Table of Contents

Introduction to the U-SQL Tutorial (version 1.0)

The U-SQL Tutorial gets developers productive on the U-SQL Language.

- Read it online
- Download the PDF

Looking for the U-SQL Language Reference?

You can find **the language reference here**.

Changes to the U-SQL Language

You can find **U-SQL Release Notes here**

Meta

This section is for content about the tutorial itself. If you want to get right unto U-SQL, you can skip this section completely.

How to Read this document

Set aside a day or two and just start reading each chapter in sequence. Every chapter introduces a simple topic. Don't skip ahead. Each chapter builds on the one before it. When you are done with all the chapters you'll have a good working knowledge of how to use U-SQL.

Send feedback

We'd love to hear it!

Send **feedback and ideas about U-SQL** or Azure Data Lake to our UserVoice forum: https://aka.ms/adlfeedback

If you need to send **suggestions for the U-SQL tutorial document** itself, contact **usqltutorial@microsoft.com**

Contributing

We welcome contributions.

Using GitBooks

If you have an account with GitBooks you can edit the tutorial

https://www.gitbook.com/book/saveenr/usql-tutorial/edit

With Git

Clone the repo

```
https://git.gitbook.com/saveenr/usql-tutorial.git
```

Change log

- 2017/08/28 - Updated SqlMap section
- 2017/08/27 - Added a tip for generating ranges of numbers and dates
- 2017/08/27 - Updated SqlArray section
- 2017/08/26 - Fixed a few typos
- 2017/07/23 - Added section on conditionally counting
- 2017/07/18 - Improved sample for CREATE TABLE AS SELECT
- 2017/07/11 - added DEPLOY RESOURCE section
- 2017/07/02 - added string literals
- 2017/07/02 - clarified aggregates
- 2017/07/01 - started initial complex types chapter
- 2017/07/01 - Added chapter on referencing assemblies
- 2017/06/26 - Moved FileSets to its own chapter
- 2017/06/25 - CSS customizations
- 2017/06/24 - Updated Joins, Window functions, and Set ops
- 2017/06/21 - First draft released

Getting Started

In this chapter you will master the mechanics of running U-SQL on your own box with Visual Studio. You will also get familiar with basic U-SQL concepts.

U-SQL versus SQL

If you come from a SQL background, you'll notice that U-SQL queries look a lot of SQL queries. Many fundamental concepts and syntactic expressions will be very familiar to those with a background in SQL.

However U-SQL is a distinct language and some of the expectations you might have from the SQL world do not carry over into U-SQL.

Running U-SQL scripts

There are two ways to run U-SQL scripts:

- You can run U-SQL scripts on your own machine. The data read and written by this script will be on you own machine. You aren't using Azure resources to do this so there is no additional cost. This method of running U-SQL scripts is called **"U-SQL Local Execution"**
- You can run U-SQL scripts in Azure in the context of a Data Lake Analytics account. The data read or written by the script will also be in Azure - typically in an Azure Data Lake Store account. You pay for any compute and storage used by the script. This is called **"U-SQL Cloud Execution"**

For most of the tutorial we will work with **U-SQL Local Execution** because we want you to learn the language first. Later on we can explore the very rich topic that is **U-SQL Cloud Execution**.

Preparing for the tutorial

Machine

- x64 CPU
- Minimum of 16 GB RAM

Operating System

- Windows 7, 8, or 10 (x64)
- Windows 10 is recommended
- You MUST use an x64 version of Windows

Visual Studio

- You can use Visual Studio 2015 or Visual Studio 2013
- Visual Studio 2015 is recommended
- You can use Visual Studio Community editions
- The tutorial was written using VS 2015 Community Edition
- Support for Visual Studio 2017 is coming soon.

Azure Data Lake Tools for Visual Studio (ADLToolsForVS)

For Visual Studio 2015 and Visual Studio 2013

Download and onstall ADLToolsForVS from **here**.

Verify ADLToolsForVS is Installed

In Visual Studio, go to **Tools menu**. If you see an item in that menu called **Data Lake** then ADLToolsForVS is installed.

Check for Updates

ADLToolsForVS is updated frequently.

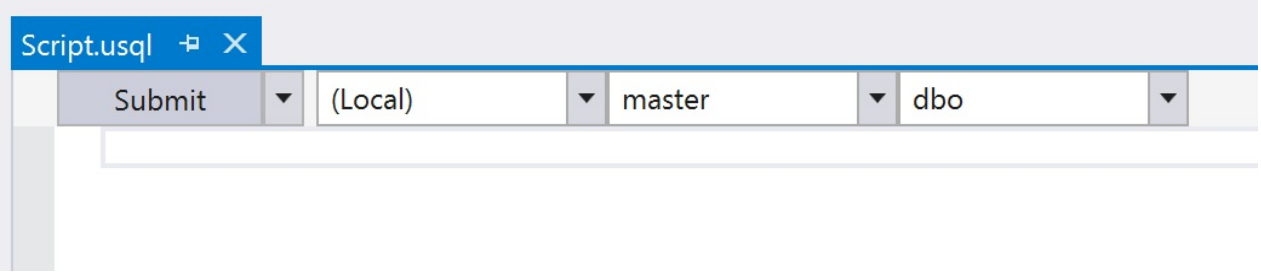
Check for new versions often. In visual studio, go to **Tools > Data Lake > Check for updates**.

Your first U-SQL Script

Launch Visual Studio. Go to **File > New > Project > Installed > Templates > Azure Data Lake > U-SQL Project**.

At this point you should notice that an empty U-SQL script has been created called "Script.usql".

Script Files

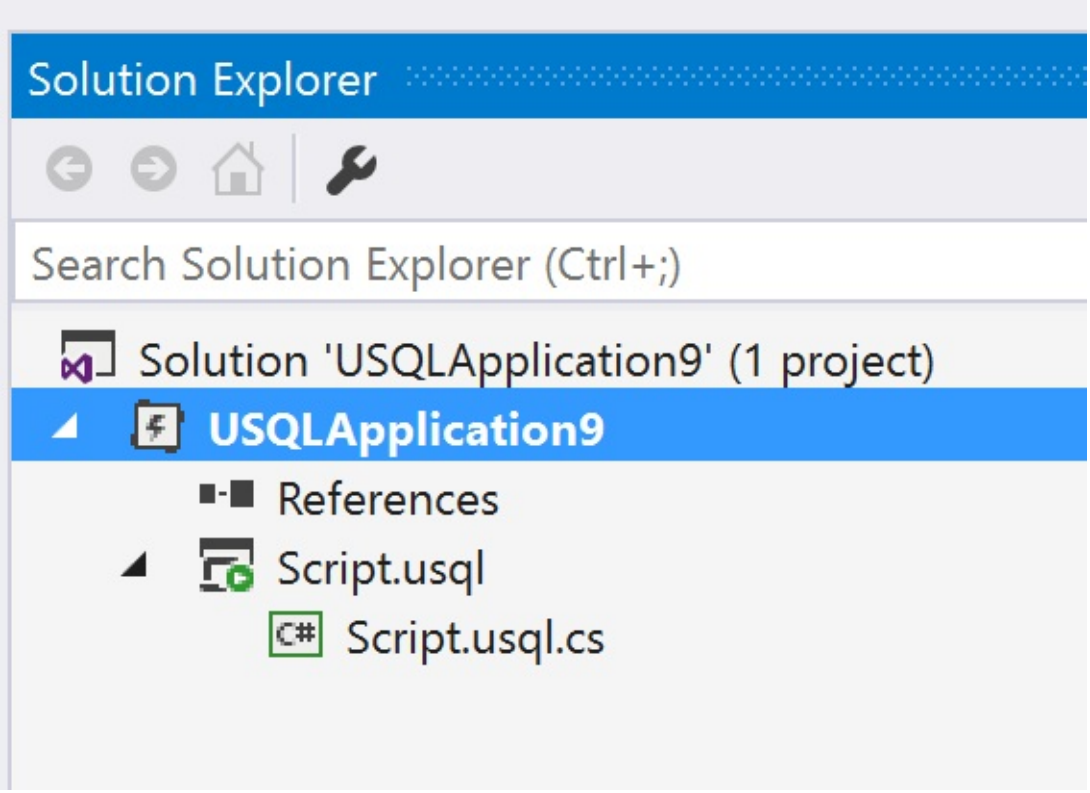


The **Submit** button will run the script. The dropdown next to the **Submit** button says `(Local)`. This means that Submit will run the script in Local Execution mode. In future chapters, we'll pick an Azure Data Lake Analytics account and run the script in Cloud Execution mode - but for now we will continue to locally execute all our U-SQL script.

You can ignore the dropdowns that say `master` and `dbo`. They will not impact the tutorial in any way.

U-SQL Projects and Code-Behind files

Make sure the Solution Explorer is visible. If it isn't go to **View > Solution Explorer**.



The solution contains single U-SQL project called `USQLApplicationX`. Inside that project is the empty `Script.usql` file.

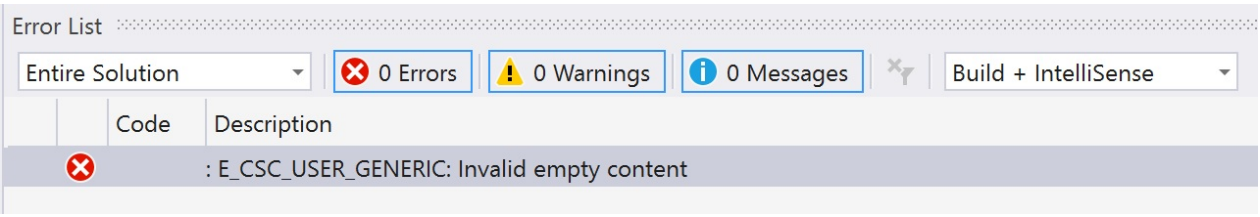
Notice that you can expand the .usql file node. It contains a C# file. This is called the **U-SQL Code-Behind** file. You will find this code-behind file very useful in later chapters.

NOTE: U-SQL Code-behind is a convenience feature of ADLToolsForVS and not a part of the U-SQL language itself.

Your First Compiler Error

Learning a programming language means learning how the compiler tells you something is wrong.

Press **Submit** to run your empty script in Local Execution mode. You should see the Error List window appear and a single error. Unsurprisingly, it complains of an empty script.



It's worth noticing a few things about this error.

- `csc` : this refers to the U-SQL compiler.

- `USER` : this means that the user (you) are the source of the problem. If it says `SYSTEM` it means something is wrong with the U-SQL compiler itself or perhaps the service.

System errors are not very common - but it will be valuable later on to always understand the distinction between **user errors** (also called **user code errors**) and system errors.

Inputs and Outputs

All U-SQL scripts transform inputs to outputs. There are different kinds of inputs and outputs but ultimately they resolve to one of two things:

- Files
- U-SQL tables

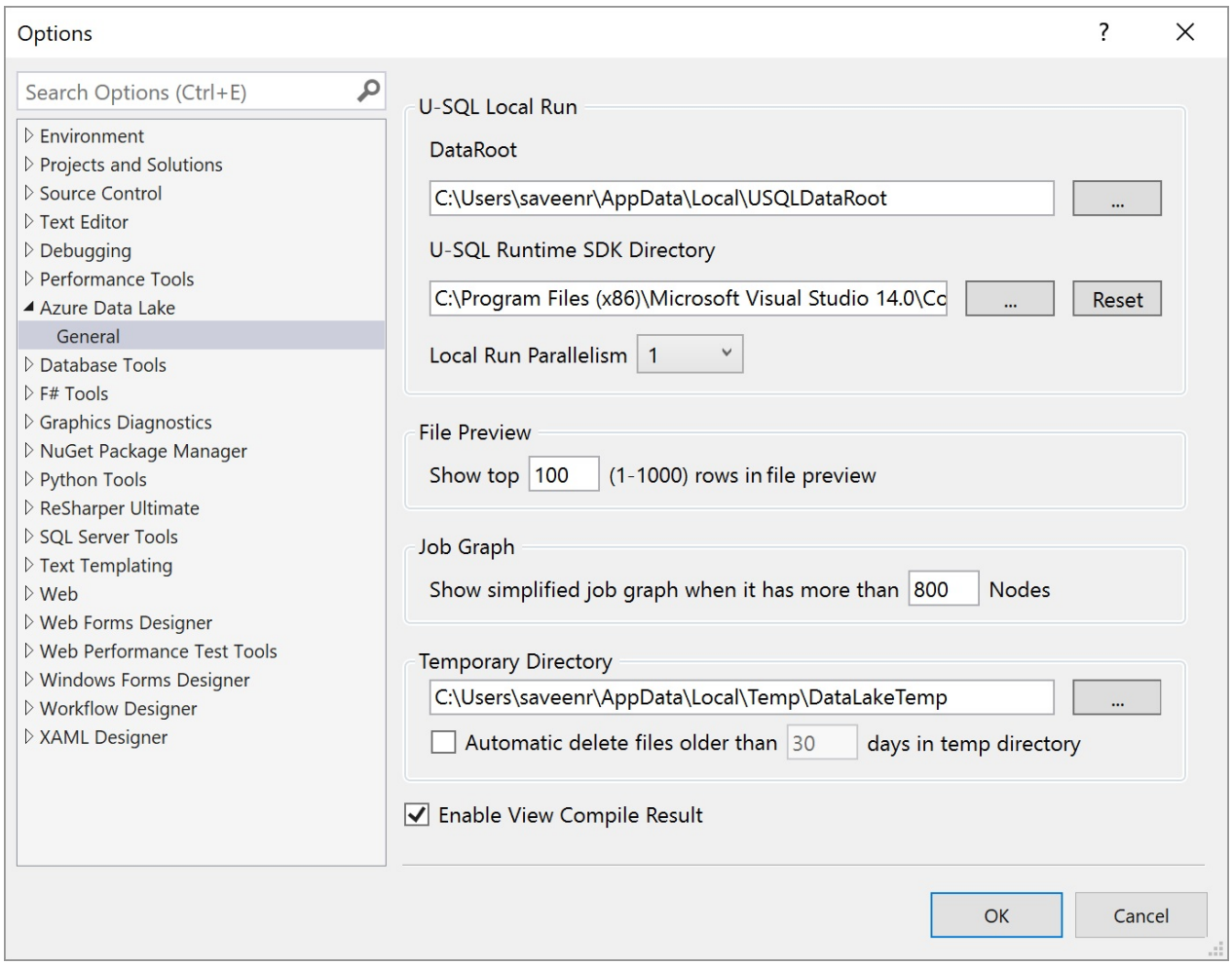
REMEMBER: compiling a U-SQL script requires that all the inputs must exist. We often describe this as "all inputs must be known at compile time". Later chapters will talk about this topic in more detail.

