Scope Tutorial > Introduction

Author: Saveen Reddy

Date: 8/4/2015

# Table of Contents

[Table of Contents 1](#_Toc404523735)

[Introduction 1](#_Toc404523736)

[Links 1](#_Toc404523737)

[Need help? Have a question? 2](#_Toc404523738)

[About Scope 2](#_Toc404523739)

[Preparation 2](#_Toc404523740)

[Core Concepts 7](#_Toc404523741)

[How to Run or Compile a Scope Script Locally 9](#_Toc404523742)

[Introduction to Processing Rowsets 15](#_Toc404523743)

[Data types in Scope 16](#_Toc404523744)

[Selection and Filtering 16](#_Toc404523745)

[Scope Expressions 18](#_Toc404523746)

[Expressions in SELECT 18](#_Toc404523747)

[LINQ and Lambdas in Expressions 21](#_Toc404523748)

[Extension Methods in Expressions 22](#_Toc404523749)

[Order of Evaluation for Expressions 23](#_Toc404523750)

[Logical Operators in Expressions 25](#_Toc404523751)

[Sorting 26](#_Toc404523752)

[Grouping and Aggregation 28](#_Toc404523753)

[System-Defined Aggregates 31](#_Toc404523754)

[Regular Expressions 34](#_Toc404523755)

[Change Log 34](#_Toc404523756)

# Introduction

This tutorial will cover the basics of using the Scope language. We will focus on getting familiar with the language while running Scope scripts locally – on your machine. You do not need access to a Cosmos virtual cluster.

# Links

The latest version of this doc and the sample files are located here: <http://aka.ms/ScopeTutorial>

# Need help? Have a question?

Send email to Cosmos Discussion: [cosmdisc@microsoft.com](mailto:cosmdisc@microsoft.com)

To join Cosmos Discussion: <http://idwebelements/GroupManagement.aspx?Group=CosmDisc&Operation=join>

# About Scope

**Scope** is the Query Language for Cosmos.

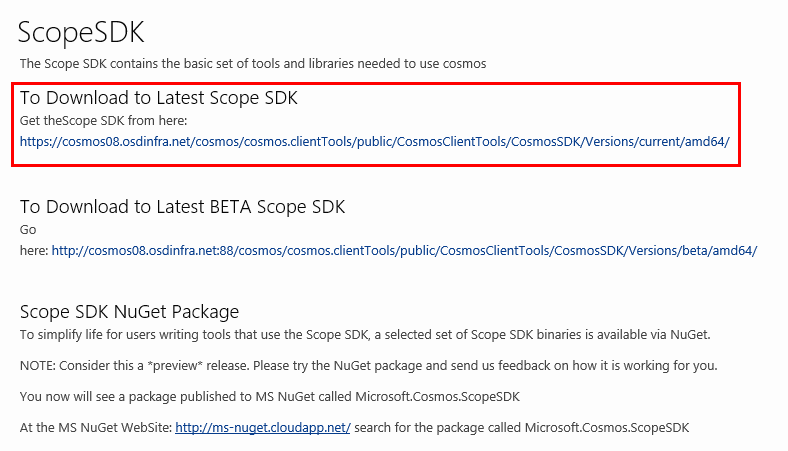
## Scope Is Not SQL

You will notice that Scope queries look at first glance similar to SQL queries. Many fundamental concepts and syntactic expressions will be very familiar to those with a background in SQL. Scope, however, is a distinct language and some of the expectations you might have from the SQL world do not carry over into Scope.

# Preparation

## Step 1: Download the Scope SDK (aka the Cosmos SDK)

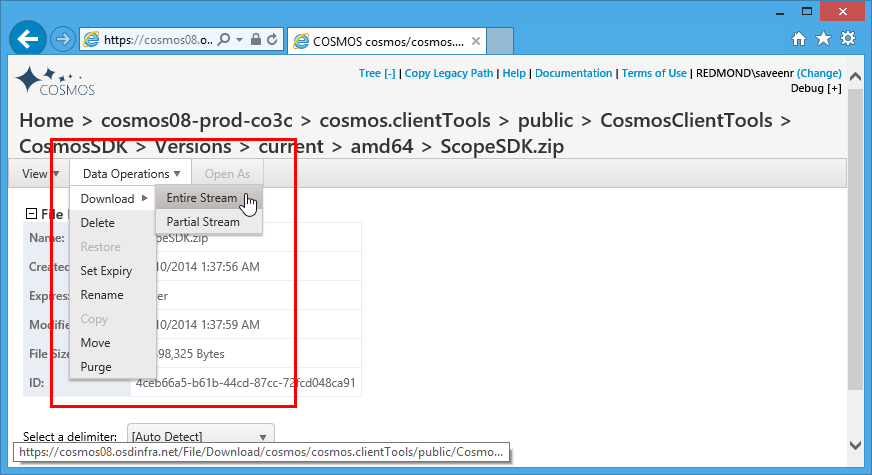
First, go to <http://aka.ms/ScopeSDK> then get the latest SDK by clicking on the link highlighted below:



That link is: <https://cosmos08.osdinfra.net/cosmos/cosmos.clientTools/public/CosmosClientTools/CosmosSDK/Versions/current/amd64/>

If you click on it you may be asked to authenticate and then you will see this page below click on ScopeSDK.zip

Then click on **Data Operations > Download > Entire Stream**

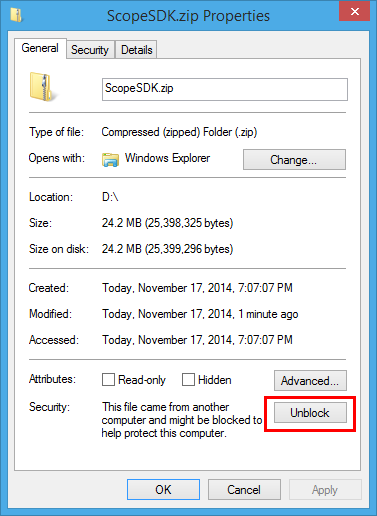


Place the ScopeSDK.zip somewhere. I usually place it at the root of the D:\ drive.

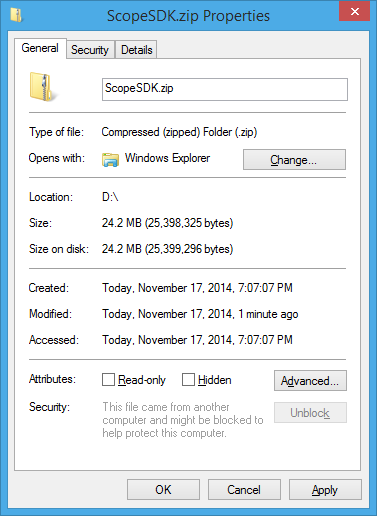
**DO NOT UNZIP IT YET**

Right click on the ScopeSDK.zip and select **Properties**.

In the **Properties** dialog if you see the **Unblock** button, then press it. If you don't see the **Unblock** button, then you are OK and there is nothing you need to do.

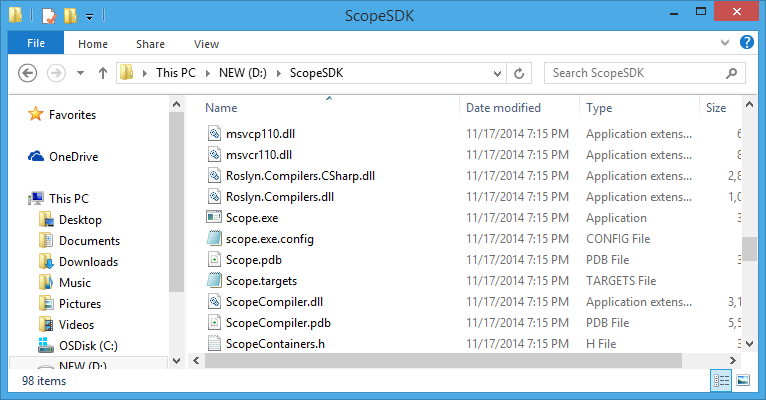


If you press the Unblock button, the properties window will look like this



Then press OK and you are finished downloading the Scope SDK.

