package design view, add an **Azure Data Lake Store File System Task**, a **Foreach Loop Container** and an **Azure Data Lake Analytics Task** in the Foreach Loop Container. The Azure Data Lake Store File System Task helps to download U-SQL files in your ADLS account to a temporary folder. The Foreach Loop Container and the Azure Data Lake Analytics Task help to submit every U-SQL file under the temporary folder to the Azure Data Lake Analytics account as a U-SQL job.




### Configure Azure Data Lake Store File System Task

1. Set **Operation** to **CopyFromADLS**.
2. Set up **AzureDataLakeConnection**, learn more about Azure Data Lake Store Connection Manager.
3. Set **AzureDataLakeDirectory**. Point to the folder storing your U-SQL scripts. Use relative path that is relative to the Azure Data Lake Store account root folder.
4. Set **Destination** to a folder that caches the downloaded U-SQL scripts. This folder path will be used in Foreach Loop Container for U-SQL job submission.



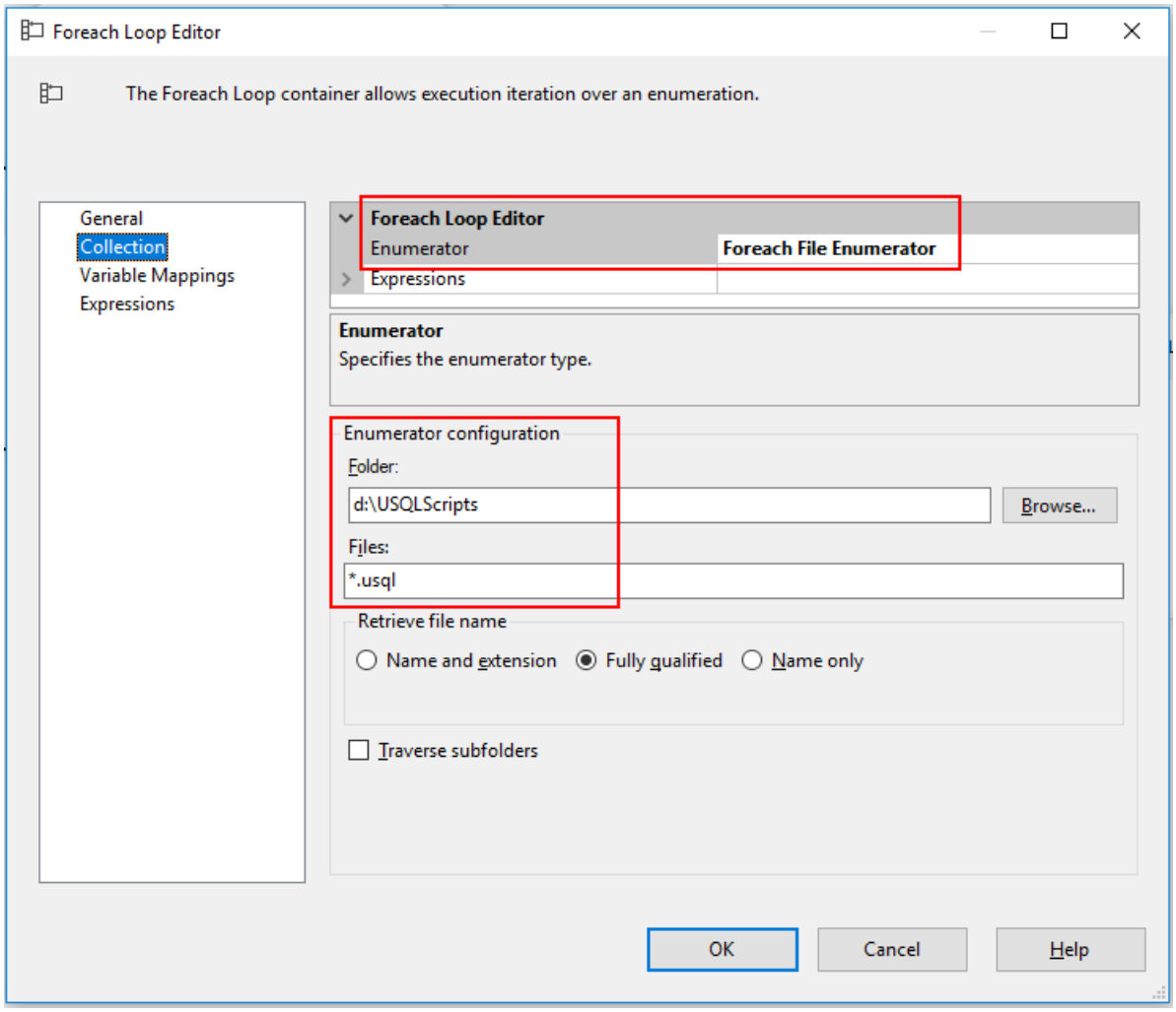

Learn more about Azure Data Lake Store File System Task.

### Configure Foreach Loop Container

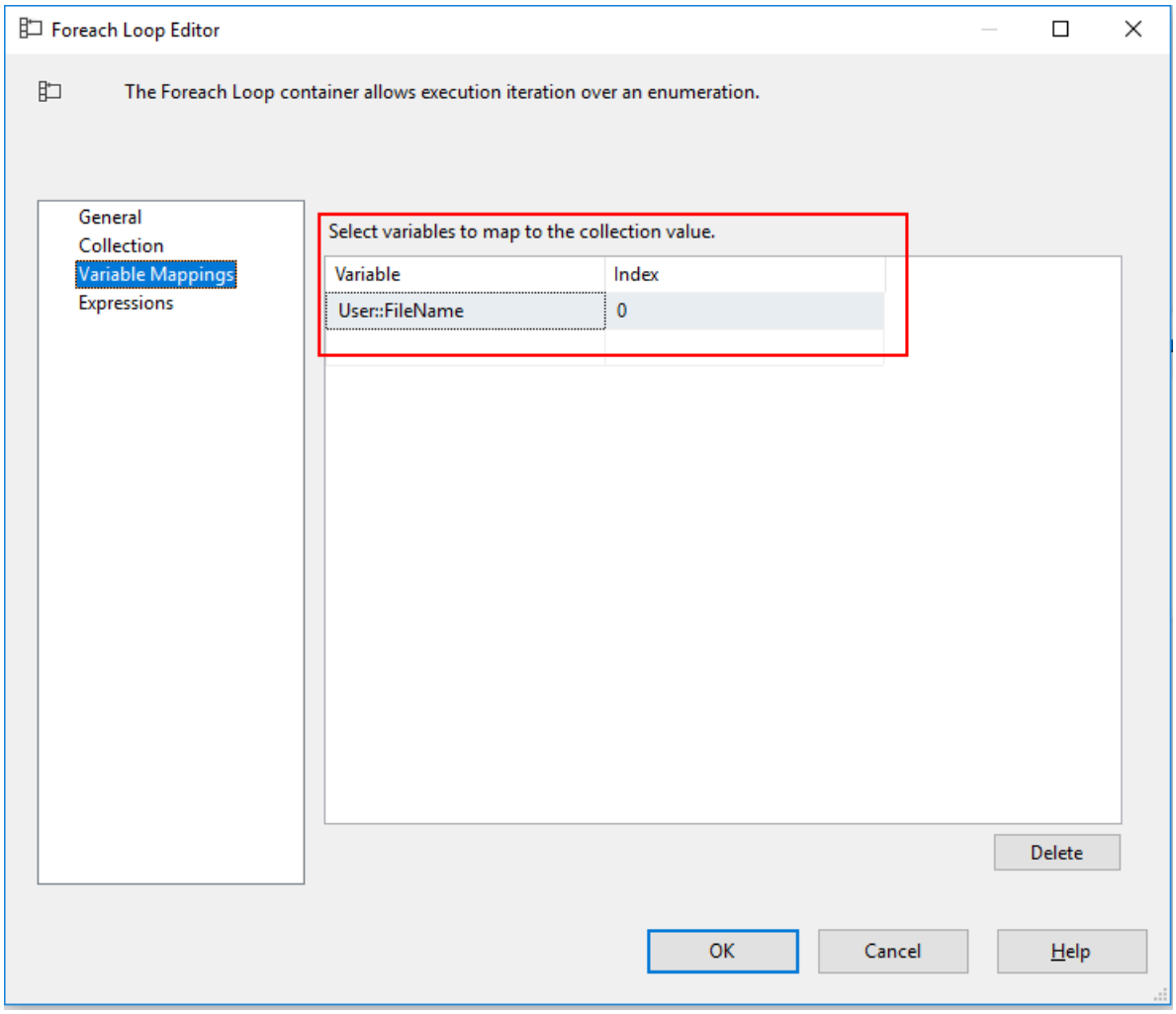
1. In **Collection** page, set **Enumerator** to **Foreach File Enumerator**.

2. Set **Folder** under **Enumerator configuration** group to the temporary folder that includes the downloaded U-SQL scripts.

3. Set **Files** under **Enumerator configuration** to `*.usql` so that the loop container only catches the files ending with `.usql`.




4. In **Variable Mappings** page, add a user defined variable to get the file name for each U-SQL file. Set the **Index** to 0 to get the file name. In this example, define a variable called `User::FileName`. This variable will be used to dynamically get U-SQL script file connection and set U-SQL job name in Azure Data Lake Analytics Task.



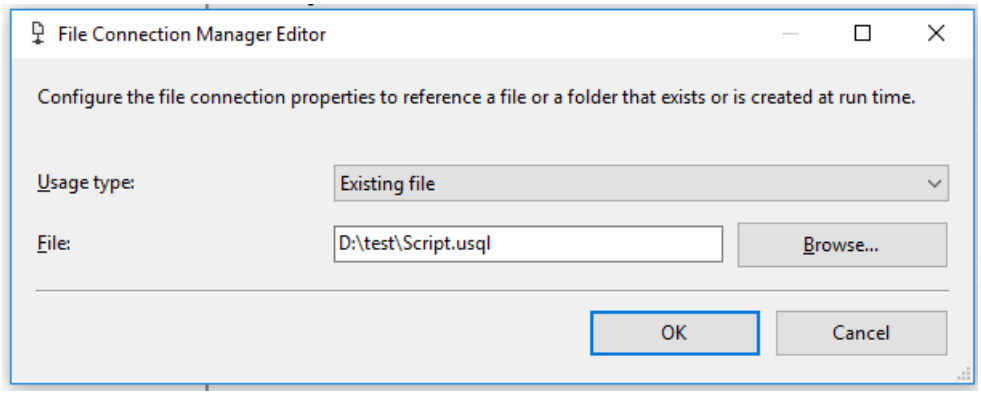

**Configure Azure Data Lake Analytics Task**

1. Set **SourceType** to **FileConnection**.
2. Set **FileConnection** to the file connection that points to the file objects returned from Foreach Loop Container.

   To create this file connection:

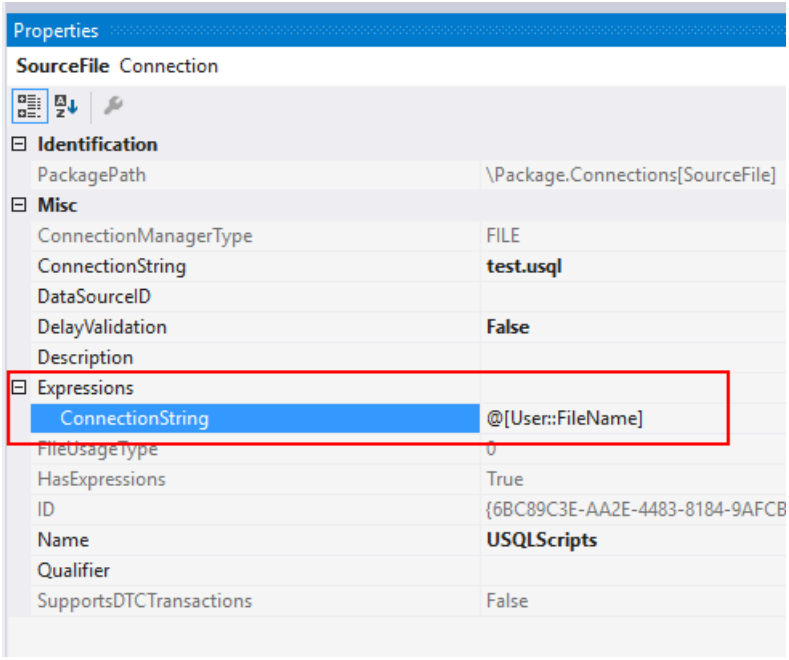
   a. Choose **<New Connection...>** in FileConnection setting.

   b. Set **Usage type** to **Existing file**, and set the **File** to any existing file's file path.




   c. In **Connection Managers** view, right-click the file connection created just now, and choose **Properties**.

   d. In the **Properties** window, expand **Expressions**, and set **ConnectionString** to the variable defined in Foreach Loop Container, for example, `@[User::FileName]`.