Location of inputs and Outputs

During U-SQL Cloud Execution the inputs/outputs must all be in the cloud - typically this means Azure Data Lake Store.

During U-SQL Local Execution the inputs/outputs must all be on your own box. There's a special name for this location: The U-**SQL Local Data Root**.

You can find the local data root by going to **Tools > Data Lake > Options and Settings**. It in the field called **DataRoot** at the top.



Copy that location and open it in Windows Explorer.

Download the `SearchLog.tsv` file into the Local Data Root from here:

https://raw.githubusercontent.com/Azure/usql/master/Examples/Samples/Data/SearchLog.tsv

Take a look at the TSV file. Some things to point out

- It is a TSV file. TSV = Tab-Separated-Value
- It has no header row
- Each row has the same number of columns

Now paste the following script in to the `Script.usql` window.

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

OUTPUT @searchlog
    TO "/SearchLog_output.tsv"
    USING Outputters.Tsv();
```

This script we've just run is very simple - all it does is copy a file. However, there it introduces many interesting topics we will discuss.

Click **Submit**.

Visual Studio will show the Job Graph window and a Console window will open. The script should run successfully.

After the script completes, look in the Local Run Data Root folder, you should see a file called `SearchLog_output.tsv`.

Congratulations, you've run your first U-SQL script!

Some things to notice

U-SQL keywords are case-sensitive

- The script contains a number of U-SQL keywords: `EXTRACT`, `FROM`, `TO`, `OUTPUT`, `USING`, etc.
- U-SQL keywords are case sensitive. Keep this in mind - it's one of the most common errors people run into.

Reading and Writing Files with EXTRACT and OUTPUT

- The `EXTRACT` statement reads from files. The built-in extractor called `Extractors.Tsv` handles Tab-Separated-Value files.
- The `OUTPUT` statement writes to files. The built-in outputter called `Outputters.Tsv` handles Tab-Separated-Value files.

We'll cover reading and writing to U-SQL tables in later chapters. Later, we'll also learn how to make custom extractors.

Schema for files and Header Rows

- From the U-SQL perspective files are "blobs" - they don't contain any usable schema information. So U-SQL supports a concept called "schema on read" - this means the developer specified the schema that is expected in the file. As you can see the names of the columns and the datatypes are specified in the `EXTRACT` statement.
- The default Extractors and Outputters cannot infer the schema from the header row - in fact by default they assume that there is no header row (this behavior can overriden)

RowSets

- The first statement in the script defines a "RowSet" called `@searchlog`. RowSets are an abstraction that represents how rows flow through a script.
- Because it comes up so often, we should clarify one thing now: RowSets are not tables, or temporary tables, or views, etc. They imply nothing about how data will be persisted.

Common compilation errors

When something is wrong with a U-SQL script it will not compile. Fortunately, of the many causes for compilation failure, there are only a few of them that account for the vast majority of compilation failures.

It will save you time to be familiar with these issues and how they manifest themselves in error messages and in the tools.

Try running each of the sample scripts below.

Incorrect casing on U-SQL identifiers

The script below uses `from` instead of `FROM`.

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

OUTPUT @searchlog
    TO @"/SearchLog_output.tsv"
    USING Outputters.Tsv();
```

Input file does not exist

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

OUTPUT @searchlog
    TO "/SearchLog_output.tsv"
    USING Outputters.Tsv();
```

Invalid C# Expression

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

OUTPUT @searchlog
    TO "/SearchLog_output.tsv"
    USING Outputters.Tsv();
```

The SearchLog sample data

Location

https://raw.githubusercontent.com/Azure/usql/master/Examples/Samples/Data/SearchLog.tsv

Meaning

The SearchLog is a small dataset that represents "sessions" that a user has when performing a search on a web search engine. Each row represents a session for a specific search query,

Schema

The `SearchLog.tsv` file doesn't contain a header row so we'll have to document the columns below in order of their appearance

- **UserId** - this is an integer representing an anonymized user
- **Start** - when started a session with the search engine
- **Region** - What geographical region the user is searching from
- **Query** - What the user searched for
- **Duration** - How long their search session lasted
- **Urls** - A semicolon-separated list All the URLs that were shown to the user in the session
- **ClickedUrls** - A subset of **Urls** that the user actually clicked on (also a semicolon-separated list)

Automatically downloading the searchlog dataset

This PowerShell script will automatically download the file for you:

```
$webclient = New-Object System.Net.WebClient
$mydocs = [environment]::getfolderpath("mydocuments")
$url = "https://raw.githubusercontent.com/Azure/usql/master/Examples/Samples/Data/Sear
chLog.tsv"
$basename = Split-Path $url -Leaf
$output_filename = Join-Path $mydocs $basename
$webclient.DownloadFile($url,$output_filename )
```

Transforming RowSets

In this chapter you'll focus on the fundamental ways in which rowsets can be created and modified.

Creating a RowSet from another RowSet

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

@output =
    SELECT *
    FROM @searchlog;

OUTPUT @output
    TO "/SearchLog_output.tsv"
    USING Outputters.Tsv();
```

Effectively this script just copies the data without transforming it.

However, if you look at the output file you will notice some things about the default behavior of `Outputters.Tsv`:

- values have been surrounded by double-quotes - which is the default behavior of `Outputters.Tsv`. You can disable the quoting by using `Outputters.Tsv( quoting:false )`.
- `DateTime` values are output in a longer standard format
- The default encoding is UTF-8 and the BOM (Byte Order Mark) is not written.

Creating constant RowSets

RowSets can be directly created in a U-SQL script.

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

DepID	DepName
31	Sales
33	Engineering
34	Clerical
35	Marketing

RowSet refinement

Rowsets can be defined from other rowsets. In fact rowsets can be defined from themselves. This is called RowSet refinement and can be a powerful way to make your code easier to read.

```
@output =
    SELECT
        *,
        Urls.Length AS UrlsLength
    FROM @searchlog;

@output =
    SELECT
        Url,
        UrlLength
    FROM @output;
```

Adding new columns with SELECT

We can create new Columns with the SELECT clause. Simply use a C# expression and give it a column name with AS.

IMPORTANT: For the sake of brevity we will start omitting the EXTRACT and OUTPUT statements in the code samples.

```
@output =
    SELECT
        *,
        (Query + Query) AS Query2
    FROM @searchlog;
```

In the example above, we added a column without having to explicitly list out all the other columns. Below you can see how we can add just specific columns.

```
@output =
    SELECT
        Region,
        Query,
        (Query + Query) AS Query2,
        Urls.Length AS UrlsLength
    FROM @searchlog;
```

Filtering records with WHERE

Now let's transform the data by filtering out records with the WHERE clause

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

@output =
    SELECT *
    FROM @searchlog
    WHERE Region == "en-gb";

OUTPUT @output
    TO "/SearchLog_output.tsv"
    USING Outputters.Tsv();
```

The SQL Logical Operators

The **AND**/**OR**/**NOT** operators can be combined with parentheses to create more complex logical expressions

```
@output =
    SELECT Start, Region, Duration
    FROM @searchlog
    WHERE (Duration >= 2*60 AND Duration <= 5*60) OR NOT (Region == "en-gb");
```

The C# Logical Operators

U-SQL also supports the C# logical operators

```
@output =
 SELECT Start, Region, Duration
 FROM @searchlog
 WHERE (Duration >= 2*60 && Duration <= 5*60) || (!(Region == "en-gb"));
```

SQL Logical Operators (AND OR) versus C# Operators (&& ||)

These operators behave the same except for their short-circuiting behavior:

- SQL-style logical operators: These DO NOT short-circuit
- C#-style logical operators: These DO short-circuit

Use the SQL-style logical operators unless you MUST have the short-circuiting behavior. The reasons why this is important are covered in a later chapter that describes the order of evaluation of predicates in expressions.

Find all the sessions occurring before a date

```
@output =
 SELECT Start, Region, Duration
 FROM @searchlog
 WHERE Start <= DateTime.Parse("2012/02/17");
```

Find all the sessions occurring between two dates

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

Filtering on calculated columns

Consider this case where SELECT is used to define a new column called `DurationInMinutes`

```
@output =
  SELECT
      Start,
      Region,
      Duration/60.0 AS DurationInMinutes
  FROM @searchlog;
```

There are a couple of approaches for filtering rows based on the `DurationInMinutes` value

The first option is to use RowSet refinement

```
@output =
  SELECT
    Start,
    Region,
    Duration/60.0 AS DurationInMinutes
  FROM @searchlog;

@output =
  SELECT *
  FROM @output
  WHERE DurationInMinutes>= 20;
```

The second option is to repeat the expression in the WHERE clause

```
@output =
  SELECT
    Start,
    Region,
    Duration/60.0 AS DurationInMinutes
  FROM @searchlog
    WHERE Duration/60.0>= 20;
```

WHERE does not work on calculated columns in the same statement

`WHERE` filters rows coming into to the statement. The DurationInMinutes column doesn't exist in the input. Therefore WHERE cannot operate on it. So, the example below will not compile

```
// SYNTAX ERROR: WHERE cannot be used with columns created by SELECT
@output =
  SELECT Start, Region,Duration/60.0 AS DurationInMinutes
  FROM @searchlog
  WHERE DurationInMinutes>= 20;
```

Numbering rows

Using the `ROW_NUMBER` windowing function aggregate is how to assign row numbers. `ROW_NUMBER` is part of Windowing Functions and that topic too complex for this tutorial. See the Windowing Functions documentation for details. However, for now we do want to show you the proper way to number rows in U-SQL using `ROW_NUMBER` because it is a popular topic.

```
@rs1 =
 SELECT
   ROW_NUMBER() OVER ( ) AS RowNumber,
   Start,
   Region
 FROM @searchlog;
```

RowSets must lead to an output

There's no syntax error here. However, compilation will fail. The reason is that all rowsets must eventually contribute to data being written to a file or table.

The following script fail will to compile the `@smallrows` does not eventually result in an output to a file or table

```
@rows =
 EXTRACT
   Name string,
   Amount int,
   FROM "/input.csv"
 USING Extractors.Csv();

@smallrows =
  SELECT Name, Amount
  FROM @rows
  WHERE Amount < 1000;

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

Escaping Column names

If you need column names that contain whitespace you can enclose the name in `[` and `]`.

The following script shows a column called **Order Number**.

```
@b =
  SELECT
    [Order Number],
    Part
  FROM @a;
```

Parameters

The `DECLARE` statement allows us to define parameters to store values for things that aren't RowSets.

We'll start with this snippet

```
@rows =
  EXTRACT
      name string,
      id int
  FROM "/data.csv"
  USING Extractors.Csv();
```

`DECLARE` can assign constant values to a name. In this case we can assign the input file to a parameter.

```
DECLARE @inputfile string = "/data.csv";

@rows =
  EXTRACT
    name string,
    id   int
  FROM @inputfile
  USING Extractors.Csv();
```

Parameter values cannot be assigned directly from a RowSet.

Values cannot be assigned from a RowSet to a DECLARE parameter.

```
// This does NOT work
DECLARE @maxval int = SELECT MAX(value) FROM data;
```

An alternative is to get a single-row RowSet with a single column and then JOIN that single-row RowSet other RowSet to get what you need.

Supported datatypes

Text

```
DECLARE @text1 string = "Hello World";
DECLARE @text2 string = @"Hello World";

DECLARE @text3 char   = 'a';
```

Datetimes

```
DECLARE @d1 DateTime = System.DateTime.Parse("1979/03/31");
DECLARE @d2 DateTime= DateTime.Now;
```

Signed numerics

```
DECLARE @a sbyte  = 0;
DECLARE @b short  = 1;

DECLARE @c int    = 2;
DECLARE @d long   = 3L;

DECLARE @e float  = 4.0f;
DECLARE @f double = 5.0;
```

Unsigned numerics

```
DECLARE @g byte   = 0;
DECLARE @h ushort = 1;
DECLARE @i uint   = 2;
DECLARE @j ulong  = 3L;
```

.NET arrays

```
DECLARE @array1 byte [] = new byte[] { 0, 1, 2, 3, 4 };
DECLARE @array2 string [] = new string[] { "foo", "bar", "beer" };
```

Complex types

```
DECLARE @m SqlMap<string, string> = new SqlMap<string, string>
    {
        {"This", "is a string in a map"},
        {"That", "is also a string in a map"}
    };
```

Miscellaneous

```
DECLARE @misc1 bool    = true;
DECLARE @misc2 Guid    = System.Guid.Parse("BEF7A4E8-F583-4804-9711-7E608215EBA6");
```

Using Expressions with DECLARE

DECLARE parameters can be expressions

```
DECLARE @a string = "BEGIN" + @text1 + "END";
DECLARE @b string = string.Format("BEGIN{0}END", @text1);
DECLARE @c string = MyHelper.GetMyName();
```

Parameter type inference

DECLARE can infer the type as shown below.

