Dynamic ETL with SSIS

By Sarah Doriss, 2013/05/03 (first published: 2010/10/19)

This article is about how to dynamically load flat files using SSIS as a shell for basic extraction functionality while all transformations and business logic run in the database. All load criteria for what to load, where to store the data, and transformation of the data happen in the database. In my design, the SSIS package downloads the flat file from an ftp site, unzips it, extracts it and then loads the data into the database. The data definition and all criteria for what needs to be loaded, extracted, and how it will be stored is saved and retrieved from the database. By storing all criteria in the database, the SSIS package no longer has to be modified for new types of files, new imports or to test a subset (say one file out of five) of a load. All business logic is processed in the database while SSIS performs basic extraction functionality.

My initial idea for designing this download and transform process was instigated by new files that were being offered from our data provider and a slow testing process associated with testing the package. We had a traditional file load set up using SSIS where all of the configuration data was stored in the package. The package became difficult to test because it needed to be modified for each test. New files would be provided by the data provider and that meant time to change the package and test it against all file downloads we were receiving. I decided to make the entire process configurable and to use SSIS for what I really needed it for which was the extracting, unzipping and importing of the data.

Weekly at my organization we download files from a data provider. The data provider has provided us with documentation of the data that it is providing to us so we know what format we are receiving. These files are provided in zip format and contain multiple flat files within the zip. Because the provider errs on the side of providing more information than less, it gives us many files which tend to be redundant and are not necessary for us to load. These include copyright information, data dictionaries (which do not change) and file processed date information. Of the files we do need to download, some consist of fixed length data, while others include delimited data that needs to be parsed. Depending on the data that we need, we may be extracting one file from a zip or many.

Related Articles

FORUM Pipe delimited VARCHAR column
Need to expand a pipe delimited string stored in a VARCHAR(4096)

SCRIPT Delimited String Parser
Parses delimited string into a table of up to 9 varchar fields.

FORUM COALESCE?Create comma delimited list field from table field
Distinct delimited list field

FORUM Double Delimited String Parsing and Re-Concatenation Help
Double Delimited

FORUM "Select Where In" using a parameter?
select intCol from Tablename where intCol in (@intList)

Tags
etl
integration services (ssis)

Write For Us
Find out more

First, let's start our design with the tables we will need for our package parameters.

The ETL_ZipFiles table contains all the zip file information received from our data provider. I give each file a filecode for lookup references in code (instead of hard coding a number into my sprocs). There is a file prefix since each file starts with a prefix and then has a date appended to it. The description and server location are self explanatory. The Run Order is the order in which the files will be run. The Active indicator indicates what files to run. The active indicator is helpful in testing when you are only testing one file as it allows you to turn off files you don't need to test. The FTP location is where to retrieve the file from.

Here is some sample data:

ETLFileId	FileCode	FilePrefix	FileDescription	ServerLocation	FolderNamePrefix	RunOrder	Active	FTPLocation
1	FA	FA_F	Weekly FA Load	\\server\drive\FA\	FA	1	1	/ftplocation/
2	FB	FB_F	Weekly FB Load	\server\drive\FB\	FB	2	1	ftplocation/
3	FC	FC_F	Weekly FC Load	\server\drive\FC\	FC	3	1	ftplocation/
4	FD	FD_F	Weekly FD Load	\server\drive\FD\	FD	4	1	ftplocation/
5	SU	SU_F	Weekly SU Load	\server\drive\SU\	SU	5	1	ftplocation/

```
CREATE TABLE [dbo].[ETL_ZipFiles](

 [ETLFileId] [int] IDENTITY(1,1) NOT NULL,

 [FileCode] [char](2) NOT NULL,

 [FilePrefix] [varchar](100) NULL,

 [FileDescription] [varchar](500) NULL,

 [ServerLocation] [varchar](500) NULL,

 [FolderNamePrefix] [varchar](100) NULL,

 [RunOrder] [int] NULL,

 [Active] [bit] NULL,

 [FTPLocation] [varchar](200) NULL
```

The following ETL_ImportFiles table now describes the files within each of the zip files in the ETL_ZipFiles table. The ImportFilename is the actual name of the file within the zip file. The LoadTableName is the name of the table where the data from the file will be imported to. Once the data has been imported in raw format (not parsed or delimited) it is then moved to the DestTableName, which is the name of the final table for the data. Data is moved to the final table after parsing or delimiting and holds the data in a format for business processing. The delimited indicator indicates whether this is a delimited file or not. If it is not, it is assumed to be a fixed length file. The delimiter is what is delimiting the file and the DestFieldToIndex indicates what field to index on the DestTableName (final output).