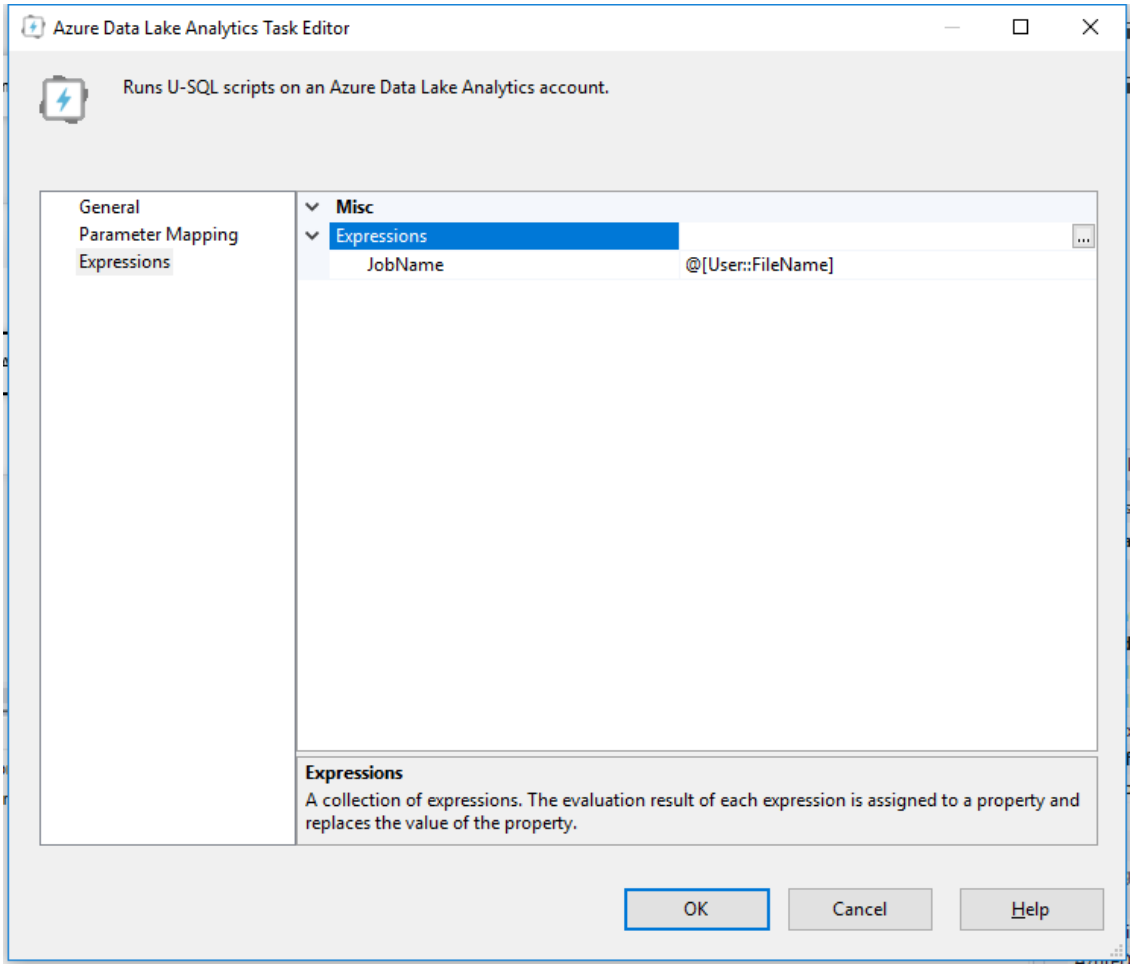

3. Set **AzureDataLakeAnalyticsConnection** to the Azure Data Lake Analytics account that you want to submit jobs to. Learn more about Azure Data Lake Analytics Connection Manager.

4. Set other job configurations. Learn More.

5. Use **Expressions** to dynamically set U-SQL job name:

   a. In **Expressions** page, add a new expression key-value pair for **JobName**.

   b. Set the value for JobName to the variable defined in Foreach Loop Container, for example, `@[User::FileName]`.

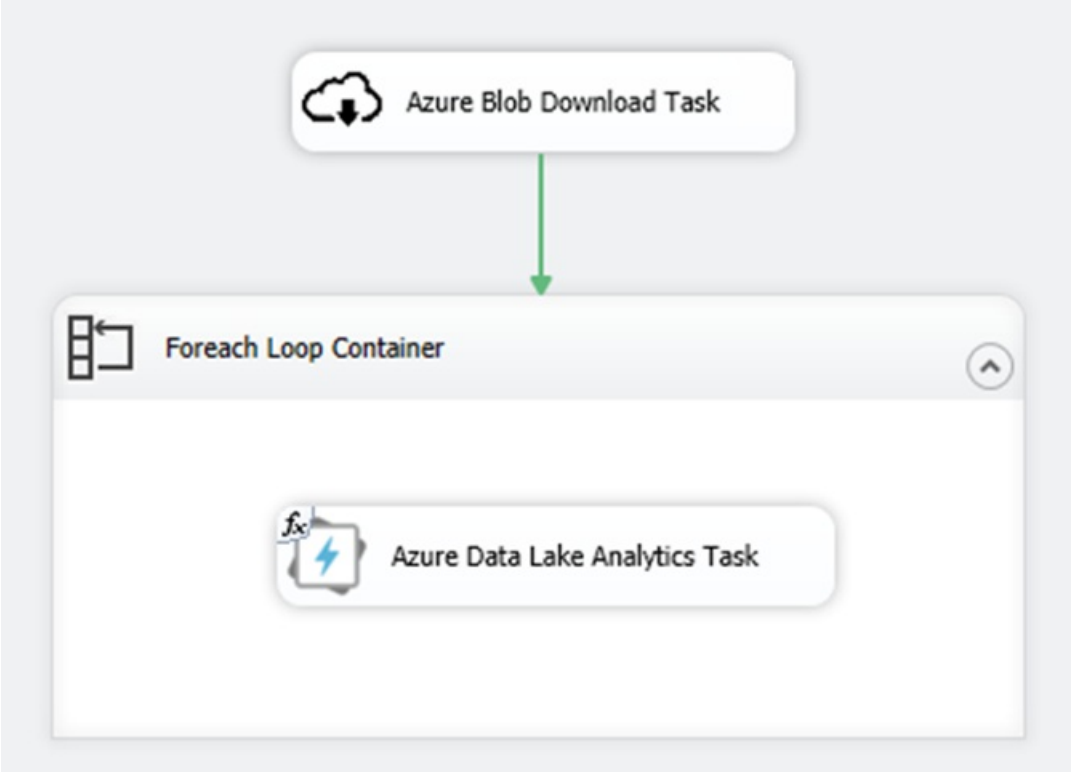


## Scenario 3-Use U-SQL files in Azure Blob Storage

You can use U-SQL files in Azure Blob Storage by using **Azure Blob Download Task** in Azure Feature Pack. This approach enables you using the scripts on cloud.

The steps are similar with Scnario 2: Use U-SQL files in Azure Data Lake Store. Change the Azure Data Lake Store File System Task to Azure Blob Download Task. Learn more about Azure Blob Download Task.

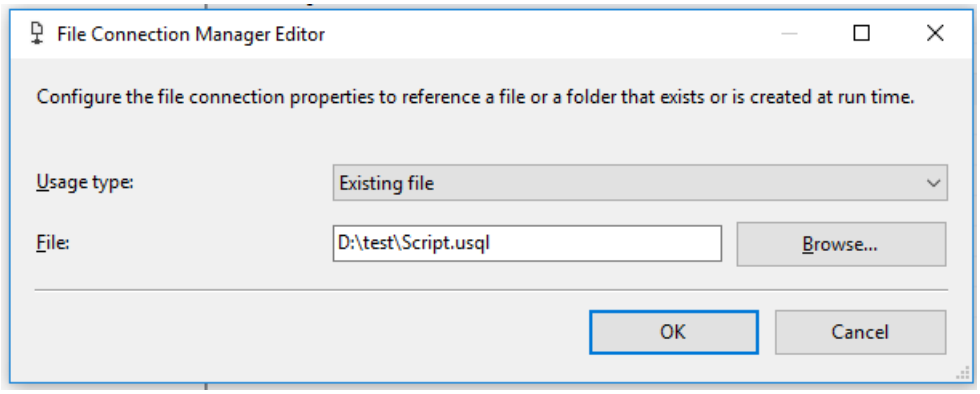
The control flow is like below.




## Scenario 4-Use U-SQL files on the local machine

Besides of using U-SQL files stored on cloud, you can also use files on your local machine or files deployed with your SSIS packages.

1. Right-click **Connection Managers** in SSIS project and choose **New Connection Manager**.
2. Select **File** type and click **Add...**.
3. Set **Usage type** to **Existing file**, and set the **File** to the file on the local machine.




4. Add **Azure Data Lake Analytics** Task and:
   a. Set **SourceType** to **FileConnection**.
   b. Set **FileConnection** to the File Connection created just now.

5. Finish other configurations for Azure Data Lake Analytics Task.

## Scenario 5-Use U-SQL statement in SSIS variable

In some cases, you may need to dynamically generate the U-SQL statements. You can use **SSIS Variable** with **SSIS Expression** and other SSIS tasks, like Script Task, to help you generate the U-SQL statement dynamically.

1. Open Variables tool window through **SSIS > Variables** top-level menu.
2. Add an SSIS Variable and set the value directly or use **Expression** to generate the value.
3. Add **Azure Data Lake Analytics Task** and:
    a. Set **SourceType** to **Variable**.
    b. Set **SourceVariable** to the SSIS Variable created just now.
4. Finish other configurations for Azure Data Lake Analytics Task.

## Scenario 6-Pass parameters to U-SQL script

In some cases, you may want to dynamically set the U-SQL variable value in the U-SQL script. **Parameter Mapping** feature in Azure Data Lake Analytics Task help with this scenario. There are usually two typical user cases:

- Set the input and output file path variables dynamically based on current date and time.
- Set the parameter for stored procedures.

Learn more about how to set parameters for the U-SQL script.

## Next steps

- Run SSIS packages in Azure
- Azure Feature Pack for Integration Services (SSIS)
- Schedule U-SQL jobs using Azure Data Factory

# How to set up a CI/CD pipeline for Azure Data Lake Analytics

10/8/2018 • 14 minutes to read • Edit Online

In this article, you learn how to set up a continuous integration and deployment (CI/CD) pipeline for U-SQL jobs and U-SQL databases.

## Use CI/CD for U-SQL jobs

Azure Data Lake Tools for Visual Studio provides the U-SQL project type that helps you organize U-SQL scripts. Using the U-SQL project to manage your U-SQL code makes further CI/CD scenarios easy.

## Build a U-SQL project

A U-SQL project can be built with the Microsoft Build Engine (MSBuild) by passing corresponding parameters. Follow the steps in this article to set up a build process for a U-SQL project.

**Project migration**

Before you set up a build task for a U-SQL project, make sure you have the latest version of the U-SQL project. Open the U-SQL project file in your editor and verify that you have these import items:

```
<!-- check for SDK Build target in current path then in USQLSDKPath-->
<Import Project="UsqlSDKBuild.targets" Condition="Exists('UsqlSDKBuild.targets')" />
<Import Project="$(USQLSDKPath)\UsqlSDKBuild.targets" Condition="!Exists('UsqlSDKBuild.targets') And
'$(USQLSDKPath)' != '' And Exists('$(USQLSDKPath)\UsqlSDKBuild.targets')" />
```

If not, you have two options to migrate the project:

- Option 1: Change the old import item to the preceding one.
- Option 2: Open the old project in the Azure Data Lake Tools for Visual Studio. Use a version newer than 2.3.3000.0. The old project template will be upgraded automatically to the newest version. New projects created with versions newer than 2.3.3000.0 use the new template.

**Get NuGet**

MSBuild doesn't provide built-in support for U-SQL projects. To get this support, you need to add a reference for your solution to the Microsoft.Azure.DataLake.USQL.SDK NuGet package that adds the required language service.

