

**SSIS for Azure and Hybrid Data Movement**

SQL Server Technical Article

**Writer:** Alexei Khalyako, Carla Sabotta, Daniel Sol, Sreedhar Pelluru, Steve Howard

**Technical Reviewer:** David Pless

**Published:** December, 2012

**Applies to:** SQL Server 2012

**Summary:** SQL Server Integration Services (SSIS) can be used effectively as a tool for moving data to and from Windows Azure SQL Database, as part of the total extract, transform, and load (ETL) solution and as part of the data movement solution. SSIS can be used effectively to move data between sources and destinations in the cloud, and in a hybrid scenario between the cloud and on-premise. This paper outlines best practices for using SSIS for cloud sources and destinations and for project planning for SSIS projects to be used with Azure or hybrid data moves, and gives an example of maximizing performance on a hybrid move by scaling out the data movement.

Copyright

This document is provided “as-is”. Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. You bear the risk of using it.

Some examples depicted herein are provided for illustration only and are fictitious.  No real association or connection is intended or should be inferred.

This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes.

© 2011 Microsoft. All rights reserved.

Contents

[Introduction 5](#_Toc343709774)

[Project Design 5](#_Toc343709775)

[Considerations and Options for Azure 5](#_Toc343709776)

[Sources and Destinations 5](#_Toc343709777)

[ODBC 6](#_Toc343709778)

[ADO.NET 8](#_Toc343709779)

[Sharding and Federations 8](#_Toc343709780)

[Importing to Federations 10](#_Toc343709781)

[SSIS Processes in Windows Azure 19](#_Toc343709782)

[Lifecycle of the SSIS VMs 20](#_Toc343709783)

[Location of SSIS Processes 20](#_Toc343709784)

[Azure to Azure Data Movement 21](#_Toc343709785)

[Hybrid Data Movement 21](#_Toc343709786)

[Dealing with Limited Bandwidth Scenarios 22](#_Toc343709787)

[Compressing Data 22](#_Toc343709788)

[Two-Stage Transfer 22](#_Toc343709789)

[Applying Scale-out Principles with WA 23](#_Toc343709790)

[Multiple Persistent VMs or SSIS Extraction Servers 23](#_Toc343709791)

[Queues 23](#_Toc343709792)

[Blob Storage 24](#_Toc343709793)

[Designing for Retry without Manual Intervention 24](#_Toc343709794)

[Incorporating Retry into a Package 24](#_Toc343709795)

[Incorporating Retry Logic in Custom Components 29](#_Toc343709796)

[Moving Retry when Building or Executing Programmatically 29](#_Toc343709797)

[Hybrid Data Movement with SQL Server running in WA VMs 29](#_Toc343709798)

[Strategies 30](#_Toc343709799)

[Load data directly into the WA VM. 31](#_Toc343709800)

[Using file transfer for moving data between the on premise server and the WA VM 32](#_Toc343709801)

[Tuning Best Practices for On-Premises and Hybrid Data Movement 41](#_Toc343709802)

[Network 41](#_Toc343709803)

[SSIS Package Data Flow 41](#_Toc343709804)

[Connection Drivers 42](#_Toc343709805)

[SQL Server Database (as destination) 42](#_Toc343709806)

[Solution: Scale-Out and Hybrid Data Movement 42](#_Toc343709807)

[Solution Requirements 43](#_Toc343709808)

[Solution Design 43](#_Toc343709809)

[Architecture 44](#_Toc343709810)

[Design Considerations 47](#_Toc343709811)

[Solution Implementation 47](#_Toc343709812)

[Hardware and Software Requirements 47](#_Toc343709813)

[Install SSIS on On-Premises and Cloud Machines 48](#_Toc343709814)

[Install Custom Compressed File Components 48](#_Toc343709815)

[Set up the SQL Azure Export and Import Queues 48](#_Toc343709816)

[Set up the SQL Azure Blob Storage 48](#_Toc343709817)

[Create Coordinator Application 48](#_Toc343709818)

[Create the Export Application 48](#_Toc343709819)

[Create Application to generate packages in the Cloud 49](#_Toc343709820)

[Using the Conditional Split Transformation to Shard Data 49](#_Toc343709821)

[Conclusion 49](#_Toc343709822)

[Appendix A – Example Coordinator Process 51](#_Toc343709823)

[Appendix B – Example Extraction Process 62](#_Toc343709824)

[Appendix C – Example Import Process 67](#_Toc343709825)

[Appendix D – Example Template Package 79](#_Toc343709826)

# Introduction

SQL Server Integration Services (SSIS) can be used effectively as a tool for moving data to and from Windows Azure SQL Database, as part of the total extract, transform, and load (ETL) solution and as part of the data movement solution. SSIS can be used effectively to move data between sources and destinations in the cloud, and in a hybrid scenario between the cloud and on-premise. This paper outlines best practices for using SSIS for cloud sources and destinations and for project planning for SSIS projects to be used with Azure or hybrid data moves, and gives an example of maximizing performance on a hybrid move by scaling out the data movement.

# Project Design

Projects that move data between cloud and on-premises data stores can involve diverse processes inside various solutions. There are often many pieces starting from the initial population of the destination, which may take data from another system or platform, through maintenance such as rebalancing datasets among changing numbers of partitions or shards, and possibly continuing with period bulk data operations or refreshes. The project design and underlying assumptions will often differ for a solution involving data in the cloud as compared to a traditional, fully on-premises data movement environment. Many learnings, experiences, and practices will still apply but changes are necessary to accommodate differences such as the fact that your environment is no longer self-contained and fully under your control as you move to a shared pool of commodity resources. These differences require a more balanced and scalable approach to be successful.

For a complete discussion of project design, please see the “SQL Server 2012 SSIS Operational and Tuning Guide” located in the MSDN library under the [Microsoft White Papers](http://msdn.microsoft.com/en-us/library/hh403491.aspx) node (http://msdn.microsoft.com/en-us/library/hh403491.aspx) for SQL Server 2012.

# Considerations and Options for Azure

The Windows Azure (WA) platform introduces several possibilities as well as challenges with SSIS. This section discusses some of these possibilities, and some options for dealing with common challenges. This section also discusses concepts to allow for scaling out a data transfer as a WA cloud solution.

## Sources and Destinations

The data sources or destinations located in WA Persistent virtual machines (VMs) can be used just as they would be when running in on-premises systems. However, not all on-premises solutions are supported for Windows Azure SQL Database. For example, Windows Azure SQL Database does not support OLE-DB, so when transferring data into or out of the Azure data platform, you should use either ODBC or ADO.NET connections.

### ODBC

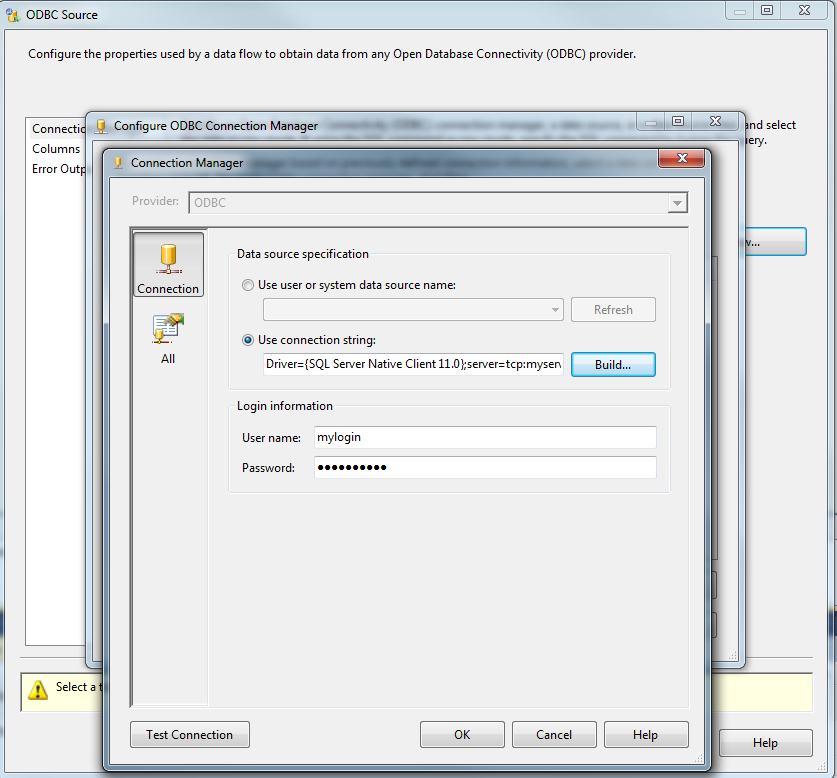
ODBC can be used with SQL Server 2012 drivers to connect to Windows Azure SQL Database. When creating a connection manager to connect to SQL Azure, do the following.

1. Choose **New** in the **Configure ODBC Connection Manage**r dialog to open the **Connection Manager** dialog.
2. Choose the option **Use connection String:**.
3. Create a connection string using the following format.

Driver={SQL Server Native Client 11.0};Server=tcp:myservername.database.windows.net;Database=mydbname;Uid=mylogin;Pwd=myPassword;

Replace “myservername,” “mydbname,” “mylogin” and “myPassword” with appropriate values.

1. Click **Build** button. The **User name** and **Password** boxes will be populated with the userid and password from the connection string.
2. Click **Test Connection** button to ensure you can make a connection.

****

A connection manager configuration using the Use connection string option in SSIS.

**NOTE:** Ensure that the Windows Azure SQL Database firewall rules are set for your server to allow connections from your IP address. You will not be able to connect if a firewall rule does not exist to allow the connection from your IP address. See [Windows Azure SQL Database Firewall](http://msdn.microsoft.com/en-us/library/windowsazure/ee621782.aspx) (<http://msdn.microsoft.com/en-us/library/windowsazure/ee621782.aspx> ) for information on Windows Azure SQL Database Firewall and instructions on how to set firewall rules.

**NOTE:** Ensure you use the Database attribute when building your connection string. If you do not specify a database, you will connect to the Master database. The [USE](http://msdn.microsoft.com/en-us/library/ms188366.aspx) command to change database context is not supported in SQL Azure.

### ADO.NET

ADO.NET can be used as a source or destination for SQL Azure. When configuring the ADO.NET connection manager for use with SQL Azure, select **Use SQL Server Authentication** in the **Connection Manager** dialog box.

## Sharding and Federations

While large databases may reside in a single database in many on-premises solutions, size and manageability limitations and a need to scale out data sources leads to sharded or federated database architectures for databases in Windows Azure SQL Database. Functional or vertical sharding is accomplished by placing different groups of related tables into different databases. Horizontal sharding is accomplished by placing different rows from the same table into different databases based on a sharding key. For more information on sharding, see [How to Shard with Windows Azure SQL Database](http://social.technet.microsoft.com/wiki/contents/articles/1926.how-to-shard-with-windows-azure-sql-database.aspx) (http://social.technet.microsoft.com/wiki/contents/articles/1926.how-to-shard-with-windows-azure-sql-database.aspx).

Vertical sharding is accomplished in SSIS by simply configuring the destinations to point to the correct database. Horizontal sharding is accomplished in SSIS with one pass through the data source, by using a conditional split to send the data to the correct shard, and having a connection manager for each destination database.

Two types of sharding logic are typically used for horizontal sharding:

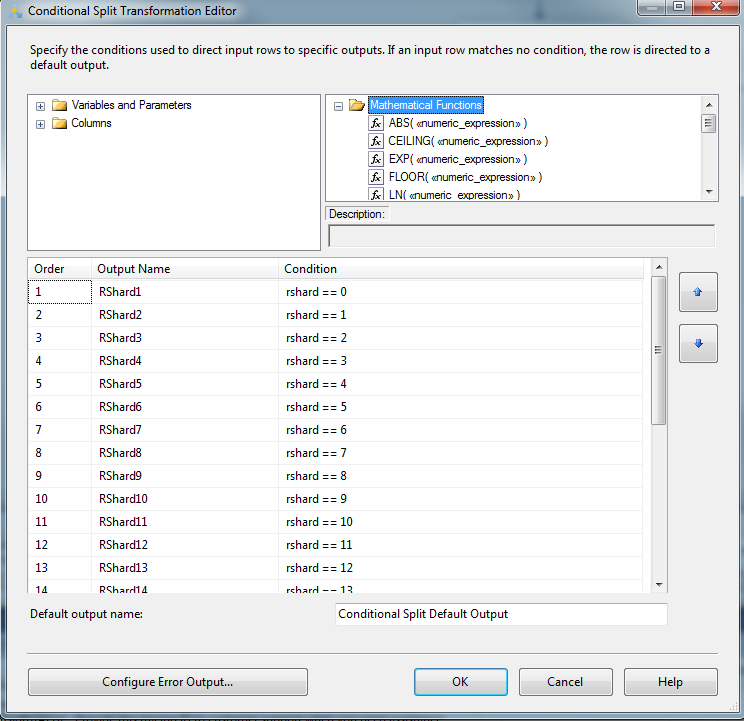
* A single value indicates the shard to which a row belongs.
* A range of key values is stored in each shard.

For the single value, the value may exist in the data that SSIS receives, or the sharding key may be calculated inside the data flow in SSIS. For ranges of values, the value typically is in the source data. For both range sharding and single value sharding, expressions are used in a conditional split in the data flow to direct each row to the proper destination.

An example of a single value for sharding is one that will use the HASHVALUE function in SQL Server or .NET code for use on GUID primary keys. For such a move, you would use SQL command for the data access mode in your SSIS ADO.NET source or ODBC source component and enter a query. An example of a query could be:

SELECT CAST(ABS(HASHBYTES(N'SHA1', CAST(USERID AS BINARY(16))) % 25)AS SMALLINT) as rshard, \* FROM SourceTable

In this example, the data will be distributed among 25 shards, so the modulo operation “%25” is used to distribute the data among the shards. The possible values for rshard in this query is 0 – 24. To distribute the data in the data flow, you would configure a conditional split as shown in the following image.



Each of the 25 outputs in this example will flow to a separate data destination. Each destination is using a separate connection manager that is configured to point to the database where its shard is located. For example, the conditional split output RShard1 will be connected to a destination named RShard1 that is configured to use a connection manager named RShard1. The connection manager points to a database named RShard1 that contains only those records with the sharding key value of 0. Note that in this case, it is not necessary to store the sharding key as the application can calculate it prior to access.

The advantage of single value sharding is its ease for importing or for calculation in the application. The disadvantage is that changing the number of shards, if a modulo is used, either introduces complexity in the calculation, or it makes it necessary to redistribute data across all shards. There is not an easy way to split a single shard. For ease of splitting single shards for scaling needs, you use range sharding. An example of range sharding is the Windows Azure SQL Database Federations implementation.

### Importing to Federations

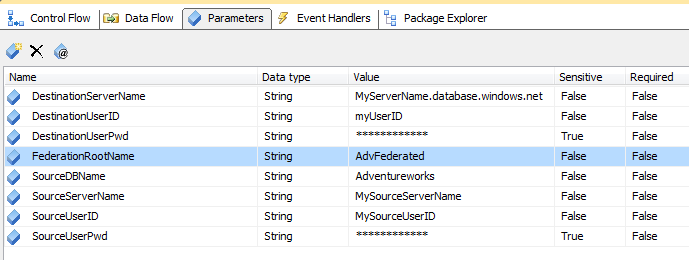
Federations shard data by defining range boundaries. Applications use federations by connecting to the root database, and issuing a [USE FEDERATION](http://msdn.microsoft.com/en-us/library/windowsazure/hh597457.aspx) (http://msdn.microsoft.com/en-us/library/windowsazure/hh597457.aspx ) command. SSIS can connect to the ROOT database, but before any bulk insert operations can be done, it must be connected to the correct federation member. Federation members are different databases that are created as federation boundaries are defined. Federation member database names are not used by most applications because they follow the paradigm of connecting to the root database, and issuing the USE FEDERATION command to direct the query to the correct federation member. In the case of SSIS, it is helpful to use a little different method to allow you to connect to and use federations, and still make your package reusable with a minimal amount of updating when moving from one environment to another.

In general, to import into a Windows Azure SQL Database federation with SSIS, do the following.

1. Define a package parameter to hold the federation root database name.
2. Define package parameters to hold the following.
   1. The connection and login information, and any source connection information necessary.
   2. The federation name to be used.

**NOTE:** Set the **Sensitive** property to **True** for parameters that contain passwords, or other connection information you consider to be sensitive. This prevents unauthorized viewing of any sensitive connection information.

When you are done, the parameters should look like this example:



Replace the parameter values with values to connect to your severs, or provide parameter values at execution to allow for connection to your system.

1. Define String variables for the following items.
   1. For each federation member that data can be imported to.

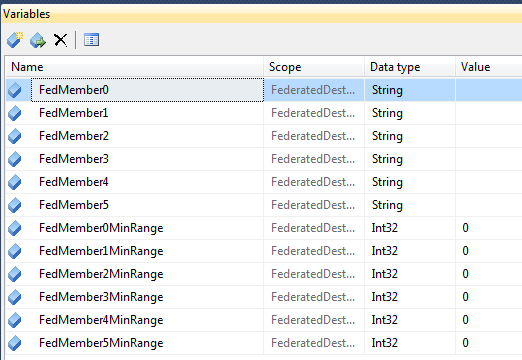
Number these sequentially for easy access in later steps. For example, use a naming convention such as “FedMember0,” “FedMember1,” “FedMember2,” continuing to the highest sequence of members that can be used.

**NOTE:** You will need to populate the values for these variables with a database that contains the destination table schema. The SSIS designer will need to connect to the database during design to retrieve the metadata. You can populate all variable values with the same name – the values will be changed by the Script task at runtime to send the data to the correct federation members.

* 1. For the minimum range boundary for each federation.

You can build this off of your member names above with a naming convention such as “FedMember0MinRange,” “FedMember1MinRange,” “FedMember2MinRange,” continuing to the highest sequence of members that can be used.