Here is some sample data. In this example there is only one file per zip:

ETLImportId	ELTFileId	FileCode	ImportFileName	LoadTableName	DestTableName	Delimited	Delimiter	DestFieldToIndex
1	1	FA	FA_File1	Load_FAFile1	FAOutput	0		FAField1
2	2	FB	FB_Extract	Load_FBExtract	FBOutput	0		FBField1
3	3	FC	FC_Data	Load_FCData	FCOutput	1	|	FCField1
4	4	FD	FD_Listing	Load_FDListing	FDOutput	1	|	FDField1
5	5	SU	SU_Extract	Load_SUExtract	SUOutput	0		SUField1

```
CREATE TABLE [dbo].[ETL_ImportFiles](

 [ETLImportFileId] [int] IDENTITY(1,1) NOT NULL,

 [ETLFileId] [int] NOT NULL,

 [FileCode] [char](2) NOT NULL,

 [ImportFileName] [varchar](100) NULL,

 [LoadTableName] [varchar](100) NULL,

 [DestTableName] [varchar](100) NULL,

 [Delimited] [bit] NOT NULL,

 [Delimiter] [char](1) NULL,

 [DestFieldToIndex] [varchar](100) NULL
```

This ETL_ImportDictionary is the data dictionary to your file. You will most likely populate this with data dictionary information from your data provider. The first field in the table is the primary key, the second two match it up to the import file (not zip) it associated with, and the FieldName corresponds to the name of the field that will be coming through. There is a ParseStartPoint and ParseEndPoint for files that are fixed length. These fields can be null or 0 for fields that are delimited. The DataTypeDescription can be a varchar or an nvarchar. With my files I am making everything into a varchar or nvarchar and the computation to the correct datatypes happens at another point after the data has been transformed in. You could change this piece of code if you wanted to transform the data to a different datatype at the time of entry. The FieldsOrder field indicates the order of the fields within the file and also once it is extracted to the table.

Here is a sample data dictionary for one file within a zip that has five fields in it. It is fixed length with 100 characters of data in a row:

ETLImportDictionaryId	ETLImportFileId	FileCode	FieldName	ParseStartPoint	ParseEndPoint	DatayType Description	Run Order
1	1	SU	Field1	1	11	varchar(11)	1
2	1	SU	Field2	12	12	varchar(1)	2
3	1	SU	Field3	13	15	varchar(3)	3
4	1	SU	Field4	16	16	varchar(1)	4

5	1	SU	Field5	17	100	varchar(84)	5

```sql
CREATE TABLE [dbo].[ETL_ImportDictionary](

 [ETLImportDictionaryId] [int] IDENTITY(1,1) NOT NULL,

 [ETLImportFileId] [int] NOT NULL,

 [FileCode] [char](2) NOT NULL,

 [FieldName] [varchar](200) NULL,

 [ParseStartPoint] [int] NULL,

 [ParseEndPoint] [int] NULL,

 [DataTypeDescription] [varchar](100) NULL,

 [FieldsOrder] [int] NULL

) ON [PRIMARY]
```

The ETL_Run table holds the run information. A run is essentially the instigation of the download and transform process for each designated group of zip files. Once you have set what zip files you are retrieving and which are active, a run is essentially the transformation and load of those designated zip files. The ETL_Run table is populated before the process is kicked off and updated when the process is completed. The table is first populated with the information required for the run and once the run is complete, the table is updated with the information that indicates that the run is now complete. So the table acts as the parameters for the process as it runs, and once the process is completed it acts as a history of previous runs. The Filename, FolderName and FileLocation all have taken the information from the ETL_ZipFiles table and used it to create the ETL run that it will start. Once the run is complete, the completed indicator will be set, the duration will be set and the imported indicator will be set. If an error occurs, the containserrors indicator is set. Each run loads the parameters from the table which are not completed. If there is an error, you would want to come back, fix it and update the table so that the run was completed before you ran your next transformation.