Now, extract the ScopeSDK.zip. I usually extract it to d:\ScopeSDK so this is what you will see. Notice the **scope.exe** tool. You'll be using this a lot.



## Step 2: The CppSdk.zip

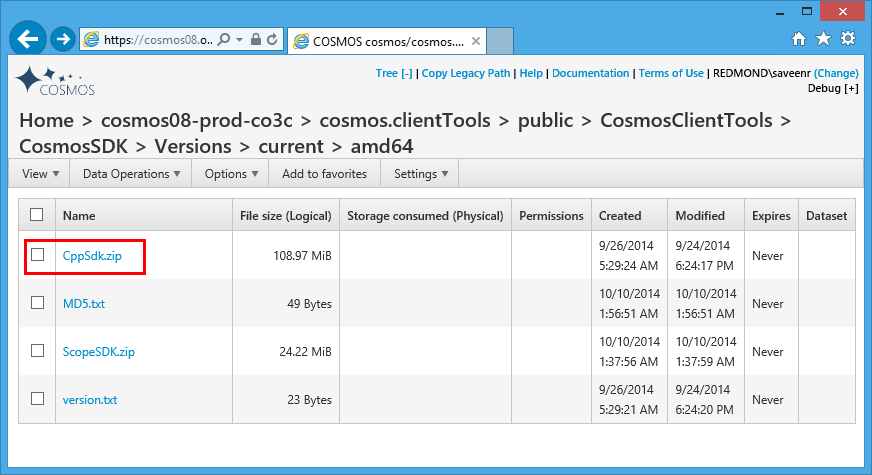
Scope actually uses the C++ compiler – the reasons why aren't important now – but they key point is that in order to successfully locally run a Scope script on your machine you'll need this SDK.

### Option 1: You Probably Already have the CppSdk installed

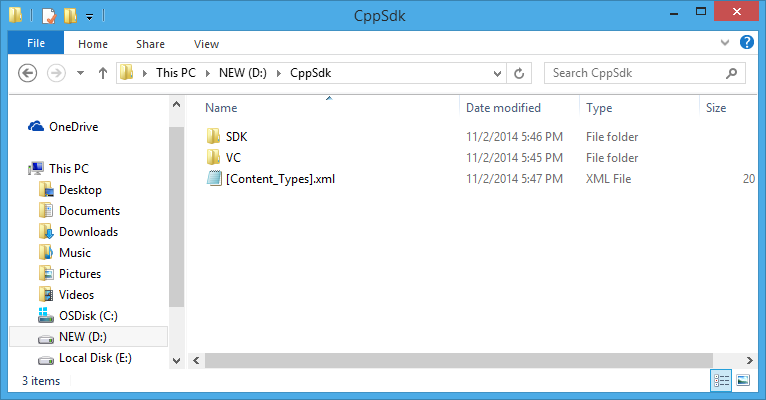
If you have Visual Studio 2012 installed then Scope will automatically use the CppSdk from that installation.

### Option 2: Manual Installation and Configuration of the CppSdk

If needed, you can download the exact CppSdk needed by Scope studio. It's in the same folder as the ScopeSDK.zip file.



I suggest you place this at the root of a drive and unzip it – I usually extract it to folder called **D:\CppSdk** which when extracted looks like this:



Then set the **Scope\_Cpp\_Sdk** environment variable to point to this location

## Step 3: Download the Cosmos Code Samples

Use Git to get the code samples <http://aka.ms/CosmosCodeSamples> - instructions on using Git are here

## To Compile and Execute Scope scripts requires an x64 System

* Unfortunate, but true.
* Use on a 64-bit OS.

# Core Concepts

## Inputs and Outputs

At a first level of approximation, a Scope script maps some input to some output via a processing step.



Inputs and outputs are **Streams** – essentially a file-like data structure that can be read from or written out. (**Views** can also be inputs. This will be covered later). In other words, streams are *persisted* entities. During processing an input stream is transformed into a **rowset.** Rowsets – similar to an intermediate table – are how Scope internally passes data during script execution**.** That rowset may be transformed to other rowsets. Finally a rowset may be persisted back out as a stream.

Imagine the simplest Scope script. It would perform an extract, create a single rowset, and then immediately write the rowset output. In other words, it merely copies a stream.

rs0 = EXTRACT

FirstName : string,

LastName : string,

Age : int

FROM

"/test\_input.tsv"

USING DefaultTextExtractor();

OUTPUT rs0

TO "/test\_output.tsv"

USING DefaultTextOutputter();

Key points:

* Because the stream in unstructured, the **EXTRACT** command explicitly names the columns and the types in the stream.
* The **OUTPUT** command sends the rows from rowset **rs0** to an output stream (in this case a local file).

PROTIP: Calling **DefaultTextExtractor** without any arguments is equivalent to **DefaultTextExtractor**( delimiter: 't' ) which explicitly identifies that the file is delimited by tabs.

PROTIP: Assume that all stream names are case-sensitive. Sure, when running locally it doesn't matter but when running in the cluster it does. Likewise, stick the "/" separator. It works both locally and for remote execution unlike "\" which only works on local execution.

## Scope Keywords are Upper-Case

Notice that some keywords are capitalized? For example: **EXTRACT**, **FROM**, and **USING**.

Scope takes capitalization very seriously, so writing **extract** instead of **EXTRACT** will cause a compilation error. Just remember to capitalize every Scope keyword in your scripts.

## Inputs

There are several kinds of inputs each with their appropriate use.

**Streams**Also referred to as *unstructured* streams, a stream is a persisted, file-like input. For example, any text file is a stream. Scope extracts data from streams and processes that data. Scope has the native ability to extract data from simple text streams (via the **DefaultTextExtractor**) which satisfies most scenarios. Developers can also build custom extractors to read data from other formats such as binary, JSON, or XML.

**Structured Streams**Structured streams are streams that are optimized for storage and access based on some unique properties (keys) of individual records. Such streams are more suitable for random data access patterns. They provide several advantages for the developer. First, structured streams are directly used without an extractor. Second, they offer greater performance than unstructured streams. Third, just as no extractor is needed, no outputter is needed.

**Views**A VIEW simplifies reading from streams. A great example of why you would use a VIEW is that it can take on the responsibility of reading multiple rowsets, doing all the joining, filtering, etc for you.

## Outputs

Processing creates an output rowset - or even multiple output rowsets. For this data to be consumed, the rowsets must also be persisted. Scope allows one to persist the output rowsets as either an (unstructured) Stream or a Structured Stream.

Just as there is a built in **DefaultTextExtractor**, Scope also provides a **DefaultTextOutputter** to persist simple streams. Again, this should satisfy most scenarios, but developers can implement their own **Outputter** should the need arise.