To add the NuGet package reference, right-click the solution in Visual Studio Solution Explorer and choose **Manage NuGet Packages**. Or you can add a file called `packages.config` in the solution folder and put the following contents into it:

```
<?xml version="1.0" encoding="utf-8"?>
<packages>
  <package id="Microsoft.Azure.DataLake.USQL.SDK" version="1.3.180620" targetFramework="net452" />
</packages>
```

**Manage U-SQL database references**

U-SQL scripts in a U-SQL project might have query statements for U-SQL database objects. In that case, you need to reference the corresponding U-SQL database project that includes the objects' definition before you build

the U-SQL project. An example is when you query a U-SQL table or reference an assembly.

Learn more about U-SQL database project.

> **NOTE**
> U-SQL database project is currently in public preview. If you have DROP statement in the project, the build fails. The DROP statement will be allowed soon.

#### Build a U-SQL project with the MSBuild command line

First migrate the project and get the NuGet package. Then call the standard MSBuild command line with the following additional arguments to build your U-SQL project:

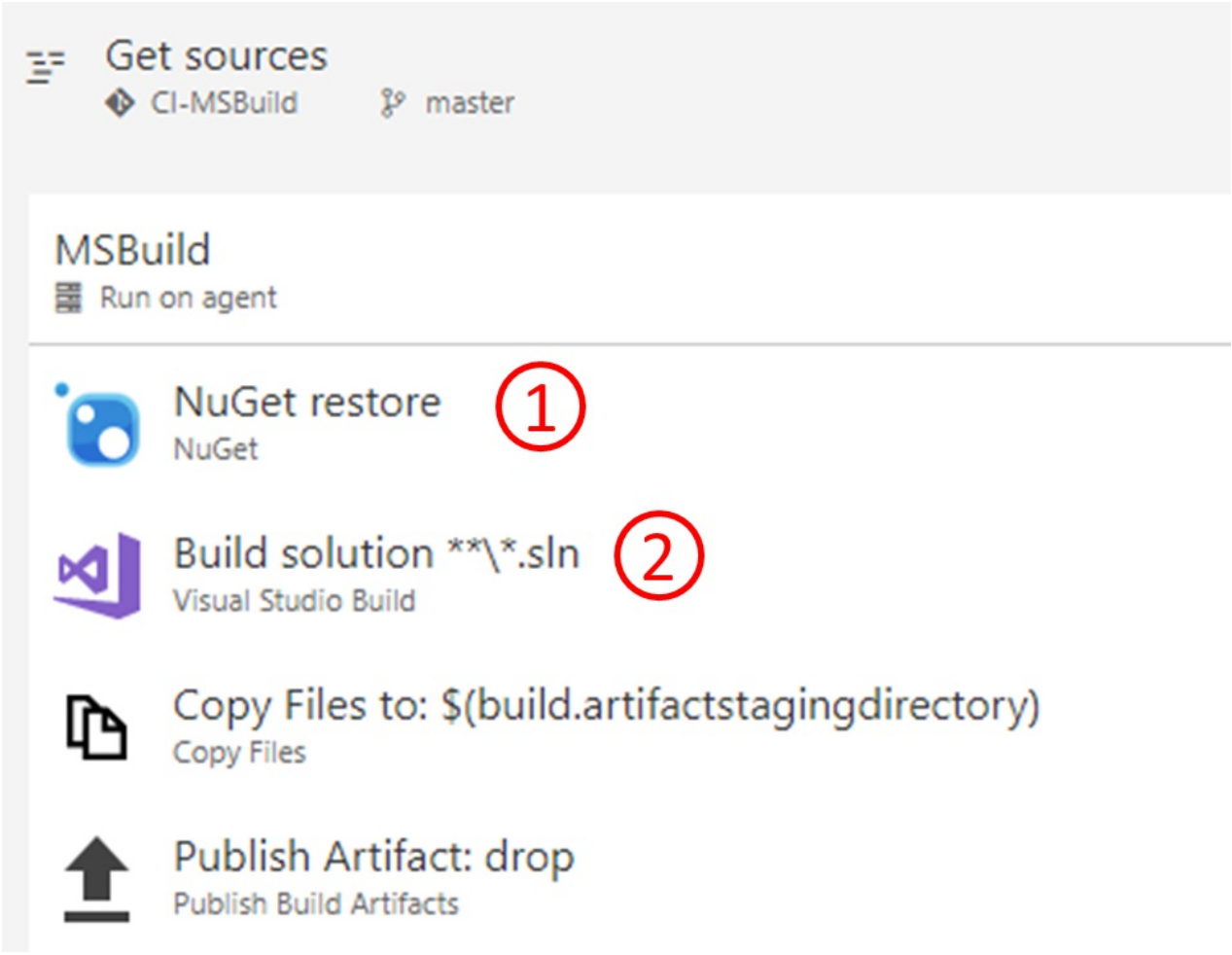
```
msbuild USQLBuild.usqlproj
/p:USQLSDKPath=packages\Microsoft.Azure.DataLake.USQL.SDK.1.3.180615\build\runtime;USQLTargetType=SyntaxCheck;
DataRoot=datarootfolder;/p:EnableDeployment=true
```

The arguments definition and values are as follows:

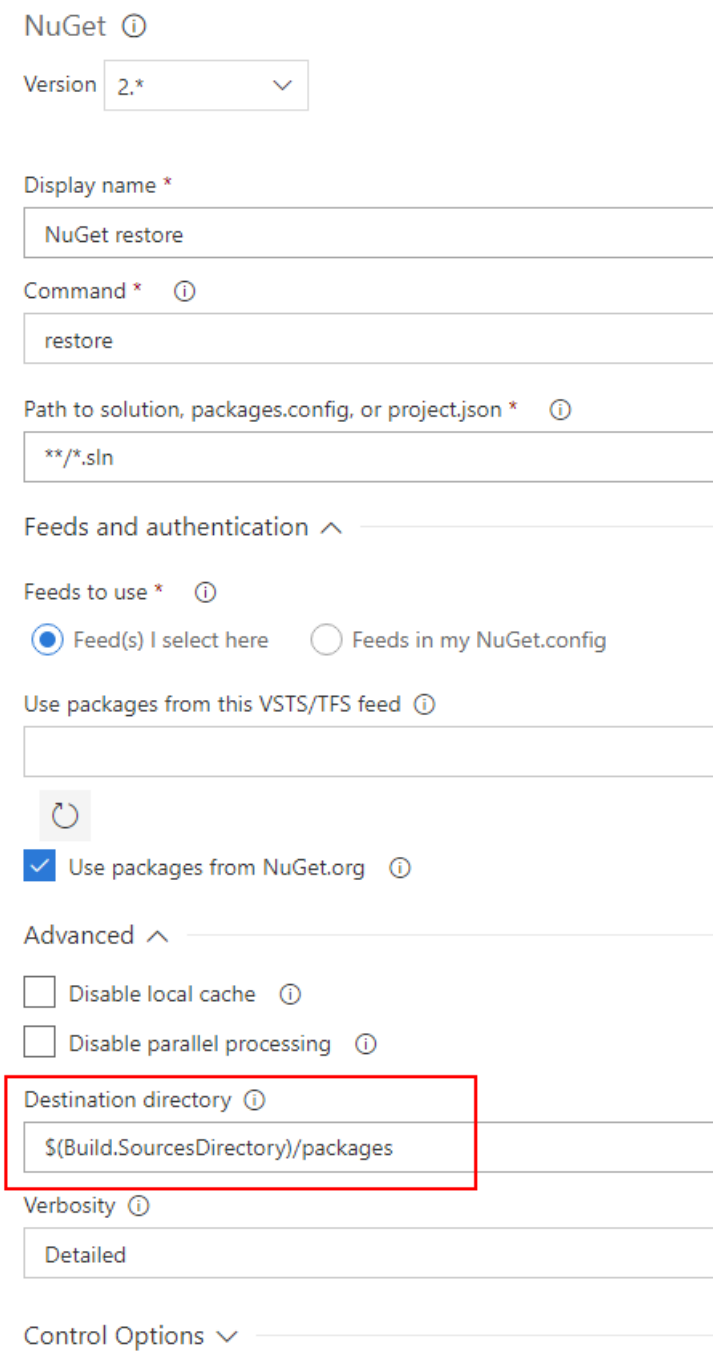
- **USQLSDKPath=\build\runtime**. This parameter refers to the installation path of the NuGet package for the U-SQL language service.
- **USQLTargetType=Merge or SyntaxCheck**:
  - **Merge**. Merge mode compiles code-behind files. Examples are **.cs**, **.py**, and **.r** files. It inlines the resulting user-defined code library into the U-SQL script. Examples are a dll binary, Python, or R code.
  - **SyntaxCheck**. SyntaxCheck mode first merges code-behind files into the U-SQL script. Then it compiles the U-SQL script to validate your code.
- **DataRoot=**. DataRoot is needed only for SyntaxCheck mode. When it builds the script with SyntaxCheck mode, MSBuild checks the references to database objects in the script. Before building, set up a matching local environment that contains the referenced objects from the U-SQL database in the build machine's DataRoot folder. You can also manage these database dependencies by referencing a U-SQL database project. MSBuild only checks database object references, not files.
- **EnableDeployment=true** or **false**. EnableDeployment indicates if it's allowed to deploy referenced U-SQL databases during the build process. If you reference a U-SQL database project and consume the database objects in your U-SQL script, set this parameter to **true**.

#### Continuous integration through Azure Pipelines

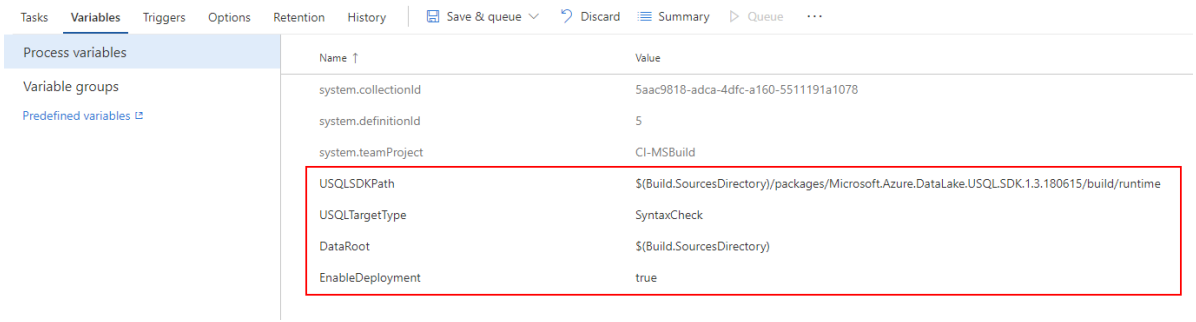
In addition to the command line, you can also use the Visual Studio Build or an MSBuild task to build U-SQL projects in Azure Pipelines. To set up a build pipeline, make sure to add two tasks in the build pipeline: a NuGet restore task and an MSBuild task.




1. Add a NuGet restore task to get the solution-referenced NuGet package that includes `Azure.DataLake.USQL.SDK`, so that MSBuild can find the U-SQL language targets. Set **Advanced** > **Destination directory** to `$(Build.SourcesDirectory)/packages` if you want to use the MSBuild arguments sample directly in step 2.




2. Set MSBuild arguments in Visual Studio build tools or in an MSBuild task as shown in the following example. Or you can define variables for these arguments in the Azure Pipelines build pipeline.




```
/p:USQLSDKPath=/p:USQLSDKPath=$(Build.SourcesDirectory)/packages/Microsoft.Azure.DataLake.USQL.SDK.1.3.180615/build/runtime /p:USQLTargetType=SyntaxCheck /p:DataRoot=$(Build.SourcesDirectory) /p:EnableDeployment=true
```

### U-SQL project build output

After you run a build, all scripts in the U-SQL project are built and output to a zip file called `USQLProjectName.usqlpack`. The folder structure in your project is kept in the zipped build output.

> **NOTE**
> Code-behind files for each U-SQL script will be merged as an inline statement to the script build output.

## Test U-SQL scripts

Azure Data Lake provides test projects for U-SQL scripts and C# UDO/UDAG/UDF:

- Learn how to add test cases for U-SQL scripts and extended C# code.
- Learn how to run test cases in Azure Pipelines.