```
DECLARE @a = "Hello World"; // string
DECLARE @b = 'a'; // char
DECLARE @c = 2; // int
DECLARE @d = 2L; // long
DECLARE @d = 4.0f; // float
DECLARE @e = 5.0; // double

DECLARE @m = new SqlMap<string, string>
    {
        {"This", "is a string in a map"},
        {"That", "is also a string in a map"}
    };
```

Native Data types

U-SQL has many built-in Native U-SQL datatypes in that U-SQL has special support for them so that using them will be more performant. In general use the U-SQL's native data types wherever possible.

Most of the datatypes below will be familiar to a C#/.NET programmer. Indeed except for MAP and ARRAY, these *ARE* the normal .NET datatypes you are used to.

Numeric signed

```
sbyte
int
long
float
double
decimal
short
```

Numeric unsigned

```
byte
uint
ulong
ushort
```

Text

```
char
string
```

Complex

```
MAP<k,v>
ARRAY<v>
```

Miscellaneous

```
bool
Guid
DateTime
byte[]
```

Nullable Data Types

In .NET types that cannot be null such as `int` are called **value types**. Types that can have a null value such as string are called **reference** types. Sometimes it is convenient though to value that is an value type but that that can also have a null value. These types are called **nullable types**.

Using nullable types in C# code

```
// This is C# code, not U-SQL
int? i = 100;
i = 200;
i = null;
```

Supported nullable types

U-SQL supports the following nullable types

```
byte?
sbyte?
int?
uint?
long?
ulong?
float?
double?
decimal?
short?
ushort?
bool?
Guid?
DateTime?
char?
```

Nullable types don't apply to reference types such as `string`

The purpose of "nullable" is enable null values for a type that doesn't already support null values. The string type - like all .NET reference types - already supports null values.

U-SQL Expressions

Overview

Clauses such as `SELECT`, `WHERE`, and `HAVING` (among others) allow you to enter U-SQL expressions.

An expression in a programming language is a combination of explicit values, constants, variables, operators, and functions that are interpreted according to the particular rules of precedence and of association for a particular programming language, which computes and then produces another value.

The simplest way of thinking of a U-SQL expression is that it is a merely C# expression with some U-SQL extensions such as the `AND`, `OR`, `NOT` operators.

Casting types

Expressions can also be converted to a different type

```
@output=
    SELECT
        Start,
        Region,
        ((double) Duration) AS DurationDouble
    FROM @searchlog;
```

Calling methods .NET Types

RowSet columns are strongly typed. U-SQL allows you to call methods defined on those types in the SELECT clause. Because U-SQL supports .NET types such as string and DateTime, you can use all the methods available on those types.

```
// Find what day of year each session took place

@output=
    SELECT
        Start,
        Region,
        Start.DayOfYear AS StartDayOfYear
    FROM @searchlog;
```

Order of Evaluation in Expressions

There's a common pattern C# developers are used to, as shown below:

```
if ( (QueryString!=null) && (QueryString.StartsWith("bing") )
{
    // do something
}
```

This pattern depends on a C# behavior (common to many languages) called "short-circuiting." Simply put, in the above example, when the code runs there's no logical reason to check both conditions if the first one returns false. Short circuiting is useful because evaluating each condition may be expensive. Thus, it is a technique that compilers use to improve the performance of C# code.

When trying to do the same thing in U-SQL there are two paths you can pick. Both are valid expressions, but one will cause problems that may not be obvious at first.

The Right Choice: Use && to keep the desired short-circuiting behavior

```
@rs1 =
    SELECT *
    FROM @data
    WHERE ((Name!=null) && (Name.StartsWith("bing"));
```

The Wrong Choice: Use AND which does NOT match the short-circuiting behavior.

```
@rs1 =
    SELECT *
    FROM @data
    WHERE ((Name!=null) AND (Name.StartsWith("bing"));
```

The second translation that uses AND will sometimes fail saying that a NullReferenceException has occurred. (sometimes = it might work on your local box but might fail in the cluster)

The reason is simple and by-design: with AND/OR U-SQL will try to perform certain optimizations that result in better performance - for example it may evaluate the second part of the expression first because it assumes that there is no relationship between the two conditions.

This is a standard optimization technique and the same thing is done in many systems such as SQL. The gain this optimization provides in performance is well worth the occasional confusion it causes for new U-SQL users - so this behavior will never change.
Summary: if you need this short-circuiting behavior use && and `||`.
As an alternative you can use the SQL-like ALL/ANY operators which are equivalent to && and ||.

You CANNOT circumvent the order of evaluation by using multiple U-SQL statements

Of course, then you'll be tempted to write your script by splitting apart the expression as shown below.

```
@rs1 =
    SELECT *
    FROM @data
    WHERE Name != null;

@rs2 =
    SELECT *
    FROM @rs1
    WHERE Name.StartsWith("bing");
```

The assumption here is that the first statement executes, before the second. This assumption is wrong.

This won't work either. U-SQL is declarative language not an imperative one. Just because `@rs1` is defined earlier than `@rs2` in the script above it does NOT imply that the WHERE condition in `@rs1` is evaluated before the WHERE in `@rs2`. U-SQL reserves the right to combine multiple statements together and perform optimizations. You MUST use the && operator if you want to perform short-circuiting.

Useful expressions

If you are a C# developer, the expressions below will seem obvious. If you have a SQL background or come from another language, please read this section. It will save you a lot of time later.

if null, pick a default value, otherwise use the value

```
stringcol1 ?? "defval"
```

The `??` is the C# null coalescing operator.

if an expression is true, pick one value, otherwise pick a different one

```
<expr> ? "was_true" : "was_false"
```

if an expression is true, pick one value, otherwise pick a different one

```
<expr> ? "was_true" : "was_false"
```

Test if null

```
<expr> == null : "was_false"
```

String equality

Case-sensitive equality

```
"ABC" == "ABC"
stringcol1 == "ABC"
```

Case-insensitive equality

```
string.Equals(
    "Abc",
    "ABC",
System.Text.StringComparison.CurrentCultureIgnoreCase )
```

String starts/ends with text

```
stringcol1.StartsWith( "prefix" )
stringcol1.EndsWith( "prefix" )
```

String contains text

```
stringcol1.Contains("prefix" )
```

Substrings

Get every character after the second one

```
stringcol1.Substring(2)
```

Get a string of 5 characters starting after the second character in the original string

```
stringcol1.Substring(2, 5)
```

String length

```
stringcol.Length
```

Merging strings

```
stringcol1 + "_foo_" + stringcol2
```

Or

```
string.Format( "{0}_foo_{1}" , stringcol1, stringcol2 );
```

Remove whitespace

From the beginning and the end of the sting

```
stringcol1.Trim()
```

From the beginning

```
stringcol1.TrimStart()
```

From the end

```
stringcol1.TrimEnd()
```

Change case

```
stringcol1.ToLower()
stringcol1.ToUpper()
```

Tips for SQL developers

Lots of people come to U-SQL from SQL and ask how U-SQL accomplishes things they are familiar with in SQL. A great example in creating an uppercase string.

Given, what we' ve covered so far A SQL developer will expect to write the following in U-SQL

```
@output =
    SELECT
        UPPER( Region ) AS NewRegion
    FROM @searchlog;
```

Unfortunately, said developer will be disappointed to find out that U-SQL has no `UPPER()` method. The C# developer, on the other hand, knows what to do: use the string type's intrinsic `ToUpper()` method.

```
@output =
    SELECT
        Region.ToUpper() AS NewRegion
    FROM @searchlog;
```

Reading and Writing Files

Built-in Extractors

U-SQL has three built-in extractors that handle text

- `Extractors.Csv()` reads comma-separated value (CSV)
- `Extractors.Tsv()` reads tab-separated value (TSV)
- `Extractors.Text()` reads delimited text files.

`Extractors.Csv` and `Extractors.Tsv` are the same as `Extractors.Text` but they default to a specific delimiter appropriate for the format they support.

Text Encoding

By default all the extractors default to UTF-8 as the encoding.

All three built-in extractors allow you to control the encoding through the **encoding** parameter as shown below.

```
@searchlog =
    EXTRACT UserId          int,
            Start           DateTime,
            Region          string,
            Query           string,
            Duration        int,
            Urls            string,
            ClickedUrls     string
    FROM "/SearchLog.tsv"
    USING Extractors.Tsv( encoding: Encoding.[ASCII] );
```

These are the supported text encodings:

```
Encoding.[ASCII]
Encoding.BigEndianUnicode
Encoding.Unicode
Encoding.UTF7
Encoding.UTF8
Encoding.UTF32
```

FileSets

We've seen that you can explicitly list all the files in the EXTRACT statement. In some cases, there might be a large number of files, so you may not want to list all the files manually every time.

FileSets make it easy to define a pattern to identify a set of files to read.

In the simplest case let's get all the files in a folder.

```
@rs =
    EXTRACT
        user   string,
        id     string,
    FROM
        "/input/{*}"
    USING Extractors.Csv();
```

Specifying a list of files that have an extension

This is very simple modification syntax. The example uses will extract all the files that end with ".csv".

```
@rs =
    EXTRACT
        user   string,
        id     string,
    FROM
        "/input/{*}.csv"
    USING Extractors.Csv();
```

Getting filenames as a column in the RowSet

Because we are reading rows from multiple files. it is convenient to for the rows to have some information about the filename it came from. We can adjust the query slightly to make this possible.

```
@rs =
    EXTRACT
        user       string,
        id         string,
        __filename string
    FROM
        "/input/{__filename}"
    USING Extractors.Csv();
```

You are probably wondering about the `__` in the column `__filename`. It isn't necessary at all, however it is useful as a way of marking that this information came from the process of extracting the file, not from the data in the file itself.

To emphasize that the naming of `__filename` and the use of the `__` prefix was completely arbitrary below is the same script with a different name (`foo`).

```
@rs =
    EXTRACT
        user string,
        id   string,
        foo  string
    FROM
        "/input/{foo}"
    USING Extractors.Csv();
```

Getting parts of a filename as a column in the RowSet

Instead of the full filename, we can also get part of the filename. The sample below shows how to get just the number part.

```
@rs =
    EXTRACT
        user       string,
        id         string,
        __filenum  int
    FROM
        "/input/data{__filenum}.csv"
    USING Extractors.Csv();
```

Notes

- The schemas for all the files in the FileSet must match the schema specified in the extract.
- The more files there are in the FileSet the longer the compilation time will take.

Using WHERE to filter the files

FileSets also let us filter the inputs so that the EXTRACT only chooses some of those files.

For example, imagine we have 10 files as shown below.

```
/input/data1.csv
/input/data2.csv
/input/data3.csv
/input/data4.csv
/input/data5.csv
/input/data6.csv
/input/data7.csv
/input/data8.csv
/input/data9.csv
/input/data10.csv
```

The following EXTRACT will read all 10 files

```
@rs =
    EXTRACT
        user       string,
        id         string,
        __filenum  int,
    FROM
        "/input/data{__filenum}.csv"
    USING Extractors.Csv();
```

However by adding a WHERE clause that uses `__filenum` only those files that are matched by the WHERE clause are read. Only 3 files are read (7,8,9).

```
@rs =
    EXTRACT
        user       string,
        id         string,
        __filenum  int,
    FROM
        "/input/data{__filenum}.csv"
    USING Extractors.Csv();

@rs =
    SELECT *
    FROM @rs
    WHERE
        ( __filenum >= 7  ) AND
        ( __filenum <= 9 );
```

FileSets with dates

File names or paths often include information about dates and times. This is often the case for log files. FileSets make it easy to handle these cases.

Image we have files that are named in this pattern `"data-YEAR-MONTH-DAY.csv"`. The following query reads all the files where with that pattern.

```
@rs =
  EXTRACT
      user    string,
      id      string,
      __date  DateTime
  FROM
    "/input/data-{__date:yyyy}-{__date:MM}-{__date:dd}.csv"
  USING Extractors.Csv();
```

Many times you'll have to restrict the files to a specific time range. This can be done by refining the RowSet with a WHERE clause.

```
@rs =
  EXTRACT
      user    string,
      id      string,
      __date  DateTime
  FROM
    "/input/data-{__date:yyyy}-{__date:MM}-{__date:dd}.csv"
  USING Extractors.Csv();

@rs =
  SELECT *
  FROM @rs
  WHERE
    date >= System.DateTime.Parse("2016/1/1") AND
    date < System.DateTime.Parse("2016/2/1");
```

In the above example we used all three parts of the date in the file path. However, you can use any parts you need and don't need to use them all. The following example shows a file path that only uses the year and month.

```
@rs =
    EXTRACT
      user    string,
      id      string,
      __date  DateTime
  FROM
    "/input/data-{__date:yyyy}-{__date:MM}.csv"
  USING Extractors.Csv();

@rs =
  SELECT *
  FROM @rs
  WHERE
    date >= System.DateTime.Parse("2016/1/1") AND
    date < System.DateTime.Parse("2016/2/1");
```

Grouping and Aggregation

Grouping, in essence, collapses multiple rows into single rows based on some criteria. Hand-in-hand with performing a grouping operation, some fields in the output rowset must be aggregated into some meaningful value (or discarded if no possible or meaningful aggregation can be done).

We can witness this behavior by building up to it in stages.

```
// list all session durations.
@output =
  SELECT Duration
  FROM @searchlog;
```

This creates a simple list of integers.

Duration
73
614
74
24
1213
241
502
60
1270
610
422
283
305
10
612
1220
691
63
30
119
732
183
630

Now, let's add all the numbers together. This yields a RowSet with exactly one row and one column.