# How to Run or Compile a Scope Script Locally

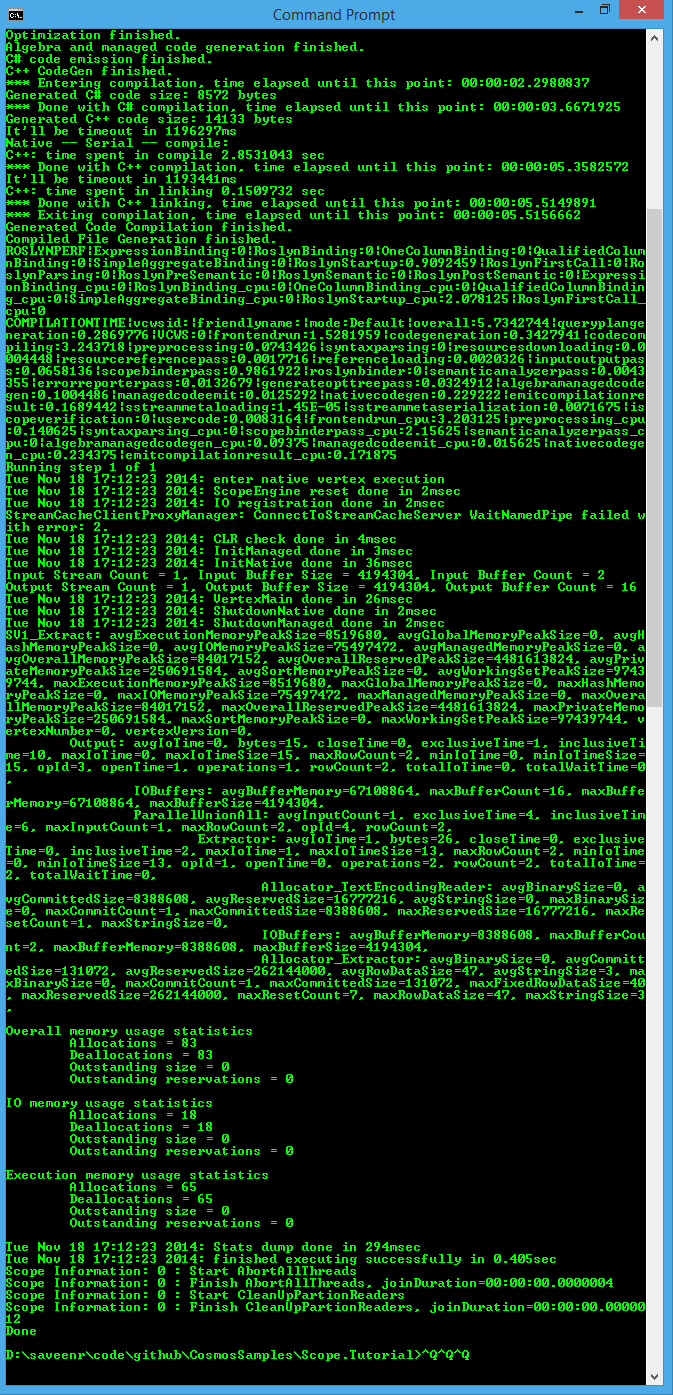
The **d:\CosmosSamples\Scope.Tutorial** folder contains a file called **test.script**. Let's see how to run it with **scope.exe**.

From the **cmd** prompt, then try the following

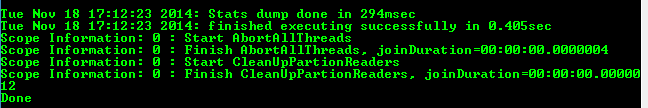
cd d:\CosmosSamples\Scope.Tutorial

d:\scopesdk\scope.exe run –i test.script

NOTE: this will create a \*LOT\* of output on the console. A successful execution would look like this:



In particular at the very end it will say "Done" for a successful result.



## Looking at Compiler Errors

Now that you've seen a successful compile. Let's see what an unsuccessful compile looks like.

Now we will deliberately introduce an error into the script.

rs0 = EXTRACT

FirstName : string,

LastName : **foobar**,

Age : int

FROM

"/test\_input.tsv"

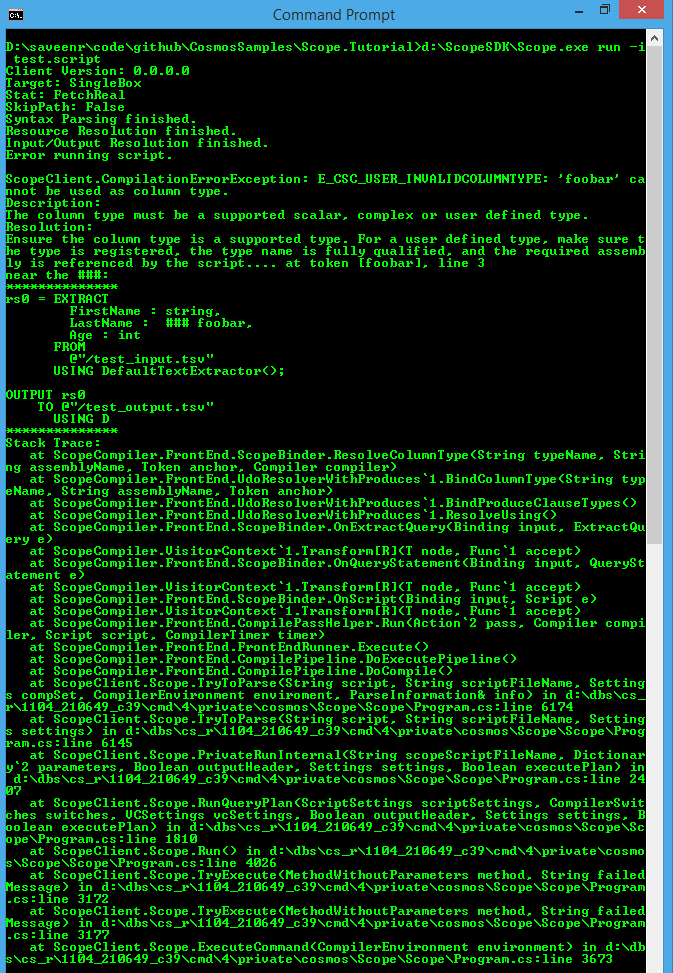
USING DefaultTextExtractor();

OUTPUT rs0

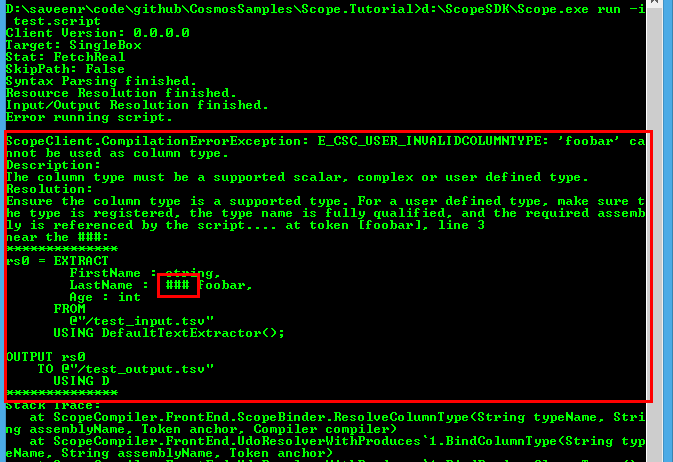
TO "/test\_output.tsv"

USING DefaultTextOutputter();

The result is:



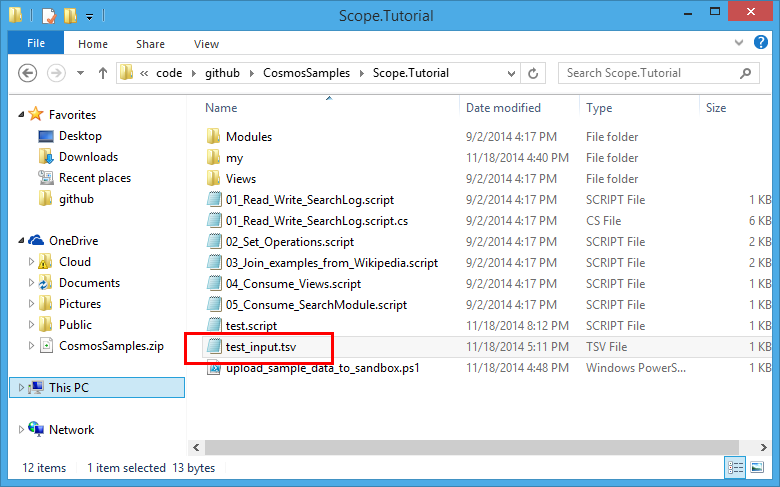
That's still long so let's zoom to the top:



You can see that there's an error code: **E\_CSC\_USER\_INVALIDCOLUMNTYPE** and the **###** sequence is used to draw your attention to the point in the script with the error.