## Deploy a U-SQL job

After you verify code through the build and test process, you can submit U-SQL jobs directly from Azure Pipelines through an Azure PowerShell task. You can also deploy the script to Azure Data Lake Store or Azure Blob storage and run the scheduled jobs through Azure Data Factory.

### Submit U-SQL jobs through Azure Pipelines

The build output of the U-SQL project is a zip file called **USQLProjectName.usqlpack**. The zip file includes all U-SQL scripts in the project. You can use the Azure PowerShell task in Pipelines with the following sample PowerShell script to submit U-SQL jobs directly from Azure Pipelines.

```
<#
    This script can be used to submit U-SQL Jobs with given U-SQL project build output(.usqlpack file).
    This will unzip the U-SQL project build output, and submit all scripts one-by-one.

    Note: the code behind file for each U-SQL script will be merged into the built U-SQL script in build
output.

    Example :
        USQLJobSubmission.ps1 -ADLAAccountName "myadlaaccount" -ArtifactsRoot "C:\USQLProject\bin\debug\" -
DegreeOfParallelism 2
#>

param(
    [Parameter(Mandatory=$true)][string]$ADLAAccountName, # ADLA account name to submit U-SQL jobs
    [Parameter(Mandatory=$true)][string]$ArtifactsRoot, # Root folder of U-SQL project build output
    [Parameter(Mandatory=$false)][string]$DegreeOfParallelism = 1
)

function Unzip($USQLPackfile, $UnzipOutput)
{
    $USQLPackfileZip = Rename-Item -Path $USQLPackfile -NewName
$([System.IO.Path]::ChangeExtension($USQLPackfile, ".zip")) -Force -PassThru
    Expand-Archive -Path $USQLPackfileZip -DestinationPath $UnzipOutput -Force
    Rename-Item -Path $USQLPackfileZip -NewName $([System.IO.Path]::ChangeExtension($USQLPackfileZip,
".usqlpack")) -Force
}

## Get U-SQL scripts in U-SQL project build output(.usqlpack file)
Function GetUsqlFiles()
{

    $USQLPackfiles = Get-ChildItem -Path $ArtifactsRoot -Include *.usqlpack -File -Recurse -ErrorAction
SilentlyContinue

    $UnzipOutput = Join-Path $ArtifactsRoot -ChildPath "UnzipUSQLScripts"
```

```
    $UnzipOutput = Join-Path $ArtifactsRoot "UnzipUSQLScripts"

    foreach ($USQLPackfile in $USQLPackfiles)
    {
        Unzip $USQLPackfile $UnzipOutput
    }

    $USQLFiles = Get-ChildItem -Path $UnzipOutput -Include *.usql -File -Recurse -ErrorAction SilentlyContinue

    return $USQLFiles
}

## Submit U-SQL scripts to ADLA account one-by-one
Function SubmitAnalyticsJob()
{
    $usqlFiles = GetUsqlFiles

    Write-Output "$($usqlFiles.Count) jobs to be submitted..."

    # Submit each usql script and wait for completion before moving ahead.
    foreach ($usqlFile in $usqlFiles)
    {
        $scriptName = "[Release].[$([System.IO.Path]::GetFileNameWithoutExtension($usqlFile.fullname))]"

        Write-Output "Submitting job for '{$usqlFile}'"

        $jobToSubmit = Submit-AzureRmDataLakeAnalyticsJob -Account $ADLAAccountName -Name $scriptName -
ScriptPath $usqlFile -DegreeOfParallelism $DegreeOfParallelism

        LogJobInformation $jobToSubmit

        Write-Output "Waiting for job to complete. Job ID:'{$($jobToSubmit.JobId)}', Name:
'$($jobToSubmit.Name)' "
        $jobResult = Wait-AzureRmDataLakeAnalyticsJob -Account $ADLAAccountName -JobId $jobToSubmit.JobId
        LogJobInformation $jobResult
    }
}

Function LogJobInformation($jobInfo)
{
    Write-Output "********************************************************************************"
    Write-Output ([string]::Format("Job Id: {0}", $(DefaultIfNull $jobInfo.JobId)))
    Write-Output ([string]::Format("Job Name: {0}", $(DefaultIfNull $jobInfo.Name)))
    Write-Output ([string]::Format("Job State: {0}", $(DefaultIfNull $jobInfo.State)))
    Write-Output ([string]::Format("Job Started at: {0}", $(DefaultIfNull  $jobInfo.StartTime)))
    Write-Output ([string]::Format("Job Ended at: {0}", $(DefaultIfNull  $jobInfo.EndTime)))
    Write-Output ([string]::Format("Job Result: {0}", $(DefaultIfNull $jobInfo.Result)))
    Write-Output "********************************************************************************"
}

Function DefaultIfNull($item)
{
    if ($item -ne $null)
    {
        return $item
    }
    return ""
}

Function Main()
{
    Write-Output ([string]::Format("ADLA account: {0}", $ADLAAccountName))
    Write-Output ([string]::Format("Root folde for usqlpack: {0}", $ArtifactsRoot))
    Write-Output ([string]::Format("AU count: {0}", $DegreeOfParallelism))

    Write-Output "Starting USQL script deployment..."

    SubmitAnalyticsJob

    Write-Output "Finished deployment..."
```

```
    Write-Output "Finished deployment..."
}

Main
```

### Deploy U-SQL jobs through Azure Data Factory

You can submit U-SQL jobs directly from Azure Pipelines. Or you can upload the built scripts to Azure Data Lake Store or Azure Blob storage and run the scheduled jobs through Azure Data Factory.

Use the Azure PowerShell task in Azure Pipelines with the following sample PowerShell script to upload the U-SQL scripts to an Azure Data Lake Store account:

```
<#
    This script can be used to upload U-SQL files to ADLS with given U-SQL project build output(.usqlpack
file).
    This will unzip the U-SQL project build output, and upload all scripts to ADLS one-by-one.

    Example :
        FileUpload.ps1 -ADLSName "myadlsaccount" -ArtifactsRoot "C:\USQLProject\bin\debug\"
#>

param(
    [Parameter(Mandatory=$true)][string]$ADLSName, # ADLS account name to upload U-SQL scripts
    [Parameter(Mandatory=$true)][string]$ArtifactsRoot, # Root folder of U-SQL project build output
    [Parameter(Mandatory=$false)][string]$DesitinationFolder = "USQLScriptSource" # Desitination folder in
ADLS
)

Function UploadResources()
{
    Write-Host "********************************************************************************"
    Write-Host "Uploading files to $ADLSName"
    Write-Host "********************************************************************************"

    $usqlScripts = GetUsqlFiles

    $files = @(get-childitem $usqlScripts -recurse)
    foreach($file in $files)
    {
        Write-Host "Uploading file: $($file.Name)"
        Import-AzureRmDataLakeStoreItem -AccountName $ADLSName -Path $file.FullName -Destination "/$(Join-Path
$DesitinationFolder $file)" -Force
    }
}

function Unzip($USQLPackfile, $UnzipOutput)
{
    $USQLPackfileZip = Rename-Item -Path $USQLPackfile -NewName
$([System.IO.Path]::ChangeExtension($USQLPackfile, ".zip")) -Force -PassThru
    Expand-Archive -Path $USQLPackfileZip -DestinationPath $UnzipOutput -Force
    Rename-Item -Path $USQLPackfileZip -NewName $([System.IO.Path]::ChangeExtension($USQLPackfileZip,
".usqlpack")) -Force
}

Function GetUsqlFiles()
{

    $USQLPackfiles = Get-ChildItem -Path $ArtifactsRoot -Include *.usqlpack -File -Recurse -ErrorAction
SilentlyContinue

    $UnzipOutput = Join-Path $ArtifactsRoot -ChildPath "UnzipUSQLScripts"

    foreach ($USQLPackfile in $USQLPackfiles)
    {
        Unzip $USQLPackfile $UnzipOutput
    }

    return Get-ChildItem -Path $UnzipOutput -Include *.usql -File -Recurse -ErrorAction SilentlyContinue
}

UploadResources
```

## CI/CD for a U-SQL database

Azure Data Lake Tools for Visual Studio provides U-SQL database project templates that help you develop, manage, and deploy U-SQL databases. Learn more about a U-SQL database project.

# Build U-SQL database project

### Get the NuGet package

MSBuild doesn't provide built-in support for U-SQL database projects. To get this ability, you need to add a reference for your solution to the Microsoft.Azure.DataLake.USQL.SDK NuGet package that adds the required language service.

To add the NuGet package reference, right-click the solution in Visual Studio Solution Explorer. Choose **Manage NuGet Packages**. Then search for and install the NuGet package. Or you can add a file called **packages.config** in the solution folder and put the following contents into it:

```
<?xml version="1.0" encoding="utf-8"?>
<packages>
  <package id="Microsoft.Azure.DataLake.USQL.SDK" version="1.3.180615" targetFramework="net452" />
</packages>
```

### Build U-SQL a database project with the MSBuild command line

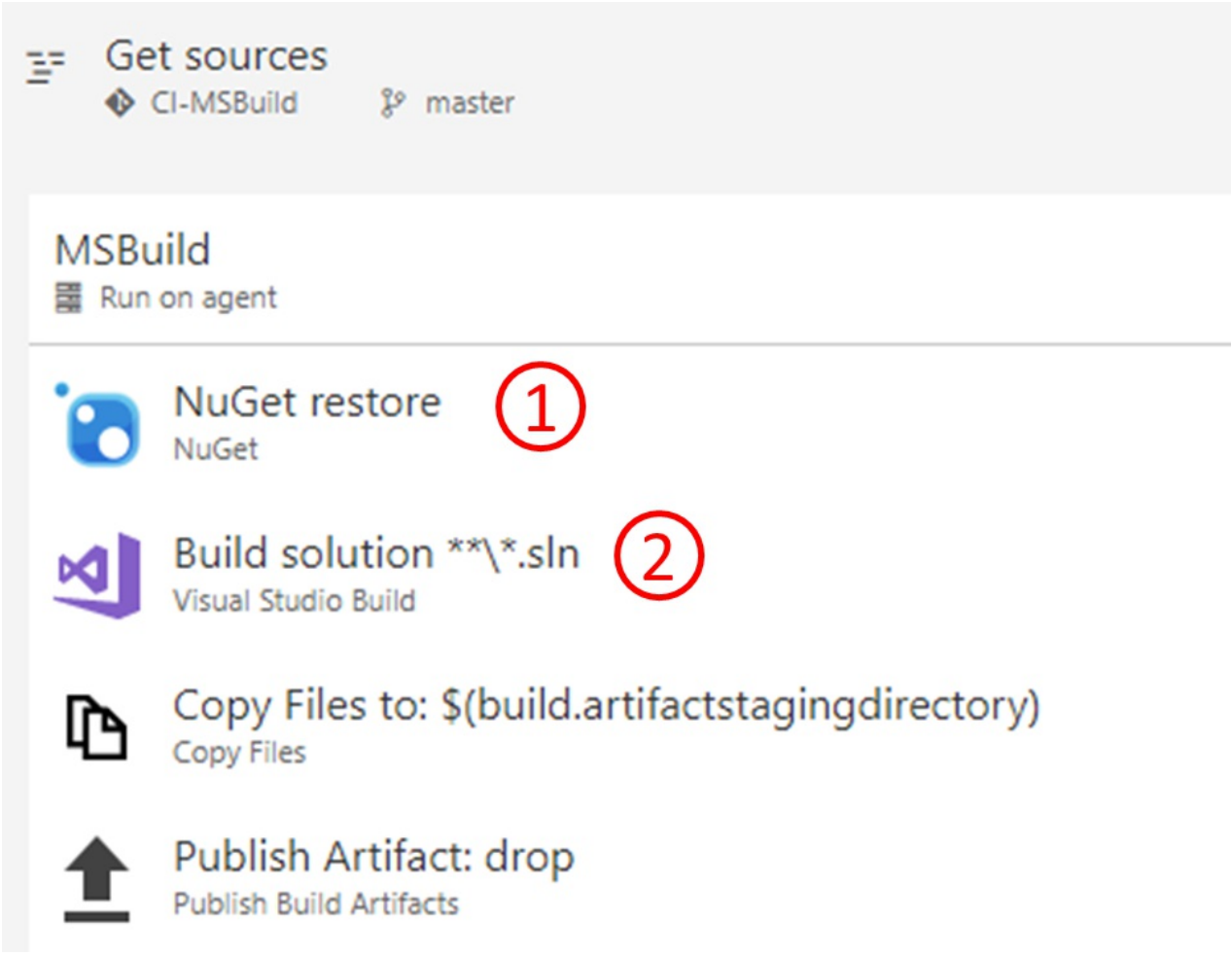
To build your U-SQL database project, call the standard MSBuild command line and pass the U-SQL SDK NuGet package reference as an additional argument. See the following example:

```
msbuild DatabaseProject.usqldbproj
/p:USQLSDKPath=packages\Microsoft.Azure.DataLake.USQL.SDK.1.3.180615\build\runtime
```