Here is a sample extract of data that might exist in your run table after your extraction process completed successfully. Notice that the FileName and the FolderName have been generated from the ETL_ZipFiles table criteria by using the FilePrefix and FolderNamePrefix fields and concatenating the run date. This allows us to name files and folders by run date so that we are not overwriting older files and we can find related files if we need to. In my package however, I have allowed files to be overwritten if the process gets kicked off again and the files are pre-existing. If I want to re-run a file that I have created a run for I can either clear out the run and re-queue it or change the completed and imported indicators. This may be necessary when testing your import files.

ETLRunId	ETLFileId	FileCode	FileName	FolderName	FileLocation	Run Start	Run End	Completed	Contains Errors	Duration	Imported	FileDate
1	1	FA	FA_F20100915	FA20100915	\\server\drive\FA20100915	2010-09-15 23:00:00	2010-09-15 23:12:00	1	0	12	1	2010-09-15 00:00:00
2	2	FB	FB_F20100915	FB20100915	\\server\drive\FB20100915	2010-09-15 23:00:00	2010-09-15 23:12:00	1	0	12	1	2010-09-15 00:00:00
3	3	FC	FC_F20100915	FC20100915	\\server\drive\FC20100915	2010-09-15 23:00:00	2010-09-15 23:12:00	1	0	12	1	2010-09-15 00:00:00
4	4	FD	FD_F20100915	FD20100915	\\server\drive\FD20100915	2010-09-15 23:00:00	2010-09-15 23:12:00	1	0	12	1	2010-09-15 00:00:00
5	5	SU	SU_F20100915	SU20100915	\\server\drive\SU20100915	2010-09-15 23:00:00	2010-09-15 23:12:00	1	0	12	1	2010-09-15 00:00:00

```
CREATE TABLE [dbo].[ETL_Run](

 [ETLRunId] [int] IDENTITY(1,1) NOT NULL,

 [ETLFileId] [int] NOT NULL,

 [FileCode] [char](2) NOT NULL,

 [FileName] [varchar](200) NOT NULL,

 [FolderName] [varchar](100) NULL,

 [FileLocation] [varchar](500) NULL,

 [RunStart] [smalldatetime] NULL,

 [RunEnd] [smalldatetime] NULL,

 [Completed] [bit] NOT NULL,
```

You have now added all the main configuration tables used in the SSIS package. One of the reasons that you cannot easily dynamically load FlatFile data with SSIS is because SSIS requires you to specify a layout for the file that is being imported. In my case, since this is dynamic and the dictionary is stored in the database and not the package, I created a trick to get around specifying detail field conversion information in the package. The way I get around this in my SSIS package is that my Load_* tables referenced in my package essentially hold two fields. One is an Identity field, and the other is called "EverythingElse". The EverythingElse field holds the entire unparsed row of data that is coming in from the provider. This essentially makes our initial load tables into holding tables of the raw extracted data. This allows us to have all the parsing/delimiting occur in the database and once the process has completed, the Load_* tables are cleared out for the next extraction process.

The following stored procedure populates the ETL_Run table from the ETL_ZipFiles table so that the parameters are set and the run is now ready to be executed. You'll notice that this sproc takes a date parameter and if it is not passed in, it determines the next Wednesday's date as the date of my file. I have not included that function with this description as it is not relevant, but feel free to create your own function depending on when your file is coming in. The calculation of the date of the next file was an important part needed for our testing because our test database is not always up to date with the most recent backups. Our files are update files and run consecutively. They cannot be run out of order. This date calculation allows the system to determine what file is needed next based on the previous run's date instead of the date closest to the time of execution.

In our process, if we ran our files out of order, our business conversion process would kick out the run because of validation that occurs to make sure we are executing the processing in the right order. However, if we have a test environment that has backups of our database from a month ago and we want to test this week's update, we would run our process without passing in the date and each run we kicked off would extract the next week's file until we got to the current week's file. A run for us is one week's worth of files, so if our database was three weeks out of date we would execute three runs (each containing in this case five files). Everyone's data extraction is different with different rules and delivery methods. The run criteria could certainly be changed here to accommodate the delivery conventions of your organization. The first time you kick off a run you will need to enter the date as a parameter because the table is empty and does not have an existing date to calculate on.

```
CREATE PROCEDURE [dbo].[Package_QueueRun]

 @RunDate smalldatetime = NULL

AS



SET NOCOUNT ON



DECLARE @FileDate varchar(8),

 @LastRunDate smalldatetime
--if run date not passed in - get the next dated file

IF @RunDate IS NULL

BEGIN
```