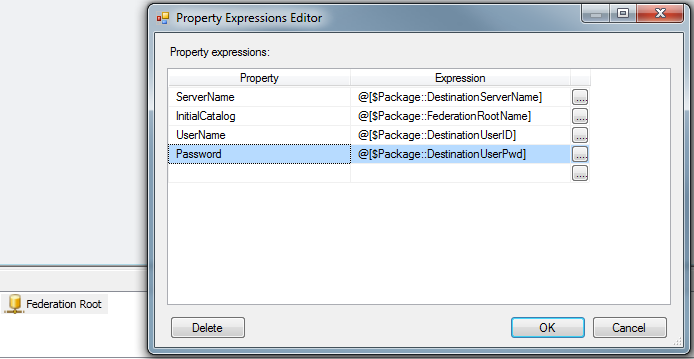
Building on the example above, with a federation with 6 members, you would have the following variables.



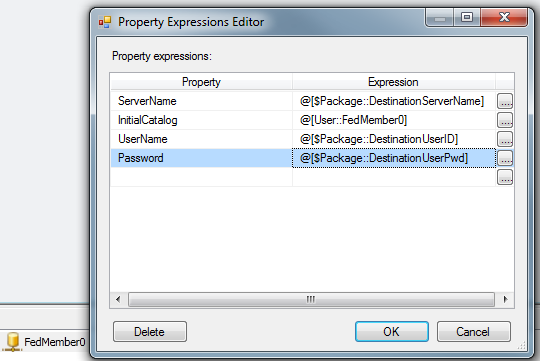
1. Create one connection manager for the root database of the federation, one for the source database, and one for each federation member database. Use expressions to set either the individual properties needed to log in and connect to the individual federation members, or use an expression to define the full connection string to connect to these federation members. Use the string variables you created in step 3 to define the “Initial Catalog” in the connection string, or the database name in the connection properties.

**NOTE:** For more information on property expressions including how to set property expressions, see [Use Property Expressions in Packages](http://technet.microsoft.com/en-us/library/ms141214.aspx) (http://technet.microsoft.com/en-us/library/ms141214.aspx).

The following example shows how to use expressions to set the connection properties for the root database of the federation, using the package parameters defined in Step 2.



The following example shows the same for the federation member database.



**NOTE:** For the InitialCatalog property, a package variable is used instead of a package parameter because at the time of package creation, you may not know the name of the federation member database. Using a package variable allows for this property to be set dynamically at runtime.

1. Although several approaches can be taken to populate the variables, this example will use a Script task. This allows everything to be set up in a single task. To use a Script task, do the following.
   1. Drag a script task to the control flow. Name it appropriately, and double-click the task to open the **Script Task Editor**.
   2. In the **ReadOnlyVariables** box, click the ellipsis button and select the package parameters you set up for the destination server name: destination federation root name, federation name, destination user id, and destination user password. After selecting each parameter and clicking **OK** in the **Select Variables** dialog box, the following appears in the **ReadOnlyVariables** box.

$Package::DestinationServerName,$Package::DestinationUserID,$Package::DestinationUserPwd,$Package::FederationName,$Package::FederationRootName

* 1. In **ReadWriteVariables** box, click the ellipsis button and select the user variables you set up for the federation member names, and the federation member minimum ranges. After selecting the variables and clicking **OK** in the **Select Variables** dialog box, the following appears in the **ReadWriteVariables** box.

User::FedMember0,User::FedMember0MinRange,User::FedMember1,User::FedMember1MinRange,User::FedMember2,User::FedMember2MinRange,User::FedMember3,User::FedMember3MinRange,User::FedMember4,User::FedMember4MinRange,User::FedMember5,User::FedMember5MinRange

* 1. Click **Edit Script**.
  2. When VstaProjects opens, click the + near the top to the left of the word “Namespaces” to expand the code region. Add this line to the region:

using System.Data.SqlClient;

* 1. For this example, with the variables given, Main should read as:

public void Main()

{

// connect to the federation root and begin building the federation map:

    // first, set up the connection and command to get info from the root:

    SqlConnection conn = new SqlConnection();

    conn.ConnectionString =

        String.Format("Data Source={0};Initial Catalog={1};User Id={2};Password={3};"

        , Dts.Variables["$Package::DestinationServerName"].Value

        , Dts.Variables["$Package::FederationRootName"].Value

        , Dts.Variables["$Package::DestinationUserID"].Value

        , Dts.Variables["$Package::DestinationUserPwd"

            ].GetSensitiveValue().ToString());

    SqlCommand cmd = new SqlCommand();

    cmd.Connection = conn;

    cmd.CommandText = "select f.federation\_id, m.member\_id, d.distribution\_name,"

        + " fd.range\_low from sys.federations f\n"

        + " join sys.federation\_members m on f.federation\_id = m.federation\_id\n"

        + " join sys.federation\_distributions d on f.federation\_id = d.federation\_id\n"

        + " join sys.federation\_member\_distributions fd"

        + " on f.federation\_id = fd.federation\_id\n"

        + " and m.member\_id = fd.member\_id\n"

        + " and d.distribution\_name = fd.distribution\_name\n"

        + "where f.name = @FedName\n"

        + "ORDER BY fd.range\_low desc; ";

    cmd.Parameters.AddWithValue("@FedName",

        Dts.Variables["$Package::FederationName"].Value.ToString());

    // you will need a second connection to connect to each federation member

    // to get the database name:

    SqlConnection conn1 = new SqlConnection();

    conn1.ConnectionString = conn.ConnectionString;

    SqlCommand fedNameCmd = new SqlCommand();

    fedNameCmd.Connection = conn1;

    // since a "USE FEDERATION" must be by itself in a batch,

    // and the process to retrieve a federation member

    // database name will involve 2 batches,

    // you cannot yet prepare a statement nor

    // create a single execution. Therefore, delay setting CommandText on fedNameCmd

    try

    {

        conn.Open();

    }

    catch (SqlException e)

    {

        Dts.Events.FireError(0,

            "Setup Connection Info For Federation, connect conn to federation root",

            e.Message, String.Empty, 0);

        Dts.TaskResult = (int)ScriptResults.Failure;

        return;

    }

    try

    {

        conn1.Open();

    }

    catch (SqlException e)

    {

        Dts.Events.FireError(0,

            "Setup Connection Info For Federation, connect conn1 to federation root",

            e.Message, String.Empty, 0);

        Dts.TaskResult = (int)ScriptResults.Failure;

        return;

    }

    // if you get here, you have both connections open.

    // safely execute cmd

    try

    {

        SqlDataReader reader =  cmd.ExecuteReader();

        // Variables to hold the results of the query:

        Int32 federation\_id;

        Int32 member\_id;

        String distribution\_name;

        Int32 range\_low;            // choose the datatype appropriately.

                                    // in my case, range key is int

        // use a variable to track which member we are on

        // and use the variable value to build the Dts variable name to set its value

        Int32 federationMember = 0;

        while (reader.Read())

        {

            federation\_id = reader.GetInt32(0);

            member\_id = reader.GetInt32(1);

            distribution\_name = reader.GetString(2);

            range\_low = reader.GetInt32(3);

            /\*

                \*

                \* Note that in this example, federation\_id and member\_id are pulled.

                \* These are only necessary if you want to query catalog views in the

                \* federation member to verify that the table is in fact

                \* federated.

                \* \*/

            // first, set the rangeLow for this federation member:

            Dts.Variables[String.Format("User::FedMember{0}MinRange",

                federationMember)].Value = range\_low;

            // next, set the database name so that the data flow

            // can connect to the correct database. note when creating

            // the command text that range\_low is used. In the ranges defined in the

            // sys.federation\_member\_distributions catalog view range\_low

            // is inclusive, and range\_high is exclusive, therefore, using on the

            // range\_low value will put you into this federation member.

            // see http://msdn.microsoft.com/en-us/library/windowsazure/hh597453

            // for more information

            try

            {

                String commandText = String.Format(

                    "USE FEDERATION {0} ({1} = {2}) WITH RESET;",

                    Dts.Variables["$Package::FederationName"].Value

                    , distribution\_name, range\_low);

                fedNameCmd.CommandText = commandText;

                fedNameCmd.ExecuteNonQuery();

                // fedNameCmd is now connected to the federation member.

                // query for the name:

                fedNameCmd.CommandText = "SELECT db\_name()";

                // set the SSIS Variable to the correct database name:

                Dts.Variables[String.Format("User::FedMember{0}",

                    federationMember++)].Value = (String)fedNameCmd.ExecuteScalar();

            }

            catch (SqlException e)

            {

                Dts.Events.FireError(0,

                    "Setup Connection Info for Federation," +

                    " Retrieving federated member names",

                    e.Message, String.Empty, 0);

                Dts.TaskResult = (int)ScriptResults.Failure;

                return;

            }

        }

    }

    catch (SqlException e)

    {

        Dts.Events.FireError(0,

            "Setup Connection Info For Federation, Processing federation root info",

            e.Message, String.Empty, 0);

        Dts.TaskResult = (int)ScriptResults.Failure;

        return;

    }

Dts.TaskResult = (int)ScriptResults.Success;

}

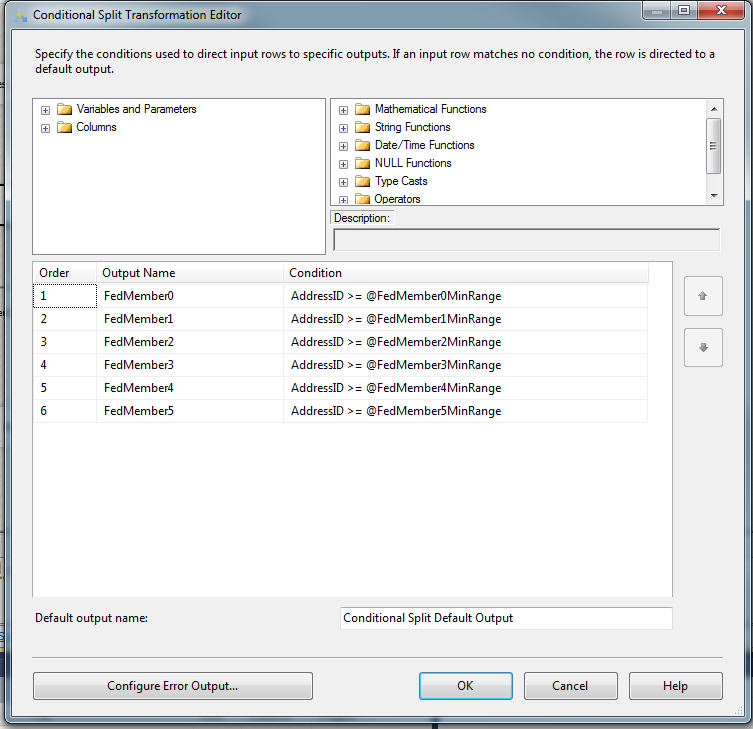
1. Create your dataflow task and components including any restart and retry logic for your dataflow. Note that if you include restart for your components, you must setup checkpoints, and set the **FailPackageOnFailure** property to **true** for the Script task, in order for variable values to be saved for restart of any package.

See [Restart Packages by Using Checkpoints](http://msdn.microsoft.com/en-us/library/ms140226.aspx) (http://msdn.microsoft.com/en-us/library/ms140226.aspx) for more information on using checkpoints to maintain state in the control flow for restarting packages. For information on designing the pipeline for restart, **Designing for Restart Without Losing Pipeline Progress** section in the “SQL Server 2012 SSIS Operational and Tuning Guide”. The guide is located in the MSDN library under the [Microsoft White Papers](http://msdn.microsoft.com/en-us/library/hh403491.aspx) node (http://msdn.microsoft.com/en-us/library/hh403491.aspx) for SQL Server 2012.

1. Connect the Script task to the Data Flow task with precedence constraints to ensure the Data Flow task does not start before the Script task executes.
2. Within the data flow, a conditional split must be used to redirect rows to the correct federation member. One output must be created for each federated member.

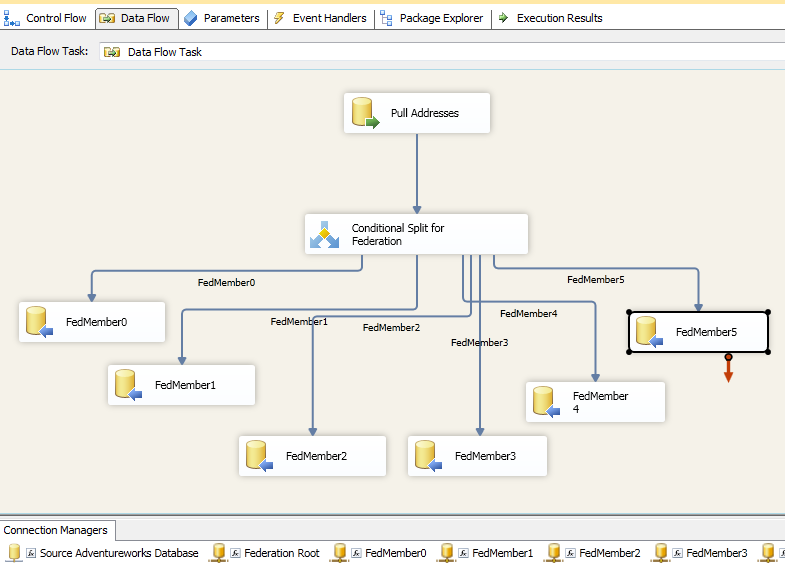
Because the query used to query for federation members returned rows ordered on range\_low descending, and federation members within SSIS are named starting from 0, FedMember0 will have the greatest federation key values. Because range\_low is inclusive, the expression must include a <= condition.

Rows are sent to the first output where their condition returns true. So starting with the greatest values range and using a >= condition, there is no need to check for the range\_high to ensure it meets both conditions. All rows with values higher than the current range are sent to their respective outputs before the rows for any given range reach their correct range. The following is an example of the Conditional Split configuration.



1. Attach the outputs of the Conditional Split transformation to the destination components for the federated members.

When you develop the package, you will need to put a default value into the FedMember0, FedMember1, and other FedMember variable values. The value needs to be the name of a database that contains a table with the same schema as your destination. This will allow SSIS to retrieve the destination metadata during design. When the package is executed, the Script task will change the variable value to the database name of the federation member into which data is inserted. The dataflow from the example package looks like this:



**TIP:** Because of the static nature of the data flow pipeline, you cannot dynamically create additional destinations in your dataflow to address shard splits. To allow for dynamically expanding federations, you must programmatically build packages. You can use the technique demonstrated in the Script task to map data to the correct federation members as you build packages programmatically.

**TRICK:** You can create static packages with room for expansion by creating more destinations in your data flow than you currently have federation members. The technique demonstrated for mapping the data dynamically to the existing federation members will send data only to the existing federation members. When you execute a split to create a new federation member, the next execution of the package will have a destination available and the Script task will use the federation metadata to direct data to the new federation member.

## SSIS Processes in Windows Azure

With the new WA Persistent VM, SQL Server is now supported in WA. SSIS is not supported in WA. This allows SSIS to be located within the same data center as Azure data sources or destinations, thus allowing for much more efficient Azure to Azure data movement than was previously possible. It also makes possible scale-out SSIS data movement in Azure by running packages across multiple Persistent VMs. In addition, data can be moved in two steps where the first step compresses data and stages the compressed data in Azure storage and the second step retrieves the compressed data and extracts it into destination located in WA. The compression allows you to move much larger amounts of data across limited bandwidth and high latency networks that you may have between your on-premises and Azure data centers.

### Lifecycle of the SSIS VMs

WA Persistent VMs with the SSIS components installed can run SSIS packages. SQL Server is not required to be running on the machine for SSIS packages to be executed. For installation options for SSIS, see [Install Integration Services](http://msdn.microsoft.com/en-us/library/ms143731.aspx) in SQL Server Books Online (http://msdn.microsoft.com/en-us/library/ms143731.aspx).

SSIS can be installed on a VM either in WA, or running in Hyper-V and uploaded to Azure. As an alternative, in the WA platform, you can create a new VM from the Gallery, and choose the Microsoft SQL Server 2012 image from the gallery. Because these machines have the SSIS service, runtime, and SDK installed on the image, you can execute your SSIS packages on the machines as soon as they are brought online. The SSISDB catalog and database is not installed on the Azure Gallery images, so to use catalog features, you must configure this along with any other application requirements prior to use. For instructions on creating the catalog, see [Create the SSIS Catalog](http://msdn.microsoft.com/en-us/library/gg471509.aspx) (http://msdn.microsoft.com/en-us/library/gg471509.aspx ). For more information about the SSISDB catalog and database, see [SSIS Catalog](http://msdn.microsoft.com/en-us/library/hh479588.aspx) (http://msdn.microsoft.com/en-us/library/hh479588.aspx ).

Whether created from a gallery image, or uploaded from a sysprepped Hyper-V image, you are charged for storage of any private gallery images you save. Computing is charged as soon as a VM is created from the image. You are warned when you shut down a VM in Azure that you will continue to accrue charges on this VM unless you delete it.

One scenario would be to create VMs for ongoing data movement purposes and schedule or run SSIS packages on these dedicated machines. This scenario works for steady data movement and ETL needs. However, for times when large amounts of data must be moved in a short amount of time, or when the amount of data needed to move surges, you may need to create a solution where images are spun up as VMs to handle the surge, and then deleted when the data movement tasks have been completed.

When SSIS VMs need to be spun up for large data movements and surges, you should install all needed components, applications, and schedules, complete all configuration, then capture an image of the machine to your private gallery so instances can be created to deal with high demand periods.

## Location of SSIS Processes

All general recommendations about executing SSIS package on machines with available resource continue to apply when using SSIS to execute packages in WA. Running SSIS packages on the same machine or VM as your production SQL Server, can take much needed resources such as memory or CPU capacity from SQL Server and therefore impact performance. When designing a data movement solution that involves moving data across restricted bandwidth or high latency networks, additional steps and conditions may need to be considered.

### Azure to Azure Data Movement

When setting up data movement solutions where both the data source and data destination is in WA, whether in SQL Server in a VM, in a Windows Azure SQL Database, or any other form, the most efficient place to put your SSIS process is in a VM in WA. In addition, consider the location within Azure of your data source and destination. The following possibilities exist.