```
// Find the total duration for all sessions combined
@output =
  SELECT
    SUM(Duration) AS TotalDuration
  FROM @searchlog;
```

Duration
9981

Now let's use the **GROUP BY** operator to break apart the totals by `Region`.

```
// find the total Duration by Region
@output =
  SELECT
    Region,
    SUM(Duration) AS TotalDuration
  FROM searchlog
  GROUP BY Region;
```

This returns:

en_ca	24
en_ch	10
en_fr	241
en_gb	688
en_gr	305
en_mx	422
en_us	8291

This is a good opportunity to explore a common use of the **HAVING** operator. We can use **HAVING** to restrict the output RowSet to those rows that have aggregate values we are interested in. For example, perhaps we want to find all the Regions where total dwell time is above some value.

```
// find all the Regions where the total dwell time is &gt; 200
@output =
  SELECT
    Region,
    SUM(Duration) AS TotalDuration
  FROM @searchlog
  GROUP BY Region
  HAVING TotalDuration > 200;
```

en-fr	241
en-gb	688
en-gr	305
en-mx	422
en-us	8291

```
// Count the number of total sessions.
@output =
  SELECT
    COUNT() AS NumSessions
  FROM @searchlog;
```

23

Count the number of total sessions by Region.

```
@output =
  SELECT
    COUNT() AS NumSessions,
    Region
  FROM @searchlog
  GROUP BY Region;
```

1	en_ca
1	en_ch
1	en_fr
2	en_gb
1	en_gr
1	en_mx
16	en_us

Count the number of total sessions by Region and include total duration for that language.

```
@output =
  SELECT
    COUNT() AS NumSessions,
    Region,
    SUM(Duration) AS TotalDuration,
    AVG(Duration) AS AvgDwellTtime,
    MAX(Duration) AS MaxDuration,
    MIN(Duration) AS MinDuration
    FROM @searchlog
  GROUP BY Region;
```

NumSessions	Region	TotalDuration	AvgDuration	MaxDuration	MinDu
1	en_ca	24	24	24	24
1	en_ch	10	10	10	10
1	en_fr	241	241	241	241
2	en_gb	688	344	614	74
1	en_gr	305	305	305	305
1	en_mx	422	422	422	422
16	en_us	8291	518.1875	1270	30

Data types coming from aggregate functions

You should be aware of how some aggregation operators deal with data types. Some aggregations will promote a numeric type to a "larger" type.

For example:

- `SUM(floatexpr)` -> double
- `SUM(doubleexpr)` -> double
- `SUM(intexpr)` -> long
- `SUM(byteexpr)` -> long

DISTINCT with Aggregates

Every aggregate function can take a DISTINCT qualifier.

For example

`COUNT(DISTINCT x)`

Notes

Aggregates can ONLY appear in a SELECT clause.

Conditionally counting

Consider the following RowSet:

```
@a  =
    SELECT * FROM
        (VALUES
            ("Contoso",    1500.0),
            ("Woodgrove",  2700.0),
            ((string)null, 6700.0)
        ) AS D( customer, amount );
```

We can easily count the number of rows:

```
@b =
    SELECT
       COUNT() AS Count1
       FROM @a;
```

But what if we wanted to count the rows only if they met a certain criteria? We can accomplish this my using the SUM operator with an expression that will return either a 1 or 0.

```
@b =
    SELECT
       COUNT() AS Count,
       SUM(customer.Contains("o") ? 1 : 0) AS CountContainsLowercaseO,
       SUM(customer==null ? 1: 0) AS CountIsNull
    FROM @a;
```

Count	CountContainsLowercaseOnt2	CountIsNull
3	2	1

Filtering Aggregated Rows

We'll start again with a simple GROUP BY

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

Region	TotalDuration
en_ca	24
en_ch	10
en_fr	241
en_gb	688
en_gr	305
en_mx	422
en_us	8291

Filtering with WHERE

You might try `WHERE` here.

```
@output =
  SELECT
    Region,
    SUM(Duration) AS TotalDuration
  FROM @searchlog
  WHERE TotalDuration > 200
  GROUP BY Region;
```

Which will cause an error. Because `WHERE` can only work on the input columns to the statement, not the output columns

We could use multiple U-SQL Statements to accomplish this

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

@output =
  SELECT *
  FROM @output
  WHERE TotalDuration > 200;
```

Filtering with HAVING

Alternatively , we can use the HAVING clause which is designed to filter columns when a GROUP BY is used..

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

You may have noticed that `SUM(Duration)` was repeated in the HAVING clause. That's because HAVING (like WHERE) cannot use columns created in the SELECT clause.

Aggregate functions

U-SQL contains several common aggregation functions:

- AVG
- COUNT
- ANY_VALUE
- FIRST_VALUE
- LAST_VALUE
- MAX
- MIN
- SUM
- VAR
- STDEV
- ARRAY_AGG
- MAP_AGG

Basic Statistics with MAX, MIN, AVG, STDEV, & SUM

These do what you expect them to do

```
@output =
    SELECT
        MAX(Duration) AS DurationMax,
        MIN(Duration) AS DurationMin,
        AVG(Duration) AS DurationAvg,
        SUM(Duration) AS DurationSum,
        VAR(Duration) AS DurationVariance,
        STDEV(Duration) AS DurationStDev,
        VARP(Duration) AS DurationVarianceP,
        STDEVP(Duration) AS DurationStDevP
    FROM @searchlog
    GROUP BY Region;
```

Notes for Statisticians

VAR & **STDEV** are the **sample version** with Bessel's correction **VARP** & **STDEVP** are the better-known **population version**.

ANY_VALUE

ANY_VALUE gets a value for that column with no implications about the where inside that rowset the value came from. It could be the first value, the last value, are on value in between. It is useful because in some scenarios (for example when using Window Functions) where you don't care which value you receive as long as you get one.

```
@output =
  SELECT
    ANY_VALUE(Start) AS FirstStart,
    Region
  FROM @searchlog
  GROUP BY Region;
```

CROSS APPLY and ARRAY_AGG

CROSS APPLY and **ARRAY_AGG** support two very common scenarios in transforming text

- CROSS APPLY can break a row apart into multiple rows
- ARRAY_AGG joins rows into a single row

What we will how is how to move back and forth between the following two RowSets

The first RowSet has a column called Urls we want to split

Region	Urls
en-us	A;B;C
en-gb	D;E;F

The second RowSet has a column called `Url` we want to merge

Region	Url
en-us	A
en-us	B
en-us	C
en-gb	D
en-gb	E
en-gb	F

Breaking apart rows with CROSS APPLY

Let's examine the searchlog again and extract the `Region` and `Urls` columns.

```
@a =
  SELECT
    Region,
    Urls
  FROM @searchlog;
```

@a looks like this:

Region	Urls
en-us	A;B;C
en-gb	D;E;F

The **Urls** column contains strings, but each string is a semicolon-separated list of URLs. We will eventually use CROSS APPLY to break that column apart. But first we must transform the string into an array that CROSS APPLY can work with.

```
@b =
  SELECT
    Region,
    SqlArray.Create(Urls.Split(';')) AS UrlTokens
  FROM @a;
```

@b looks like this

Region	Urls
en-us	SqlArray{"A","B","C"}
en-gb	SqlArray{"D","E","F"}

Now we can use CROSS APPLY to break the rows apart.

```
@c =
  SELECT
    Region,
    Token AS Url
  FROM @b
   CROSS APPLY EXPLODE (UrlTokens) AS r(Token);
```

@c looks like this

Region	Url
en-us	A
en-us	B
en-us	C
en-gb	D
en-gb	E
en-gb	F

Merging rows with ARRAY_AGG

Now, let's reverse the scenario and merge the Url column for each region with ARRAY_AGG

First, we'll merge the Urls together into an array with ARRAY_AGG

```
@d =
  SELECT
    Region,
    ARRAY_AGG<string>(Url) AS UrlsArray
  FROM @c
  GROUP BY Region;
```

@d looks like this

Region	UrlsArray
en-us	SqlArray{"A","B","C"}
en-gb	SqlArray{"D","E","F"}

Now that we have arrays of strings, we will collapse the each array into a string using `string.Join`.

```
@e =
  SELECT
    Region,
    string.Join(";", UrlsArray) AS Urls
  FROM @d;
```

Finally @e looks like this. We are back to where we started.

Region	Urls
en-us	A;B;C
en-gb	D;E;F

The U-SQL Catalog

The U-SQL Catalog is the way U-SQL organizes data and code for re-use and sharing.

Catalog organization

```
ADLA Account
|
|--Catalog
   |
   |--Databases
      |
      |--Schemas
      |  |--Tables
      |  |--Views
      |  |--Table-valued functions
      |  |--Procedures
      |  |--Assemblies
      |  |--Credentials
      |  |--External Data Sources
      |
      |--Assemblies
```

- Every ADLA account has a single U-SQL catalog. The catalog cannot be deleted.
- Each U-SQL catalog contains one or more U-SQL databases.
- Every catalog has a `master` database that cannot be deleted.
- Each U-SQL database can contain code and data.
- Data is stored in the form of U-SQL tables.
- U-SQL code is stored in a database in the form of views, table-valued functions, procedures.
- .NET code (.NET assemblies) is stored in the database in the form of U-SQL assemblies.

Creating a database

To create a database is simple.

```
CREATE DATABASE MyDB;
```

This command will fail if the database already exists. Often, it will be the case that you want to create a database only if it does not exist. In this case the following command is used.

```
CREATE DATABASE IF NOT EXISTS MyDB;
```

Deleting a Database

```
DROP DATABASE MyDB;
```

DROP DATABASE IF EXISTS MyDB;

Reusing U-SQL code with a table-valued function (TVF)

Many of the scripts in the tutorial have required reading from the searchlog and the code to read from the searchlog is shown below

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

Now instead of writing this code over and over in every script we will store the code as a TVF in a database. The name of this function is going to be `MyDB.dbo.ExtractSearchLog`. The `dbo` part of that name is the "schema".

```
CREATE FUNCTION MyDB.dbo.ExtractSearchLog()
RETURNS @rows
AS BEGIN
  @rows =
    EXTRACT UserId          int,
            Start           DateTime,
            Region          string,
            Query           string,
            Duration        int,
            Urls            string,
            ClickedUrls     string
    FROM "/SearchLog.tsv"
    USING Extractors.Tsv();
    RETURN;
END;
```

Notice that CREATE FUNCTION indicates it will return a RowSet call @rows. Then in the @rows RowSet is defined.

Now that the TVF is created, we can call it this way.

```
@searchlog = MyDB.dbo.ExtractSearchLog();
```

U-SQL Tables

U-SQL tables offer a way to store data in form that preserves it schema and organize data to high-performance queries which is very important as data sizes grow.

Creating a table from a RowSet

If you have a RowSet creating a table from it is very simple. The one thing you must remember is that a table must have an index defined. The table gets its schema from the schema of the rowset.

```
@customers  =
  SELECT * FROM
    (VALUES
       ("Contoso",   123 ),
       ("Woodgrove", 456 )
    ) AS D( Customer, Id  );

DROP TABLE IF EXISTS MyDB.dbo.Customers;

CREATE TABLE MyDB.dbo.Customers
(
    INDEX idx
    CLUSTERED(customer ASC)
    DISTRIBUTED BY HASH(customer)
) AS SELECT * FROM @customers;
```

Creating an empty table and filling it latter

If you need don't have the data available at the time of table creation. You can create an empty table as shown below. Notice that this time the schema has to be specified.

```
DROP TABLE IF EXISTS MyDB.dbo.Customers;

CREATE TABLE MyDB.dbo.Customers
(
    Customer string,
    Id int,
    INDEX idx
        CLUSTERED(Customer ASC)
        DISTRIBUTED BY HASH(Customer)
);
```

Then separately, you can fill the table.

```
@customers  =
  SELECT * FROM
    (VALUES
       ("Contoso",   123 ),
       ("Woodgrove", 456 )
    ) AS D( Customer, Id  );

INSERT INTO MyDB.dbo.Customers
    SELECT * FROM @customers;
```

Reading from a table

```
@rs =
  SELECT *
  FROM MyDB.dbo.Customers;
```

Assemblies

Step 1: create a .NET assembly

Create a .NET assembly with a filename of `OrdersLib.dll` with this code. In this case we will simply crate a single static string in the assembly that we will reuse in a U-SQL script.

```
namespace OrdersLib
{
  public static class Helpers
  {
    public static string CustPrefix = "CUST_";
  }
}
```

Step 2 Upload the assembly into the default ADLS store of the ADLA account

For example place the assembly in this location

```
"/DLLs/OrdersLib.dll"
```

Step 3: Register the assembly in a U-SQL database

For example place the assembly in this location

```
CREATE ASSEMBLY MyDB.OrdersLibAsm
  FROM @"/DLLs/OrdersLib.dll";
```

The identifier `OrdersLibAsm` is the name the U-SQL catalog uses for the assembly. It is independent of the assembly filename `"OrdersLib.dll"` .

The dll has been copied to the U-SQL database called MyDB. You can now safely delete the file from `"/DLLs/OrdersLib.dll"` .

Step 4: Run a script that uses the assembly

The `REFERENCE ASSEMBLY` statement makes the code from the assembly named `OrdersLibAsm` to the script.

```
REFERENCE ASSEMBLY MyDB.OrdersLibAsm;

@customers  =
  SELECT * FROM
    (VALUES
       ("Contoso",   123 ),
       ("Woodgrove", 456 )
    ) AS D( Customer, Id  );

@customers =
  SELECT
    (OrdersLib.Helpers.CustPrefix + Customer) AS Customer,
    Id
  FROM @customers;
```

Packages

This documentation will arrive on 2017/07/01

Using assemblies

A U-SQL Assembly let's you share code.