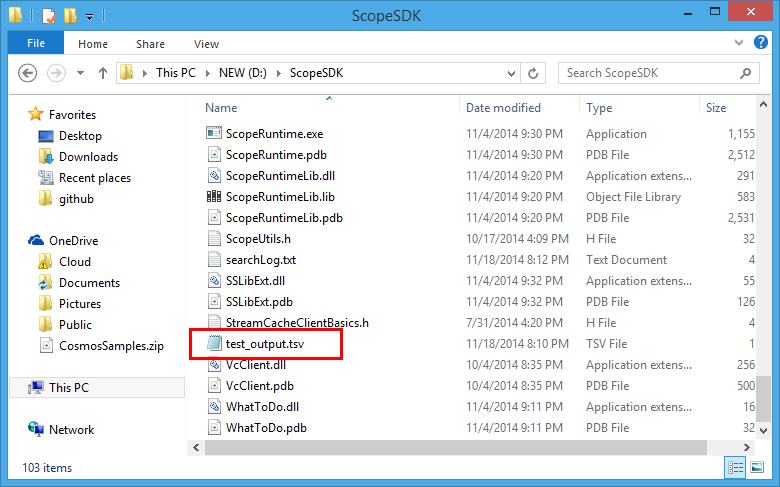
## Locating the Input and the Output

If you list the files in of **d:\CosmosSamples\Scope.Tutorial** you'll see the input file "test\_input.tsv"



But you've run the script successfully, where is the output?

By default the output will be placed in the same folder that scope.exe is located. You can verify this by looking at d:\scopesdk



That's really not what you'd want and doesn't match the behavior when a script is run in the cluster.

Fortunately for the run command the way to accomplish what you need comes in the form of several parameters to scope.exe

I've found the simplest thing to do for the tutorial is to set the both to the location of the tutorial files to the same location. In particular there is already a folder prepared for you that will mimic the root of a virtual cluster it is located at **d:\CosmoSamples\VCROOT**. To use this folder just specify it as shown below

d:\scopesdk\scope.exe run –i test.script

-INPUT\_PATH %CD%;d:\CosmosSamples\VCROOT

-OUTPUT\_PATH d:\CosmosSamples\VCROOT

-workingRoot %TEMP%

-RESOURCE\_PATH %CD%;d:\CosmosSamples\VCROOT

## Compiling a Script without executing it

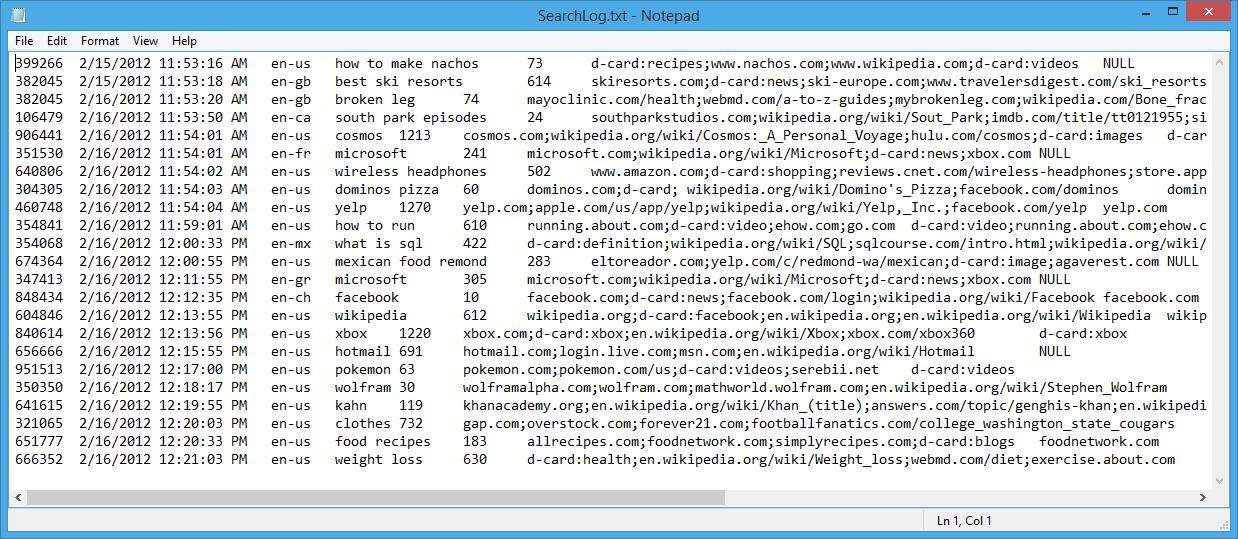
If you just want to verify that the Scope script compiles correctly – perhaps to check the syntax then the command looks like this

d:\scopesdk\scope.exe **compile** –i test.script *<Add the other parameters here>*

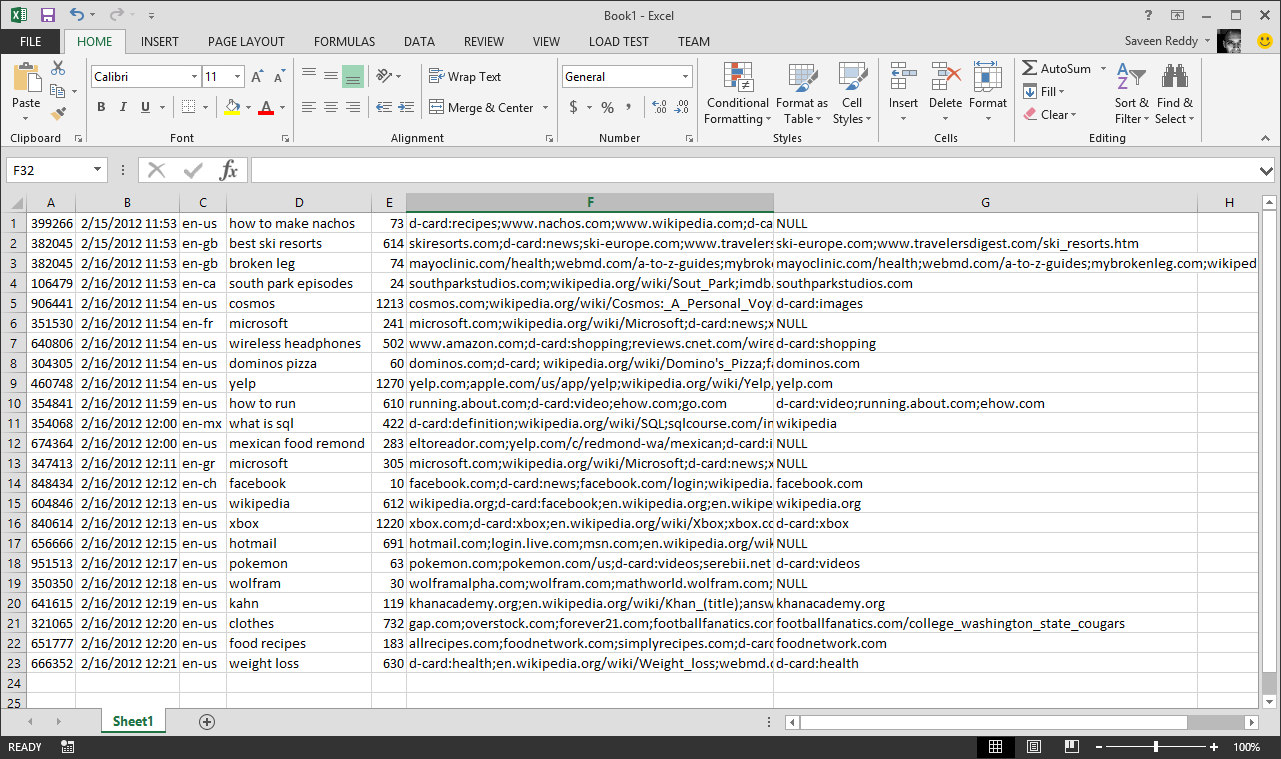
## The SearchLog Sample Data

There are quite a few sample data sets in the Scope Tutorial. The one we will initially work with is called "/my/SampleData/SearchLog.txt". It is slightly more complex than the rest and is useful for demonstrating many aspects of Scope so we'll spend some time understanding it.