* When the source and destination are both in the same data center, be sure to put your SSIS process in a VM in that same data center.
* When the source and destination are in different data centers, you can take one of two approaches depending on the amount of data and time required.
  + For small amounts of uncompressed data, it may be more efficient to run the SSIS process in the same data center as either your source or destination data, and not create any intermediate files.
  + If the data files are compressed files, put your SSIS process in the same data center as your destination. First, copy or move the compressed files to the data center where the SSIS process and destination are located, and process the data in this data center.
  + If the data source is uncompressed, but the destination is a compressed file, run the SSIS process in the same data center as your source, and send the compressed file to the destination
  + For large amounts of uncompressed data, design your data movement solution to extract to compressed files, move the compressed files to the same data center as your destination, and use a separate SSIS process in the same data center as your destination to decompress the data and import to your destination.

### Hybrid Data Movement

Hybrid data movement is the movement between an on-premises data store and a cloud data store. For these moves, you will always have a restricted bandwidth and high latency network. The scenarios for this are the same as for moving data between data centers in Azure to Azure data moves. For a description of the scenarios, see the [Azure to Azure Data Movement](#AzureToAzure) section in this article.

* For small amounts of uncompressed data, it may be more efficient to run the SSIS process in the same data center as either your source or destination data, and not create any intermediate files.
* If the data files are compressed files, put your SSIS process in the same data center as your destination. First, copy or move the compressed files to the data center where the SSIS process and destination are located, and process the data in this data center.
* If the data source is uncompressed, but the destination is a compressed file, run the SSIS process in the same datacenter as your source, and send the compressed file to the destination
* For large amounts of uncompressed data, design your data movement solution to extract to compressed files, move the compressed files to the same data center as your destination, and use a separate SSIS process in the same datacenter as your destination to decompress the data and import to your destination.

## Dealing with Limited Bandwidth Scenarios

When a limited bandwidth or high latency network is between your source and destination, for movement of data of any size, it is much more efficient to compress the data prior to sending it across the limited network, and only decompress it after it has arrived in the destination data center. Visualize this scenario like an hourglass where the limited network is the restriction in the middle. If you could compress the sand in the hourglass prior to it passing through the center, then decompress it afterwards, the sand could pass through to the bottom much more quickly. In cases of hybrid data movement, this two-stage transfer approach can be several times faster than transferring the uncompressed data across the limited network directly to the destination.

### Compressing Data

Compressing data before sending it across a limited network saves bandwidth, and helps to alleviate some network latency. When the data is being moved out of Azure, it also helps to minimize the egress charges. When using cloud storage as an intermediate location, it also helps to minimize the storage and transaction costs.

Data such as text data or XML data is highly compressible. Data such as jpeg files stored in SQL Tables is not very compressible. The amount of compression you are able to get depends on the data type you are compressing.

To compress the data, you can extract to a data file such as an SSIS Raw File or flat file, or you can use custom components that extract directly to a compressed file in order to extract and compress at the same time. You can download such custom compressed file components from [CodePlex](http://compfilecomponents.codeplex.com/) (http://compfilecomponents.codeplex.com/).

### Two-Stage Transfer

In a two stage transfer, data is first extracted to a file. The file is then moved from the source location to the destination location. A separate process at the destination location picks up the file and imports it into the destination. In hybrid data movement, using a two-stage transfer where the file that is transferred from the source data center to the destination data center can be much more efficient than a single SSIS process transferring the data.

You can run SSIS in persistent VMs to handle the extraction or load processes to sources or destinations in the WA Platform. You can run SSIS in on-premises machines to handle the extraction or loading in those locations.

The hybrid scale-out example that is given in the [Scale-Out and Hybrid Data Movement](#_Scale-Out_and_Hybrid) section in this article is an example of a two-stage transfer.

## Applying Scale-out Principles with WA

Scaling out execution of an application involves moving parts of the solution to separate servers to allow for parallel execution. SSIS can be used in scaled-out data movement solutions by breaking down extraction and loading into separate services and executing them simultaneously. Data sources such as SQL Server can serve data to numerous such extraction processes simultaneously, and data destinations such as SQL Server or Windows Azure SQL Database allow for multiple concurrent import processes either in the same table, or across multiple tables. When sources or destinations are sharded or federated, multiple processes may also be employed to extract or import to different shards or federation members.

### Multiple Persistent VMs or SSIS Extraction Servers

Scaling out the data movement will be done by running multiple processes across multiple machines for the extraction in parallel, staging in parallel, and parallel importing of the data. Running across multiple machines allows you to take advantage of processing power of several, smaller machines. The asynchronous processing allows you to more effectively use limited resources such as network since it is likely that not all processes will be trying to upload at the exact same time.

When you design a scale-out application, you need a way of distributing the work. You can do this in various ways. The stored procedures in the SSISDB database allow you to execute packages on remote machines from one centralized location. Queues allow applications to retrieve units of work as they are ready to process them.

### Queues

Unlike catalog execution from a centralized location, queues can be used to allow processes to self-schedule – retrieving records only as they are ready to process, and not sit idly when more work is ready to be done. Queues allow units of work to be distributed among processes running asynchronously. WA queues are one queuing mechanism that can be used to distribute such work.

Two steps are required to remove a message from a WA queue.

1. The message is popped.
2. The message is deleted.

A Timespan object is passed to the queue when a message is popped. The Timespan indicates the amount of time the record should remain invisible to all other processes polling the queue. If the process completes before the timespan expires, it can delete the message, and the message will not be placed back into the queue. If the Timespan expires without the message being deleted, the message will be placed back onto the queue with an incremented dequeue count. To use queues effectively you should do the following.

* Keep the units of work the process will do with the message, small enough that the work can be completed within a reasonable timespan. In this case, the unit of work is defined by the extraction of the data chunk.
* Set the timespan to a reasonable length to allow for the work to be done, but short enough to allow for the message to timeout and be retried by other processes within the time allowed for your overall transfer.
* Check the dequeue count on the message, especially if an exception occurs during processing. It is possible for an error condition to be raised while processing a unit of work that cannot be resolved by the process. If the dequeue count goes above a reasonable number of retries, treat it as a poison record and process it as such. This may mean entering a record elsewhere to indicate that this unit of work could not be completed, and processing this unit of work manually.

The two steps that are required for removing a message from a WA queue, builds retry into an application. If a machine fails while processing a unit of work, another process will pop the message after the timeout expires and process the unit of work. This adds robustness to your scale-out application.

### Blob Storage

Using blob storage to stage files such as data files or SSIS packages, allows for multiple processes running in multiple locations to access these files as needed, even if the machines do not share security privileges with users on the other machines. Each process needs only to have the URL and access key to the storage account to be able to access the files.

# Designing for Retry without Manual Intervention

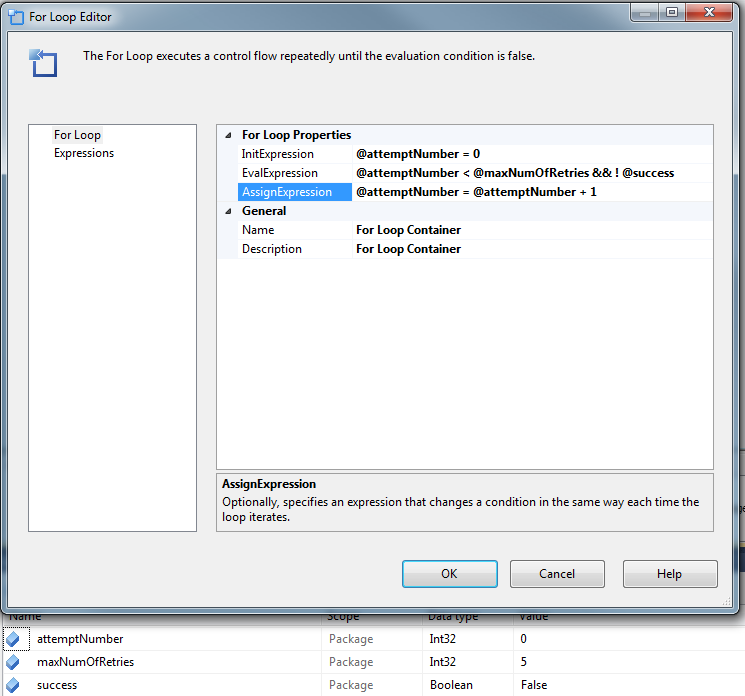
SSIS Source and destination components do not incorporate retry logic. However, the SSIS package developer is not helpless do deal with this situation. Designing your packages so that you can restart your SSIS packages without losing the progress in your data flow also enables you to retry automatically after a failure with some additional consideration. In cases such as moving data from or to Windows Azure SQL Database, you may need to retry automatically to handle transient fault conditions such as being throttled.

This section builds on the concepts demonstrated in the last section and shows an example of simple retry logic in your package. The section then demonstrates using a chunked approach that enables retry for each chunk in order to add robustness to your package during times when transient faults may be more likely.

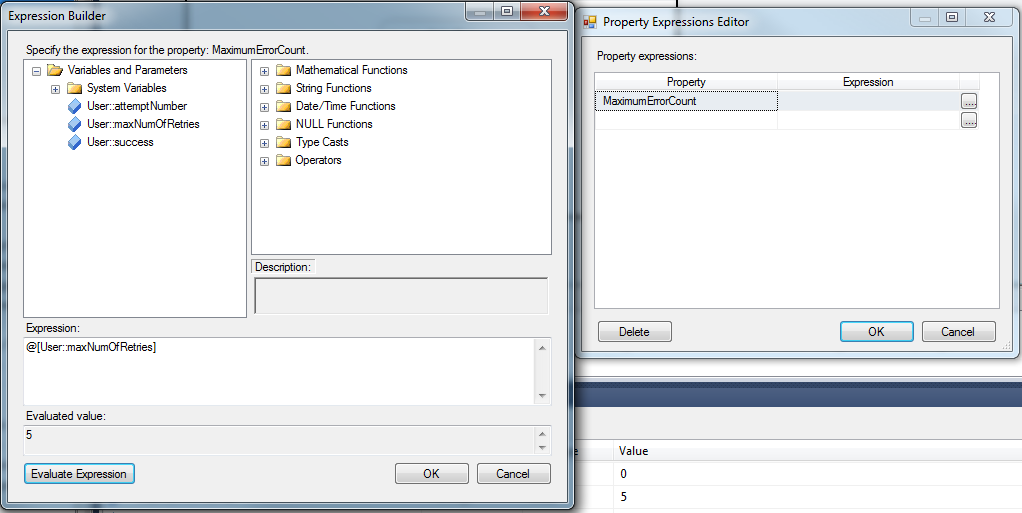
## Incorporating Retry into a Package

For cases of a single package, incorporate retry by doing the following.

1. **Determine the maximum number of times the package should retry before failing.** For the sake of this example, we will determine that a component will be allowed a maximum of 5 retries before failing the package. Create a variable in your package named “maxNumOfRetries.” Make this variable of type “int” and give it the value of 5. This will be used in expressions in the package. You can change the number of retries by changing only the default value of the variable.
2. **Set up a variable to store your success status.** Create a new variable in your SSIS package. Name this new variable “attemptNumber”
3. Use a FOR loop to retry up to the maximum number of retries if the data flow is not successful.



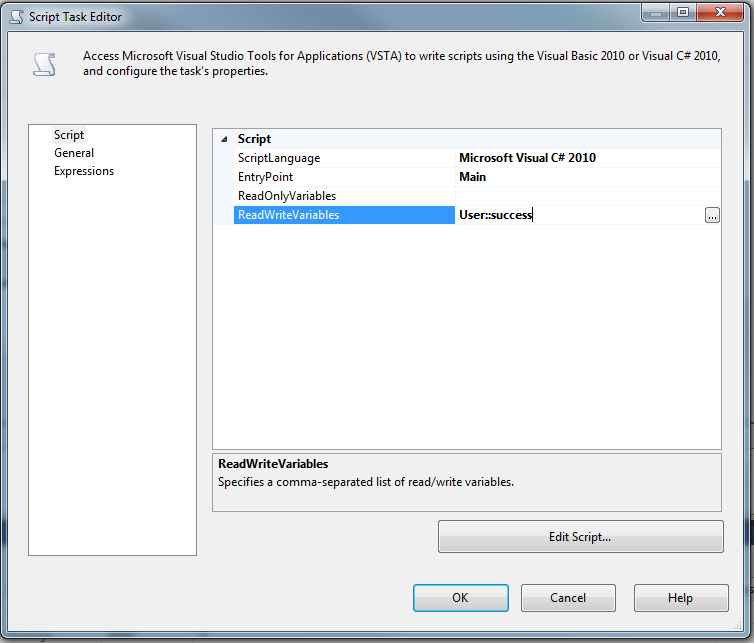
1. Put your data flow task inside the FOR loop.
2. Set the **MaximumErrorCount** property on the FOR loop to the maximum number of times the data flow should be retried. You do this with an expression that uses the maxNumOfRetries variable you set up in step 1.

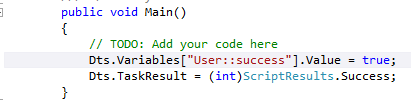


1. Use Execute SQL tasks as shown in the last section to find the minimum key values at the source and the maximum key values at the destination.
2. Place your Execute SQL tasks in the FOR loop.

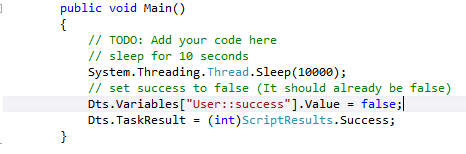
For a simple retry, this can be done inside the FOR loop. For the more advanced retries, this may appear elsewhere in the control flow prior to the FOR loop.

1. If the Execute SQL tasks were used in step 6, connect the success constraint from the Execute SQL Task to the Data Flow task.
2. Connect the Success precedence constraint from the Data Flow task to a script task that sets the success status variable to **true** to break out of the FOR loop. An example of the configuration of the script task is below

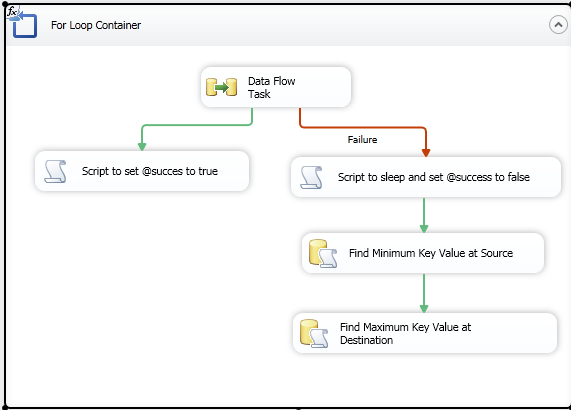




1. Connect a Failure precedence constraint from the Data Flow task to another Script task to
   1. Sleep for 10 seconds if you are in a Windows Azure SQL Database environment where throttling is a possibility.
   2. Set the value of the success variable to false.



1. For each task in the FOR loop, set the **FailPackageOnFailure** property to **false**. In this configuration, only when the FOR loop fails after using all of the configured retry attempts will it report failure, thus failing the package.
2. Configure Execute SQL tasks after the failure to again check for the minimum key values at the source, and check for maximum key values at the destination so that progress can be resumed without re-doing any work. Set these up as described in the **Designing for Restart Without Losing Pipeline Progress** section in the “SQL Server 2012 SSIS Operational and Tuning Guide”. The guide is located in the MSDN library under the [Microsoft White Papers](http://msdn.microsoft.com/en-us/library/hh403491.aspx) node (http://msdn.microsoft.com/en-us/library/hh403491.aspx) for SQL Server 2012.



Each time an error in the dataflow occurs, if the maximum number of retries has not been reached, then the process goes back to the point of finding the maximum key value at the destination, and the process continues by processing from the point that is currently committed at the destination. If there are multiple destinations in the dataflow, begin after the lowest key value to have safely arrived in the destination, and use a conditional split, or error flow to handle any rows that would cause a primary key violation.

## Incorporating Retry Logic in Custom Components

Retry logic is most efficient when put into source or destination components. You can develop custom source or destination components for SSIS and incorporate transient fault handling such as is demonstrated in the [Transient Fault Handling Framework for SQL Azure, Windows Azure Storage, Service Bus & Cache](http://windowsazurecat.com/2011/02/transient-fault-handling-framework/) (<http://windowsazurecat.com/2011/02/transient-fault-handling-framework/> ) article.

## Moving Retry when Building or Executing Programmatically

The previous example showed an example of putting retry logic into the SSIS package. When you execute packages programmatically, you also have the option of adding the retry logic. You accomplish this by checking the execution status after package execution, and retrying if the package execution was unsuccessful. The solution demonstrated in the [Scale-Out and Hybrid Data Movement](#_Scale-Out_and_Hybrid_1) section later in this article uses this method. You can refer to the solution in Appendix C for an example of this.

# Hybrid Data Movement with SQL Server running in WA VMs

Another possibility for data movement between WA and on-premises SQL Servers is movement to and from databases residing in WA Persistent VMs. Many of the principles remain the same as with data movement to and from Windows Azure SQL Database, but there are some key differences.

The following are special considerations .

* **Compress data prior to moving across the Wide Area Network (WAN).** The low bandwidth network continues to be a potential bottleneck for hybrid data movement. Regardless of the method used to transfer the data across this network, for large amounts of data, it is more efficient to compress the data prior to moving it across the restricted bandwidth network, then decompressing after the data has moved. FTP servers can be set up to transfer data directly to the SSIS servers, or the data can be moved to WA Blob storage as an intermediate staging location.
* **Use a separate SSIS server.** To maximize processing power with minimal impact to your SQL Server running in a WA VM, use a separate SSIS server rather than running packages directly on the SQL Server VM.
* **Tune your packages.** Tuning of packages remains key to overall performance.
* **Design for minimally logged operations when possible.** Recommendations in [The Data Loading Performance Guide](http://msdn.microsoft.com/en-us/library/dd425070(v=sql.100).aspx) (http://msdn.microsoft.com/en-us/library/dd425070(v=sql.100).aspx ) still apply to loading to SQL Server running in WA Persistent VMs.
* **Ensure your SQL Server is configured for running in a WA Persistent VM.** The recommendations include the following.
  + **Move tempdb off of local storage and onto data drives.**
  + **Put databases onto WA Data drives.**
  + **Place the transaction log on a separate WA drive from the drives where the data files are located.**
  + **Use multiple WA Data Drives, and place data files on separate drives.**
  + **Tune the SQL Server memory to prevent external contention between SQL Server and any other processes running on the VM.** If you opt to run your SSIS packages on the same VM as your SQL Server instance, remember that the SSIS package executes in a separate process that can compete with the SQL Server instance for memory, processing, and disk resources.