Although a U-SQL assembly is often used to share .NET code - specifically a single .NET assembly. As you'll see in this section. A U-SQL assembly can do much more than that.

System assemblies

Some assemblies are part of the .NET Base Class Library. They aren't in the U-SQL catalog, but we still need a way to make them available to a U-SQL script.

```
REFERENCE SYSTEM ASSEMBLY [System.Xml];
```

Notice that no database name is provided.

User-provided assemblies

Referencing assemblies in the U-SQL Catalog

```
CREATE ASSEMBLY MyDB.OrdersLibAsm
  FROM @"/DLLs/OrdersLib.dll";
```

Below is the standard syntax for referencing an assembly that is in the U-SQL catalog.

```
REFERENCE SYSTEM ASSEMBLY [DBName].[AssemblyName];
```

Simplifying code with the USING statement

Similar, to C#'s using statement, U-SQL lets you provide an alias for a namespace, this improve the readability of your code.

```
USING Exc = System.Exceptions;
```

Referencing assemblies in the U-SQL Catalog of other ADLA accounts

If the assembly you want is in another ADLA account, and you have access to that assembly you can reference it as shown below.

```
REFERENCE SYSTEM ASSEMBLY [Account][DBName].[AssemblyName];
```

Window Functions

Window functions were introduced to the ISO/ANSI SQL Standard in 2003. U-SQL adopts a subset of window functions as defined by the ANSI SQL Standard.

Window functions are used to do computation within sets of rows called windows. Windows are defined by the OVER clause. Window functions solve some key scenarios in a highly efficient manner.

This tutorial uses two sample datasets to walk you through some sample scenario where you can apply window functions.

The window functions are categorized into:

- Reporting aggregation functions, such as SUM or AVG
- Ranking functions, such as DENSE_RANK, ROW_NUMBER, NTILE, and RANK
- Analytic functions, such as cumulative distribution, percentiles, or accesses data from a previous row in the same result set without the use of a self-join

Sample data

QueryLog sample dataset

QueryLog represents a list of what people searched for in search engine. Each query log includes:

- Query - What the user was searching for.
- Latency - How fast the query came back to the user in milliseconds.
- Vertical - What kind of content the user was interested in (Web links, Images, Videos). Copy and paste the following script into your U-SQL project for constructing the QueryLog rowset:

```
@querylog =
    SELECT * FROM ( VALUES
        ("Banana"  , 300, "Image" ),
        ("Cherry"  , 300, "Image" ),
        ("Durian"  , 500, "Image" ),
        ("Apple"   , 100, "Web"   ),
        ("Fig"     , 200, "Web"   ),
        ("Papaya"  , 200, "Web"   ),
        ("Avocado" , 300, "Web"   ),
        ("Cherry"  , 400, "Web"   ),
        ("Durian"  , 500, "Web"   ) )
    AS T(Query,Latency,Vertical);
```

Employees sample dataset

The Employee dataset includes the following fields:

- EmpID - Employee ID
- EmpName Employee name
- DeptName - Department name
- DeptID - Deparment ID
- Salary - Employee salary

Copy and paste the following script into your U-SQL project for constructing the Employees rowset:

```
@employees =
    SELECT * FROM ( VALUES
        (1, "Noah",   "Engineering", 100, 10000),
        (2, "Sophia", "Engineering", 100, 20000),
        (3, "Liam",   "Engineering", 100, 30000),
        (4, "Emma",   "HR",          200, 10000),
        (5, "Jacob",  "HR",          200, 10000),
        (6, "Olivia", "HR",          200, 10000),
        (7, "Mason",  "Executive",   300, 50000),
        (8, "Ava",    "Marketing",   400, 15000),
        (9, "Ethan",  "Marketing",   400, 10000) )
    AS T(EmpID, EmpName, DeptName, DeptID, Salary);
```

Compare window functions to Grouping

Windowing and Grouping are conceptually related by also different. It is helpful to understand this relationship.

Use aggregation and Grouping

The following query uses an aggregation to calculate the total salary for all employees:

```
@result =
    SELECT
        SUM(Salary) AS TotalSalary
    FROM @employees;
```

The result is a single row with a single column. The $165000 is the sum of of the Salary value from the whole table.

TotalSalary
165000

The following statement use the GROUP BY clause to calculate the total salary for each department:

```
@result=
    SELECT DeptName, SUM(Salary) AS SalaryByDept
    FROM @employees
    GROUP BY DeptName;
```

The results are:

DeptName	SalaryByDept
Engineering	60000
Executive	50000
HR	30000
Marketing	25000

The sum of the SalaryByDept column is $165000, which matches the amount in the last script. In both these cases the number of there are fewer output rows than input rows: Without GROUP BY, the aggregation collapses all the rows into a single row. With GROUP BY, there are N output rows where N is the number of distinct values that appear in the data, In this case, you will get 4 rows in the output.

Use a window function

The OVER clause in the following sample is empty. This defines the "window" to include all rows. The SUM in this example is applied to the OVER clause that it precedes. You could read this query as: “The sum of Salary over a window of all rows”.

```
@result=
    SELECT
        EmpName,
        SUM(Salary) OVER( ) AS SalaryAllDepts
    FROM @employees;
```

Unlike GROUP BY, there are as many output rows as input rows:

EmpName	SalaryAllDepts
Noah	165000
Sophia	165000
Liam	165000
Emma	165000
Jacob	165000
Olivia	165000
Mason	165000
Ava	165000
Ethan	165000

The value of 165000 (the total of all salaries) is placed in each output row. That total comes from the "window" of all rows, so it includes all the salaries. The next example demonstrates how to refine the "window" to list all the employees, the department, and the total salary for the department. PARTITION BY is added to the OVER clause.

```
@result=
    SELECT
        EmpName,
        DeptName,
        SUM(Salary) OVER( PARTITION BY DeptName ) AS SalaryByDept
    FROM @employees;
```

The results are:

EmpName	DeptName	SalaryByDept
Noah	Engineering	60000
Sophia	Engineering	60000
Liam	Engineering	60000
Mason	Executive	50000
Emma	HR	30000
Jacob	HR	30000
Olivia	HR	30000
Ava	Marketing	25000
Ethan	Marketing	25000

Again, there are the same number of input rows as output rows. However each row has a total salary for the corresponding department.

Reporting aggregation functions

Window functions also support the following aggregates:

- COUNT
- SUM
- MIN
- MAX
- AVG
- STDEV
- VAR
- STDEVP
- VARP

The syntax:

```
<AggregateFunction>( [DISTINCT] <expression>) [<OVER_clause>]
```

Note:

- By default, aggregate functions, except COUNT, ignore null values.
- When aggregate functions are specified along with the OVER clause, the ORDER BY clause is not allowed in the OVER clause.

SUM

The following example adds a total salary by department to each input row:

```
@result=
    SELECT
        *,
        SUM(Salary) OVER( PARTITION BY DeptName ) AS TotalByDept
    FROM @employees;
```

EmpID	EmpName	DeptName	DeptID	Salary	TotalByDept
1	Noah	Engineering	100	10000	60000
2	Sophia	Engineering	100	20000	60000
3	Liam	Engineering	100	30000	60000
7	Mason	Executive	300	50000	50000
4	Emma	HR	200	10000	30000
5	Jacob	HR	200	10000	30000
6	Olivia	HR	200	10000	30000
8	Ava	Marketing	400	15000	25000
9	Ethan	Marketing	400	10000	25000

COUNT

The following example adds an extra field to each row to show the total number employees in each department.

```
@result =
    SELECT
        *,
        COUNT(*) OVER(PARTITION BY DeptName) AS CountByDept
    FROM @employees;
```

EmpID	EmpName	DeptName	DeptID	Salary	CountByDept
1	Noah	Engineering	100	10000	3
2	Sophia	Engineering	100	20000	3
3	Liam	Engineering	100	30000	3
7	Mason	Executive	300	50000	1
4	Emma	HR	200	10000	3
5	Jacob	HR	200	10000	3
6	Olivia	HR	200	10000	3
8	Ava	Marketing	400	15000	2
9	Ethan	Marketing	400	10000	2

MIN and MAX

The following example adds an extra field to each row to show the lowest salary of each department:

```
@result =
    SELECT
        *,
        MIN(Salary) OVER ( PARTITION BY DeptName ) AS MinSalary
    FROM @employees;
```

EmpID	EmpName	DeptName	DeptID	Salary	MinSalary
1	Noah	Engineering	100	10000	10000
2	Sophia	Engineering	100	20000	10000
3	Liam	Engineering	100	30000	10000
7	Mason	Executive	300	50000	50000
4	Emma	HR	200	10000	10000
5	Jacob	HR	200	10000	10000
6	Olivia	HR	200	10000	10000
8	Ava	Marketing	400	15000	10000
9	Ethan	Marketing	400	10000	10000

Analytic functions

Analytic functions are used to understand the distributions of values in windows. The most common scenario for using analytic functions is the computation of percentiles.

Supported analytic window functions

- CUME_DIST
- PERCENT_RANK
- PERCENTILE_CONT
- PERCENTILE_DISC

CUME_DIST

CUME_DIST computes the relative position of a specified value in a group of values.

For a column "X", It calculates the percent of rows that have an X less than or equal to the current X in the same window.

For a row R, assuming ascending ordering, the **CUME_DIST** of R is the number of rows with values lower than or equal to the value of R, divided by the number of rows evaluated in the partition or query result set. **CUME_DIST** returns numbers in the range `0 < x <= 1`

```
CUME_DIST()
    OVER (
        [PARTITION BY <identifier, > …[n]]
        ORDER BY >identifier, > …[n] [ASC|DESC]
    ) AS <alias>
```

The following example uses the CUME_DIST function to compute the latency percentile for each query within a vertical.

```
@result=
    SELECT
        *,
        CUME_DIST() OVER(PARTITION BY Vertical ORDER BY Latency) AS CumeDist
    FROM @querylog;
```

The results:

Query	Latency	Vertical	CumeDist
Durian	500	Image	1
Banana	300	Image	0.666666666666667
Cherry	300	Image	0.666666666666667
Durian	500	Web	1
Cherry	400	Web	0.833333333333333
Fig	300	Web	0.666666666666667
Fig	200	Web	0.5
Papaya	200	Web	0.5
Apple	100	Web	0.166666666666667

- There are 6 rows in the partition where partition key is "Web" (4th row and down)
- There are 6 rows with the value equal or lower than 500, so the CUME_DIST equals to 6/6=1
- There are 5 rows with the value equal or lower than 400, so the CUME_DIST equals to 5/6=0.83
- There are 4 rows with the value equal or lower than 300, so the CUME_DIST equals to 4/6=0.66
- There are 3 rows with the value equal or lower than 200, so the CUME_DIST equals to 3/6=0.5. There are two rows with the same latency value.
- There is 1 row with the value equal or lower than 100, so the CUME_DIST equals to 1/6=0.16.

Usage notes:

- Tie values always evaluate to the same cumulative distribution value.
- NULL values are treated as the lowest possible values.
- You must specify the ORDER BY clause to calculate CUME_DIST.
- CUME_DIST is similar to the PERCENT_RANK function
- Note: The ORDER BY clause is not allowed if the SELECT statement is not followed by OUTPUT. Thus ORDER BY clause in the OUTPUT statement determines the display order of the resultant rowset.

PERCENTILE_CONT & PERCENTILE_DISC

These two functions calculates a percentile based on a continuous or discrete distribution of the column values.

- **PERCENTILE_CONT** calculates a percentile based on a **continuous distribution**. It will interpolate values.
- **PERCENTILE_DISC** calculates the percentile based on a **discrete distribution**. It will always return one of the input values and will not interpolate a value.

Syntax

```
[PERCENTILE_CONT | PERCENTILE_DISC] ( numeric_literal )
    WITHIN GROUP ( ORDER BY <identifier> [ASC | DESC] )
    OVER ( [PARTITION BY <identifier,>…[n] ] ) AS <alias>
```

- `numeric_literal` - The percentile to compute. range = [0.0, 1.0].
- `WITHIN GROUP ( ORDER BY <identifier> [ASC | DESC])` - Within each partition, compute the percentile on `identifier`. The default sort order is ascending.
- `OVER ( [PARTITION BY <identifierm,...> [n] ] )` - Defines the partitions.

Note: Any nulls in the data set are ignored.

PERCENTILE_CONT vs PERCENTILE_DISC

You can see how **PERCENTILE_CONT** and **PERCENTILE_DISC** differ in the example below which tries to find the median (percentile=0.50) value for Latency within each Vertical

```
@result =
    SELECT
        Vertical,
        Query,
        PERCENTILE_CONT(0.5)
            WITHIN GROUP (ORDER BY Latency)
            OVER ( PARTITION BY Vertical ) AS PercentileCont50,
        PERCENTILE_DISC(0.5)
            WITHIN GROUP (ORDER BY Latency)
            OVER ( PARTITION BY Vertical ) AS PercentileDisc50
    FROM @querylog;
```

The results:

Query	Latency	Vertical	PercentileCont50	PercentilDisc50
Banana	300	Image	300	300
Cherry	300	Image	300	300
Durian	500	Image	300	300
Apple	100	Web	250	200
Fig	200	Web	250	200
Papaya	200	Web	250	200
Fig	300	Web	250	200
Cherry	400	Web	250	200
Durian	500	Web	250	200

Look at the median for the `Web` vertical.

- **PERCENTILE_CONT** gives the median as 250 even though no query in the web vertical had a latency of 250.
- **PERCENTILE_DISC** gives median for Web as 200, which is an actual value found in the input rows.