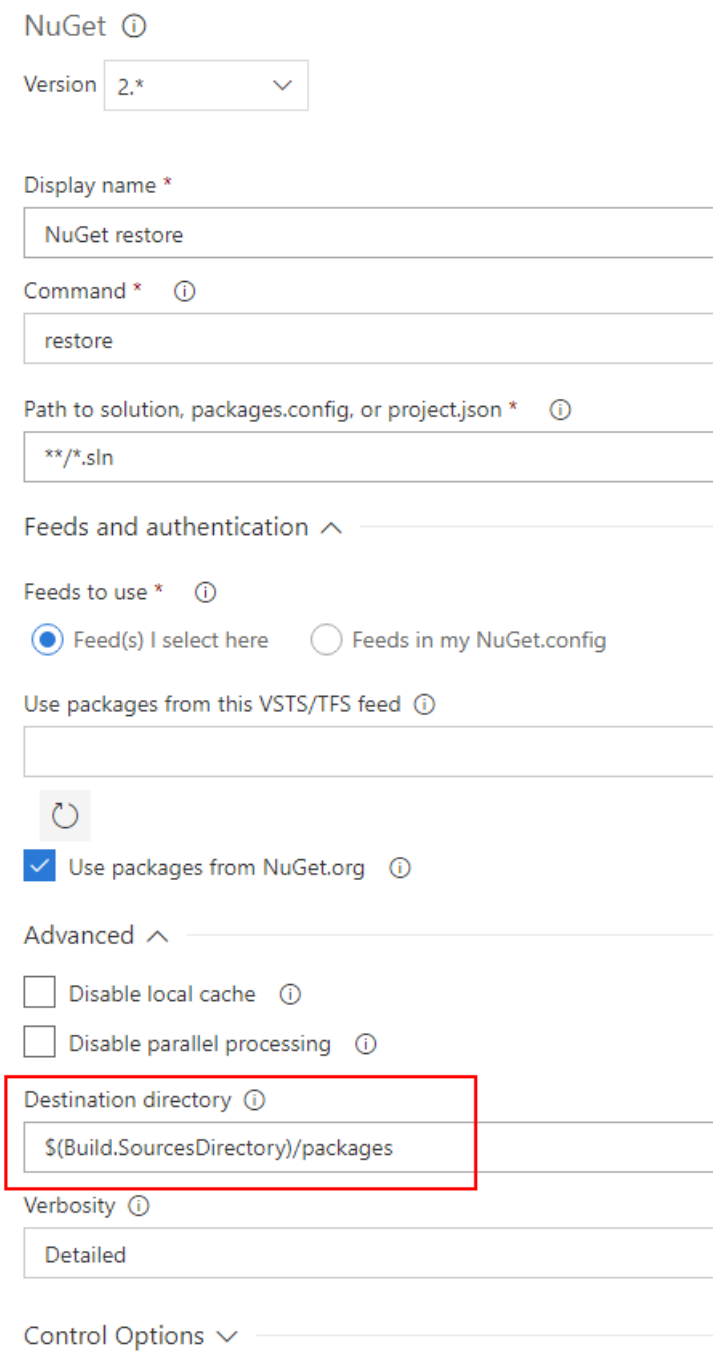
The argument `USQLSDKPath=<U-SQL Nuget package>\build\runtime` refers to the install path of the NuGet package for the U-SQL language service.

### Continuous integration with Azure Pipelines

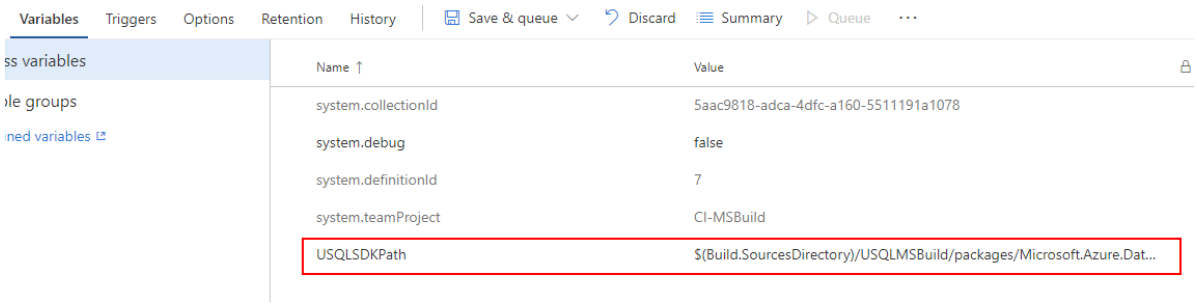
In addition to the command line, you can use Visual Studio Build or an MSBuild task to build U-SQL database projects in Azure Pipelines. To set up a build task, make sure to add two tasks in the build pipeline: a NuGet restore task and an MSBuild task.




1. Add a NuGet restore task to get the solution-referenced NuGet package, which includes `Azure.DataLake.USQL.SDK`, so that MSBuild can find the U-SQL language targets. Set **Advanced** > **Destination directory** to `$(Build.SourcesDirectory)/packages` if you want to use the MSBuild arguments sample directly in step 2.




2. Set MSBuild arguments in Visual Studio build tools or in an MSBuild task as shown in the following example. Or you can define variables for these arguments in the Azure Pipelines build pipeline.




```
/p:USQLSDKPath=/p:USQLSDKPath=$(Build.SourcesDirectory)/packages/Microsoft.Azure.DataLake.USQL.SDK.1.3.180615/build/runtime
```

**U-SQL database project build output**

The build output for a U-SQL database project is a U-SQL database deployment package, named with the suffix `.usqldbpack`. The `.usqldbpack` package is a zip file that includes all DDL statements in a single U-SQL script in a DDL folder. It includes all **.dlls** and additional files for assembly in a temp folder.

## Test table-valued functions and stored procedures

Adding test cases for table-valued functions and stored procedures directly isn't currently supported. As a workaround, you can create a U-SQL project that has U-SQL scripts that call those functions and write test cases for them. Take the following steps to set up test cases for table-valued functions and stored procedures defined in the U-SQL database project:

1. Create a U-SQL project for test purposes and write U-SQL scripts calling the table-valued functions and stored procedures.
2. Add a database reference to the U-SQL project. To get the table-valued function and stored procedure definition, you need to reference the database project that contains the DDL statement. Learn more about database references.
3. Add test cases for U-SQL scripts that call table-valued functions and stored procedures. Learn how to add test cases for U-SQL scripts.

## Deploy U-SQL database through Azure Pipelines

`PackageDeploymentTool.exe` provides the programming and command-line interfaces that help deploy U-SQL database deployment packages, **.usqldbpack**. The SDK is included in the U-SQL SDK NuGet package, located at **build/runtime/PackageDeploymentTool.exe**. By using `PackageDeploymentTool.exe`, you can deploy U-SQL databases to both Azure Data Lake Analytics and local accounts.

> **NOTE**
> PowerShell command-line support and Azure Pipelines release task support for U-SQL database deployment is currently pending.

Take the following steps to set up a database deployment task in Azure Pipelines:

1. Add a PowerShell Script task in a build or release pipeline and execute the following PowerShell script. This task helps to get Azure SDK dependencies for `PackageDeploymentTool.exe` and `PackageDeploymentTool.exe`. You can set the **-AzureSDK** and **-DBDeploymentTool** parameters to load the dependencies and deployment tool to specific folders. Pass the **-AzureSDK** path to `PackageDeploymentTool.exe` as the **-AzureSDKPath** parameter in step 2.

```
<#
    This script is used for getting dependencies and SDKs for U-SQL database deployment.
    PowerShell command line support for deploying U-SQL database package(.usqldbpack file) will come
soon.

    Example :
        GetUSQLDBDeploymentSDK.ps1 -AzureSDK "AzureSDKFolderPath" -DBDeploymentTool
"DBDeploymentToolFolderPath"
#>

param (
    [string]$AzureSDK = "AzureSDK", # Folder to cache Azure SDK dependencies
    [string]$DBDeploymentTool = "DBDeploymentTool", # Folder to cache U-SQL database deployment tool
    [string]$workingfolder = "" # Folder to execute these command lines
)

if ([string]::IsNullOrEmpty($workingfolder))
{
```

```
    $scriptpath = $MyInvocation.MyCommand.Path
    $workingfolder = Split-Path $scriptpath
}
cd $workingfolder

echo "workingfolder=$workingfolder, outputfolder=$outputfolder"
echo "Downloading required packages..."

iwr https://www.nuget.org/api/v2/package/Microsoft.Azure.Management.DataLake.Analytics/3.5.1-preview -
outf Microsoft.Azure.Management.DataLake.Analytics.3.5.1-preview.zip
iwr https://www.nuget.org/api/v2/package/Microsoft.Azure.Management.DataLake.Store/2.4.1-preview -outf
Microsoft.Azure.Management.DataLake.Store.2.4.1-preview.zip
iwr https://www.nuget.org/api/v2/package/Microsoft.IdentityModel.Clients.ActiveDirectory/2.28.3 -outf
Microsoft.IdentityModel.Clients.ActiveDirectory.2.28.3.zip
iwr https://www.nuget.org/api/v2/package/Microsoft.Rest.ClientRuntime/2.3.11 -outf
Microsoft.Rest.ClientRuntime.2.3.11.zip
iwr https://www.nuget.org/api/v2/package/Microsoft.Rest.ClientRuntime.Azure/3.3.7 -outf
Microsoft.Rest.ClientRuntime.Azure.3.3.7.zip
iwr https://www.nuget.org/api/v2/package/Microsoft.Rest.ClientRuntime.Azure.Authentication/2.3.3 -outf
Microsoft.Rest.ClientRuntime.Azure.Authentication.2.3.3.zip
iwr https://www.nuget.org/api/v2/package/Newtonsoft.Json/6.0.8 -outf Newtonsoft.Json.6.0.8.zip
iwr https://www.nuget.org/api/v2/package/Microsoft.Azure.DataLake.USQL.SDK/ -outf USQLSDK.zip

echo "Extracting packages..."

Expand-Archive Microsoft.Azure.Management.DataLake.Analytics.3.5.1-preview.zip -DestinationPath
Microsoft.Azure.Management.DataLake.Analytics.3.5.1-preview -Force
Expand-Archive Microsoft.Azure.Management.DataLake.Store.2.4.1-preview.zip -DestinationPath
Microsoft.Azure.Management.DataLake.Store.2.4.1-preview -Force
Expand-Archive Microsoft.IdentityModel.Clients.ActiveDirectory.2.28.3.zip -DestinationPath
Microsoft.IdentityModel.Clients.ActiveDirectory.2.28.3 -Force
Expand-Archive Microsoft.Rest.ClientRuntime.2.3.11.zip -DestinationPath
Microsoft.Rest.ClientRuntime.2.3.11 -Force
Expand-Archive Microsoft.Rest.ClientRuntime.Azure.3.3.7.zip -DestinationPath
Microsoft.Rest.ClientRuntime.Azure.3.3.7 -Force
Expand-Archive Microsoft.Rest.ClientRuntime.Azure.Authentication.2.3.3.zip -DestinationPath
Microsoft.Rest.ClientRuntime.Azure.Authentication.2.3.3 -Force
Expand-Archive Newtonsoft.Json.6.0.8.zip -DestinationPath Newtonsoft.Json.6.0.8 -Force
Expand-Archive USQLSDK.zip -DestinationPath USQLSDK -Force

echo "Copy required DLLs to output folder..."

mkdir $AzureSDK -Force
mkdir $DBDeploymentTool -Force
copy Microsoft.Azure.Management.DataLake.Analytics.3.5.1-preview\lib\net452\*.dll $AzureSDK
copy Microsoft.Azure.Management.DataLake.Store.2.4.1-preview\lib\net452\*.dll $AzureSDK
copy Microsoft.IdentityModel.Clients.ActiveDirectory.2.28.3\lib\net45\*.dll $AzureSDK
copy Microsoft.Rest.ClientRuntime.2.3.11\lib\net452\*.dll $AzureSDK
copy Microsoft.Rest.ClientRuntime.Azure.3.3.7\lib\net452\*.dll $AzureSDK
copy Microsoft.Rest.ClientRuntime.Azure.Authentication.2.3.3\lib\net452\*.dll $AzureSDK
copy Newtonsoft.Json.6.0.8\lib\net45\*.dll $AzureSDK
copy USQLSDK\build\runtime\*.* $DBDeploymentTool
```