The SearchLog sample dataset is represents user activity on a search engine. It is a simple text file you can open in Notepad.



If you were to load this into excel it would look like this:



This file doesn't contain a header row so we'll have to document the columns below:

* **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)

To EXTRACT data from this file, we would use this Scope statement

searchlog =

EXTRACT UserId : int,

Start : DateTime,

Region : string,

Query : string,

Duration : int,

Urls : string,

ClickedUrls : string

FROM "/my/SampleData/SearchLog.txt"

USING DefaultTextExtractor();

## The SearchLog View

In Scope for unstructured streams, you'll have to repeat that same extract statement over and over again. However, the Scope Tutorial comes with a VIEW – a topic we'll cover in greater depth later - that lets you avoid having to type this into every script.

See how simple this becomes with a view.

**searchlog = VIEW "Views/SearchLog.view";**

# Data types in Scope

## Native Data types

Scope supports most .NET data types. The full list is shown below.

|  |  |  |
| --- | --- | --- |
| **Category** | **Data type** | **Nullable Data type** |
|  | |  |  |
| **Miscellaneous** | | bool  Guid  datetime  byte[] | bool?  Guid?  datetime? |
| **Numeric** | | byte  sbyte  int  uint  long  ulong  float  double  decimal  short  ushort | byte?  sbyte?  int?  uint?  long?  ulong?  float?  double?  decimal?  short?  ushort? |
| **Text** | | char  string | char? |
|  | |  |  |
| **Complex** | | MAP<k,v>  ARRAY<v> |  |

## User-Defined Data Types

Scope also allows you to create your own user-defined types that you can use for columns in rowsets. Consult Part 2 of the Scope Tutorial to learn more.

# Selection and Filtering

The **SELECT** clause, as in SQL, allows you to pick the columns of interest.

searchlog = VIEW "Views/SearchLog.view";

// Find all the session dates, Durations

rs1 =

SELECT Start, Region, Duration

FROM searchlog;

OUTPUT rs1

TO "/my/output.txt";

And again, as with SQL, the **WHERE and HAVING** clauses allow you to filter data. Logical operators such as **AND** and **OR** are supported.

searchlog = VIEW "Views/SearchLog.view";

// Find all the sessions in the en-gb region

rs1 =

SELECT Start, Region, Duration

FROM searchlog

WHERE Region == "en-gb";

OUTPUT rs1

TO "/my/Outputs/output.txt";

Notice the use of "==" in the example above instead of "=". This is because expressions in the **SELECT** statement are true C# expressions.

The **AND** operator and **OR** operator can be combined with parentheses for more complex expressions

// Find all the sessions lasting between 2 and 5 minutes OR that are in the en-gb Region

rs1 =

SELECT Start, Region, Duration

FROM searchlog

WHERE (Duration >= 2\*60 AND Duration <= 5\*60) OR (Region == "en-gb");

OUTPUT rs1

TO "/my/Outputs/output.txt";

Because there is no C# literal for the **DateTime** type we have to make use of the **DateTime.Parse()** method as shown below.

// Find all the sessions occurring between two dates

rs1 =

SELECT Start, Region, Duration

FROM searchlog

WHERE Start >= DateTime.Parse("2012/02/16") AND Start <= DateTime.Parse("2012/02/17");

OUTPUT rs1

TO "/my/Outputs/output.txt";

# Refining RowSets

A rowset can be created from itself – this allows you to refine a rowset one statement at a time which may be useful for debugging or simply to make your script easier to use.

rs1 =

SELECT Start, Region, Duration

FROM searchlog;

rs1 =

SELECT \*

FROM rs1

WHERE Start >= DateTime.Parse("2012/02/16") AND Start <= DateTime.Parse("2012/02/17");

# Scope Expressions

## Understanding Scope Expressions

Clauses such as SELECT, WHERE, and HAVING (and others) allow you to enter **expressions** – in particular **Scope 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 Scope expression is that it is a merely C# expression with some Scope extensions such as the AND, OR, NOT operators.

## A Tip for SQL Developers

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

A SQL developer will expect to write the following in Scope

rs1 = SELECT UPPER( Region ) AS NewRegion

FROM searchlog;

But will be disappointed to find out that Scope has no **UPPER()** method. The C# developer knows what to: just use the string type's intrinsic **ToUpper()** method.

rs1 =

SELECT

Region.ToUpper() AS NewRegion

FROM searchlog;

# Expressions in SELECT

Specific columns can be picked for the output rowset. In the example below two columns are retrieved and a third is calculated. Note that when an expression is used to calculate a value then you must assign that column a name via the **AS** keyword.

rs1 =

SELECT

Start,

Region,

Duration + 1.0 **AS Duration2**

FROM searchlog;

## Type Casting

Expressions can also be converted to a different type

rs1 =   
 SELECT

Start,

Region,

**((double) Duration) AS DurationDouble**

FROM searchlog;

## Using .NET Types

Rowset columns are strongly typed. Scope allows you to call methods defined on those types in the **SELECT** clause.

// Find what day of year each session took place

rs1 =   
 SELECT

Start,

Region,

Start.DayOfYear AS StartDayOfYear

FROM searchlog;

## Creating New Objects with Constructors

You can use standard C# expressions to create new objects

rs1 =

SELECT

Foo,

new MyType( Bar ) AS Beer

FROM data;

There's even support for using Type Initializers with constructors:

rs1 =

SELECT

Foo,

new MyType { MyProperty=Bar } AS Beer

FROM data;

## Creating .NET Collections

Similar to the ability to create new objects, even collections can be created and initialized.

rs1 =

SELECT

Foo,

**new List<int> {1,2,3} AS Beer**

FROM data;

rs2 =

SELECT

Foo,

**new int[] {1,2,3} AS Beer**

FROM data;

rs3 =

SELECT

Foo,

**new [] {1,2,3} AS Beer**

FROM data;

## #CS Blocks and User-Defined Functions

A calculation can be implemented in C# code then later used in an expression. The code can be stored in a separate DLL or as part of a #CS block as shown below

rs1 =   
 SELECT

Start,

Region,

MyHelper.SecondsToMinutes(Duration) AS DurationInMinutes

FROM searchlog;

OUTPUT rs1

TO "/my/output.txt";

#CS

public static class MyHelper

{

public static double SecondsToMinutes(int seconds)

{

double minutes = seconds/60.0;

return minutes;