PERCENT_RANK

PERCENT_RANK calculates the relative rank of a row within a group of rows. PERCENT_RANK is used to evaluate the relative standing of a value within a rowset or partition. The range of values returned by PERCENT_RANK is greater than 0 and less than or equal to 1.

```
PERCENT_RANK( )
    OVER (
        [PARTITION BY <identifier>; …[n]]
        ORDER BY <identifier,> …[n] [ASC|DESC]
    ) AS <alias>;
```

Notes

- The first row in any set has a PERCENT_RANK of 0.
- NULL values are treated as the lowest possible values.
- PERCENT_RANK is similar to the CUME_DIST the function. Unlike CUME_DIST, PERCENT_RANK is always 0 for the first row.

The following example uses the PERCENT_RANK function to compute the latency percentile for each query within a vertical. The PARTITION BY clause is specified to partition the rows in the result set by the vertical. The ORDER BY clause in the OVER clause orders the rows in each partition. The value returned by the PERCENT_RANK function represents the rank of the queries' latency within a vertical as a percentage.

```
@result=
    SELECT
        *,
        PERCENT_RANK()
            OVER (PARTITION BY Vertical ORDER BY Latency) AS PercentRank
    FROM @querylog;
```

Query	Latency:int	Vertical	PercentRank
Banana	300	Image	0
Cherry	300	Image	0
Durian	500	Image	1
Apple	100	Web	0
Fig	200	Web	0.2
Papaya	200	Web	0.2
Fig	300	Web	0.6
Cherry	400	Web	0.8
Durian	500	Web	1

Window Ranking Functions

Ranking functions return a ranking value (a long) for each row in each partition as defined by the PARTITION BY and OVER clauses. The ordering of the rank is controlled by the ORDER BY in the OVER clause.

The following are supported ranking functions:

- RANK
- DENSE_RANK
- ROW_NUMBER
- NTILE

Syntax:

```
[RANK() | DENSE_RANK() | ROW_NUMBER() | NTILE(<numgroups>)]
    OVER (
    [PARTITION BY <identifier, > …[n]]
    [ORDER BY <identifier, > …[n] [ASC|DESC]]
    ) AS <alias>
```

The ORDER BY clause is optional for ranking functions. If ORDER BY is specified then it determines the order of the ranking. If ORDER BY is not specified then U-SQL assigns values based on the order it reads record. Thus resulting into non deterministic value of row number, rank or dense rank in the case were order by clause is not specified.

NTILE requires an expression that evaluates to a positive integer. This number specifies the number of groups into which each partition must be divided. This identifier is used only with the NTILE ranking function.

For more details on the OVER clause, see U-SQL reference.

ROW_NUMBER, RANK, and DENSE_RANK

ROW_NUMBER, RANK, and DENSE_RANK all assign numbers to rows in a window. Rather than cover them separately, it's more intuitive to see how They respond to the same input.

```
@result =
SELECT
*,
ROW_NUMBER() OVER (PARTITION BY Vertical ORDER BY Latency) AS RowNumber,
RANK() OVER (PARTITION BY Vertical ORDER BY Latency) AS Rank,
DENSE_RANK() OVER (PARTITION BY Vertical ORDER BY Latency) AS DenseRank
FROM @querylog;
```

Note the OVER clauses are identical. The result:

Query	Latency:int	Vertical	RowNumber	Rank	DenseRank
Banana	300	Image	1	1	1
Cherry	300	Image	2	1	1
Durian	500	Image	3	3	2
Apple	100	Web	1	1	1
Fig	200	Web	2	2	2
Papaya	200	Web	3	2	2
Fig	300	Web	4	4	3
Cherry	400	Web	5	5	4
Durian	500	Web	6	6	5

ROW_NUMBER

Within each Window (Vertical,either Image or Web), the row number increases by 1 ordered by Latency.

RANK

Different from ROW_NUMBER(), RANK() takes into account the value of the Latency which is specified in the ORDER BY clause for the window.

RANK starts with (1,1,3) because the first two values for Latency are the same. Then the next value is 3 because the Latency value has moved on to 500. The key point being that even though duplicate values are given the same rank, the RANK number will "skip" to the next ROW_NUMBER value. You can see this pattern repeat with the sequence (2,2,4) in the Web vertical.

DENSE_RANK

DENSE_RANK is just like RANK except it doesn't "skip" to the next ROW_NUMBER, instead it goes to the next number in the sequence. Notice the sequences (1,1,2) and (2,2,3) in the sample.

Remarks

- If ORDER BY is not specified than ranking function will be applied to rowset without any ordering. This will result into non deterministic behavior on how ranking function is applied
- There is no guarantee that the rows returned by a query using ROW_NUMBER will be ordered exactly the same with each execution unless the following conditions are true.
- Values of the partitioned column are unique.
- Values of the ORDER BY columns are unique.
- Combinations of values of the partition column and ORDER BY columns are unique.

NTILE

NTILE distributes the rows in an ordered partition into a specified number of groups. The groups are numbered, starting at one.

The following example splits the set of rows in each partition (vertical) into 4 groups in the order of the query latency, and returns the group number for each row.

The Image vertical has 3 rows, thus it has 3 groups.

The Web vertical has 6 rows, the two extra rows are distributed to the first two groups. That's why there are 2 rows in group 1 and group 2, and only 1 row in group 3 and group 4.

```
@result =
    SELECT
        *,
        NTILE(4) OVER(PARTITION BY Vertical ORDER BY Latency) AS Quartile
    FROM @querylog;
```

The results:

Query	Latency	Vertical	Quartile
Banana	300	Image	1
Cherry	300	Image	2
Durian	500	Image	3
Apple	100	Web	1
Fig	200	Web	1
Papaya	200	Web	2
Fig	300	Web	2
Cherry	400	Web	3
Durian	500	Web	4

NTILE takes a parameter ("numgroups"). Numgroups is a positive int or long constant expression that specifies the number of groups into which each partition must be divided.

If the number of rows in the partition is evenly divisible by numgroups then the groups will have equal size.

If the number of rows in a partition is not divisible by numgroups, this will cause groups of two sizes that differ by one member. Larger groups come before smaller groups in the order specified by the OVER clause.

For example:

- 100 rows divided into 4 groups: [25, 25, 25, 25]
- 102 rows divided into 4 groups: [26, 26, 25, 25]

Assign Globally Unique Row Number

It's often useful to assign a globally unique number to each row. This is easy (and more efficient than using a reducer) with the ranking functions.

```
@result =
    SELECT
        *,
        ROW_NUMBER() OVER () AS RowNumber
    FROM @querylog;
```

Top N Records in group via RANK, DENSE_RANK or ROW_NUMBER

Many users want to select only TOP n rows per group. This is not possible with the traditional GROUP BY.

You have seen the following example at the beginning of the Ranking functions section. It doesn't show top N records for each partition:

```
@result =
    SELECT
        *,
        ROW_NUMBER() OVER (PARTITION BY Vertical ORDER BY Latency) AS RowNumber,
        RANK() OVER (PARTITION BY Vertical ORDER BY Latency) AS Rank,
        DENSE_RANK() OVER (PARTITION BY Vertical ORDER BY Latency) AS DenseRank
    FROM @querylog;
```

The results:

Query	Latency	Vertical	Rank	DenseRank	RowNumber
Banana	300	Image	1	1	1
Cherry	300	Image	1	1	2
Durian	500	Image	3	2	3
Apple	100	Web	1	1	1
Fig	200	Web	2	2	2
Papaya	200	Web	2	2	3
Fig	300	Web	4	3	4
Cherry	400	Web	5	4	5
Durian	500	Web	6	5	6

TOP N with DENSE RANK

The following example returns the top 3 records from each group with no gaps in the sequential rank numbering of rows in each windowing partition.

```
@result =
    SELECT
        *,
        DENSE_RANK()
            OVER (PARTITION BY Vertical ORDER BY Latency) AS DenseRank
    FROM @querylog;

@result =
    SELECT *
    FROM @result
    WHERE DenseRank <= 3;
```

The results:

Query	Latency	Vertical	DenseRank
Banana	300	Image	1
Cherry	300	Image	1
Durian	500	Image	2
Apple	100	Web	1
Fig	200	Web	2
Papaya	200	Web	2
Fig	300	Web	3

TOP N with RANK

```
@result =
    SELECT
        *,
        RANK()
            OVER (PARTITION BY Vertical ORDER BY Latency) AS Rank
    FROM @querylog;

    @result =
        SELECT *
        FROM @result
        WHERE Rank <= 3;
```

The results:

Query	Latency	Vertical	Rank
Banana	300	Image	1
Cherry	300	Image	1
Durian	500	Image	3
Apple	100	Web	1
Fig	200	Web	2
Papaya	200	Web	2

TOP N with ROW_NUMBER

```
@result =
    SELECT
        *,
        ROW_NUMBER()
            OVER (PARTITION BY Vertical ORDER BY Latency) AS RowNumber
    FROM @querylog;

@result =
    SELECT *
    FROM @result
    WHERE RowNumber <= 3;
```

The results:

Query	Latency	Vertical	RowNumber
Banana	300	Image	1
Cherry	300	Image	2
Durian	500	Image	3
Apple	100	Web	1
Fig	200	Web	2
Papaya	200	Web	3

Any N Records per group

Just as you sometimes want to get the "TOP n rows per group". Sometimes you just want "ANY n rows per group".

For example, "list any 10 customers per zipcode".

```
@result =
    SELECT
        *,
        ROW_NUMBER()
            OVER (PARTITION BY ZipCode ORDER BY 1) AS RowNumber
    FROM @customers;

@result =
    SELECT *
    FROM @result
    WHERE RowNumber <= 10;
```

This technique makes use of the "ORDER BY 1" pattern - which effectively means no ordering is performed.

Getting the Rows that have the maximum (or minimum) value for a Column within a partition

Another scenario easily done through the ranking functions, is finding the row that contains the max value in a partition

Returning to our original input data set, imagine we want to partition by Vertical and within each vertical find the row that has the maximum value for latency.

Query	Latency	Vertical
Banana	300	Image
Cherry	300	Image
Durian	500	Image
Apple	100	Web
Fig	200	Web
Papaya	200	Web
Fig	300	Web
Cherry	400	Web
Durian	500	Web

The desired output for is as follows. As clearly 500 is the maximum latency in both Image and Web

Query	Latency	Vertical
Durian	500	Image
Durian	500	Web

The U-SQL that accomplishes this uses ROW_NUMBER

```
@results =
     SELECT
         Query,
         Latency,
         Vertical,
         ROW\_NUMBER() OVER (PARTITION BY Vertical ORDER BY Latency DESC) AS rn
     FROM @querylog;

@results =
     SELECT
         Query,
         Latency,
         Vertical
     FROM @results
     WHERE rn==1;
```

To retrieve the row with the minimum value for each partition, in the OVER clause change the **DESC** to **ASC**.

Set operations

Set operations are a way of merging rowsets together based on set theoretic operations such as union (UNION). intersection (INTERSECT), complement (EXCEPT).

Sample data

Let's define two RowSets: `@a` and `@b`. Notice that both RowSets have duplicate rows.

```
@a  =
    SELECT * FROM
        (VALUES
            (1,    "Smith"),
            (1,    "Smith"),
            (2,    "Brown"),
            (3,    "Case")
        ) AS D( DepID, DepName );
```

```
@b  =
    SELECT * FROM
        (VALUES
            (1,    "Smith"),
            (1,    "Smith"),
            (1,    "Smith"),
            (2,    "Brown"),
            (4,    "Dey"),
            (4,    "Dey")
        ) AS D( DepID, DepName );
```

UNION

UNION combines two rowsets.

UNION ALL

As you can see UNION ALL clearly leaves in duplicate rows.

```
@union_all =
    SELECT * FROM @a
    UNION ALL
    SELECT * FROM @b;
```

DepID	Name
1	Smith
1	Smith
2	Brown
3	Case
1	Smith
1	Smith
1	Smith
2	Brown
4	Dey
4	Dey

UNION DISTINCT

UNION DISTINCT discards duplicate rows.

```
@union_distinct =
    SELECT * FROM @a
    UNION DISTINCT
    SELECT * FROM @b;
```

DepID	Name
1	Smith
2	Brown
3	Case
4	Dey

Schema requirements

UNION by default require that the RowSets have the same schema.

- Each column must have the same name and data type
- The columns must appear in the same order in the rowset schema

Finding Common Rows with INTERSECT

Sometimes, we only care about the rows both rowsets have *in common*. We use the **INTERSECT** operator to accomplish this. **INTERSECT ALL** preserves duplicates while **INTERSECT** removes duplicates.

INTERSECT ALL

```
@intersect_all =
    SELECT * FROM @a
    INTERSECT ALL
    SELECT * FROM @b;
```

@intersect_all

DepID	name
1	Smith
1	Smith
2	Brown

INTERSECT DISTINCT

```
@intersect_distinct =
    SELECT * FROM @a
    INTERSECT DISTINCT
    SELECT * FROM @b;
```

@intersect_distinct

DepID	name
1	Smith
2	Brown

Finding Rows That Are NOT in the Other RowSet with EXCEPT

The **EXCEPT** operator returns all the rows in the left RowSet that *are not* in the right RowSet.

EXCEPT ALL

```
@a_except_b_all =
   SELECT * FROM @a
   EXCEPT ALL
   SELECT * FROM @b;
```

@a_except_b_all

DepID	name
3	Case

```
@b_except_a_all =
   SELECT * FROM @b
   EXCEPT ALL
   SELECT * FROM @a;
```

@b_except_a_all

DepID	name
1	Smith
4	Dey
4	Dey

EXCEPT DISTINCT

```
@a_except_b_distinct =
   SELECT * FROM @a
   EXCEPT DISTINCT
   SELECT * FROM @b;
```

@a_except_b_distinct

DepID	name
3	Case

```
@b_except_a_distinct =
   SELECT * FROM @b
   EXCEPT DISTINCT
   SELECT * FROM @a;
```

@b_except_a_distinct

DepID	name
4	Dey

OUTER UNION

By default UNION requires both RowSets to have matching schemas. OUTER UNION allows the schemas to be different. If one RowSet is missing a column that the other has, that row will be included in the result with a default value for the missing columns.