For ,ore information on tuning recommendations for SQL Server running in a WA VM, see [Running SQL Server in Windows Azure Virutual Machines – Performance Guidelines for Preview](http://sqlcat.com/sqlcat/b/technicalnotes/archive/2012/06/08/running-sql-server-in-windows-azure-virtual-machine-performance-guidelines-for-preview.aspx) (<http://sqlcat.com/sqlcat/b/technicalnotes/archive/2012/06/08/running-sql-server-in-windows-azure-virtual-machine-performance-guidelines-for-preview.aspx> ) .

## Strategies

The typical flow of data in the on-premises configuration looks like following:

* Data Sources contain the information which has to be loaded into the database. The role of the Data Sources could play the table in another database or flat file.
* SSIS runs the package which picks the data from Data Source and process it into the relational data base with or without doing additional transformations.
* In order to satisfy data moving speed, the SLA SSIS package may be designed using multiple streams processing data in parallel according to the data loading best practices

**Original on Premise configuration:**



The Hybrid solution assumes that the data will be stored not only on the server in the local infrastructure, but that some data will be available in the WA too. In this scenario, the originally used ETL flow would additional modification to extract data into the databases in the Cloud.

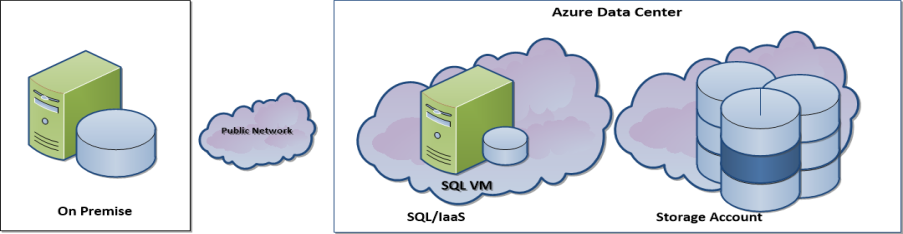
**Hybrid configuration:**



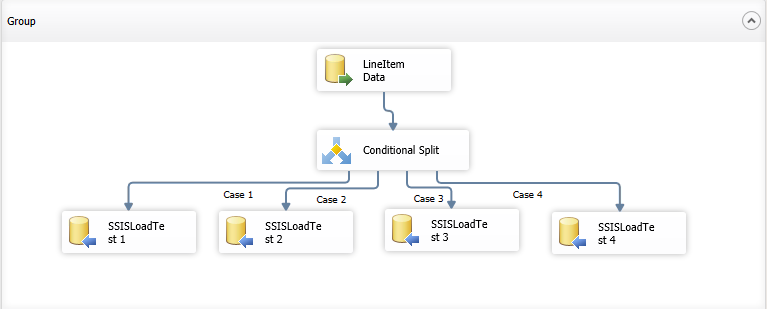
Let’s look deeper at how the data loading strategies can look alike.

### Load data directly into the WA VM.

The simplest scenario of pushing data into the WA environment would involve Azure databases as the destination.



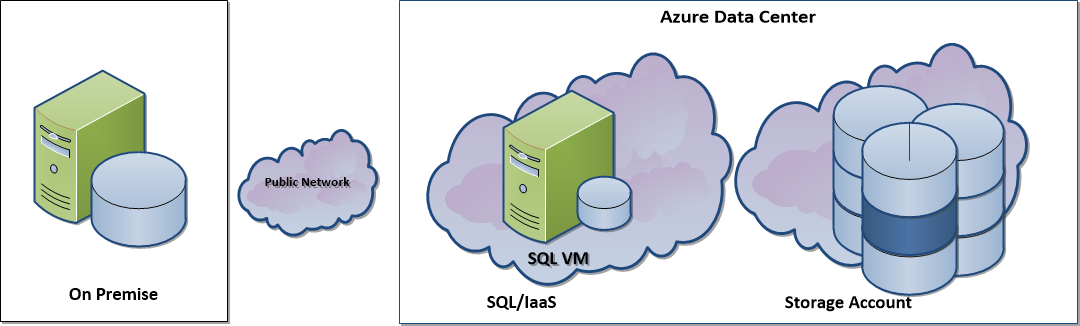
In this case, if we look at the SSIS package example, it should not be very different from how it looked when we developed it for the on-premise ETL.

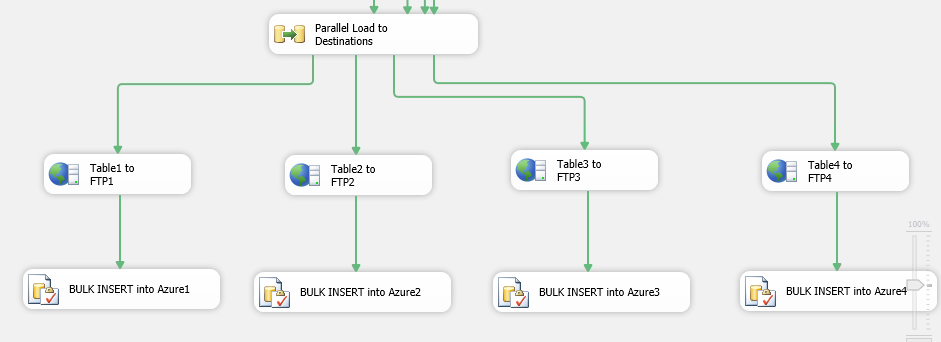


SSIS Package Example

### Using file transfer for moving data between the on premise server and the WA VM

An alternative strategy for loading data from the on-premises server into the destination VM in WA is to leverage file transfer. The data loading process in this case will involve loading data into a flat file, transferring the file over the network into the VM share, and bulk inserting the data into the destination table.





SSIS Package Example

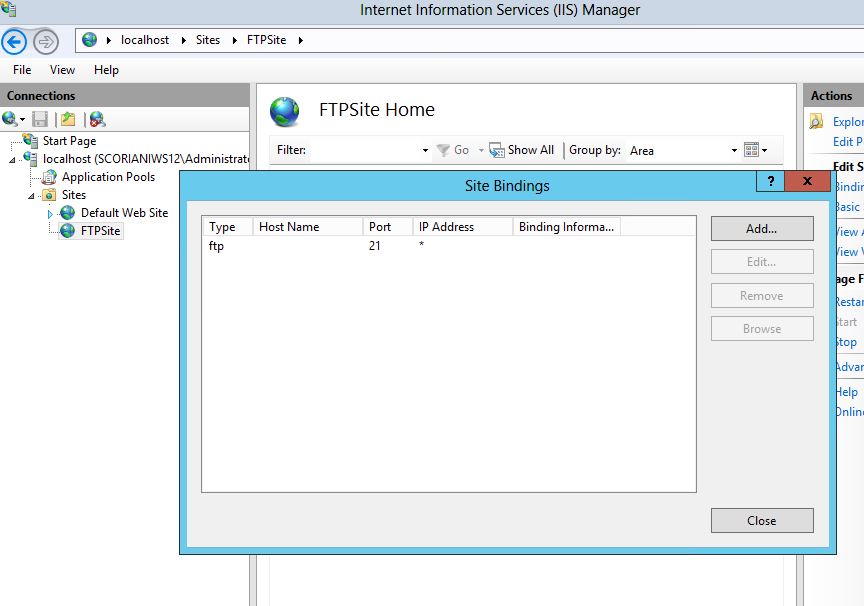
However, this scenario requires setting up the file shares, where temporary files with data will be created and stored. In order to make files accessible for the WA VM, you may decide to use a folder share. This requires joining the VM into the domain where the on-premise server resides.

One of the benefits of using this configuration is that SSIS will be using Flat File components and stay relatively simple.

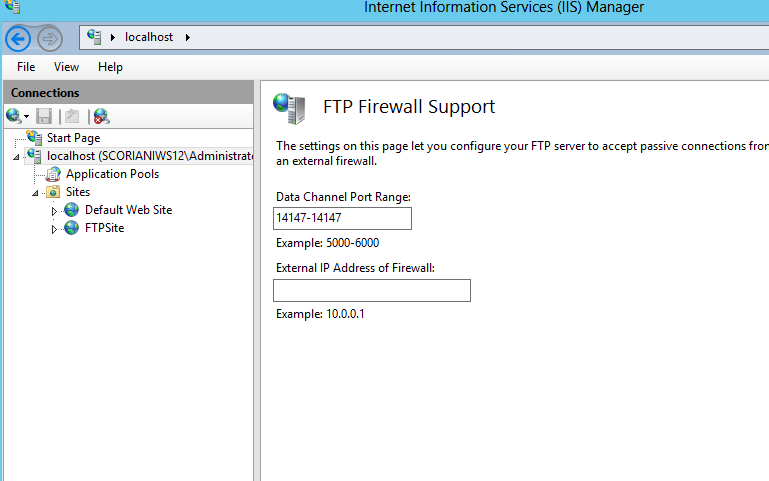
However, in some cases the process of joining the WA VM into the domain might be problematic, mostly due to a company’s internal organizational restrictions. In this case, using an FTP service may give certain flexibility. You would use the service to transfer files over the public network between the on-premise machine and the WA VM. This requires setting up an FTP Server on the WA VM.

**To set up an FTP Server on the WA VM**

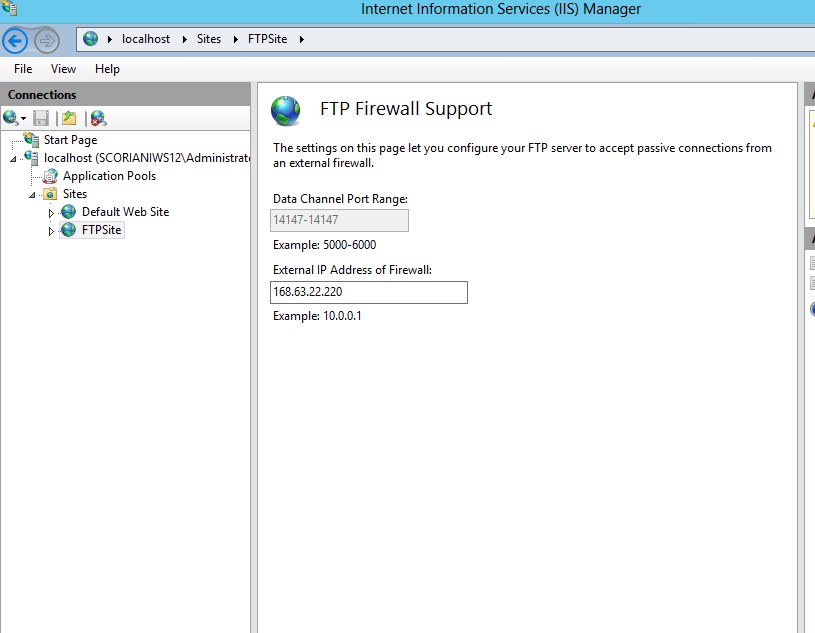
1. Install IIS with FTP feature using Server Manager on the VM operating system (Windows Server 2008/R2/2012)
2. Create a standard FTP site with your preferred configuration and security settings. For more information, see [Set Up FTP in IIS 7](http://technet.microsoft.com/en-us/library/cc771012(WS.10).aspx) (http://technet.microsoft.com/en-us/library/cc771012(v=ws.10).aspx).



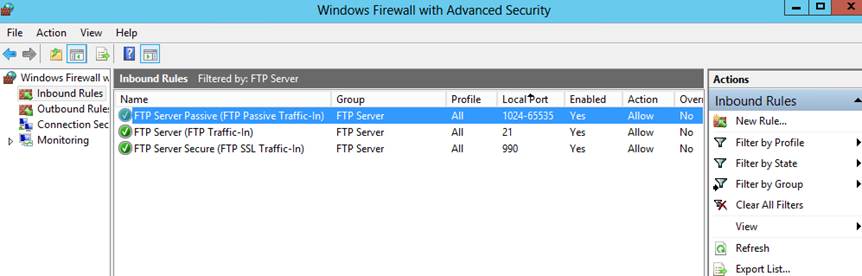
1. In the IIS management console, at the server level, open FTP Firewall Support settings and define a ***single*** port range for Data Channel (for example, 14147). This will be used by the FTP server to establish a data channel back to the client. It’s important to have a fixed one because it will be necessary to define a VM Endpoint to make it work. Note that this also means the FTP server will be accessible only using passive connections. The active connections need to communicate over a dynamic port range that won’t be possible to configure with WAVM Endpoints.



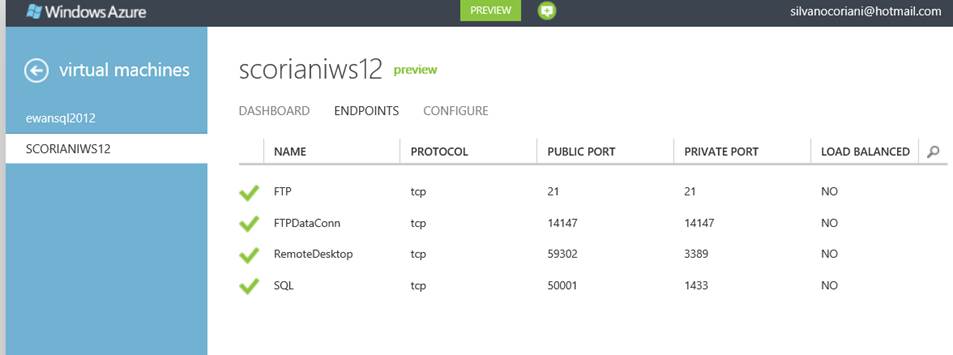
1. In the **External IP Address of Firewall** text box, enter your VM public Virtual IP (VIP) address. Otherwise, some FTP clients, such as the SSIS FTP task, receive an error while the server is communicating the internal VM IP address to the client.



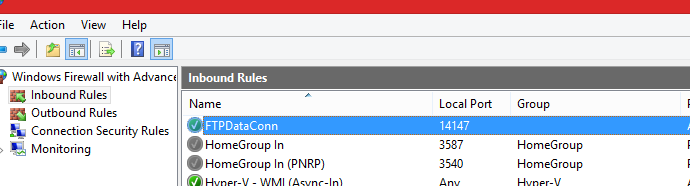
1. Make sure that during the FTP Service setup, all the relevant firewall rules will be created (21 and the passive range are the most important).



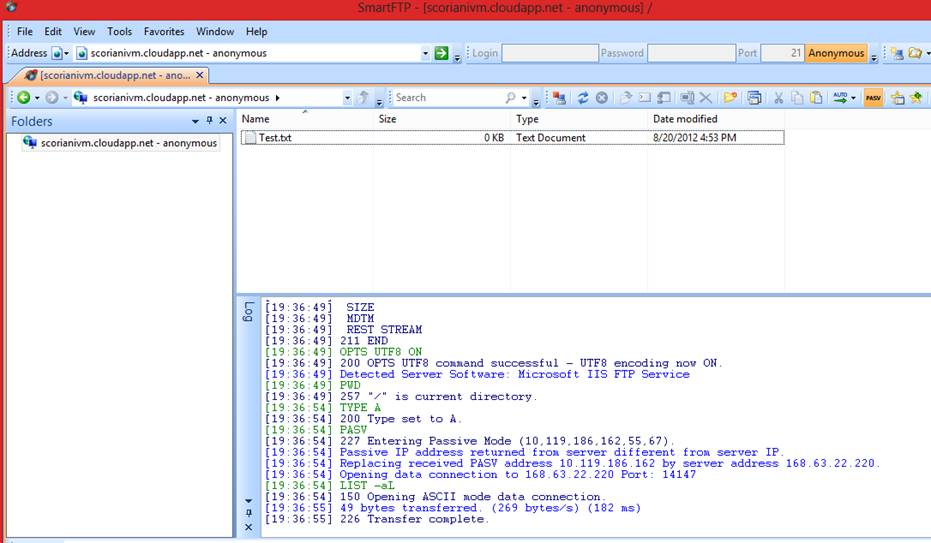
1. Create two endpoints for FTP control and data (21 for one endpoint, which was defined in a previous step) using the WA Portal. Then reboot the VM.



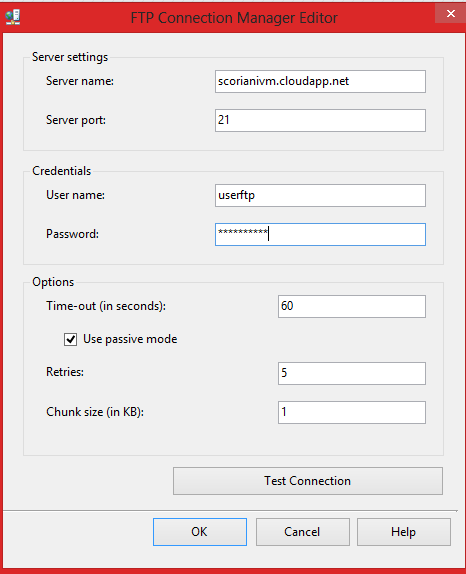
1. On-premises, on the SSIS Server machine, open the data connection port that was specified as FTP data connection port (for example, 14147).