2. Add a **Command-Line task** in a build or release pipeline and fill in the script by calling `PackageDeploymentTool.exe`. `PackageDeploymentTool.exe` is located under the defined **$DBDeploymentTool** folder. The sample script is as follows:

   - Deploy a U-SQL database locally:

     ```
     PackageDeploymentTool.exe deploylocal -Package <package path> -Database <database name> -
     DataRoot <data root path>
     ```

   - Use interactive authentication mode to deploy a U-SQL database to an Azure Data Lake Analytics account:

```
PackageDeploymentTool.exe deploycluster -Package <package path> -Database <database name> -
Account <account name> -ResourceGroup <resource group name> -SubscriptionId <subscript id> -
Tenant <tenant name> -AzureSDKPath <azure sdk path> -Interactive
```

- Use **secrete** authentication to deploy a U-SQL database to an Azure Data Lake Analytics account:

```
PackageDeploymentTool.exe deploycluster -Package <package path> -Database <database name> -
Account <account name> -ResourceGroup <resource group name> -SubscriptionId <subscript id> -
Tenant <tenant name> -ClientId <client id> -Secrete <secrete>
```

- Use **certFile** authentication to deploy a U-SQL database to an Azure Data Lake Analytics account:

```
PackageDeploymentTool.exe deploycluster -Package <package path> -Database <database name> -
Account <account name> -ResourceGroup <resource group name> -SubscriptionId <subscript id> -
Tenant <tenant name> -ClientId <client id> -Secrete <secrete> -CertFile <certFile>
```

### PackageDeploymentTool.exe parameter descriptions

**Common parameters**

| PARAMETER | DESCRIPTION | DEFAULT VALUE | REQUIRED |
|---|---|---|---|
| Package | The path of the U-SQL database deployment package to be deployed. | null | true |
| Database | The database name to be deployed to or created. | master | false |
| LogFile | The path of the file for logging. Default to standard out (console). | null | false |
| LogLevel | Log level: Verbose, Normal, Warning, or Error. | LogLevel.Normal | false |

**Parameter for local deployment**

| PARAMETER | DESCRIPTION | DEFAULT VALUE | REQUIRED |
|---|---|---|---|
| DataRoot | The path of the local data root folder. | null | true |

**Parameters for Azure Data Lake Analytics deployment**

| PARAMETER | DESCRIPTION | DEFAULT VALUE | REQUIRED |
|---|---|---|---|
| Account | Specifies which Azure Data Lake Analytics account to deploy to by account name. | null | true |
| ResourceGroup | The Azure resource group name for the Azure Data Lake Analytics account. | null | true |

| PARAMETER | DESCRIPTION | DEFAULT VALUE | REQUIRED |
| --- | --- | --- | --- |
| SubscriptionId | The Azure subscription ID for the Azure Data Lake Analytics account. | null | true |
| Tenant | The tenant name is the Azure Active Directory (Azure AD) domain name. Find it in the subscription management page in the Azure portal. | null | true |
| AzureSDKPath | The path to search dependent assemblies in the Azure SDK. | null | true |
| Interactive | Whether or not to use interactive mode for authentication. | false | false |
| ClientId | The Azure AD application ID required for non-interactive authentication. | null | Required for non-interactive authentication. |
| Secrete | The secrete or password for non-interactive authentication. It should be used only in a trusted and secure environment. | null | Required for non-interactive authentication, or else use SecreteFile. |
| SecreteFile | The file saves the secrete or password for non-interactive authentication. Make sure to keep it readable only by the current user. | null | Required for non-interactive authentication, or else use Secrete. |
| CertFile | The file saves X.509 certification for non-interactive authentication. The default is to use client secrete authentication. | null | false |
| JobPrefix | The prefix for database deployment of a U-SQL DDL job. | Deploy_ + DateTime.Now | false |

## Next steps

- How to test your Azure Data Lake Analytics code.
- Run U-SQL script on your local machine.
- Use U-SQL database project to develop U-SQL database.

# Test your Azure Data Lake Analytics code

9/10/2018 • 4 minutes to read • Edit Online

Azure Data Lake provides the U-SQL language, which combines declarative SQL with imperative C# to process data at any scale. In this document, you learn how to create test cases for U-SQL and extended C# UDO (user-defined operator) code.

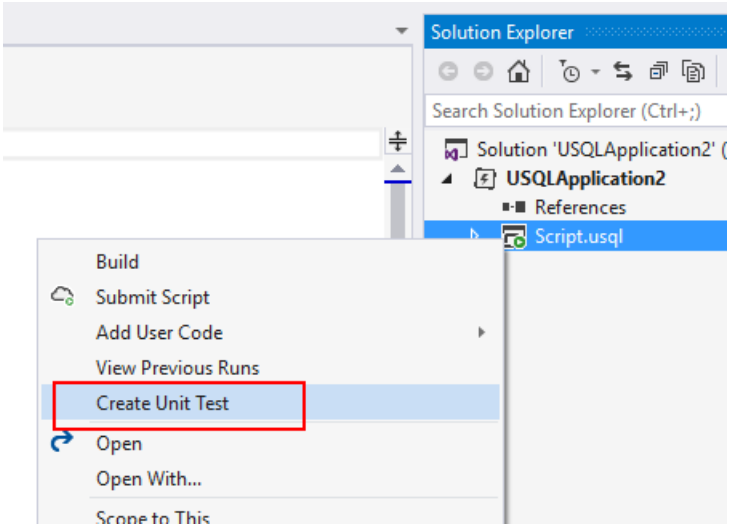
## Test U-SQL scripts

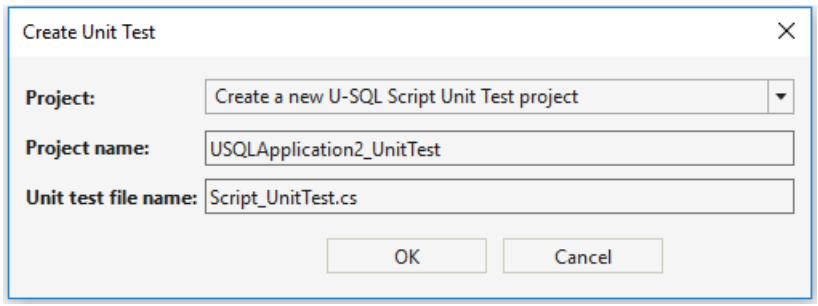
The U-SQL script is compiled and optimized for executable code to run across machines on the cloud or on your local machine. The compilation and optimization process treats the entire U-SQL script as a whole. You can't do a traditional "unit test" for every statement. However, by using the U-SQL test SDK and the local run SDK, you can do script-level tests.

### Create test cases for U-SQL script

Azure Data Lake Tools for Visual Studio enables you to create U-SQL script test cases.

1. Right-click a U-SQL script in Solution Explorer, and then select **Create Unit Test**.
2. Create a new test project or insert the test case into an existing test project.

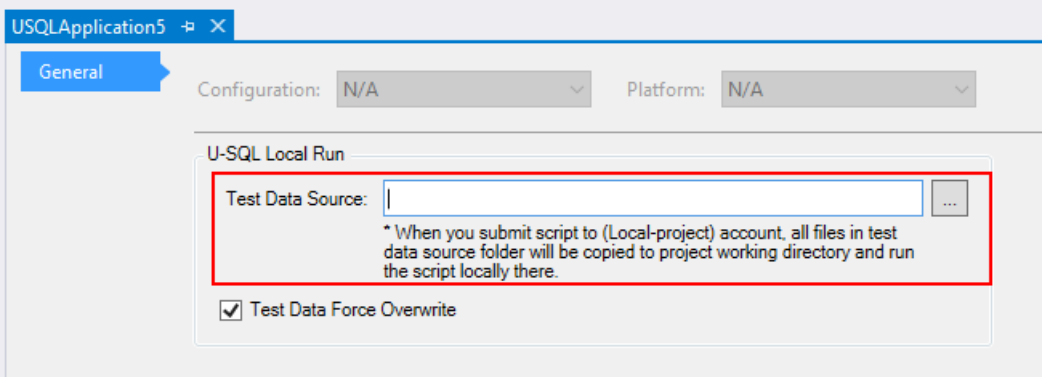



### Manage the test data source

When you test U-SQL scripts, you need test input files. You can manage the test data by configuring **Test Data Source** in the U-SQL project properties.

When you call the `Initialize()` interface in the U-SQL test SDK, a temporary local data root folder is created under the working directory of the test project, and all files and subfolders (and files under subfolders) in the test data source folder are copied to the temporary local data root folder before you run the U-SQL script test cases. You can add more test data source folders by splitting the test data folder path with a semicolon.




### Manage the database environment for testing

If your U-SQL scripts use or query with U-SQL database objects (for example, when calling stored procedures) then you need to initialize the database environment before running U-SQL test cases. The `Initialize()` interface in the U-SQL test SDK helps you deploy all databases that are referenced by the U-SQL project to the temporary local data root folder in the working directory of the test project.

Learn more about how to manage U-SQL database project references for a U-SQL project.

### Verify test results

The `Run()` interface returns a job execution result. 0 means success, and 1 means failure. You can also use C# assert functions to verify the outputs.

### Run test cases in Visual Studio

A U-SQL script test project is built on top of a C# unit test framework. After you build the project, you can run all test cases through **Test Explorer** > **Playlist**. Alternatively, right-click the .cs file, and then select **Run Tests**.

## Test C# UDOs

### Create test cases for C# UDOs

You can use a C# unit test framework to test your C# UDOs (user-defined operators). When testing UDOs, you need to prepare corresponding **IRowset** objects as inputs.

There are two ways to create an IRowset object:

- Load data from a file to create IRowset:

```
//Schema: "a:int, b:int"
USqlColumn<int> col1 = new USqlColumn<int>("a");
USqlColumn<int> col2 = new USqlColumn<int>("b");
List<IColumn> columns = new List<IColumn> { col1, col2 };
USqlSchema schema = new USqlSchema(columns);

//Generate one row with default values
IUpdatableRow output = new USqlRow(schema, null).AsUpdatable();

//Get data from file
IRowset rowset = UnitTestHelper.GetRowsetFromFile(@"processor.txt", schema, output.AsReadOnly(),
discardAdditionalColumns: true, rowDelimiter: null, columnSeparator: '\t');
```

- Use data from a data collection to create IRowset:

```
//Schema: "a:int, b:int"
USqlSchema schema = new USqlSchema(
    new USqlColumn<int>("a"),
    new USqlColumn<int>("b")
);

IUpdatableRow output = new USqlRow(schema, null).AsUpdatable();

//Generate Rowset with specified values
List<object[]> values = new List<object[]>{
    new object[2] { 2, 3 },
    new object[2] { 10, 20 }
};

IEnumerable<IRow> rows = UnitTestHelper.CreateRowsFromValues(schema, values);
IRowset rowset = UnitTestHelper.GetRowsetFromCollection(rows, output.AsReadOnly());
```

#### Verify test results

After you call UDO functions, you can verify the results through the schema and Rowset value verification by using C# assert functions. You can use sample code in a U-SQL C# UDO unit test sample project through **File > New > Project** in Visual Studio.

#### Run test cases in Visual Studio

After you build the test project, you can run all test cases though **Test Explorer > Playlist**, or right-click the .cs file and choose **Run Tests**.

## Run test cases in Azure DevOps

Both **U-SQL script test projects** and **C# UDO test projects** inherit C# unit test projects. The Visual Studio test task in Azure DevOps can run these test cases.

#### Run U-SQL test cases in Azure DevOps

For a U-SQL test, make sure you load `CPPSDK` on your build machine, and then pass the `CPPSDK` path to USqlScriptTestRunner(cppSdkFolderFullPath: @"").