NOTE: OUTER UNION only supports ALL. It does not support DISTINCT.

The following script will union the two rowsets `@left` and `@right` with the partially overlapping schema on columns `A` and `K` while filling in `null` into the "missing cells" of column `C` and `0` as the default value for type `int` for column `B`.

```
@left =
    SELECT *
    FROM (VALUES ( 1, "x", (int?) 50 ),
                 ( 1, "y", (int?) 60 )
         ) AS L(K, A, C);

@right =
    SELECT *
    FROM (VALUES ( 5, "x", 1 ),
                 ( 6, "x", 2 ),
                 (10, "y", 3 )
         ) AS R(B, A, K);

@res =
    SELECT * FROM @left
    OUTER UNION BY NAME ON (*)
    SELECT * FROM @right;
```

The result is:

K	A	C	B
1	"x"	50	0
1	"x"		5
1	"y"	60	0
2	"x"		6
3	"y"		10

Joins

A JOIN is a way of combining rowsets together based on key
Wikipedia article on SQL Joins

Sample data

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

```
@employees =
SELECT * FROM
    (VALUES
    ("31", "Rafferty"),
    ("33", "Jones"),
    ("33", "Heisenberg"),
    ("34", "Robinson"),
    ("34", "Smith"),
    ((string)null, "Williams")
    ) AS D( DepID, EmpName );
```

CROSS JOIN

```
@cross_join =
    SELECT
        @departments.DepID AS DepID_Dep,
        @employees.DepID AS DepID_Emp,
        @employees.EmpName,
        @departments.DepName
    FROM @employees CROSS JOIN @departments;
```

DepIDDep	DepIDEmp	EmpName	DepName
31	31	Rafferty	Sales
31	31	Rafferty	Sales

|31|33|Jones|Sales| |31|33|Heisenberg|Sales| |31|34|Robinson|Sales| |31|34|Smith|Sales| |31|ERR|Williams|Sales| |33|31|Rafferty|Engineering| |33|33|Jones|Engineering| |33|33|Heisenberg|Engineering| |33|34|Robinson|Engineering| |33|34|Smith|Engineering| |33|ERR|Williams|Engineering| |34|31|Rafferty|Clerical| |34|33|Jones|Clerical|

34	33	Heisenberg	Clerical		34	34	Robinson	Clerical		34	34	Smith	Clerical
34	ERR	Williams	Clerical		35	31	Rafferty	Marketing		35	33	Jones	Marketing
35	33	Heisenberg	Marketing		35	34	Robinson	Marketing		35	34	Smith	Marketing
35	NULL	Williams	Marketing

CROSS JOIN

```
@cross_join =
    SELECT
        @departments.DepID AS DepID_Dep,
        @employees.DepID AS DepID_Emp,
        @employees.EmpName,
        @departments.DepName
     FROM
        @employees CROSS JOIN @departments;
```

DepID_Dep	DepID_Emp	EmpName	DepName
31	31	Rafferty	Sales
31	33	Jones	Sales
31	33	Heisenberg	Sales
31	34	Robinson	Sales
31	34	Smith	Sales
31	NULL	Williams	Sales
33	31	Rafferty	Engineering
33	33	Jones	Engineering
33	33	Heisenberg	Engineering
33	34	Robinson	Engineering
33	34	Smith	Engineering
33	NULL	Williams	Engineering
34	31	Rafferty	Clerical
34	33	Jones	Clerical
34	33	Heisenberg	Clerical
34	34	Robinson	Clerical
34	34	Smith	Clerical
34	NULL	Williams	Clerical
35	31	Rafferty	Marketing
35	33	Jones	Marketing
35	33	Heisenberg	Marketing
35	34	Robinson	Marketing
35	34	Smith	Marketing
35	NULL	Williams	Marketing

SEMIJOIN

SEMIJOIN is a way to filter a RowSet using another RowSet.

A simple example is: "Find all the employees in this RowSet A, where the employee appears also in RowSet B".

There are two variants:

- **LEFT SEMIJOIN** -> Give only those rows in the left rowset that have a matching row in the right rowset.
- **RIGHT SEMIJOIN** -> Give only those rows in the right rowset that have a matching row in the left rowset.

NOTE: If you leave out **LEFT** or **RIGHT**, and instead simply write **SEMIJOIN** then what you get is **LEFT SEMIJOIN**. Do not leave out **LEFT** or **RIGHT** always explicitly it.

Find all employees that are in valid departments

```
@left_semijoin1 =
    SELECT
        @employees.DepID,
        @employees.EmpName
    FROM @employees
    LEFT SEMIJOIN @departments
        ON @employees.DepID == @departments.DepID;
```

DepID	EmpName
31	Rafferty
33	Jones
33	Heisenberg
34	Robinson
34	Smith

Find all departments that has an employee listed in the employee RowSet.

```
@left_semijoin2 =
    SELECT
        @departments.DepID,
        @departments.DepName
    FROM @departments
    LEFT SEMIJOIN @employees
    ON @departments.DepID == @employees.DepID;
```

DepID	DepName
31	Sales
33	Engineering
34	Clerical

ANTISEMIJOIN

An **ANTISEMIJOIN** is the opposite of **SEMIJOIN**

There are two variants:

- **LEFT ANTISEMIJOIN** -> Give only those rows in the left RowSet that DO NOT have a matching row in the right rowset.
- **RIGHT ANTISEMIJOIN** -> Give only those rows in the right RowSet that DO NOT have a matching row in the left RowSet.

Find all departments that don’t have an employee listed in the employees RowSet.

```
@left_antisemijoin =
    SELECT
        @departments.DepID,
        @departments.DepName
    FROM @departments
    LEFT ANTISEMIJOIN @employees
        ON @departments.DepID == @employees.DepID;
```

DepID	DepName
35	Marketing

Complex Types

Nullability

Complex types are reference types. You MUST check if they are null first before using them.

Maximum size of a complex value

There is no specific limit inherent to Complex Types

However, rowsets must still fit within the limits defined by U-SQL (currently no row can be bigger than 4MB)

Persistence

U-SQL Tables

Complex Types can be stored in and read from U-SQL Tables without any additional effort.

Files

To read/write Complex Types with files, you must use your own Custom Outputter & Extractor.

ARRAYs

Overview

- An ordered list of values (all of the same type)
- Immutable
- Single-Dimension only

Creating arrays initialized with data

You can create a SQlArray using the SqlArray constructor or use the SqlArray.Create static method

The SqlArray.Create static method can also be used to create arrays.

```
@output =
    SELECT *
    FROM
    ( VALUES
        ( "West Virginia", new SqlArray<string>( new string [] { "Charleston", "Huntin
gton", "Parkersburg", "Morgantown", "Wheeling"} ) ),
        ( "Wisconsin", SqlArray.Create( new string [] { "Milwaukee", "Madison", "Green
 Bay", "Kenosha", "Racine"} ) )

    ) AS T(State, Cities);
```

Creating Empty arrays

```
new SqlArray<string> ( )
```

or

```
new SqlArray<string> { }
```

Creating an ARRAY from an .NET Array

```
DECLARE @wv_cities = new string [] { "Charleston", "Huntington", "Parkersburg", "Morga
ntown", "Wheeling"};
DECLARE @wi_cities = new string [] { "Milwaukee", "Madison", "Green Bay", "Kenosha", "
Racine" };

@cities =
    SELECT *
    FROM
    ( VALUES
        ( "West Virginia", new SqlArray<string>( @wv_cities ) ),
        ( "Wisconsin", SqlArray.Create( @wi_cities ) )

    ) AS T(State, Cities);
```

Sample data

```
@cities =
    SELECT *
    FROM
    ( VALUES
        ( "Vermont","Burlington,Essex,South,Burlington,Colchester,Rutland" ),
        ( "Virginia","Virginia Beach,Norfolk,Chesapeake,Richmond,Newport News" ),
        ( "Washington","Seattle,Spokane,Tacoma,Vancouver,Bellevue" ),
        ( "West Virginia", "Charleston,Huntington,Parkersburg,Morgantown,Wheeling"),
        ( "Wisconsin","Milwaukee,Madison,Green Bay,Kenosha,Racine"),
        ( "Wyoming","Cheyenne,Casper,Laramie,Gillette,Rock Springs" )
    ) AS T(State, Cities);

@cities =
    SELECT
        State,
        SqlArray.Create( Cities.Split(',') ) AS Cities
    FROM @cities;
```

State string	Cities SqlArray
Vermont	SqlArray{ "Burlington", "Essex", "South", "Burlington", "Colchester", "Rutland" }
Virginia	SqlArray{ "Virginia Beach", "Norfolk", "Chesapeake", "Richmond", "Newport News" }
Washington	SqlArray{ "Seattle", "Spokane", "Tacoma", "Vancouver", "Bellevue" }
West Virginia	SqlArray{ "Charleston", "Huntington", "Parkersburg", "Morgantown", "Wheeling" }
Wisconsin	SqlArray{ "Milwaukee", "Madison", "Green Bay", "Kenosha", "Racine" }
Wyoming	SqlArray{ "Cheyenne", "Casper", "Laramie", "Gillette", "Rock Springs" }

Array Indexing

Use the array indexing operator [n] where n is a long. The first index is 0 – just like .NET

```
@output =
    SELECT
        State,
        Cities[0] AS FirstCity
    FROM @output;
```

State	FirstCity
Vermont	Burlington
Virginia	Virginia Beach
Washington	Seattle
West Virginia	Charleston
Wisconsin	Milwaukee
Wyoming	Cheyenne

Removing members

```
@output =
    SELECT
        State ,
        SqlArray.Create( Cities.Where( c=>c.StartsWith("C") ) ) AS Cities
    FROM @output;
```

State	Cities
Vermont	SqlArray{ "Colchester" }
Virginia	SqlArray{ "Chesapeake" }
Washington	SqlArray{ }
West Virginia	SqlArray{ "Charleston" }
Wisconsin	SqlArray{ }
Wyoming	SqlArray{ "Cheyenne", "Casper" }

Counting members

```
@output =
    SELECT
        State ,
        SqlArray.Create( Cities.Where( c=>c.StartsWith("C") ) ) AS Cities
    FROM @output;

@output =
    SELECT
        State ,
        Cities ,
        Cities.Count AS NumCities
    FROM @output;
```

State	Cities	NumCities
Vermont	SqlArray{ "Colchester" }	1
Virginia	SqlArray{ "Chesapeake" }	1
Washington	SqlArray{ }	0
West Virginia	SqlArray{ "Charleston" }	1
Wisconsin	SqlArray{ }	0
Wyoming	SqlArray{ "Cheyenne", "Casper" }	2

MAPs

Overview

Creating a SqlMap from constant values

The following snippet creates two rows and each has a SqlMap where the key (K) is a string and the value is also a string.

```
@projectmembers =
    SELECT *
    FROM
    ( VALUES
        ( "Website", new SqlMap<string,string> {
                {"Mallory", "PM"},
                {"Bob", "Dev"} ,
                {"Alice", "Dev"} ,
                {"Stan", "Dev"} ,
                {"Chris", "UX"} ,
            }
        ),
        ( "DB", new SqlMap<string,string> {
                {"Ted", "Test"},
                {"Joe", "Dev"} ,
                {"Chuck", "Dev"}
            }
        )
)
AS T(Project, Members);
```

Project	Members
Website	SqlMap{ Alice=Dev; Bob=Dev; Chris=UX; Mallory=PM; Stan=Dev }
DB	SqlMap{ Chuck=Dev; Joe=Dev; Ted=Test }

Creating Empty maps

```
new SqlMap<string,string> ( )
```

or

```
new SqlMap<string,string> { }
```

Maps from Maps: Removing members based on keys

```
@output =
    SELECT Project,
           new SqlMap<string,string>(Members.Where(kv => kv.Key != "Mallory")) AS Memb
ers
    FROM @projectmembers;
```

Project	Members
Website	SqlMap{ Alice=Dev; Bob=Dev; Chris=UX; Stan=Dev }
DB	SqlMap{ Chuck=Dev; Joe=Dev; Ted=Test }

Alternatively you can use the SqlMap.Create static method instead

```
@output =
    SELECT Project,
           SqlMap.Create(Members.Where(kv => kv.Key != "Mallory")) AS Members
    FROM @projectmembers;
```

Project	Members
Website	SqlMap{ Alice=Dev; Bob=Dev; Chris=UX; Stan=Dev }
DB	SqlMap{ Chuck=Dev; Joe=Dev; Ted=Test }

Combining rows into maps with MAP_AGG

```
@projectmembers =
    SELECT *
    FROM
    ( VALUES
        ( "Website","Mallory", "PM" ),
        ( "Website","Bob", "Dev" ),
        ( "Website","Alice", "Dev" ) ,
        ( "Website","Stan", "Dev" ) ,
        ( "Website","Chris", "UX" ) ,
        ( "DB", "Ted", "Test" ),
        ( "DB", "Joe", "Dev" ) ,
        ( "DB", "Chuck", "Dev" )
    )
AS T(Project, Employee, Role);

@projectmembers =
    SELECT Project,
           MAP_AGG<string, string>(Employee, Role) AS Members
    FROM @projectmembers_raw
    GROUP BY Project;
```

Project	Members
DB	SqlMap{ Chuck=Dev; Joe=Dev; Ted=Test }
Website	SqlMap{ Alice=Dev; Bob=Dev; Chris=UX; Mallory=PM; Stan=Dev }

Removing members based on values

```
@output =
    SELECT Project,
           SqlMap.Create(Members.Where(kv => kv.Value != "Dev")) AS Members
    FROM @projectmembers;
```

Project	Members
Website	SqlMap{ Chris=UX; Mallory=PM }
DB	SqlMap{ Ted=Test }

counting members

```
@output =
    SELECT Project,
           Members,
           Members.Count AS Count
FROM @projectmembers;
```

Project	Members	Count
Website	SqlMap{ Alice=Dev; Bob=Dev; Chris=UX; Mallory=PM; Stan=Dev }	5
DB	SqlMap{ Chuck=Dev; Joe=Dev; Ted=Test }	3

Retrieving values

```
@output =
    SELECT
        Project,
        Members["Mallory"] AS MalloryRole
   FROM @projectmembers;
```

Note that if the key is missing then the default value for the type is returned

Project	MalloryRole
Website	PM
DB	null

Checking if a key exists

```
@output =
    SELECT
        Project,
        Members.ContainsKey("Mallory") AS ContainsMallory
FROM @projectmembers;
```

Project	ContainsMallory
Website	True
DB	False

Extending U-SQL

There are 5 kinds of User-Defined entities in U-SQL

- User-Defined **Functions** (UDFs)
- User-Defined **Types** (UDTs)
- User-Defined **Aggregators** (UDAggs)
- User-defined **Operators** (UDOs)
- User-Defined **Appliers**

All of them are defined by .NET code. C# is not required. Any .NET language will work.