}

}

#ENDCS

## Filtering on Calculated Columns in SELECT: WHERE versus HAVING

As in SQL, keep in mind that **WHERE** operates on input rows and **HAVING** on output rows. Consider a case in which we've used an expression to create a new column:

rs1 =   
 SELECT Start, Region, Duration/60.0 AS DurationInMinutes

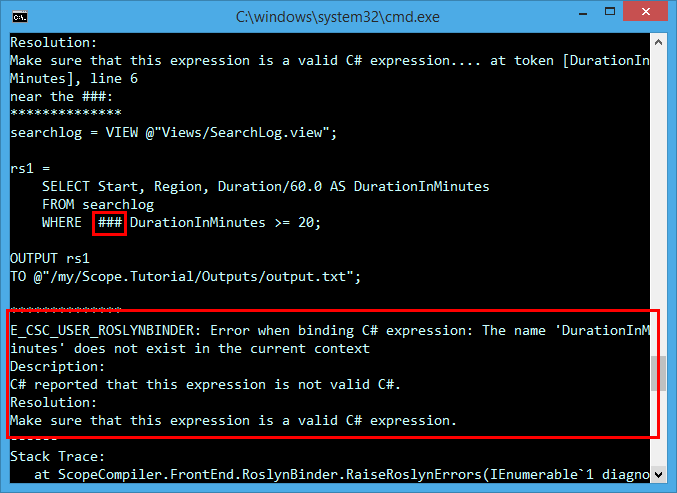
FROM searchlog

WHERE DurationInMinutes >= 20;

OUTPUT rs1

TO "/my/Outputs/output.txt";

This doesn't work even though the intention seems very clear. The code above will result in this error:



This is a concrete example of the difference between the **WHERE** and **HAVING** clauses. DurationInMinutes is NOT part of the input rows, it is created as part of the output rows. Thus to achieve filtering of these expressions we must use the **HAVING** clause.

rs1 =   
 SELECT

Start,

Region,

Duration/60.0 AS DurationInMinutes

FROM searchlog

HAVING DurationInMinutes >= 20;

Alternatively, you could use a new rowset to achieve the same effect.

rs1 =   
 SELECT

Start,

Region,

Duration,

Duration /60.0 AS DurationInMinutes

FROM searchlog;

rs2 =   
 SELECT \*

FROM rs1

WHERE DurationInMinutes >= 20;

# LINQ and Lambdas in Expressions

Scope supports **Language-Integrated Query** (LINQ) in expressions.

In the example below, the **Where()** LINQ extension method is used to filter for a particular set of URLs. **Where()**, like most LINQ methods, returns an **IEnumerable<T>** value. Because Scope does not support interfaces as column types, the **ToList()** LINQ extension method is used to convert it to a collection type that Scope does support.

rs1 =   
 SELECT

Urls.Split(';').Where(u=> u.StartsWith("http:")).ToList() AS HttpUrls

FROM searchlog;

NOTE: **DefaultTextOutputter** does not automatically serialize Collection types such as List<T>. You'll need to convert this back to a type that it supports such as a string as shown below.

rs1 =   
 SELECT

Urls.Split(';').Where(u=> u.StartsWith("http:")).ToList() AS HttpUrls

FROM searchlog;

rs2 =

SELECT

string.Join( ";" , HttpUrls) AS HttpUrls

FROM rs1;

NOTE: a **REFERENCE** to **System.Linq** is automatically added to your scripts

To learn more about what can be expressed in link consult: [**101 LINQ Samples**](http://code.msdn.microsoft.com/101-LINQ-Samples-3fb9811b)

## Anonymous Types Are Not Supported

Creating anonymous types is valid C#, however it is NOT supported in Scope. For example the following script will not work:

rs1 =

SELECT

Foo,

new {a=1, n=2} AS Beer

FROM data;

## LINQ Query Syntax

The **LINQ Method Syntax** was demonstrated above, but Scope also supports the **LINQ Query Syntax**.

rs =   
 SELECT

(from u in urls where u.StartsWith("http:")).ToList() AS HttpUrls

FROM clicks;

To learn more go here: <http://msdn.microsoft.com/en-us/library/vstudio/bb397947.aspx>

# Extension Methods in Expressions

You can also create your own custom extension methods and use them in a Scope expression. Simply create an extension method in a separate C# DLL and then reference the DLL from your Scope script.

rs =   
 SELECT

urls.GetHttpUrls() AS HttpUrls

FROM clicks;

// This must be present in a separate DLL

public static class MyExtensions

{

public static List<string> GetHttpUrls(this IList<string> urls)

{

return urls.Where(u=> u.StartsWith("http:")).ToList();

}

}

In the future, support may be enabled for extension methods in the #CS code blocks

## Testing For Membership with the IN Operator

You can use the **IN** operator as shown below to test for membership in a set of values.

rs =

SELECT FirstName, LastName, JobTitle

FROM People

WHERE JobTitle IN ("Design Engineer", "Tool Designer", "Marketing Assistant");

Keep in mind that the Scope **IN** operator does not offer all the features of the SQL **IN** operator.

See [IN Operator Documentation here](https://microsoft.sharepoint.com/teams/Cosmos/Wiki/IN%20Operator.aspx)

# Order of Evaluation for Expressions

**Read this section. No joke, it's really important.**

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 Scope 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 Scope 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 Scope 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 &&/||.

**You CANNOT circumvent the order of evaluation by using multiple 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

***First*** *I'll get the non-null objects and* ***then*** *I can avoid the null reference issue.*

This won’t work either. Scope 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. Scope reserves the right to combine multiple statements together and perform optimizations. **You MUST use the && operator if you want to perform short-circuiting.**

# Finding Distinct Rows with DISTINCT

rs1 =   
 SELECT DISTINCT

Region

FROM searchlog;

|  |
| --- |
| en\_ca |
| en\_ch |
| en\_fr |
| en\_gb |
| en\_gr |
| en\_mx |
| en\_us |

# Logical Operators in Expressions

## The C# Logical Operators: ||, &&, and !

To make it more straightforward for SQL developers, the equivalents to the C# logical operators are supported as shown below

|  |  |  |
| --- | --- | --- |
|  | **C#** | **SQL** |
| **Logical AND with short-circuiting** | (a && b && c) | ALL( a, b, c) |
| **Logical OR with short-circuiting** | (a || b || c) | ANY( a, b, c) |
| **Logical NOT** | !a | NOT(a) |

We still prefer you to use the C# expressions over the SQL style.

## The Conditional Operator and Ternary IF

Just like C, C++, and C#, Scope has the [conditional operator](file://sharepoint/DavWWWRoot/sites/cosmos/Cosmos%20Reference%20Materials/Scope-Tutorial/Older), whose form is shown below:

cond ? a : b

So, for example, it is possible to write:

rs1 =

SELECT

Region,

**(Duration>300 ? "long" :"short") AS DwellType**

FROM searchlog;

Scope has as another way of expressing this same construct with its **TERNARY IF** operator.

The general form for **TERNARY IF** is

IF (<cond>, <a>, <b> )

An example is below:

rs1 =

SELECT

Region,

**IF (Duration>300, "long", "short") AS DwellType**

FROM searchlog;

There's really no difference in using the conditional operator versus TERNARY IF, but the Scope team would prefer you to use the conditional operator as this is a standard C# expression.

PROTIP: Enclose :? In parenthesis

Consider the following two expressions – they are not equivalent:

// Case 1

IF(c1, true, false) AND IF(c2, true, false)

// Case 2

c1 ? true : false AND c2 ? true : false

The reason is that with C# precedence rules, the second expression will parse as:

c1 ? true : (false AND c2 ? True : false)

To match the original Case, use parentheses.

(c1 ? True : false) AND (c2 ? True : false)

# Sorting

You can sort rowsets by using the **ORDER BY** operator. Specifying the **ASC** and **DESC** keyword controls whether the sort is ascending or descending, respectively.