#### What is CPPSDK?

CPPSDK is a package that includes Microsoft Visual C++ 14 and Windows SDK 10.0.10240.0. This is the environment that's needed by the U-SQL runtime. You can get this package under the Azure Data Lake Tools for Visual Studio installation folder:

- For Visual Studio 2015, it is under
  `C:\Program Files (x86)\Microsoft Visual Studio 14.0\Common7\IDE\Extensions\Microsoft\Microsoft Azure Data Lake Tools for Visual Studio 2015\X.X.XXXX.X\CppSDK`
- For Visual Studio 2017, it is under
  `C:\Program Files (x86)\Microsoft Visual Studio\2017\Enterprise\SDK\ScopeCppSDK`

#### Prepare CPPSDK in the Azure DevOps build agent

The most common way to prepare the CPPSDK dependency in Azure DevOps is as follows:

1. Zip the folder that includes the CPPSDK libraries.
2. Check in the .zip file to your source control system. (The .zip file ensures that you check in all libraries under the CPPSDK folder so that some files aren't ignored by ".gitignore".)
3. Unzip the .zip file in the build pipeline.
4. Point `USqlScriptTestRunner` to this unzipped folder on the build machine.

#### Run C# UDO test cases in Azure DevOps

For a C# UDO test, make sure to reference the following assemblies, which are needed for UDOs. If you reference them through the Nuget package Microsoft.Azure.DataLake.USQL.Interfaces, make sure you add a NuGet Restore task in your build pipeline.

- Microsoft.Analytics.Interfaces
- Microsoft.Analytics.Types
- Microsoft.Analytics.UnitTest

## Next steps

- How to set up CI/CD pipeline for Azure Data Lake Analytics
- Run U-SQL script on your local machine
- Use U-SQL database project to develop U-SQL database

# Run and test U-SQL with Azure Data Lake U-SQL SDK

8/27/2018 • 11 minutes to read • Edit Online

When developing U-SQL script, it is common to run and test U-SQL script locally before submit it to cloud. Azure Data Lake provides a Nuget package called Azure Data Lake U-SQL SDK for this scenario, through which you can easily scale U-SQL run and test. It is also possible to integrate this U-SQL test with CI (Continuous Integration) system to automate the compile and test.

If you care about how to manually local run and debug U-SQL script with GUI tooling, then you can use Azure Data Lake Tools for Visual Studio for that. You can learn more from here.

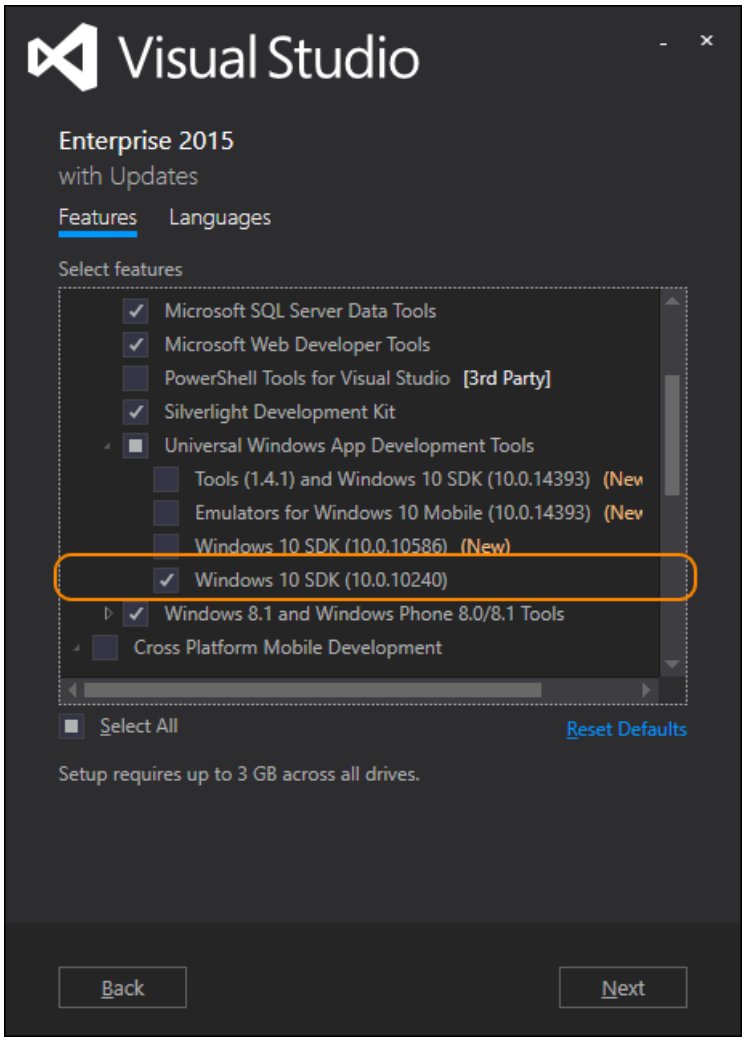
## Install Azure Data Lake U-SQL SDK

You can get the Azure Data Lake U-SQL SDK here on Nuget.org. And before using it, you need to make sure you have dependencies as follows.

**Dependencies**

The Data Lake U-SQL SDK requires the following dependencies:

- Microsoft .NET Framework 4.6 or newer.
- Microsoft Visual C++ 14 and Windows SDK 10.0.10240.0 or newer (which is called CppSDK in this article). There are two ways to get CppSDK:
    - Install Visual Studio Community Edition. You'll have a \Windows Kits\10 folder under the Program Files folder--for example, C:\Program Files (x86)\Windows Kits\10. You'll also find the Windows 10 SDK version under \Windows Kits\10\Lib. If you don't see these folders, reinstall Visual Studio and be sure to select the Windows 10 SDK during the installation. If you have this installed with Visual Studio, the U-SQL local compiler will find it automatically.




- Install Data Lake Tools for Visual Studio. You can find the prepackaged Visual C++ and Windows SDK files at C:\Program Files (x86)\Microsoft Visual Studio 14.0\Common7\IDE\Extensions\Microsoft\ADL Tools\X.X.XXXX.X\CppSDK. In this case, the U-SQL local compiler cannot find the dependencies automatically. You need to specify the CppSDK path for it. You can either copy the files to another location or use it as is.

## Understand basic concepts

**Data root**

The data-root folder is a "local store" for the local compute account. It's equivalent to the Azure Data Lake Store account of a Data Lake Analytics account. Switching to a different data-root folder is just like switching to a different store account. If you want to access commonly shared data with different data-root folders, you must use absolute paths in your scripts. Or, create file system symbolic links (for example, **mklink** on NTFS) under the data-root folder to point to the shared data.

The data-root folder is used to:

- Store local metadata, including databases, tables, table-valued functions (TVFs), and assemblies.
- Look up the input and output paths that are defined as relative paths in U-SQL. Using relative paths makes it easier to deploy your U-SQL projects to Azure.

**File path in U-SQL**

You can use both a relative path and a local absolute path in U-SQL scripts. The relative path is relative to the specified data-root folder path. We recommend that you use "/" as the path separator to make your scripts compatible with the server side. Here are some examples of relative paths and their equivalent absolute paths. In these examples, C:\LocalRunDataRoot is the data-root folder.

| RELATIVE PATH | ABSOLUTE PATH |
| --- | --- |
| /abc/def/input.csv | C:\LocalRunDataRoot\abc\def\input.csv |
| abc/def/input.csv | C:\LocalRunDataRoot\abc\def\input.csv |
| D:/abc/def/input.csv | D:\abc\def\input.csv |

### Working directory

When running the U-SQL script locally, a working directory is created during compilation under current running directory. In addition to the compilation outputs, the needed runtime files for local execution will be shadow copied to this working directory. The working directory root folder is called "ScopeWorkDir" and the files under the working directory are as follows:

| DIRECTORY/FILE | DIRECTORY/FILE | DIRECTORY/FILE | DEFINITION | DESCRIPTION |
| --- | --- | --- | --- | --- |
| C6A101DDCB470506 | | | Hash string of runtime version | Shadow copy of runtime files needed for local execution |
| | Script_66AE4909AA0ED06C | | Script name + hash string of script path | Compilation outputs and execution step logging |
| | | _script_.abr | Compiler output | Algebra file |
| | | _ScopeCodeGen_.* | Compiler output | Generated managed code |
| | | _ScopeCodeGenEngine_.* | Compiler output | Generated native code |
| | | referenced assemblies | Assembly reference | Referenced assembly files |
| | | deployed_resources | Resource deployment | Resource deployment files |
| | | xxxxxxxx.xxx[1..n]_*.* | Execution log | Log of execution steps |

## Use the SDK from the command line

### Command-line interface of the helper application

Under SDK directory\build\runtime, LocalRunHelper.exe is the command-line helper application that provides interfaces to most of the commonly used local-run functions. Note that both the command and the argument switches are case-sensitive. To invoke it:

```
LocalRunHelper.exe <command> <Required-Command-Arguments> [Optional-Command-Arguments]
```

Run LocalRunHelper.exe without arguments or with the **help** switch to show the help information:

```
> LocalRunHelper.exe help

    Command 'help' :  Show usage information
    Command 'compile' :  Compile the script
    Required Arguments :
        -Script param
                Script File Path
    Optional Arguments :
        -Shallow [default value 'False']
                Shallow compile
```

In the help information:

- **Command** gives the command's name.
- **Required Argument** lists arguments that must be supplied.
- **Optional Argument** lists arguments that are optional, with default values. Optional Boolean arguments don't have parameters, and their appearances mean negative to their default value.

#### Return value and logging

The helper application returns **0** for success and **-1** for failure. By default, the helper sends all messages to the current console. However, most of the commands support the **-MessageOut path_to_log_file** optional argument that redirects the outputs to a log file.

#### Environment variable configuring

U-SQL local run needs a specified data root as local storage account, as well as a specified CppSDK path for dependencies. You can both set the argument in command-line or set environment variable for them.

- Set the **SCOPE_CPP_SDK** environment variable.

  If you get Microsoft Visual C++ and the Windows SDK by installing Data Lake Tools for Visual Studio, verify that you have the following folder:

  ```
    C:\Program Files (x86)\Microsoft Visual Studio 14.0\Common7\IDE\Extensions\Microsoft\Microsoft Azure
  Data Lake Tools for Visual Studio 2015\X.X.XXXX.X\CppSDK
  ```

  Define a new environment variable called **SCOPE_CPP_SDK** to point to this directory. Or copy the folder to the other location and specify **SCOPE_CPP_SDK** as that.

  In addition to setting the environment variable, you can specify the **-CppSDK** argument when you're using the command line. This argument overwrites your default CppSDK environment variable.

- Set the **LOCALRUN_DATAROOT** environment variable.

  Define a new environment variable called **LOCALRUN_DATAROOT** that points to the data root.

  In addition to setting the environment variable, you can specify the **-DataRoot** argument with the data-root path when you're using a command line. This argument overwrites your default data-root environment variable. You need to add this argument to every command line you're running so that you can overwrite the default data-root environment variable for all operations.

#### SDK command line usage samples

##### Compile and run

The **run** command is used to compile the script and then execute compiled results. Its command-line arguments are a combination of those from **compile** and **execute**.

```
LocalRunHelper run -Script path_to_usql_script.usql [optional_arguments]
```

The following are optional arguments for **run**:

| ARGUMENT | DEFAULT VALUE | DESCRIPTION |
|---|---|---|
| -CodeBehind | False | The script has .cs code behind |
| -CppSDK | | CppSDK Directory |
| -DataRoot | DataRoot environment variable | DataRoot for local run, default to 'LOCALRUN_DATAROOT' environment variable |
| -MessageOut | | Dump messages on console to a file |
| -Parallel | 1 | Run the plan with the specified parallelism |
| -References | | List of paths to extra reference assemblies or data files of code behind, separated by ';' |
| -UdoRedirect | False | Generate Udo assembly redirect config |
| -UseDatabase | master | Database to use for code behind temporary assembly registration |
| -Verbose | False | Show detailed outputs from runtime |
| -WorkDir | Current Directory | Directory for compiler usage and outputs |
| -RunScopeCEP | 0 | ScopeCEP mode to use |
| -ScopeCEPTempPath | temp | Temp path to use for streaming data |
| -OptFlags | | Comma-separated list of optimizer flags |

Here's an example:

```
LocalRunHelper run -Script d:\test\test1.usql -WorkDir d:\test\bin -CodeBehind -References
"d:\asm\ref1.dll;d:\asm\ref2.dll" -UseDatabase testDB –Parallel 5 -Verbose
```

Besides combining **compile** and **execute**, you can compile and execute the compiled executables separately.