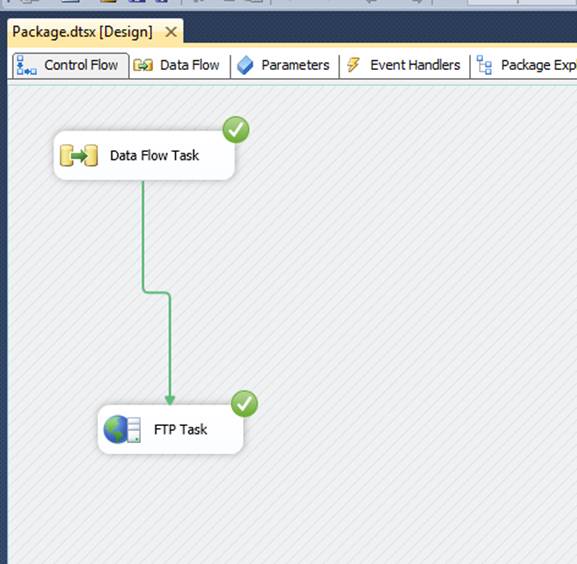
1. Use an FTP client that supports passive connections (for example, Smart FTP or FTPZilla) to check that that it is possible to set up a connection with the FTP Server on the WA VM side. Confirm that the FTP folder is accessible. Use the fully qualified name of the VM FTP Server to establish the data connection with the port you specified (for example, 14147).



1. Create an FTP Task and configure the connection appropriately, making sure that the fully qualified server name is used and the **Use passive mode** box is checked.

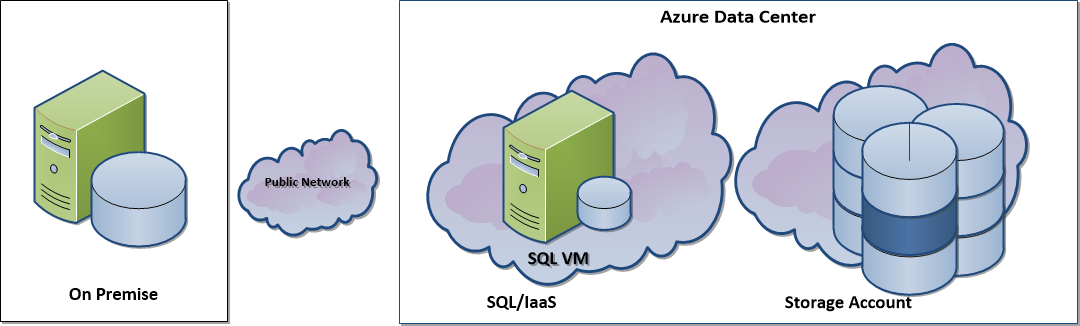


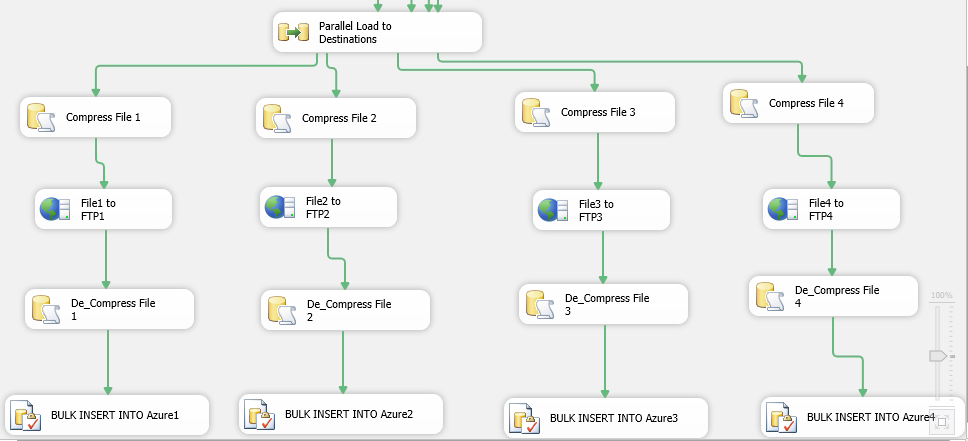
You can now design your SSIS packages and upload/download data directly from your Azure VM, as shown in the following example.



#### Special case: using compression to reduce overhead due to transferring data files over the network.

In cases when the network bandwidth would be very limited, it might be very beneficial to compress the data files before pushing them over the network.





SSIS does not provide compression components out of the box. However, this gap might be closed by developing custom SSIS components as described in the [Compressing Data](#_Compressing_Data) section of this article. Another option might be more preferable to some DBAs: implementing a CLR procedure that can be called from the SSIS package and executed on both on premise and the WA VM to compress and decompress data.

The following is an example of a CLR procedure for compressing and decompressing data.

using System.Collections.Generic;

using System.Linq;

using System.Text;

using Microsoft.SqlServer.Server;

using System.Data.SqlTypes;

using System.IO;

using System.IO.Compression;

namespace zip

{

public class Program

{

public static void comp(string option, string inputpath, string outputpath)

{

//option compression

if (option == "comp")

{

SqlContext.Pipe.Send("Compression Selected");

FileInfo fileToCompress = new FileInfo(inputpath);

FileInfo compressedFile = new FileInfo(outputpath);

Compress(fileToCompress, compressedFile);

return;

}

//option decompressor

else if (option == "decomp")

{

SqlContext.Pipe.Send("Decompression Selected\n");

FileInfo fileToDecompress = new FileInfo(inputpath);

FileInfo decompressedFile = new FileInfo(outputpath);

Decompress(fileToDecompress, decompressedFile);

return;

}

//help...because we all need this! :-)

else if (option == "help")

{

help();

return;

}

}

//Compression Code

public static void Compress(FileInfo fileToCompress, FileInfo compressedFile)

{

using (FileStream originalFileStream = fileToCompress.OpenRead())

{

if ((File.GetAttributes(fileToCompress.FullName) & FileAttributes.Hidden) != FileAttributes.Hidden & fileToCompress.Extension != ".gz")

{

using (FileStream compressedFileStream = File.Create(compressedFile + ".gz"))

{

using (GZipStream compressionStream = new GZipStream(compressedFileStream, CompressionMode.Compress))

{

originalFileStream.CopyTo(compressionStream);

SqlContext.Pipe.Send("Compression Complete\n");

}

}

}

}

}

//Help Code

public static void help()

{

SqlContext.Pipe.Send("Help for CLR File Compressor\n");

SqlContext.Pipe.Send("The Proc Expects 3 params (ALL LOWERCASE):\n");

SqlContext.Pipe.Send("Param 1 : Option, 'comp' - compression, 'decomp' - decompression\n");

SqlContext.Pipe.Send("Param 2 : Input Path<FileName>.<Extention>, so Input file to compress or decompress\n");

SqlContext.Pipe.Send("Param 3 : Input Path<FileName>.<Extention>, so file to compress or decompress, if compressing, do not use file extention\n");

SqlContext.Pipe.Send("Security Settings, DB Option, TrustWorthy = ON, Assembly Security = EXTERNAL\_ACCESS\n");

}

//DeCompression Code

public static void Decompress(FileInfo fileToDecompress, FileInfo decompressedFile)

{

using (FileStream originalFileStream = fileToDecompress.OpenRead())

{

string currentFileName = fileToDecompress.FullName;

string newFileName = currentFileName.Remove(currentFileName.Length - fileToDecompress.Extension.Length);

using (FileStream decompressedFileStream = File.Create(decompressedFile + ""))

{

using (GZipStream decompressionStream = new GZipStream(originalFileStream, CompressionMode.Decompress))

{

decompressionStream.CopyTo(decompressedFileStream);

SqlContext.Pipe.Send("Decompression Complete\n");

}

}

}

}

}

}

## Tuning Best Practices for On-Premises and Hybrid Data Movement

The following can impact performance of data movement.

* Network
* SSIS Package Data Flow
* Connection Drivers
* SQL Server Database (as destination)

### Network

Typically we get quite a valuable performance boost by setting up the Jumbo Frames in the network configuration and configuring the packet size setting in the SQL Server or SSIS connection strings.

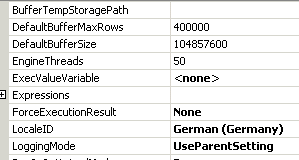
However, these settings have no visible impact on the performance in hybrid data movement, apparently because we have to go over multiple networks.

For more information, see the **Tuning network settings** section “SQL Server 2012 SSIS Operational and Tuning Guide”. The guide is located in the MSDN library under the [Microsoft White Papers](http://msdn.microsoft.com/en-us/library/hh403491.aspx) node (http://msdn.microsoft.com/en-us/library/hh403491.aspx) for SQL Server 2012.

### SSIS Package Data Flow

There are three configuration settings for the SSIS Data Flow that are relevant to hybrid data movement, **when** the server resides on premises.

* DefaultBufferMaxRows
* DefaultBufferSize
* EngineThreads



These settings allow you to better utilize the SSIS server resources and read data efficiently from the data source.

For more information, see the **SSIS Package settings** section “SQL Server 2012 SSIS Operational and Tuning Guide”. The guide is located in the MSDN library under the [Microsoft White Papers](http://msdn.microsoft.com/en-us/library/hh403491.aspx) node (http://msdn.microsoft.com/en-us/library/hh403491.aspx) for SQL Server 2012.

### Connection Drivers

We tested both OLEDB and ADO.NET driver performance to compare loading speed. There was no significant performance difference using either of the drivers.

### SQL Server Database (as destination)

Doing the following is recommended for both on-premises data movement and hybrid data movement, to ensure that the data is loaded in the fastest possible way.

* Bulk load data
* Ensure that the loading process is minimally logged.
* Avoid SORT

In the hybrid data movement, using SORT has an even worse impact on the performance then when SORT is used for on-premises data movement. We see SORT used in the hybrid solution when loading into a clustered index table.

# Solution: Scale-Out and Hybrid Data Movement

In this section, we will step through a scale-out hybrid data movement solution. This solution will include SSIS for the high-speed pipeline and components that make the design of a data flow easy. It will include the WA components to allow for elastic scale of each part of the solution. It will use a two-step transfer approach in order to minimize network bandwidth consumption and to allow for minimal elapsed time from the beginning to the end of the data transfer.

The examples for the full solution are in Appendix A, B, and C.

## Solution Requirements

To implement the solution, you need the compressed file source and compressed file destination components which you can download from [CodePlex](http://compfilecomponents.codeplex.com/) (<http://compfilecomponents.codeplex.com/>).

**NOTE:** You could build your own custom compressed file components, or compress files after extracting to file. These components were built to allow for compression as the data is extracted, and to allow for multiple blobs to be stored within a single data file.

This solution will also require the WA SDK which can be downloaded from <http://www.windowsazure.com/en-us/develop/downloads/>. The SDK is necessary to use the Azure queues and Azure blog storage.

## Solution Design

The time to extract and move data can normally be expressed as T(e) + T(l) where T(e) is time to extract and T(l) is time to load. Tools such as SSIS allow T(e) and T(l) to run simultaneously so that the time to perform the movement operation is the greater of T(e) and T(l). When a limited bandwidth network is placed into the topology, T(l) or T(e) will be affected by the bottleneck . We use a two-stage strategy to minimize this bottleneck.

For the remainder of the discussion, the following terms will be used to denote elapsed times:

* T(t) = Total elapsed time
* T(e) = Elapsed time for extraction to file
* T(c) = Elapsed time for compression
* T(u) = Elapsed time for upload to staging
* T(d) = Elapsed time for decompression
* T(l) = Elapsed time for loading from file

With a two-stage transfer, when everything is run sequentially, the time for the transfer is T(t) = T(e) + T(c) + T(u) + T(d) + T(l) . You can compress T(t) by running as many of these steps as possible in parallel. In this solution, we’ll do the following.

* Extract directly to a compressed file. This combines T(e) and T(c) into the greater of T(e) and T(c). In this example, that will be T(c). This is done by using the Compressed File Destination from CodePlex.
* Upload compressed files only across the limited bandwidth network, you minimize T(u).
* Decompress directly into the SSIS pipeline for uploading. This combines T(d) and T(l) into the greater of T(d) and T(l). In this example, that will be T(l).

With this parallelization, you now have T(t) = T(c) + T(u) + T(l). If you can extract from each table independently, then you can begin scaling out the data movement, thus reducing the T(c) and T(l).

In a situation where most of the data is in a single table, this may not result in much improvement in the overall elapsed time of the data movement operation.

To overcome this challenge, you need to have several processes running simultaneously on small chunks of data from each table. This allows each process to extract a small chunk and upload it to the staging location where it can be picked up and loaded by processes in the destination data center, thus allowing data from each table to be uploaded even while data from that same table is being extracted in the other data center. With sufficient compression, the network bottleneck is eliminated, and the total time T(t) is reduced to very near the greater of t(c) and T(l) that is close to the scenario you have without the limited network in the topology. Chunking the data also prevents a single large table from being the limitation to improvement of the overall elapsed time.

### Architecture

Using WA Platform components, the topology of the solution is shown in the following diagram.



In this topology

* WA Blob Storage is used to hold the compressed data files for staging.
* One WA Queue, the export queue, will contain messages about which SSIS packages should be executed next, to export data from the on-premises data source to a compressed data file and upload that to blob storage. This is stage 1 of the movement.
* One WA queue, the import queue, will contain messages about which compressed data files have been uploaded to the staging environment and are ready to import into the destination.
* The import servers in Azure are WA VMs. These machines run one or more processes that poll the import queue to see when a compressed data file is ready to be processed.
* When a record is successfully popped from the import queue, that process does the following:
  + Downloads the compressed data file staged in WA Storage
  + Builds an SSIS package programmatically to extract the data from that data file and load that into a sharded Windows Azure SQL Database destination.
  + Executes the package.
  + Deletes the record it popped from the import queue so that it will not timeout and be processed by another process.
* The import processes cannot process files until they are uploaded and the message is enqueued. Those tasks are handled by processes running on the Export Servers in the On-Premises setup. Multiple export processes may run on each export server. These processes poll the export queue to see what chunk of what table can be extracted.
* When a record is successfully popped from the export queue, that process does the following:
  + Downloads an SSIS package from the WA Blob storage that will extract that chunk of data to a compressed data file.
  + Executes the package.
  + Uploads the compressed data file into staging in WA Blob Storage.
  + Deletes the record it popped from the export queue so that chunk will not be processed by any other export process.
* The export processes cannot export until they have a record in the export queue telling them what SSIS package to execute. A separate process creates these. Since the chunks must be defined using ranges on keys, to prevent complexity in coordinating this, this process reads which tables are to be moved, and with each table, it creates SSIS packages to extract chunks of a predetermined number of rows of data. When the SSIS package is created, it uploads it to the WA Blob Storage and puts a message into the export queue so the export processes can retrieve and execute the process.
* The messages placed in the export queue are just the names of the SSIS packages the Export processes must download and execute.
* The messages placed into the import queue are just the names of the data files the export processes are staging in blob storage.
* In order to allow for multiple chunks of data from each table, a naming convention of <schema>.<table>.<sequence>.<extension> can be used for both the SSIS packages, and the data files.

For example, the first chunk from table Production.TransactionHistory would be extracted by an SSIS package named Production.TransactionHistory.0.dtsx and the data would be placed into a file named Production.TransactionHistory.0.dat. The second chunk would be extracted by an SSIS packaged named Production.TransactionHistory.1.dtsx and the data would be placed into a file named Production.TransactionHistory.1.dat, and that sequenced naming would continue until all the data was set to be extracted.

* Queries are used as the source in the SSIS packages. If a chunk size was determined to be 1,000,000 rows, then the query to extract the first chunk would contain “WHERE keyCol BETWEEN 1 AND 1000000 ORDER BY KeyCol”. Using an “Order By” on the clustered index allows for sequential insertion into the destination which allows for the page split optimization for sequential inserts.
* Because queues are used to distribute work among running processes, the number of export or import processes can be increased or decreased without reconfiguring the process or rewriting code.
* Azure queue messages are given a timespan for a timeout at the time they are popped from the queue. If the message has not been deleted before the timeout period, it is placed back onto the queue. This results in a retry if a process dies while exporting or importing.

### Design Considerations

For this solution, the destination is a sharded SQL Azure destination. When developing for Azure, you need to include retry logic for transient fault handling. The current ADO.NET or ODBC destinations do not have this logic built in. Custom components could be developed; another method is to re-execute the package, designing the data flow to deal with primary key violations. Retry is built in at another layer with the normal operations of queues.

## Solution Implementation

3 separate applications are needed for this solution.

* **Coordinator Application**. Defines the chunks of data to be extracted and builds the SSIS packages accordingly. A C# example of a coordinator is in Appendix A.
* **Export Application**. Executes SSIS packages to extract data and upload it to blob storage. A C# example is in Appendix B.
* **Application to generate packages in the Cloud**. Retrieves data from blob storage and imports it into the destination. A C# example is in Appendix C.

### Hardware and Software Requirements

To truly run this as a scale-out solution, you need multiple machines on-premises to run the SSIS extraction process, and multiple VMs in Azure to run the import SSIS processes. Each server or VM can run multiple processes.

### Install SSIS on On-Premises and Cloud Machines

In order for the processes to use SSIS, the SSIS runtime and SDK must be installed on the machines. The SSIS service and SQL Server are not required to run the processes. However, if you’re planning to use the SSIS catalog features for monitoring, the SQL Server and the SSIS catalog are required.

### Install Custom Compressed File Components

The solution makes use of the SSIS Compressed File source and destination current available at [CodePlex](http://compfilecomponents.codeplex.com/) (http://compfilecomponents.codeplex.com/). These components must be installed using the instructions at http://msdn.microsoft.com/en-us/library/ms403356.

### Set up the SQL Azure Export and Import Queues

Two queues are required – one for each stage of the movement. Create these in the storage account that will be used for the data movement solution. The export queue will be used by the processes to execute the extraction, and the import queue will be used by the import processes.

### Set up the SQL Azure Blob Storage

Both the SSIS packages and the data files will be staged in WA blob storage. Create a container for these files.

### Create Coordinator Application

The coordinator is the one piece of the solution that is not scaled out. It does not communicate with the other processes directly, but instead reads the list of tables that are to be transferred, creates the SSIS packages to extract chunks of a pre-determined size from these tables, uploads the SSIS packages to WA Blob Storage, and enqueues a message in the export queue. This process determines what tables are to be transferred in what order, and what chunk sizes are to be used. The export and import processes can all be running and polling the queues, but the data transfer begins when this process begins enqueuing messages on the export queue. The C# example for this solution is in Appendix A.