// List the sessions in increasing order of Duration

rs1 =

SELECT

Start,

Region,

Duration

FROM searchlog

ORDER BY Duration ASC;

OUTPUT rs1

TO "/my/Outputs/output.txt";

// List the sessions in decreasing order of Duration

rs1 =

SELECT

Start,

Region,

Duration

FROM searchlog

ORDER BY Duration DESC;

OUTPUT rs1

TO "/my/Outputs/output.txt";

# Getting the Exactly N Rows or the TOP N based on a Column

You can use the TOP operator to limit the number of rows you get

rs1 =   
 SELECT TOP 5

Region,

Duration

FROM searchlog;

Without an **ORDER BY** clause though the above statement is simply fetching *any* 5 rows it can. This is often not what you want. Instead add an **ORDER BY** on some field as shown below.

rs1 =   
 SELECT TOP 5

Region,

Duration

FROM searchlog  
 ORDER BY Duration DESC;

# 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 Scope using ROW\_NUMBER because it is a popular topic.

@rs1 =

SELECT

ROW\_NUMBER() OVER ( ) AS RowNumber,

Start,

Region

FROM @searchlog

ORDER BY Start;

|  |  |  |
| --- | --- | --- |
| 1 | 2/16/2012 11:53:50 AM | en\_ca |
| 2 | 2/16/2012 12:12:35 PM | en\_ch |
| 3 | 2/16/2012 11:54:01 AM | en\_fr |
| 4 | 2/15/2012 11:53:18 AM | en\_gb |
| 5 | 2/16/2012 11:53:20 AM | en\_gb |
| 6 | 2/16/2012 12:11:55 PM | en\_gr |
| 7 | 2/16/2012 12:00:33 PM | en\_mx |
| 8 | 2/16/2012 12:15:55 PM | en\_us |
| 9 | 2/16/2012 12:13:56 PM | en\_us |
| 10 | 2/16/2012 12:17:00 PM | en\_us |
| 11 | 2/16/2012 12:18:17 PM | en\_us |
| 12 | 2/16/2012 12:19:55 PM | en\_us |
| 13 | 2/16/2012 12:20:03 PM | en\_us |
| 14 | 2/16/2012 12:20:33 PM | en\_us |
| 15 | 2/16/2012 12:21:03 PM | en\_us |
| 16 | 2/15/2012 11:53:16 AM | en\_us |
| 17 | 2/16/2012 11:54:01 AM | en\_us |
| 18 | 2/16/2012 11:54:02 AM | en\_us |
| 19 | 2/16/2012 11:54:03 AM | en\_us |
| 20 | 2/16/2012 11:54:04 AM | en\_us |
| 21 | 2/16/2012 11:59:01 AM | en\_us |
| 22 | 2/16/2012 12:00:55 PM | en\_us |
| 23 | 2/16/2012 12:13:55 PM | en\_us |

# 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.

rs1 =

SELECT

Duration

FROM searchlog;

OUTPUT rs1

TO "/my/Outputs/output.txt";

This creates a simple list of integers.

|  |
| --- |
| 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

rs1 =

SELECT

SUM(Duration) AS TotalDuration

FROM searchlog;

OUTPUT rs1

TO "/my/Outputs/output.txt";

|  |
| --- |
| 9981 |

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

// find the total Duration by Region

rs1 =

SELECT

Region,

SUM(Duration) AS TotalDuration

FROM searchlog

GROUP BY Region;

OUTPUT rs1

TO "/my/Outputs/output.txt";

|  |  |
| --- | --- |
| 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 > 200

rs1 =

SELECT

Region,

SUM(Duration) AS TotalDuration

FROM rs1

GROUP BY Region

HAVING TotalDuration > 200;

OUTPUT rs1 TO "/my/Outputs/output.txt";

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

// Count the number of total sessions.

rs1 =

SELECT

COUNT() AS NumSessions

FROM searchlog;

OUTPUT rs1

TO "/my/Outputs/output.txt";

|  |
| --- |
| 23 |

Count the number of total sessions by Region.

rs1 =

SELECT

COUNT() AS NumSessions,

Region

FROM searchlog

GROUP BY Region;

OUTPUT rs1

TO "/my/Outputs/output.txt";

|  |  |
| --- | --- |
| 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.

rs1 = 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;

OUTPUT rs1

TO "/my/Outputs/output.txt";

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| NumSessions:long | Region | TotalDuration:long | AvgDuration:double? | MaxDuration:int | MinDuration:int |
| 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 |

## A Note: Data types Coming from Aggregations

You should be aware of how some aggregation operators deal with data types.

For example, the input data type is double:

* SUM(double) -> double
* COUNT(double) -> long(int64)

But if the input data type is numeric (long/int/short/byte, etc.):

* SUM(type) -> long(int64)
* COUNT(type) -> long(int64)

## Where You Can Use Aggregates in a Query

Aggregates can ONLY appear in a SELECT clause.

## DISTINCT with Aggregates

Every aggregate function can take a **DISTINCT** qualifier.

For example

COUNT(DISTINCT x)

**DISTINCT** also works for user-defined aggregates.

MyAggregator(DISTINCT x,y,z).

# System-Defined Aggregates

Scope contains several common aggregation functions:

* ARGMAX
* AVG
* COUNT
* COUNTIF
* ANY\_VALUE
* FIRST
* LAST
* LIST
* MAX
* MIN
* SUM
* VAR \*
* STDEV \*

## Getting a value with ANY\_VALUE and FIRST

**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 where you don't care which value you receive as long as you get one.

rs1 =

SELECT

ANY\_VALUE(Start) AS FirstStart,

Region

FROM searchlog

GROUP BY Region;

**FIRST** is badly named. It does NOT guarantee you will receive the first value in a rowset. Instead it behaves exactly like **ANY\_VALUE**. Avoid using **FIRST**, instead use **ANY\_VALUE**.

rs1 =

SELECT

FIRST(Start) AS FirstStart,

Region

FROM searchlog

GROUP BY Region;

## There is one key difference between **ANY\_VALUE** and **FIRST**

* **ANY\_VALUE** on some type T will return T?
* **FIRST** on some type T will return T

## Conditionally Counting with COUNTIF

Sometimes we need to count things, but only if a certain condition holds. For example, let's start with getting the total sessions per Region.

rs1 =

SELECT

Region,

COUNT() AS NumSessions

FROM searchlog

GROUP BY Region;

Now we want to compare the total count with the count of sessions that had a large dwell time, more than 600 seconds. We can accomplish this via **COUNTIF**.

rs2 =

SELECT

Region,

COUNT() AS NumSessions,

COUNTIF( Duration > 600 ) AS NumLongSessions

FROM searchlog

GROUP BY Region;

|  |  |  |
| --- | --- | --- |
| Region | NumSessions:long | NumLongSessions:long |
| en-ca | 1 | 0 |
| en-ch | 1 | 0 |
| en-fr | 1 | 0 |
| en-gb | 2 | 1 |
| en-gr | 1 | 0 |
| en-mx | 1 | 0 |
| en-us | 16 | 8 |

## ARGMAX

The **ARGMAX** function is simple:

**ARGMAX(a, b)** = Find the row with the maximum value for column a, from that row return the value for b.

Imagine we have some employee data:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **FirstName** | **LastName** | **Tenure:int** | **Title** | **Department** |
| Joe | Smith | 3897 | Paralegal | Legal |
| Sally | Johnson | 8897 | CFO | Exec |
| Trent | Michaels | 43 | Intern | Engineering |
| Joshua | Phillips | 373 | Developer | Engineering |
| Alice | Edwards | 513 | Tester | Engineering |

What's the title of the employee who has the longest tenure? This is simply answered with ARGMAX.

rs0 =

EXTRACT

FirstName:string,

LastName:string,

Tenure:int,

Title:string

FROM "/my/SampleData/tenures.tsv"

USING DefaultTextExtractor();

rs1 =

SELECT

ARGMAX( Tenure, LastName ) AS MostTentured

FROM rs0;

If you OUTPUT rs1 you will get this:

|  |
| --- |
| **MostTentured** |
| Johnson |

**ARGMAX** like all the other aggregates works with **GROUP BY** letting us find the most tenured employee by department.

rs2 =

SELECT

Department,

ARGMAX( Tenure, LastName ) AS MostTentured

GROUP BY Department

FROM rs0;

|  |  |
| --- | --- |
| **Department** | **MostTentured** |
| Engineering | Edwards |
| Exec | Johnson |
| Legal | Smith |

rs3 =

SELECT

Department,

ARGMAX( Tenure, LastName ) AS MostTenured,

ARGMAX( Tenure, Tenure ) AS Tenure

GROUP BY Department

FROM rs0;

|  |  |  |
| --- | --- | --- |
| **Department** | **MostTenured** | **Tenure:int** |
| Engineering | Edwards | 513 |
| Exec | Johnson | 8897 |
| Legal | Smith | 3897 |

PROTIP: Instead of ARGMAX, use the ROW\_NUMBER() and DENSE\_RANK() Analytic Windowing Functions. They are more powerful.