#### Compile a U-SQL script

The **compile** command is used to compile a U-SQL script to executables.

```
LocalRunHelper compile -Script path_to_usql_script.usql [optional_arguments]
```

The following are optional arguments for **compile**:

| ARGUMENT | DESCRIPTION |
|---|---|
| -CodeBehind [default value 'False'] | The script has .cs code behind |

| ARGUMENT | DESCRIPTION |
| --- | --- |
| -CppSDK [default value ''] | CppSDK Directory |
| -DataRoot [default value 'DataRoot environment variable'] | DataRoot for local run, default to 'LOCALRUN_DATAROOT' environment variable |
| -MessageOut [default value ''] | Dump messages on console to a file |
| -References [default value ''] | List of paths to extra reference assemblies or data files of code behind, separated by ';' |
| -Shallow [default value 'False'] | Shallow compile |
| -UdoRedirect [default value 'False'] | Generate Udo assembly redirect config |
| -UseDatabase [default value 'master'] | Database to use for code behind temporary assembly registration |
| -WorkDir [default value 'Current Directory'] | Directory for compiler usage and outputs |
| -RunScopeCEP [default value '0'] | ScopeCEP mode to use |
| -ScopeCEPTempPath [default value 'temp'] | Temp path to use for streaming data |
| -OptFlags [default value ''] | Comma-separated list of optimizer flags |

Here are some usage examples.

Compile a U-SQL script:

```
LocalRunHelper compile -Script d:\test\test1.usql
```

Compile a U-SQL script and set the data-root folder. Note that this will overwrite the set environment variable.

```
LocalRunHelper compile -Script d:\test\test1.usql –DataRoot c:\DataRoot
```

Compile a U-SQL script and set a working directory, reference assembly, and database:

```
LocalRunHelper compile -Script d:\test\test1.usql -WorkDir d:\test\bin -References
"d:\asm\ref1.dll;d:\asm\ref2.dll" -UseDatabase testDB
```

#### Execute compiled results

The **execute** command is used to execute compiled results.

```
LocalRunHelper execute -Algebra path_to_compiled_algebra_file [optional_arguments]
```

The following are optional arguments for **execute**:

| ARGUMENT | DEFAULT VALUE | DESCRIPTION |
| --- | --- | --- |

| ARGUMENT | DEFAULT VALUE | DESCRIPTION |
| --- | --- | --- |
| -DataRoot | '' | Data root for metadata execution. It defaults to the **LOCALRUN_DATAROOT** environment variable. |
| -MessageOut | '' | Dump messages on the console to a file. |
| -Parallel | '1' | Indicator to run the generated local-run steps with the specified parallelism level. |
| -Verbose | 'False' | Indicator to show detailed outputs from runtime. |

Here's a usage example:

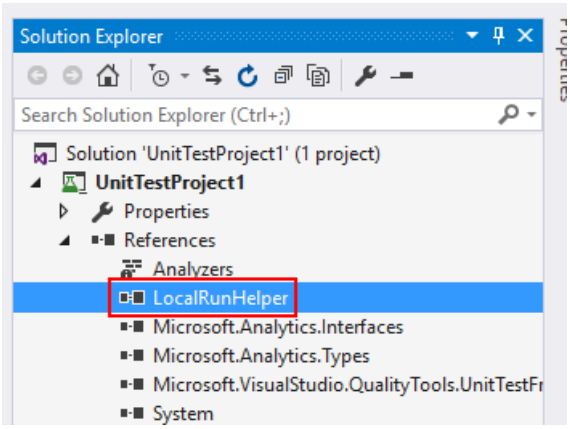
```
LocalRunHelper execute -Algebra d:\test\workdir\C6A101DDCB470506\Script_66AE4909AA0ED06C\__script__.abr –
DataRoot c:\DataRoot –Parallel 5
```

## Use the SDK with programming interfaces

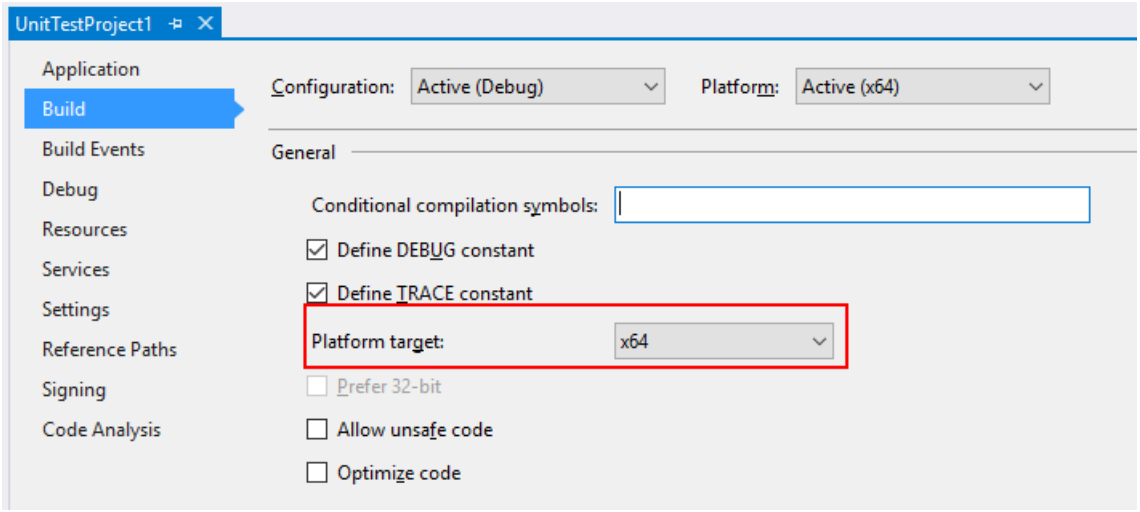
The programming interfaces are all located in the LocalRunHelper.exe. You can use them to integrate the functionality of the U-SQL SDK and the C# test framework to scale your U-SQL script local test. In this article, I will use the standard C# unit test project to show how to use these interfaces to test your U-SQL script.

**Step 1: Create C# unit test project and configuration**

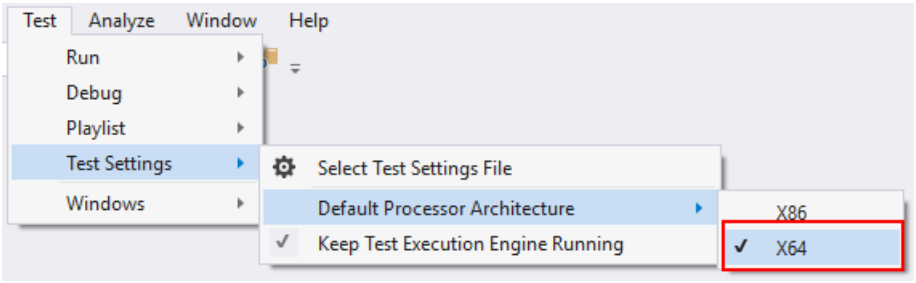
- Create a C# unit test project through File > New > Project > Visual C# > Test > Unit Test Project.
- Add LocalRunHelper.exe as a reference for the project. The LocalRunHelper.exe is located at \build\runtime\LocalRunHelper.exe in Nuget package.




- U-SQL SDK **only** support x64 environment, make sure to set build platform target as x64. You can set that through Project Property > Build > Platform target.




- Make sure to set your test environment as x64. In Visual Studio, you can set it through Test > Test Settings > Default Processor Architecture > x64.




- Make sure to copy all dependency files under NugetPackage\build\runtime\ to project working directory which is usually under ProjectFolder\bin\x64\Debug.

**Step 2: Create U-SQL script test case**

Below is the sample code for U-SQL script test. For testing, you need to prepare scripts, input files and expected output files.

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using System.IO;
using System.Text;
using System.Security.Cryptography;
using Microsoft.Analytics.LocalRun;

namespace UnitTestProject1
{
    [TestClass]
    public class USQLUnitTest
    {
        [TestMethod]
        public void TestUSQLScript()
        {
            //Specify the local run message output path
            StreamWriter MessageOutput = new StreamWriter("../../../log.txt");

            LocalRunHelper localrun = new LocalRunHelper(MessageOutput);

            //Configure the DateRoot path, Script Path and CPPSDK path
            localrun.DataRoot = "../../../";
            localrun.ScriptPath = "../../../Script/Script.usql";
            localrun.CppSdkDir = "../../../CppSDK";

            //Run U-SQL script
            localrun.DoRun();
```

```
            //Script output
            string Result = Path.Combine(localrun.DataRoot, "Output/result.csv");

            //Expected script output
            string ExpectedResult = "../../../ExpectedOutput/result.csv";

            Test.Helpers.FileAssert.AreEqual(Result, ExpectedResult);

            //Don't forget to close MessageOutput to get logs into file
            MessageOutput.Close();
        }
    }
}

namespace Test.Helpers
{
    public static class FileAssert
    {
        static string GetFileHash(string filename)
        {
            Assert.IsTrue(File.Exists(filename));

            using (var hash = new SHA1Managed())
            {
                var clearBytes = File.ReadAllBytes(filename);
                var hashedBytes = hash.ComputeHash(clearBytes);
                return ConvertBytesToHex(hashedBytes);
            }
        }

        static string ConvertBytesToHex(byte[] bytes)
        {
            var sb = new StringBuilder();

            for (var i = 0; i < bytes.Length; i++)
            {
                sb.Append(bytes[i].ToString("x"));
            }
            return sb.ToString();
        }

        public static void AreEqual(string filename1, string filename2)
        {
            string hash1 = GetFileHash(filename1);
            string hash2 = GetFileHash(filename2);

            Assert.AreEqual(hash1, hash2);
        }
    }
}
```

### Programming interfaces in LocalRunHelper.exe

LocalRunHelper.exe provides the programming interfaces for U-SQL local compile, run, etc. The interfaces are listed as follows.

### Constructor

public LocalRunHelper([System.IO.TextWriter messageOutput = null])

| PARAMETER | TYPE | DESCRIPTION |
| --- | --- | --- |
| messageOutput | System.IO.TextWriter | for output messages, set to null to use Console |

### Properties

| PROPERTY | TYPE | DESCRIPTION |
| --- | --- | --- |
| AlgebraPath | string | The path to algebra file (algebra file is one of the compilation results) |
| CodeBehindReferences | string | If the script has additional code behind references, specify the paths separated with ';' |
| CppSdkDir | string | CppSDK directory |
| CurrentDir | string | Current directory |
| DataRoot | string | Data root path |
| DebuggerMailPath | string | The path to debugger mailslot |
| GenerateUdoRedirect | bool | If we want to generate assembly loading redirection override config |
| HasCodeBehind | bool | If the script has code behind |
| InputDir | string | Directory for input data |
| MessagePath | string | Message dump file path |
| OutputDir | string | Directory for output data |
| Parallelism | int | Parallelism to run the algebra |
| ParentPid | int | PID of the parent on which the service monitors to exit, set to 0 or negative to ignore |
| ResultPath | string | Result dump file path |
| RuntimeDir | string | Runtime directory |
| ScriptPath | string | Where to find the script |
| Shallow | bool | Shallow compile or not |
| TempDir | string | Temp directory |
| UseDataBase | string | Specify the database to use for code behind temporary assembly registration, master by default |
| WorkDir | string | Preferred working directory |

### Method

| METHOD | DESCRIPTION | RETURN | PARAMETER |
| --- | --- | --- | --- |
| public bool DoCompile() | Compile the U-SQL script | True on success | |
| public bool DoExec() | Execute the compiled result | True on success | |
| public bool DoRun() | Run the U-SQL script (Compile + Execute) | True on success | |
| public bool IsValidRuntimeDir(string path) | Check if the given path is valid runtime path | True for valid | The path of runtime directory |

## FAQ about common issue

**Error 1:**

E_CSC_SYSTEM_INTERNAL: Internal error! Could not load file or assembly 'ScopeEngineManaged.dll' or one of its dependencies. The specified module could not be found.

Please check the following:

- Make sure you have x64 environment. The build target platform and the test environment should be x64, refer to **Step 1: Create C# unit test project and configuration** above.
- Make sure you have copied all dependency files under NugetPackage\build\runtime\ to project working directory.

## Next steps

- To learn U-SQL, see Get started with Azure Data Lake Analytics U-SQL language.
- To log diagnostics information, see Accessing diagnostics logs for Azure Data Lake Analytics.
- To see a more complex query, see Analyze website logs using Azure Data Lake Analytics.
- To view job details, see Use Job Browser and Job View for Azure Data Lake Analytics jobs.
- To use the vertex execution view, see Use the Vertex Execution View in Data Lake Tools for Visual Studio.