In the example, the only thing the enqueued message needs to contain is the name of the SSIS package to be retrieved and executed. Since multiple chunks of data can be retrieved from a single table, a naming convention like <Schema>.<Table>.<sequence>.dtsx can be used. An example file name using this convention would be Production.TransactionHistory.10.dtsx.

### Create the Export Application

The export application polls the export queue, pops records from the export queue when they exist, downloads the SSIS package to export a chunk of data, executes the SSIS package, uploads the data file, enqueues a message on the import queue to signal the import processes that this file is uploaded, and deletes the record it popped from the export queue. The message enqueued to the import queue need only contain the name of the data file. Since multiple chunks of data can be extracted from a single table, a naming convention like <Schema>.<Table>.<Sequence>.dat will work. An example data file name would be Production.TransactionHistory.10.dat. An example export application is in Appendix B.

### Create Application to generate packages in the Cloud

This application polls the import queue to see what data files have been staged in WA blob storage. When a message is dequeued, this process downloads the file from blob storage, builds an SSIS package based on the data it can gather, executes the package to import the data into Windows Azure SQL Database, and deletes the record it dequeued from the import queue. With proper naming conventions, most of the metadata needed to create the SSIS package is available.

* The message popped from the import queue contains the name of the source file.
* The source file name contains the schema name and table name where the data is to be imported.
* The file contains the metadata necessary to create the SSIS dataflow pipeline.

In this example, the connection information and number of shards are the only thing not known at this point. This information can be stored in a database, in WA tables, or can even be passed in as command line parameters depending on your requirements. Build this application to take advantage of the metadata convention you create.

Use the import queue messages as described in the previous section.

An example of this application that generates packages in the cloud is in Appendix C.

### Using the Conditional Split Transformation to Shard Data

When sending data to a sharded destination, use the conditional split to direct data to the proper destination as described in the [Sharding and Federations](#_Sharding_and_Federations) section earlier in this article.

# Conclusion

SQL Server Integration Services (SSIS) can be used effectively as a tool for moving data to and from Windows Azure SQL Database, as part of the total extract, transform, and load (ETL) solution and as part of the data movement solution. SSIS can be used effectively to move data between sources and destinations in the cloud, and in a hybrid scenario between the cloud and on-premise. This paper outlined best practices for using SSIS for cloud sources and destinations and for project planning for SSIS projects to be used with Azure or hybrid data moves, and gave an example of maximizing performance on a hybrid move by scaling out the data movement.

**For more information:**

<http://www.microsoft.com/sqlserver/>: SQL Server Web site

<http://technet.microsoft.com/en-us/sqlserver/>: SQL Server TechCenter

<http://msdn.microsoft.com/en-us/sqlserver/>: SQL Server DevCenter

Did this paper help you? Please give us your feedback. Tell us on a scale of 1 (poor) to 5 (excellent), how would you rate this paper and why have you given it this rating? For example:

* Are you rating it high due to having good examples, excellent screen shots, clear writing, or another reason?
* Are you rating it low due to poor examples, fuzzy screen shots, or unclear writing?

This feedback will help us improve the quality of white papers we release.

[Send feedback](mailto:sqlfback@microsoft.com?subject=White%20Paper%20Feedback:%20SSIS%20Hybrid%20and%20Azure).

# Appendix A – Example Coordinator Process

The C# code in this appendix creates SSIS packages to extract chunks of data from all tables in a database to a compressed file destination, upload these packages to blob storage, and enqueue a message containing the package name to the export queue. This is the application discussed in the [Scale-Out and Hybrid Data Movement](#_Scale-Out_and_Hybrid) section earlier in this article.

To use this as is, you must add references to Microsoft.SqlServer.DtsPipelineWrap; Microsoft.SQLServer.DtsRuntimeWrap; Microsoft.SQLServer.ManagedDTS; Microsoft.SqlServer.TxScript; and Microsoft.WindowsAzure.StorageClient. You also must download and install the Compressed File Components from http://compfilecomponents.codeplex.com/.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Reflection;

using Microsoft.SqlServer.Dts.Runtime;

using Microsoft.SqlServer.Dts.Pipeline.Wrapper;

using System.Data;

using System.Data.SqlClient;

// for the Azure access:

using Microsoft.WindowsAzure;

using Microsoft.WindowsAzure.StorageClient;

namespace CreateSSISFilesFromTemplate

{

    class MakePackages

    {

        static void Main(string[] args)

        {

            computedColumns = new HashSet<String>();

            MakePath();

            app = new Application();

            // get the list of tables:

            getTables();

            // initialize key one time:

            initializeCloudInfo();

            for (int i = 0; i < tables.Length; i++ )

            {

                makeSSISPackage(i);

                if (deleteLocalFiles)

                    System.IO.File.Delete(packagePath + files[i] + ".dtsx");

            }

           Console.WriteLine("Press Any Key to Continue...");

           Console.ReadKey();

        }

        /\*

         \* Convenience function to allow for simple, one-line

         \* enqueueing of packages to the export queue

         \*

         \* \*/

        private static void enqueSSISPackage(String fileName)

        {

            CloudQueueMessage msg = new CloudQueueMessage(fileName);

            queue.AddMessage(msg);

        }

        /\*

         \* uploadSSISPackage just uploads a single file to

         \* blob storage.

         \*

         \* \*/

        private static void uploadSSISPackage(String fileName)

        {

            // put together the file name to upload:

            CloudBlob blob = container.GetBlobReference(fileName);

            blob.Metadata["MetaName"] = fileName;

            // upload the file:

            blob.UploadFile(packagePath + fileName);

        }

        /\*

         \* Called one time at the beginning, the initializeCloudInfo uses the

         \* cloud info that was stored and gets the references needed for

         \* cloud blob and queue access throughout execution

         \*

         \* \*/

        private static void initializeCloudInfo()

        {

            // first, just connect to the azure storage account:

            key = new Microsoft.WindowsAzure.StorageCredentialsAccountAndKey(

                accountName, primaryAccessKey);

            account = new Microsoft.WindowsAzure.CloudStorageAccount(key, true);

            // get the blob client and container:

            client = account.CreateCloudBlobClient();

            container = new CloudBlobContainer(blobContainerName, client);

            container.CreateIfNotExist();

            BlobContainerPermissions permissions = container.GetPermissions();

            container.SetPermissions(permissions);

            // set up to work with the queue

            queueClient = account.CreateCloudQueueClient();

            queue = queueClient.GetQueueReference(queueName);

            queue.CreateIfNotExist();

        }

        /\*

         \* The makeSSISPackage Method does most of the work of creating

         \* a package from a template and uploading that package to blob

         \* storage. This is called with an integer telling it which table

         \* in the array of tables to create and upload an SSIS file for. It is

         \* called one time for each table for which an SSIS package for extraction is

         \* to be created.

         \* \*/

        private static void makeSSISPackage(int i)

        {

            Console.WriteLine("Tablename: " + tables[i] + " filename: " + files[i]);

            Package p = app.LoadPackage(templatePath + templatePackageName, null);

            // regenerate the GUID so we don't have conflicts on execution:

            p.RegenerateID();

            // take out the dataflow components:

            // Set the servername parameter

            // the ServerName parameter must exist in the package

            // and be used in the server connection

            p.Parameters["ServerName"].Value = serverName;

            p.Connections[0].AcquireConnection(null);

            MainPipe dataFlowTask = (MainPipe)((TaskHost)p.Executables[

                dataflowTaskName]).InnerObject;

            // make sure it is not null:

            if (dataFlowTask == null)

                return;

            // we now have the dataflow task, so rip its guts out.

            // delete the path:

            dataFlowTask.PathCollection.RemoveAll();

            // after the path is deleted, remove all the inputs,

            // and create a new input

            dataFlowTask.ComponentMetaDataCollection["CompressedFileDest"

                ].InputCollection.RemoveAll();

            // instantiate and provide component properties

            CManagedComponentWrapper outputInstance =

                dataFlowTask.ComponentMetaDataCollection["CompressedFileDest"

                ].Instantiate();

            outputInstance.ProvideComponentProperties();

            // set it to the new table

            CManagedComponentWrapper instance =

                dataFlowTask.ComponentMetaDataCollection["Source"].Instantiate();

            instance.ProvideComponentProperties();

            // ProvideComponentProperties resets the name. Change it back:

            dataFlowTask.ComponentMetaDataCollection["OLE DB Source"].Name = "Source";

            if (dataFlowTask.ComponentMetaDataCollection["Source"

                ].RuntimeConnectionCollection.Count > 0)

            {

                dataFlowTask.ComponentMetaDataCollection["Source"

                    ].RuntimeConnectionCollection[0].ConnectionManager =

                        DtsConvert.GetExtendedInterface(p.Connections[0]);

                            dataFlowTask.ComponentMetaDataCollection["Source"

                                ].RuntimeConnectionCollection[0].ConnectionManagerID =

                            p.Connections[0].ID;

                }

            // set the query, and the range values

            p.Parameters["SQLSourceQuery"].Value = "SELECT \* FROM " + tables[i];

            // for the moment, just set a range. we'll adjust before deployment:

            p.Parameters["BeginRange"].Value = 1.ToString();

            p.Parameters["EndRange"].Value = 1000000.ToString();

            // set other parameter names necessary:

            p.Parameters["DataFile"].Value = tables[i] + "." +

                fileSequence.ToString() + ".dat";

            p.Parameters["MaxRowsInBuffer"].Value = getMaxRowsInBuffer(tables[i]);

            p.Parameters["KeyColName"].Value = keyCols[i];

            // set the source component access mode

            // 0 is "Table or view", 3 is "SQL Query"

            instance.SetComponentProperty("AccessMode", 3);

            instance.SetComponentProperty("SqlCommandVariable",

                "User::FullSQLStatement");

            // now the metadata is all going to be invalidated

            // reset it:

            instance.Validate();

             instance.AcquireConnections(null);

            try

            {

                instance.ReinitializeMetaData();

            }

            catch (Exception e)

            {

                Console.WriteLine("Exception encountered on {0}\nMessage is: {1}",

                    tables[i], e.Message);

            }

            instance.ReleaseConnections();

            // set the Connection Manager

            if (dataFlowTask.ComponentMetaDataCollection["CompressedFileDest"

                ].RuntimeConnectionCollection.Count > 0)

            {

                dataFlowTask.ComponentMetaDataCollection["CompressedFileDest"

                    ].RuntimeConnectionCollection[0].ConnectionManager =

                    DtsConvert.GetExtendedInterface(p.Connections[1]);

                dataFlowTask.ComponentMetaDataCollection["CompressedFileDest"

                    ].RuntimeConnectionCollection[0].ConnectionManagerID =

                p.Connections[1].ID;

            }

            // create a path to connect the two

            IDTSPath100 path = dataFlowTask.PathCollection.New();

            // connect the two components:

            path.AttachPathAndPropagateNotifications(

                dataFlowTask.ComponentMetaDataCollection["Source"

].OutputCollection[0],

                dataFlowTask.ComponentMetaDataCollection["CompressedFileDest"

                    ].InputCollection[0]);

            // now, select all columns to be output:

            IDTSVirtualInput100 input =

                dataFlowTask.ComponentMetaDataCollection["CompressedFileDest"

                    ].InputCollection[0].GetVirtualInput();

            foreach (IDTSVirtualInputColumn100 column in

                input.VirtualInputColumnCollection)

            {

                // call the setUsageType method of the destination

                // to add each avaialble virtual input column as an input column

                try

                {

                    if (!computedColumns.Contains(String.Format("{0}.{1}", tables[i],

                        column.Name)))

                        outputInstance.SetUsageType(

                            dataFlowTask.ComponentMetaDataCollection[

"CompressedFileDest"].InputCollection[0].ID,

                            input, column.LineageID, DTSUsageType.UT\_READONLY);

                }

                catch (Exception e)

                {

                    Console.WriteLine(e.Message.ToString());

                }

            }

            // Validate the component so that all properties remain in tact.

            instance.Validate();

            instance.AcquireConnections(null);

            try

            {

                instance.ReinitializeMetaData();

            }

            catch (Exception e)

            {

                Console.WriteLine("Exception encountered on {0}\nMessage is: {1}", tables[i],

                    e.Message);

            }

            instance.ReleaseConnections();

            // save the package with the correct sequence numbers:

            // need to add the sequence number on the file

            // for this example, the query gets exact chunk sizes.

            // it is not very efficient, but all it needs to do

            // is get the first packages uploaded quickly to

            // get the export processes busy, then stay ahead

            // of the export processes. This is adequate for

            // that while maintaining the exact size

            // create the files by querying for the chunks:

            String SQLQuery = String.Format("SET NOCOUNT ON \n"

                + "DECLARE @minVal NVARCHAR(max) \n"

                + "declare @maxVal NVARCHAR(max) \n"

                + "declare @startVal NVARCHAR(max) \n"

                + "declare @chunkSize INT \n"

                + "set @startVal = @p0 \n"

                + "set @chunkSize = @p1 \n"

                + "\n"

                + "IF @startVal IS NULL \n"

                + " SELECT @minVal = MIN({0}) FROM {1} \n"

                + "ELSE \n"

                + " SELECT @minVal = MIN({0}) FROM {1} WHERE {0} > @startVal \n"

                + "\n"

                + "SELECT @maxVal = MAX({0}) FROM (SELECT TOP (@ChunkSize) {0} "

                + "FROM {1} WHERE {0} >= @MinVal ORDER BY {0}) chunk \n"

                + "SELECT @minVal, @MaxVal", keyCols[i], tables[i]);

            String connStr = "Data Source=" + serverName + ";Initial Catalog="

                + dbName + ";Integrated Security=true;";

            // create a prepared statement:

            try

            {

                SqlConnection conn = new SqlConnection();

                conn.ConnectionString = connStr;

                conn.Open();

                SqlCommand cmd = new SqlCommand(SQLQuery);

                cmd.CommandTimeout = 0;

                cmd.Connection = conn;

                SqlParameter p0 = cmd.CreateParameter();

                p0.Direction = ParameterDirection.Input;

                p0.DbType = DbType.AnsiString;

                p0.Value = DBNull.Value;

                p0.ParameterName = @"@p0";

                SqlParameter p1 = cmd.CreateParameter();

                p1.Direction = ParameterDirection.Input;

                p1.DbType = DbType.Int32;

                p1.Value = getMaxRange(tables[i]);

                p1.ParameterName = @"@p1";

                cmd.Parameters.Add(p0);

                cmd.Parameters.Add(p1);

                // now set up and read the values:

                Boolean wasNull = false;

                int j;

                j = 0;

                while (!wasNull)

                {

                    SqlDataReader rdr = cmd.ExecuteReader();

                    rdr.Read(); // only one record being returned by the results

                    if (rdr.IsDBNull(0))

                    {

                        wasNull = true;

                    }

                    else

                    {

                        String beginRange = rdr.GetString(0);

                        String endRange = rdr.GetString(1);

                        Console.WriteLine("beginRange: {0}, EndRange: {1}",

                            beginRange, endRange);

                        String fileName = String.Format("{0}.{1}.dtsx", files[i], j);

                        p.Parameters["BeginRange"].Value = beginRange;

                        p.Parameters["EndRange"].Value = endRange;

                        p.Parameters["DataFile"].Value = String.Format("{0}.{1}.dat",

                            files[i], j);

                        app.SaveToXml(packagePath + fileName, p, null);

                        // upload the package to blob storage blob:

                        uploadSSISPackage(fileName);

                        Console.WriteLine("Enqueueing {0}", fileName);

                        // enqueue the message

                        enqueSSISPackage(fileName);

                        // setup for the next execution.

                        p0.Value = endRange;

                    }

                    j++;

                    rdr.Close();

                }

                conn.Close();

            }

            catch (SqlException e)

            {

                Console.WriteLine("SQL Exception encountered: {0}", e.Message);

                Console.WriteLine(e.StackTrace);

            }

        }

        /\*

         \* Make sure the directory exists for the template and for

         \* the working directory

         \*

         \* \*/

        private static void MakePath()

        {

            if (! System.IO.Directory.Exists(templatePath))

                System.IO.Directory.CreateDirectory(templatePath);

            if (! System.IO.Directory.Exists(packagePath))

                System.IO.Directory.CreateDirectory(packagePath);

        }

        /\*

         \* Hardcoded exceptions to the chunk size in this case

         \* A more flexible approach would be to store this elsewhere

         \* and read that configuration at runtime

         \*

         \* \*/

        private static int getMaxRange(String tabName)

        {

            if (tabName == "[Production].[ProductPhoto]" || tabName ==

"[Production].[ProductPhoto1]")

                return 5000;

            if (tabName == "[Sales].[Individual]" || tabName ==

"[Sales].[Individual1]")

                return 2000000;

            if (tabName == "[Production].[TransactionHistory]" || tabName ==

"[Production].[TransactionHistory1]")

                return 10000000;

            // default case

            return 1000000;

        }

        /\*

         \* Setting the buffer size is critical to SSIS package performance

         \* Packages with BLOB values need to be set with much lower MaxRowsInBuffer

         \* values.

         \*

         \* Although this is hardcoded, another approach would be to store this

         \* externally and call read that configuration at runtime.

         \* \*/

        private static int getMaxRowsInBuffer(String tabName)

        {

            if (tabName == "[Production].[ProductPhoto]"

                || tabName == "[Production].[ProductPhoto1]")

                return 10;

            if (tabName == "[Sales].[Individual]"

                || tabName == "[Sales].[Individual1]")

                return 20;

            if (tabName == "[Production].[TransactionHistory]"

                || tabName == "[Production].[TransactionHistory1]")

                return 10000;

            // default case is 1 million for test purposes

            return 10000;

        }

        /\*

         \* The approach taken for this example is to read all the tables from the

         \* catalog views and order them by the number of rows DESC. This serves as

         \* an example, but for other moves that may not involve all tables, it may

         \* be better to store these in another table or file and read those at

         \* runtime as these are read at runtime. This allows for the list to

         \* change without coding changes.