## An Important Fact about VAR and STDEV

For the Statisticians, variance (**VAR**) and standard deviation (**STDEV**) are the **sample version** with Bessel's correction, **not** the better-known **population version**.

# Regular Expressions

Regular expressions provide advanced text matching capabilities in a terse specification. Regular Expressions are supported in the WHERE and HAVING clauses.

## Finding Simple Patterns

// Find all the sessions where the query contained the word pizza (but not pizzeria, for example)

rs1 =

SELECT

Start,

Region,

Duration

FROM searchlog

WHERE REGEX(@"\bpizza.\*\b").IsMatch(Query);

OUTPUT rs1

TO "/my/Outputs/output.txt";

## Extracting a REGEX Match

Sometimes you'll need to "pull" out a substring and promote that to a column.

For example, if there is a column called Name and its value can look like "--------Cosmos01------", "foooCosmos11bar," etc. and we want to pull out the "Cosmos<number>" parts, then REGEX and Scope make this pretty easy to do.

rs1 =

SELECT

Name,

REGEX(@"Cosmos[0-9]\*").Match(Name).Value AS CosmosCluster

FROM data;

NOTE: That the above example is case-sensitive, so it won't match "cosmos08" but will match "Cosmos08".

# Notes on Rowsets

## You cannot extract a single scalar value from a rowset

This is a common question for developers new to Scope. Often they want to store a value from a rowset in a scalar value. For example, they may want the MAX for some column in a rowset and then re-use that value somewhere else and will try to write code like this:

maxcost =

SELECT

MAX( Cost ) As MaxCost

FROM data;

There’s nothing wrong with this syntactically – it will compile. However, maxcost will not be a double it will be a rowset with a single row a single column called MaxCost.

There is no way in Scope to return a single scalar value like this. Your alternative is to get a one-row rowset and then combine (JOIN) that with another rowset to achieve the computation you want.

# Breaking Rows Apart with CROSS APPLY

Let's examine the search log again.

rs1 =

SELECT

Region,

Urls

FROM searchlog;

The query above returns something 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. What happens if we want to break apart the **Urls** field so that only a URL is present on every row? For example, below is what we want to see:

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

This is a perfect job for the **CROSS APPLY** operator.

rs1 =

SELECT

Region,

Urls

    FROM searchlog;

rs2 =

SELECT

Region,

**SplitUrls AS Url**

FROM rs1

**CROSS APPLY Urls.Split(';') AS SplitUrls;**

NOTE: The transformation above is possible to perform programmatically with **PROCESSORS** – but **CROSS APPLY** is always preferred to custom processors.

# CROSS APPLY with Multiple Columns

Occasionally you'll see data organized in the manner shown below:

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

In this structure each value in the Region column is a set of values that correspond 1-to-1 with a value in the Urls column.

What you clearly want is to "zip" the pairs of values together to get this:

|  |  |
| --- | --- |
| Region | Result |
| en-us | A |
| en-us | B |
| en-us | C |
| en-gb | D |
| en-gb | E |
| en-gb | F |

With a little bit of extra code, **CROSS APPLY** works in this case also.

rs1 =

SELECT

Tup.Item1 AS Region,

Tup.Item2 AS Result

FROM rs0

CROSS APPLY Regions.Split(';').Zip(Result.Split(';'), Tuple.Create) AS Tup;

# Putting Rows Together with LIST

The **LIST** aggregate operator performs the opposite of **CROSS APPLY**.

For example, if we start with this:

|  |  |
| --- | --- |
| Region | Result |
| en-us | A |
| en-us | B |
| en-us | C |
| en-gb | D |
| en-gb | E |
| en-gb | F |

But we want this as the output:

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

This is exactly what the **LIST** operator does. In the example below you will see rowset r1 taken apart by **CROSS APPLY** and then reconstructed as rowset r3 via the **LIST** operator.

rs1 =

SELECT

Region,

Urls

FROM searchlog;

rs2 =

SELECT

Region,

SplitUrls AS Url

FROM rs1

CROSS APPLY Urls.Split(';') AS SplitUrls;

rs3 =

SELECT

Region,

string.Join(";" , LIST(Url).ToArray() ) AS Urls

FROM rs2

GROUP BY Region;

PROTIP: LIST offers no guarantees on order. So you may end up with C;A;B for example for en-us.

PROTIP: In general, Cosmos treats data homogenously with no respect for order unless explicit ordering instructions are provided. Keep this in mind anytime you find yourself relying on order.

# Putting Rows Together with ARRAY\_AGG

**ARRAY\_AGG** works a lot like **LIST** but is the preferred way – for reasons will get into – for merging lists of column values

rs1 =

SELECT

Region,

Urls

FROM searchlog;

rs2 =

SELECT

Region,

SplitUrls AS Url

FROM rs1

CROSS APPLY Urls.Split(';') AS SplitUrls;

rs3 =

SELECT

Region,

string.Join(";" , ARRAY\_AGG(Url) ) AS Urls

FROM rs2

GROUP BY Region;

## ARRAY\_AGG versus List

The general guidance is **always choose ARRAY\_AGG if possible** because it is more efficient and more optimizable than LIST.

The functional difference between **ARRAY\_AGG** and **LIST** is:

* **LIST** can handle \*any\* type
* **ARRAY\_AGG** can handle only the store native Scope types.

## ARRAY\_AGG versus Array<T> and List<T>

Array<T> and List<T> are collections from the **System.Collections** namespace. You are free to use them in scope, but again **always choose ARRAY\_AGG if possible** because it is more efficient and more optimizable than using the standard .Net collection types.

# Miscellaneous

## Writing Debug Information in C# Code

Sometimes you need to log output in your C# code. The correct way to do this is to use the Debug Streams feature.

Consult [this document for more information](https://microsoft.sharepoint.com/teams/Cosmos/Documents/Scope/Scope-User-Code-Debug-Streams.docx?Web=1):

## String Literals

This tutorial uses different kinds of string literals.

* The Regular C# String Literal
* The Verbatim C# String Literal – these string literals begin with a @ character

You may use either in Scope. The key difference are in the handling of embedded quotation marks, backslashes, and newlines as shown in the table below.

|  |  |  |
| --- | --- | --- |
|  | Regular | Verbatim |
| Simple string | "Foo" | @"Foo" |
| Quotation marks | "\"Hello\" I said" | @"""Hello"" I said" |
| Slashes | "a/b/c" | @"a/b/c" |
| Backslashes | "a\\b\\c" | @"a\b\c" |
| newlines | "a\r\nb\r\nc" | @"a  b  c" |

# Change Log

<https://microsoft.sharepoint.com/teams/Cosmos/Documents/Scope/Tutorial/Scope_Tutorial_Change_Log.docx>