User-Defined Functions

User defined functions are normal static methods on a .NET Class. Below is the code for a simple class.

```
namespace OrdersLib
{
  public static class Helpers
  {
    public static string Normalize(string s)
    { return s.ToLower(); }
  }
}
```

Assuming you registered the assembly that contains this class using the name "MyDB.OrdersLibAsm", you can use the assembly and the UDF as shown below.

```
REFERENCE ASSEMBLY MyDB.OrdersLibAsm;

@customers =
  SELECT * FROM
    (VALUES
      ("Contoso", 123 ),
      ("Woodgrove", 456 )
    ) AS D( Customer, Id );

@customers =
  SELECT
    OrdersLib.Helpers.Normalize(Customer) AS Customer,
    Id
  FROM @customers;
```

User-Defined Aggregators (UDAggs)

UDAggs allow you to define your own aggregate functions for U-SQL for use with GROUP BY.

Creating an aggregator

Here is a simple example of a UDAgg that manually implements a custom version of Sum

```
using Microsoft.Analytics.Interfaces;

namespace MVA_UDAgg
{
    public class MySum : IAggregate<int, long>
    {
        long total;
        public override void Init() { total = 0; }
        public override void Accumulate(int value) { total += value; }
        public override long Terminate() { return total; }
    }
}
```

Using the UDAgg

In the SELECT clause use the `AGG<T>` operator where `T` is the name of your UDAgg;

```
AGG<UDAgg>(InputCol) AS OutputCOl
```

Here's a full script:

```
REFERENCE ASSEMBLY AssemblyContainingUDagg;

@t =
  SELECT * FROM
    (VALUES
      ("2016/03/31","1:00","mrys","@saveenr great demo yesterday", 7 ),
      ("2016/03/31","7:00","saveenr","@mrys Thanks! U-SQL RuL3Z!", 4 )
    ) AS D( date, time, author, tweet , retweets);

@results =
    SELECT
        AGG<MVA_UDAgg.MySum>(retweets) AS totalretweets
    FROM @t
    GROUP BY date;

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

Recursive Aggregators

If the operation in your is associative (https://en.wikipedia.org/wiki/Associative_property\) then you should mark your aggregator with an attribute to indicate that the UDAgg is "recursive" (a.k.a "associative"). This will improve the performance substantially of your U-SQL script has to aggregate a lot of data with your UDAgg because the UDAgg can be parallelized.

The snippet below shows how to do this.

```
[SqlUserDefinedReducer(IsRecursive = true)]
public class MySum : IAggregate<int, long>
{
// your code here
}
```

Do not just blindly add the `IsRecursive` property to your UDAggs. Make sure the UDAggs support the associative property.

DEPLOY RESOURCE

Scenario

Suppose you are using a .NET library - for example one that maps IP addresses to locations. THen suppose that this .NET library loads its IP database from a file on disk - maybe a file called `ip.data`.

Now if we want to use this library in U-SQL we have to do two things: (1) make sure the .NET library is available to the script via a U-SQL assembly and (2) make sure the data file is available to the script with the `DEPLOY RESOURCE` statement.

A simple Hello World example

The simple example below shows how it would be done with a C# expression directly in the script.

```
DEPLOY RESOURCE "/helloworld.txt";

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

@departments =
    SELECT DepID, DepName, System.[IO].File.ReadAllText("helloworld.txt") AS Message
    FROM @departments;

OUTPUT @departments
    TO "/departments.tsv"
    USING Outputters.Tsv();
```

Using a processor

```
DEPLOY RESOURCE "/helloworld.txt";

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


@departments =
     PROCESS @departments
     PRODUCE DepID int,
             DepName string,
             HelloWorld string
     USING new Demo.HelloWorldProcessor();

OUTPUT @departments
    TO "/departments.tsv"
    USING Outputters.Tsv();
```

```
using Microsoft.Analytics.Interfaces;
using Microsoft.Analytics.Types.Sql;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;

namespace Demo
{
    [SqlUserDefinedProcessor]
    public class HelloWorldProcessor : IProcessor
    {
        private string hw;

        public HelloWorldProcessor()
        {
            this.hw = System.IO.File.ReadAllText("helloworld.txt");
        }

        public override IRow Process(IRow input, IUpdatableRow output)
        {
            output.Set<int>("DepID", input.Get<int>("DepID"));
            output.Set<string>("DepName", input.Get<string>("DepName"));
            output.Set<string>("HelloWorld", hw);
            return output.AsReadOnly();
        }
    }
}
```

User-Defined Types

Concepts

In order to create a UDT in U-SQL there are two thing that have to be created:

- The UDT type itself
- A formatter that can conver the UDT into a string and can transform a string into an instance of the UDT

Scenario

We will define a UDT - and its corresponding formatter - for a "Bits" UDT. This UDFT is a simple bitarray with a textual representation that looks like "1100101010".

UDT and formatter skeletons

First let's look at the fundamental structure of the Bits type.

```
[SqlUserDefinedType(typeof(BitFormatter))]
public struct Bits
{
}
```

Notice that the only thing the Bits UDT requires is that it identifies its corresponding formatter.

And here is the skeleton of its corresponding formatter.

```
public class BitFormatter : Microsoft.Analytics.Interfaces.IFormatter<Bits>
{
    public BitFormatter()
    { ... }

    public void Serialize(
        Bits instance,
        IColumnWriter writer,
        ISerializationContext context)
    { ... }

    public Bits Deserialize(
         IColumnReader reader,
         ISerializationContext context)
    { ... }
}
```

Full code for the UDT

```
namespace MyUDTExamples
{
    [SqlUserDefinedType(typeof(BitFormatter))]
    public struct Bits
    {
        System.Collections.BitArray bitarray;

        public Bits(string s)
        {
            this.bitarray = new System.Collections.BitArray(s.Length);
            for (int i = 0; i<s.Length; i++)
            {
                this.bitarray[i] = (s[s.Length-i-1] == '1' ? true : false);
            }
        }

        public int ToInteger()
        {
            int value = 0;
            for (int i = 0; i < this.bitarray.Length; i++)
                { if (bitarray[i]) { value += (int)System.Math.Pow(2, i); } }
            return value;
        }

        public override string ToString()
        {
            var sb = new System.Text.StringBuilder(this.bitarray.Length);
            for (int i = 0; i < this.bitarray.Length; i++)
            { sb.Append(this.bitarray[i] ? "1" : "0"); }
            return sb.ToString();
        }
    }

}
```

Full code for the UDT's formatter

```
namespace MyUDTExamples
{

    public class BitFormatter : Microsoft.Analytics.Interfaces.IFormatter<Bits>
    {
        public BitFormatter()
        {
        }

        public void Serialize(
            Bits instance,
            IColumnWriter writer,
            ISerializationContext context)
        {
            using (var w = new System.IO.StreamWriter(writer.BaseStream))
            {
                var bitstring = instance.ToString();
                w.Write(bitstring);
                w.Flush();
            }
        }

        public Bits Deserialize(
            IColumnReader reader,
            ISerializationContext context)
        {
            using (var w = new System.IO.StreamReader(reader.BaseStream))
            {
                string bitstring = w.ReadToEnd();
                var bits = new Bits(bitstring);
                return bits;
            }
        }
    }

}
```

Using the UDT

```
@products  =
    SELECT * FROM
        (VALUES
            ("Apple", "0000"),
            ("Cherry", "0001"),
            ("Banana", "1001"),
            ("Orange", "0110")
        ) AS
              D( bitstring );

@products =
    SELECT
       ProductCode
       BitString,
       new MyUDTExamples.Bits(BitString) AS Bits
    FROM @products;
```

Persisting UDTs

Persisting UDTs into a file

UDTs cannot be persistent directly into a file using a default U-SQL outputter. You'll have to manually convert your UDT's value to a U-SQL-supported datataype.

```
@products2 =
    SELECT
        ProductCode
        BitString,
        Bits.ToInteger() AS BitInt
    FROM @products;
```

Now it can be persisted into a text file:

```
OUTPUT @products2
    TO "/output.csv"
    USING Outputters.Csv();
```

Persisting UDTs into a U-SQL table

UDTs cannot be persistent directly into a U-SQL table. You'll have to manually convert your UDT's value to a U-SQL-supported datataype.

```
CREATE TABLE MyDB.dbo.MyTable
(
    INDEX idx
    CLUSTERED(ProductCode ASC)
    DISTRIBUTED BY HASH(ProductCode)
) AS SELECT * FROM @products2;
```

Tips

Generating ranges of numbers and dates

Many common scenarios for U-SQL developers require constructing a RowSet made up of a simple range of numbers or dates, for example the integers from 1 to 10. In this blog post we'll take a look at options for doing this in U-SQL. In the process, we'll get a chance to learn how to use some common U-SQL features:

- Creating RowSets from constant values
- Using CROSS JOIN
- Using SELECT to map integers to DateTimes
- Using CREATE TABLE to create a table directly from a RowSet. This is sometimes called "CREATE TABLE AS SELECT" and often abbreviated as "CTAS".

First, we'll begin by using the VALUES statement to create a simple RowSet of integers from 0 to

```
@numbers_10 =
    SELECT*
    FROM (VALUES
          (0),
          (1),
          (2),
          (3),
          (4),
          (5),
          (6),
          (7),
          (8),
          (9)
    ) AS T(Value);
```

This technique is simple. However, it's disadvantages are: (1) the script will need to manually list all the numbers (2) there is an upper limit on the number of items allowed in VALUES in a U-SQL script. Currently that limit is 10,000 items.

The CROSS JOIN statement helps us generate larger lists of numbers easily. In the following snippet, CROSS JOIN is used to generate 100 integers from 0 to 99.

```
@numbers_100 =
    SELECT (a.Value*10 + b.Value) AS Value
    FROM @numbers_10 AS a
        CROSS JOIN @numbers_10 AS b;
```

We can apply CROSS JOIN again to increase this to generate 10,000 integers from 0 to 9,999 as shown in the following sample:

```
@numbers_10000 =
    SELECT (a.Value*100 + b.Value) AS Value
    FROM @numbers_100 AS a CROSS JOIN @numbers_100 AS b;
```

Finally, if you want a range of dates, use SELECT to map each integer to a DateTime value as shown below.

```
DECLARE @StartDate = DateTime.Parse("1979-03-31");

@numbers_10000 = ...;

@result =
    SELECT
        Value,
        @StartDate.AddDays( Value ) AS Date
    FROM @numbers_10000;
```

Putting it all together, the full script looks like this:

```
DECLARE @StartDate = DateTime.Parse("1979-03-31");

@numbers_10 =
    SELECT
        *
    FROM
    (VALUES
        (0),
        (1),
        (2),
        (3),
        (4),
        (5),
        (6),
        (7),
        (8),
        (9)
    ) AS T(Value);

@numbers_100 =
    SELECT (a.Value*10 + b.Value) AS Value
    FROM @numbers_10 AS a CROSS JOIN @numbers_10 AS b;

@numbers_10000 =
    SELECT (a.Value*100 + b.Value) AS Value
    FROM @numbers_100 AS a CROSS JOIN @numbers_100 AS b;

@result =
    SELECT
        Value,
        @StartDate.AddDays( Value ) AS Date
    FROM @numbers_10000;

OUTPUT @result TO "/res.csv" USING Outputters.Csv(outputHeader:true);
```

Ultimately, it may be more convenient to store these ranges in a U-SQL Table rather than regenerating them every time in a script. This is easy with CTAS. The snippet below shows how to create a table using CREATE TABLE from a RowSet that contains these numbers. Notice that in this case CREATE TABLE does not require the schema to be specified. The scheme is inferred from the SELECT clause.

```
CREATE DATABASE IF NOT EXISTS MyDB;
DROP TABLE IF EXISTS MyDB.dbo.Numbers_10000;

CREATE TABLE MyDB.dbo.Numbers_10000
(
 INDEX idx
 CLUSTERED(Value ASC)
 DISTRIBUTED BY RANGE(Value)
) AS SELECT * FROM @numbers_10000
```

Now retrieving any desired range is simple by using SELECT on the table followed with a WHERE clause:

```
// get the range of numbers from 1 to 87
@a =
    SELECT Value
    FROM MyDB.dbo.Numbers_10000
    WHERE Value >=1 AND Value <= 87;
```