         \* \*/

        private static void getTables()

        {

            String connectionString = "Data Source=" + serverName +

                ";Initial Catalog=" + dbName + ";Integrated Security=true;";

            // build the query string we need to retrieve the list of tables:

            String queryString = "SELECT t.file\_name, t.table\_name, t.rows, "

                + "(SELECT TOP 1 sc.name FROM sys.indexes i \n"

                + " JOIN sys.index\_columns c ON i.index\_id = c.index\_id \n"

                + " JOIN sys.columns sc ON c.object\_id = sc.object\_id \n"

                + " WHERE i.is\_primary\_key = 1 AND c.object\_id = t.object\_id \n"

                + " order by c.column\_id ) as first\_key\_col \n"

                + "FROM  \n"

                + "( \n"

                + " SELECT schema\_name(t.schema\_id) + '.' + t.name AS "

                + " [file\_name], \n"

                + " '[' + schema\_name(t.schema\_id) + ']' + '.' + "

                + "'[' +  t.name + ']' AS [table\_name], \n"

                + " SUM(p.rows) AS rows, t.object\_id \n"

                + " FROM sys.tables t \n"

                + " JOIN sys.partitions p on "

                + "t.object\_id = p.object\_id  \n"

                + " LEFT JOIN sys.indexes i ON t.object\_id = "

                + "i.object\_id AND i.is\_primary\_key = 1 \n"

                + " WHERE t.type\_desc = 'USER\_TABLE' \n"

                + "    GROUP BY t.schema\_id, t.name, t.object\_id \n"

                + ") AS t ORDER BY rows DESC \n";

            // set up the connection and query:

            SqlConnection conn = new SqlConnection();

            conn.ConnectionString = connectionString;

            conn.Open();

            SqlCommand cmd = new SqlCommand(queryString, conn);

            SqlDataReader rdr = cmd.ExecuteReader();

            System.Collections.ArrayList fileList =

                new System.Collections.ArrayList();

            System.Collections.ArrayList tableList =

                new System.Collections.ArrayList();

            System.Collections.ArrayList keyList =

                new System.Collections.ArrayList();

            while (rdr.Read())

            {

                fileList.Add(rdr.GetString(0));

                tableList.Add(rdr.GetString(1));

                keyList.Add(rdr.GetString(3));

            }

            rdr.Close();

            tables = (String[]) tableList.ToArray(typeof(String));

            files =  (String[])fileList.ToArray(typeof(String));

            keyCols = (String[])keyList.ToArray(typeof(String));

            // get the list of computed columns

            // this is to allow them to be ignored if needed:

            cmd.CommandText = "SELECT '[' + schema\_name(t.schema\_id) + ']' + "

                + " '.' + '[' +  t.name + ']' AS [table\_name], \n"

                + "         c.name FROM sys.tables t JOIN sys.columns c "

                + " ON t.object\_id = c.object\_id \n"

                + "         WHERE c.is\_computed = 1;";

            cmd.CommandTimeout = 0;

            rdr = cmd.ExecuteReader();

            while (rdr.Read())

            {

                computedColumns.Add(rdr.GetString(0) + "." + rdr.GetString(1));

            }

            conn.Close();

        }

        // Constants or variables to be used in the package

        // shown in this example to show what configurations were used

        // a better way would be to store these externally to make the

        // code reusable. Load the values at runtime either via

        // comman line parameters, external table, or file

        // path to the template package

        private const String templatePath = @"c:\MigrationPOC\";

        // path to store SSIS packages

        private const String packagePath = @"c:\packages\";

        private const String templatePackageName = @"TemplatePackage.dtsx";

        // name of source server

        private const String serverName = @"changed";

        // name of database

        private const String dbName = @"AdventureWorks";

        private const String pipelineString = @"SSIS.pipeline";

        private const String BlobURL = "changed.blob.core.windows.net";

        // get from storage account

        private const String primaryAccessKey = "changed";

        // where packages are stored

        private const String blobContainerName = "exports";

        // Azure storage account to use

        private const String accountName = "changed";

        private const String queueName = "export";

        private const String queueUrl = "changed.queue.core.windows.net";

        // the name of the dataflow task in the template

        private const String dataflowTaskName = "MoveTheData";

        /\*

         \* The variables below here are not configurations, but

         \* rather are used in the application

         \*

         \* \*/

        private static Application app;

        private static String[] tables;

        private static String[] files;

        private static String[] keyCols;

        private static int fileSequence = 0;

        private static bool deleteLocalFiles = true;

        private static Microsoft.WindowsAzure.StorageCredentialsAccountAndKey key;

        private static CloudStorageAccount account;

        private static CloudBlobClient client;

        private static CloudBlobContainer container;

        private static CloudQueueClient queueClient;

        private static CloudQueue queue;

        private static HashSet <String> computedColumns;

    }

}

# Appendix B – Example Extraction Process

The C# code in this appendix is an implementation of the program to download SSIS packages from blob storage, execute them, and upload them to blob storage as described in the [Scale-Out and Hybrid Data Movement](#_Scale-Out_and_Hybrid) section earlier in this article. It works with the code in Appendix A and Appendix C as part of the full solution.

This code requires references to Microsoft.SQLServer.ManagedDTS; and Microsoft.WindowsAzure.StorageClient.

using System;

using System.Data;

using Microsoft.SqlServer.Dts.Runtime;

using Microsoft.WindowsAzure.StorageClient;

using Microsoft.WindowsAzure;

using System.IO;

using System.IO.Compression;

namespace PopAndExtract

{

    public class PopAndExtract

    {

        public static void Main()

        {

            // setup configuration info for the movement

            PopAndExtract p = new PopAndExtract();

            p.BlobURL = "changed.blob.core.windows.net";

            p.PrimaryAccessKey = "changed";

            p.containerName = "exports";

            p.accountName = "changed";

            p.setWorkingPath(@"d:\DataLanding\");

            p.queueName = "export";

            p.queueURL = "changed.queue.core.windows.net";

            p.importQueueName = "import";

            p.getContainerAndQueue();

            p.deleteSourceFiles = true;

            // set up to be able to execute the package:

            p.app = new Microsoft.SqlServer.Dts.Runtime.Application();

            // start popping and executing:

            int iteration = 0;

            TimeSpan timeSpan = new TimeSpan(0, 10, 0);

            CloudQueueMessage msg = p.queue.GetMessage(timeSpan);

            p.fileName = (msg == null) ? "" : msg.AsString.Replace("[", ""

                ).Replace("]", "");

            while (true)

            {

                p.fileName = (msg == null) ? "" : msg.AsString.Replace("[", ""

                    ).Replace("]", "");

                if (msg != null && p.fileName != null && p.fileName.Length > 0)

                {

                    Console.WriteLine("Retrieving from blob to file: {0}",

                        p.fileName);

                    CloudBlob blob = p.container.GetBlobReference(p.fileName);

                    int maxIterations = 5;

                    int iterations = 0;

                    bool successDownloading = false;

                    {

                        while (!successDownloading && iterations++ < maxIterations)

                        {

                            try

                            {

                                blob.DownloadToFile(p.workingPath + p.fileName);

                                // we get here only if we were successful

                                successDownloading = true;

                            }

                            catch (Exception e)

                            {

                                Console.WriteLine(

                                    "Exception encountered while downloading {0}. " +

                                    "Iteration {1} of a maximum of {2}.",

                                    p.fileName, iteration, maxIterations);

                                Console.WriteLine("The error message was: {0}",

                                    e.Message);

                                if (iteration < maxIterations)

                                    Console.WriteLine("Retrying ... \n\n");

                                else

                                {

                                    Console.WriteLine("failed ... \n\n");

                                }

                            }

                        }

                    }

                    // next, execute the package:

                    DateTime begin = DateTime.Now;

                    Console.WriteLine("Preparing and executing package: {0}, " +

                        "Began at {1}", p.dtsxFileName, begin);

                    p.setupAndExecutePackage();

                    Console.WriteLine(p.result.ToString() + "\n\n" +

                        p.result.GetHashCode());

                    DateTime end = DateTime.Now;

                    Console.WriteLine("Executing package completed at {0}", end);

                    Console.WriteLine("Total time executing was {0}",

                        end.Subtract(begin).TotalSeconds);

                    // upload the file

                    String cabFileName = p.workingPath + p.dataFileName;

                    bool successUploading = false;

                    iteration = 0;

                    begin = DateTime.Now;

                    Console.WriteLine("Begin uploading at {0}", begin);

                    while (!successUploading && iteration++ < maxIterations)

                    {

                        Console.WriteLine("uploading {0}", p.dataFileName);

                        try

                        {

                            blob = p.container.GetBlobReference(p.dataFileName);

                            blob.UploadFile(p.workingPath + p.dataFileName);

                            successUploading = true;

                        }

                        catch (Exception e)

                        {

                           Console.WriteLine("Exception encountered while uploading”+

                                "{0}. Iteration {1} of a maximum of {2}.",

                                p.fileName, iteration, maxIterations);

                            Console.WriteLine("The error message was: {0}",

e.Message);

                            if (iteration < maxIterations)

                                Console.WriteLine("Retrying ... \n\n");

                            else

                            {

                                Console.WriteLine("failed ... \n\n");

                                p.success = false;

                            }

                        }

                        Console.WriteLine("\n\n");

                    }

                    end = DateTime.Now;

                    Console.WriteLine("Upload completed at {0}", end);

                    Console.WriteLine("Total time uploading was {0} seconds",

                        end.Subtract(begin).TotalSeconds);

                    // don't put it in the upload queue unless it was successful

// in uploading.

                    if (!successUploading)

                        continue;

                    CloudQueueMessage inputMessage = new CloudQueueMessage(

p.dataFileName);

                    if (p.success)

                        p.importQueue.AddMessage(inputMessage);

                    p.queue.DeleteMessage(msg);

                    // when you get here, you have completed the full upload.

                    // now cleanup and move to the next message:

                    if (p.deleteSourceFiles)

                    {

                        // delete the dtsx package:

                        blob = p.container.GetBlobReference(p.fileName);

                        blob.DeleteIfExists();

                        //delete the local dtsx package copy

                        System.IO.File.Delete(p.workingPath + p.dtsxFileName);

                        // delete the local data file:

                        System.IO.File.Delete(p.workingPath + p.dataFileName);

                    }

                    // cleanup complete

                    msg = p.queue.GetMessage(timeSpan);

                    p.fileName = (msg == null) ? "" : msg.AsString.Replace("[", ""

                        ).Replace("]", "");

                }

                while (msg == null)

                {

                    System.Threading.Thread.Sleep(5000);

                    msg = p.queue.GetMessage(timeSpan);

                }

                iteration = 0;

            }

        }

        public void setupAndExecutePackage()

        {

            Package p = app.LoadPackage(workingPath + fileName, null);

            dataFileName = p.Parameters["DataFile"].Value.ToString();

            // now execute the package:

            try

            {

                result = p.Execute();

                success = true;

            }

            catch (Exception e)

            {

                Console.WriteLine(fileName);

                Console.WriteLine(e.Message);

                Console.WriteLine(e.StackTrace);

                Console.WriteLine();

                success = false;

            }

        }

        public void setWorkingPath(String pWorkingPath)

        {

            if (!pWorkingPath.Substring(pWorkingPath.Length - 1).Equals(@"\"))

                pWorkingPath = pWorkingPath + @"\";

            workingPath = pWorkingPath;

        }

        public void getContainerAndQueue()

        {

            Microsoft.WindowsAzure.StorageCredentialsAccountAndKey key

                = new Microsoft.WindowsAzure.StorageCredentialsAccountAndKey(

                    accountName, PrimaryAccessKey);

            CloudStorageAccount account =

                new Microsoft.WindowsAzure.CloudStorageAccount(key, true);

            CloudBlobClient client = account.CreateCloudBlobClient();

            container = new CloudBlobContainer(containerName, client);

            container.CreateIfNotExist();

            BlobContainerPermissions permissions = container.GetPermissions();

            container.SetPermissions(permissions);

            // get the queues

            CloudQueueClient queueClient = account.CreateCloudQueueClient();

            queue = queueClient.GetQueueReference(queueName);

            importQueue = queueClient.GetQueueReference(importQueueName);

        }

        // variables used to hold current info on the work being done.

        String BlobURL;

        String PrimaryAccessKey;

        String containerName;

        String fileName;

        String accountName;

        String queueName;

        String importQueueName;

        CloudBlobContainer container;

        CloudQueue queue;

        CloudQueue importQueue;

        String dataFileName;

        String workingPath;

        String dtsxFileName;

        String queueURL;

        Microsoft.SqlServer.Dts.Runtime.Application app;

        DTSExecResult result;

        bool success;

        bool deleteSourceFiles;

        String databaseName = @"adventureworks";

        String SQLLogin = "sa";

        String SQLPwd = "changed";

        String SQLServerName = @"changed";

        Boolean useWindowsAuthentication = true;

        string messageText;

    }

 }

# Appendix C – Example Import Process

The C# code in this appendix is an example of the application that imports data in the scale-out solution, as outlined in the [Scale-Out and Hybrid Data Movement](#_Scale-Out_and_Hybrid) section earlier in this article. This program will poll the import queue, download the data file, build the SSIS package programmatically, and import the data into shards.

This code requires references to Microsoft.SqlServer.DTSPipelineWrap; Microsoft.SqlServer.DTSRuntimeWrap; Microsoft.SQLServer.ManagedDTS; and Microsoft.WindowsAzure.StorageClient.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using Microsoft.WindowsAzure;

using Microsoft.WindowsAzure.StorageClient;

using Microsoft.WindowsAzure.StorageClient.Protocol;

using System.Reflection;

using Microsoft.SqlServer.Dts.Runtime;

using Microsoft.SqlServer.Dts.Pipeline.Wrapper;

using System.Data;

using System.Data.SqlClient;

namespace ImportIntoAzure

{

    class Program

    {

        static void Main(string[] args)

        {

            if (args.Length > 0)

                setupConfigs();

            // make sure the paths exist to download files into and to unzip into

            MakePath();

            // initiate the cloud connection to download the info:

            initializeCloudInfo();

            // loop through and process files:

            TimeSpan t = new TimeSpan(1, 0, 0);

            while (true)

            {

                try

                {

                    CloudQueueMessage msg;

                    msg = queue.GetMessage(t);

                    while (msg == null)

                    {

                        System.Threading.Thread.Sleep(5000); // sleep for 5 seconds

                        msg = queue.GetMessage(t);

                    }

                    // deal with the possibility of a poison record

                    // for this example, just delete it.

                    // you may want to record this elsewhere to deal with it

                    // separately.

                    if (msg.DequeueCount > 5)

                    {

                        // delete the poison message:

                        queue.DeleteMessage(msg);

                        Console.WriteLine("Deleting poison message: {0}", msg.AsString);

                        continue;

                    }

                    // get our table and file names straight.

                    String[] nameParts = msg.AsString.Split('.');

                    tableName = "\"" + nameParts[0] + "\".\"" + nameParts[1] + "\"";

                    dataFileName = msg.AsString;

                    // download the file:

                    downloadAndUncab();

                    // Create and execute the package:

                    DateTime begin = DateTime.Now;

                    Console.WriteLine(

                        "Beginning creation of package and import at {0}", begin);

                    makeAndExecuteSSISPackage();

                    // execution will throw an exception if it was not successful

                    // in the maximum number of allowed retries.

                    // if that happens, this will drop to the catch block

                    // which will skip the delete of the message.

                    DateTime end = DateTime.Now;

                    Console.WriteLine("Import completed at {0}", end);

                    Console.WriteLine("Elapsed time for import was {0} seconds",

                        end.Subtract(begin).TotalSeconds);

                    // get rid of the intermediate files if so configured:

                    if (dropIntermediateFiles)

                    {

                        deleteIntermediateFiles();

                    }

                    // and delete the message:

                    queue.DeleteMessage(msg);

                }

                catch (Exception e)

                {

                    // put in a log event here

                    Console.WriteLine(tableName);

                    Console.WriteLine(e.StackTrace);

                    Console.WriteLine();

                    if (dropIntermediateFiles)

                        deleteIntermediateFiles();

                }

            }

        }

        private static void makeAndExecuteSSISPackage()

        {

            Application app = new Application();

            Package p = new Package();

            // add a dataflow task:

            Executable exec = p.Executables.Add("STOCK:PipelineTask");

            TaskHost thMainPipe = exec as TaskHost;

            thMainPipe.Name = "MoveDataToAzure";

            thMainPipe.MaximumErrorCount = 1000;

            MainPipe dataFlowTask =

                thMainPipe.InnerObject as MainPipe;

            thMainPipe.Properties["DefaultBufferMaxRows"].SetValue(

                thMainPipe, getMaxRowsInBuffer(tableName));

            thMainPipe.Properties["EngineThreads"].SetValue(

                thMainPipe, numShards + 10);

            thMainPipe.Properties["IsolationLevel"].SetValue(

                thMainPipe, 4096);

            thMainPipe.Properties["DefaultBufferSize"].SetValue(

                thMainPipe, 10485760 \* 5);  // this test has large objects

            IDTSComponentMetaData100 CompressedFileSource

                = dataFlowTask.ComponentMetaDataCollection.New();

            CompressedFileSource.ComponentClassID

                = app.PipelineComponentInfos["CompressedFileSource"].CreationName;

            CManagedComponentWrapper CompressedFileWrapper

                = CompressedFileSource.Instantiate();

            CompressedFileWrapper.ProvideComponentProperties();

            CompressedFileSource.Name = "CompressedFileSource";

            // create the connection manager for teh conditional split

            // and get the metadata so we can connect downstream:

            // set the connection manager for the CompressedFileSource:

            if (CompressedFileSource.RuntimeConnectionCollection.Count < 1)

            {

                CompressedFileSource.RuntimeConnectionCollection.New();

            }

            // now I know for sure there is a connection collection there.

            // I need a connection manager for it:

            ConnectionManager sourceConn = p.Connections.Add("FILE");

            sourceConn.Name = "SourceFile";

            sourceConn.ConnectionString = dataFilePath + dataFileName;

            sourceConn.Properties["FileUsageType"].SetValue(sourceConn, 0);

            CompressedFileSource.RuntimeConnectionCollection[0].ConnectionManager =

                DtsConvert.GetExtendedInterface(p.Connections["SourceFile"]);

            CompressedFileSource.RuntimeConnectionCollection[0].ConnectionManagerID =

                p.Connections["SourceFile"].ID;

            // the connection manager should be set

            // now reinitilize metadata

            CompressedFileWrapper.AcquireConnections(null);

            CompressedFileWrapper.ReinitializeMetaData();

            CompressedFileWrapper.ReleaseConnections();

            // Now add a conditional split for sharding:

            IDTSComponentMetaData100 conditionalSplit

                = dataFlowTask.ComponentMetaDataCollection.New();

            conditionalSplit.ComponentClassID

                = app.PipelineComponentInfos["Conditional Split"].CreationName;

            CManagedComponentWrapper conditionalSplitWrapper

                = conditionalSplit.Instantiate();

            conditionalSplitWrapper.ProvideComponentProperties();

            conditionalSplit.Name = "Shard splitter";

            // attach the input to the conditional split:

            IDTSPath100 path = dataFlowTask.PathCollection.New();

            path.AttachPathAndPropagateNotifications(

                CompressedFileSource.OutputCollection[0],

                conditionalSplit.InputCollection[0]);

            // set the properties of the column we will use in expressions:

            IDTSInput100 splitInput = conditionalSplit.InputCollection[0];

            IDTSInputColumnCollection100 splitInputColumns

                = splitInput.InputColumnCollection;

            IDTSVirtualInput100 splitVirtualInput

                = splitInput.GetVirtualInput();

            IDTSVirtualInputColumnCollection100 splitVirtualInputColumns

                = splitVirtualInput.VirtualInputColumnCollection;

            conditionalSplitWrapper.SetUsageType(

                splitInput.ID, splitVirtualInput,

                splitVirtualInputColumns[0].LineageID,

                DTSUsageType.UT\_READONLY);

            // create the conditions for the split

            // this is for demo purposes only as these are just

            // using modulo.

            IDTSOutputColumn100 conditionalColumn

                = CompressedFileSource.OutputCollection[0].OutputColumnCollection[0];

            String conditionalExpression = "";

            if (conditionalColumn.DataType

                    == Microsoft.SqlServer.Dts.Runtime.Wrapper.DataType.DT\_WSTR

                || conditionalColumn.DataType

                    == Microsoft.SqlServer.Dts.Runtime.Wrapper.DataType.DT\_STR

                || conditionalColumn.DataType

                    == Microsoft.SqlServer.Dts.Runtime.Wrapper.DataType.DT\_NTEXT)

                conditionalExpression +=

                    "CODEPOINT ([" + conditionalColumn.Name + "]) % "

                        + numShards.ToString();

            else

                conditionalExpression += "[" +

                    conditionalColumn.Name + "] % " + numShards.ToString();

            // we now have the metadata set,

            // so next add the destination connection adapter:

            for (int i = 0; i < numShards; i++)

            {

                String shardName

                    = String.Format("Shard{0}", i.ToString().PadLeft(2, '0'));

                IDTSOutput100 splitOutput

                    = conditionalSplitWrapper.InsertOutput(

                        DTSInsertPlacement.IP\_BEFORE,

                        conditionalSplit.OutputCollection[0].ID);

                splitOutput.Name = String.Format(shardName);

                splitOutput.Description = "Output to Shard";

                splitOutput.IsErrorOut = false;

                conditionalSplitWrapper.SetOutputProperty(splitOutput.ID,

                    "EvaluationOrder", i);

                conditionalSplitWrapper.SetOutputProperty(splitOutput.ID,

                    "FriendlyExpression", String.Format("ABS({0}) == {1}",

                    conditionalExpression, i));

                // add a connection manager where the data will go to

                // connect the conditional output to:

                ConnectionManager SQLAzureConnection = p.Connections.Add(

                    "ADO.NET:SQL");

                SQLAzureConnection.ConnectionString = getConnectionString(i);

                SQLAzureConnection.Description = String.Format(

                    "Connection to {0}", shardName);

                SQLAzureConnection.Name = String.Format(

                    "{0} Connection", shardName);

                // now create the destination adapter:

                IDTSComponentMetaData100 ADODestination

                    = dataFlowTask.ComponentMetaDataCollection.New();

                ADODestination.ComponentClassID

                    = app.PipelineComponentInfos["ADO NET Destination"].CreationName;

                ADODestination.ValidateExternalMetadata = true;

                CManagedComponentWrapper destWrapper

                    = ADODestination.Instantiate();

                destWrapper.ProvideComponentProperties();

                // name it AFTER calling ProvideComponentProperties

                // since that method resets its name

                ADODestination.Name = String.Format(

                    "ADO NET Destination for Shard{0}", i.ToString().PadLeft(2, '0'));

                destWrapper.SetComponentProperty("TableOrViewName", tableName);

                destWrapper.SetComponentProperty("BatchSize", 500);

                if (ADODestination.RuntimeConnectionCollection.Count > 0)

                {

                    // set the connection manager to above

                    ADODestination.RuntimeConnectionCollection[0].ConnectionManager =

                        DtsConvert.GetExtendedInterface(SQLAzureConnection);

                    ADODestination.RuntimeConnectionCollection[0

                        ].ConnectionManagerID =

                        SQLAzureConnection.ID;

                }

                // connect the source and destination:

                destWrapper.AcquireConnections(null);

                destWrapper.ReinitializeMetaData();

                destWrapper.ReleaseConnections();

                path = dataFlowTask.PathCollection.New();

                // connect the output for ths shard to the new destinaton:

                path.AttachPathAndPropagateNotifications(

                    conditionalSplit.OutputCollection[shardName],

                    ADODestination.InputCollection["ADO NET Destination Input"]);

                IDTSVirtualInput100 virtualInput

                    = ADODestination.InputCollection["ADO NET Destination Input"

                        ].GetVirtualInput();

                IDTSInput100 destInput

                    = ADODestination.InputCollection["ADO NET Destination Input"];

                IDTSInputColumnCollection100 destInputCols

                    = destInput.InputColumnCollection;

                IDTSExternalMetadataColumnCollection100 destExtCols

                    = destInput.ExternalMetadataColumnCollection;

                IDTSOutputColumnCollection100 sourceColumns

                    = CompressedFileSource.OutputCollection[0].OutputColumnCollection;

                // Map the columns

                foreach (IDTSOutputColumn100 outputCol in sourceColumns)

                {

                    IDTSExternalMetadataColumn100 extCol

                        = (IDTSExternalMetadataColumn100)destExtCols[outputCol.Name];

                    if (extCol != null)

                    {

                        virtualInput.SetUsageType(

                            outputCol.ID, DTSUsageType.UT\_READONLY);

                        IDTSInputColumn100 inputCol

                            = destInputCols.GetInputColumnByLineageID(outputCol.ID);

                        if (inputCol != null)

                        {

                            //map the input column with an external metadata column

                            destWrapper.MapInputColumn(

                                destInput.ID, inputCol.ID, extCol.ID);

                        }

                    }

                }

                // last thing, set the output to redirect error rows:

                ADODestination.InputCollection[0].ErrorOrTruncationOperation

                    = "Probably a primary key violation on retry";

                ADODestination.InputCollection[0].ErrorRowDisposition

                    = DTSRowDisposition.RD\_RedirectRow;

                ADODestination.Validate();

            }

            // execute the package:

            string localFileName = dataFilePath + @"importPackages\"

                + tableName.Replace("\"", "") + ".dtsx";

            Console.WriteLine("Importing {0}", dataFileName);

            results = p.Execute();

            // You can add in code to get the full log messages

            // from the executing package.

            if (saveLocalPackages)

                app.SaveToXml(localFileName, p, null);

            Console.WriteLine(results.ToString());

            // add code to reprocess failures.

            int retries = 1;

            int maxRetries = 5;

            while ((p.ExecutionResult == DTSExecResult.Failure

                || p.ExecutionStatus == DTSExecStatus.Abend) && retries++ < maxRetries)

            {

                Console.WriteLine("Error executing import of ", dataFileName);

                Console.WriteLine();

                ErrorEnumerator en = p.Errors.GetEnumerator();

                while ((en.MoveNext()) && en.Current != null)

                    Console.WriteLine(en.Current.Description);

                results = p.Execute();

                // throw an exception if we didn't get it in the

                // maximum number of retries

                if ((p.ExecutionResult == DTSExecResult.Failure

                    || p.ExecutionStatus == DTSExecStatus.Abend)

                    && retries>= maxRetries)

                    throw new Exception (

                        String.Format(

                            "Unable to process the file {0} in {1} attempts.",

                            fileName, maxRetries));

            }

        }

        private static String getConnectionString(int shardNo)

        {

            String curServer;

            if (shardNo > 149)

                curServer = serverName4;

            else if (shardNo > 99)

                curServer = serverName3;

            else if (shardNo > 49)

                curServer = serverName2;

            else

                curServer = serverName;

            //String Retval = String.Format("Data Source=

            String lDbName = "Shard";

            if (shardNo < 10)

                lDbName += "0";

            lDbName += shardNo.ToString();

            //Console.WriteLine(lDbName);

            return String.Format(

                "Data Source={0};Initial Catalog={1};User ID={2}@{0};" +

                "Password={3}; Connect Timeout=60;"

                , curServer, lDbName, dbAccountName, accountPwd);

        }

        private static void MakePath()

        {

            if (!System.IO.Directory.Exists(dataFilePath))

                System.IO.Directory.CreateDirectory(dataFilePath);

            if (!System.IO.Directory.Exists(downloadPath))

                System.IO.Directory.CreateDirectory(downloadPath);

            // make sure the directories are terminated properly:

            if (!dataFilePath.Trim().EndsWith(@"\"))

                dataFilePath = dataFilePath.Trim() + @"\";

            if (!downloadPath.Trim().EndsWith(@"\"))

                downloadPath = downloadPath.Trim() + @"\";

        }

        // download the file:

        private static void downloadAndUncab()

        {

            CloudBlob blob = container.GetBlobReference(dataFileName);

            //String lBlobFileName = downloadPath + fileName;

            String lDataFileName = dataFilePath + dataFileName;

            int iteration = 0;

            bool success = false;

            int maxIterations = 5;

            DateTime begin = DateTime.Now;

            Console.WriteLine("Beginning download at {0}", begin);

            while (!success && iteration++ < maxIterations)

            {

                try

                {

                    blob.DownloadToFile(lDataFileName);

                    // if we get here, then it was successful

                    success = true;

                }

                catch (Exception e)

                {

                    if (iteration < maxIterations)

                    {

                        Console.WriteLine("Exception encountered while downloading " +

                            "{0}.Iteration {1} of a maximum of {2}."

                            , fileName, iteration, maxIterations);

                        Console.WriteLine("The error message was: {0}", e.Message);

                        Console.WriteLine("Retrying ... \n\n");

                    }

                    else

                    {

                        Console.WriteLine("Exception encountered while downloading " +

                            "{0}. Iteration {1} of a maximum of {2}."

                            , fileName, iteration, maxIterations);

                        Console.WriteLine("The error message was: {0}", e.Message);

                        Console.WriteLine("No more retries ... \n\n");

                        //rethrow the exception:

                        throw e;

                    }

                }

            }

            DateTime end = DateTime.Now;

            Console.WriteLine("Completed download at {0}", end);

            Console.WriteLine("Elapsed time for download was {0} seconds",

                end.Subtract(begin).TotalSeconds);

        }

        private static void initializeCloudInfo()

        {

            // first, just connect to the azure storage account:

            key = new Microsoft.WindowsAzure.StorageCredentialsAccountAndKey(

                accountName, primaryAccessKey);

            account = new Microsoft.WindowsAzure.CloudStorageAccount(key, true);

            // get the blob client and container:

            client = account.CreateCloudBlobClient();

            client.Timeout = new TimeSpan(1, 0, 0);

            container = new CloudBlobContainer(blobContainerName, client);

            container.CreateIfNotExist();

            BlobContainerPermissions permissions = container.GetPermissions();

            container.SetPermissions(permissions);

            // set up to work with the queue

            queueClient = account.CreateCloudQueueClient();

            queue = queueClient.GetQueueReference(queueName);

            queue.CreateIfNotExist();

        }

        private static int getMaxRowsInBuffer(String tabName)

        {

            if (tabName == "\"Production\".\"ProductPhoto\""

                || tabName == "\"Production\".\"ProductPhoto1\"")

                return 10;

            if (tabName == "\"Sales\".\"Individual\""

                || tabName == "\"Sales\".\"Individual1\"")

                return 20;

            if (tabName == "\"Production\".\"TransactionHistory\""

                || tabName == "\"Production\".\"TransactionHistory1\"")

                return 10000;

            // default case is ten thousand for test purposes

            return 10000;

        }

        private static void deleteIntermediateFiles()

        {

            Uri filePath = new Uri(downloadPath);

            System.IO.DirectoryInfo dldi = new System.IO.DirectoryInfo(

                filePath.AbsolutePath);

            filePath = new Uri(dataFilePath);

            System.IO.DirectoryInfo dfdi = new System.IO.DirectoryInfo(

                filePath.AbsolutePath);

            foreach (System.IO.FileInfo fi in

                dldi.GetFiles(tableName.Replace("\"", "") + "\*.\*"))

            {

                fi.Delete();

            }

            foreach (System.IO.FileInfo fi in

                dfdi.GetFiles(tableName.Replace("\"", "") + "\*.\*"))

            {

                fi.Delete();

            }

        }

        private static void setupConfigs()

        {

            /\*

             \* -d is data file and download path

             \* -s is the number of shards

             \* \*/

            String[] configs = Environment.GetCommandLineArgs();

            for (int i = 0; i < configs.Length; i++)

                Console.WriteLine("parameter {0} is {1}", i, configs[i]);

            for (int i = 1; i < configs.Length; i += 2)

            {

                if (!configs[i].StartsWith(@"-") && !configs[i].StartsWith(@"/"))

                {

                    IncorrectArgs();

                }

                if (!(i + 1 < configs.Length))

                    IncorrectArgs();

                if (!(configs[i].Substring(1, 1).Equals(

                    "d", StringComparison.Ordinal)

                        || configs[i].Substring(1, 1).Equals(

                        "s", StringComparison.Ordinal)

                    ))

                    IncorrectArgs();

                else if (configs[i].Trim().EndsWith("d", StringComparison.Ordinal))

                    dataFilePath = configs[i + 1].Trim();

                else if (configs[i].Trim().EndsWith("s", StringComparison.Ordinal))

                    if (!Int32.TryParse(configs[i + 1].Trim(), out numShards))

                        IncorrectArgs();

            }

        }

        private static void IncorrectArgs()

        {

            String progname = Environment.CommandLine.Split(' ')[0];

            Console.WriteLine("Usage is: {0} -d datafilePath -s 20", progname);

            Environment.Exit(1);

        }

        /\*

         \* Class variables

         \* Make to run as a static app

         \* \*/

        // ideally, instead of putting values into your code,

        // you would store it externally so you can reuse the

        // app without recoding. This example puts them directly

        // into the code only.

        private static String dataFilePath = @"c:\MigrationPOC\";

        private static String downloadPath = @"c:\MigrationPOC\";

        private static String serverName = "changed.database.windows.net";

        private static String serverName2 = "changed2.database.windows.net";

        private static String serverName3 = "changed3.database.windows.net";

        private static String serverName4 = "changed4.database.windows.net";

        private static String dbName = @"adventureworks";

        private static String BlobURL = @"changed.blob.core.windows.net";

        private static String primaryAccessKey = "changed";

        private static String blobContainerName = @"exports";

        private static String accountName = "changed";

        private static String dbAccountName = "changed";

        private static String accountPwd = "changed";

        private static String queueName = "import";

        private static bool saveLocalPackages = false;

        // other variables used in the program

        private static Application app;

        private static String fileName;

        private static String tableName;

        private static String dataFileName;

        private static Microsoft.WindowsAzure.StorageCredentialsAccountAndKey key;

        private static bool deleteFilesFromBlob = true;

        private static bool dropIntermediateFiles = true;

        private static CloudStorageAccount account;

        private static CloudBlobClient client;

        private static CloudBlobContainer container;

        private static CloudQueueClient queueClient;

        private static CloudQueue queue;

        private static DTSExecResult results;

        private static int numShards = 10;

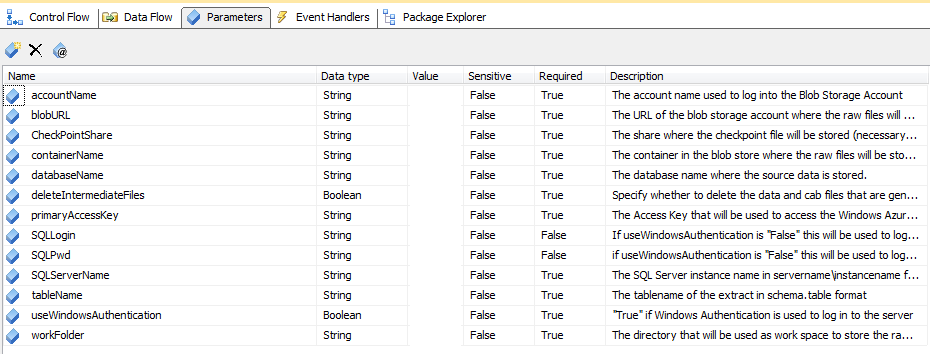
        private static string workingDir;

    }

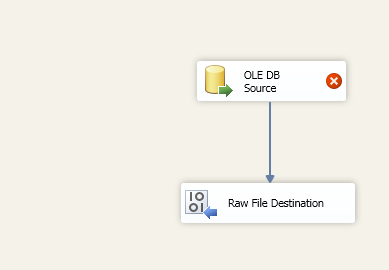
}

# Appendix D – Example Template Package

This appendix shows key parameters and data flow components for the template package used by the code in Appendix A. Note that using a template package is entirely optional as a package can be built in memory without the use of a template. A template was used as a demonstration. As an additional tip, package parameters are a good place to store additional information for subsequent processes without using Azure Tables, Files, or other external data sources.



Parameters used in the template package. These are used in creating connection strings in connection managers and setting up for execution of the package. They can be changed by the program using this template prior to saving the version to be executed.



Data Flow of the template package.