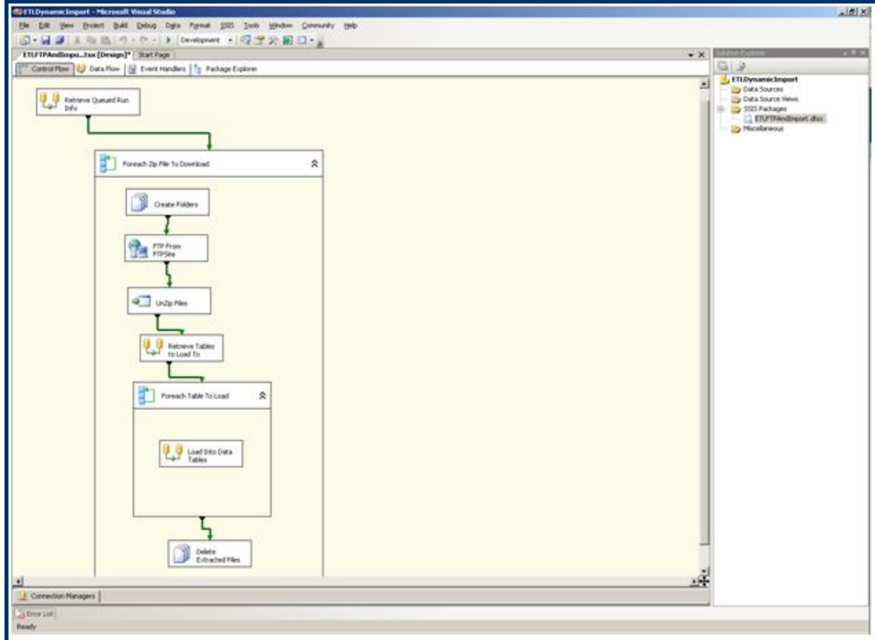
The next step is to kick off the Package_CreateLoadTables sproc, which will drop the load tables if they exist and create the load tables referenced in the ETL_ImportFiles table. All of this is only acting on files that are active.

The load tables created in this sproc contain the Id and EverythingElse fields. For my load tables, a varchar(500) is sufficient. You may need to change this to nvarchar or increase the size depending on the row length of the file that you are retrieving.

```
CREATE PROCEDURE [dbo].[Package_CreateLoadTables]

AS


SET NOCOUNT ON


DECLARE @SQL nvarchar(max),

 @DropSQL nvarchar(max)
SET @SQL = ''

SET @DropSQL = ''

--Create DROP TABLE SQL

SELECT @DropSQL = @DropSQL +
```

Once the Package_QueueRun and the Package_CreateLoadTables sprocs have been kicked off, we can now run our SSIS package.

This is what my SSIS package looks like:



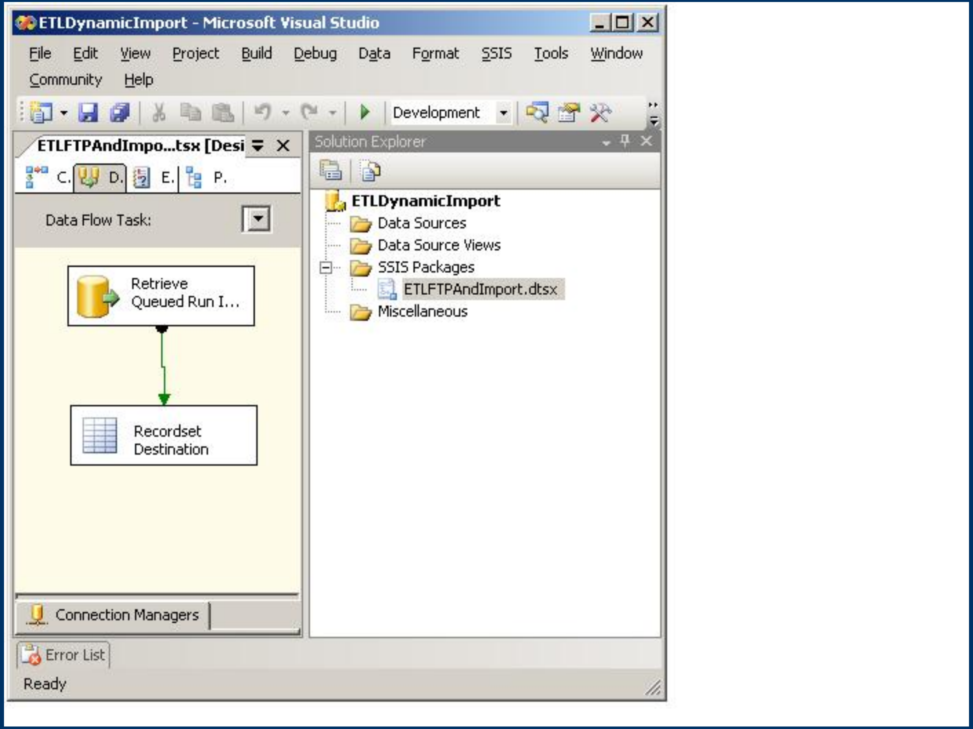
🔍 Zoom in | Open in new window

The first task retrieves the run information. It is a data flow task that runs the following sproc and then dumps the information into a Recordset Destination to be iterated on.

```
CREATE PROCEDURE [dbo].[Package_RunInfo]

AS

BEGIN

 SET NOCOUNT ON;

SELECT [ETLRunId]

 ,mr.[FileCode]

 ,mr.[FileName]

 ,mr.[FolderName]

 ,mr.[FileLocation] as ZipLocation

 ,mr.[FileLocation] + 'ExtractFiles\' as FileLocation
```

There are two different locations that I use: one is the ZipLocation and the other is the FileLocation. The file location is where the files are unzipped to. Once the load is complete, this folder is then deleted so that we are not carrying more files than needed. We keep all of our original zips, but not the zips and the extracted files.

Here is the dataflow task for the run:

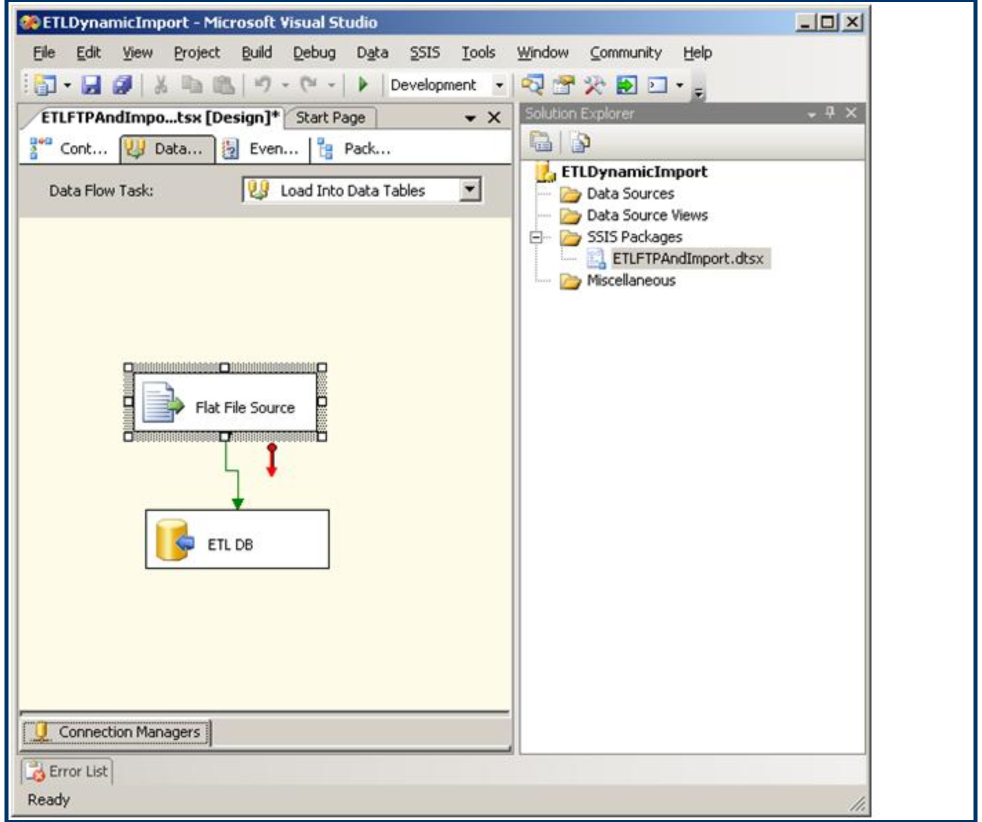


🔍 Zoom in | Open in new window

Once the recordset is populated, there is a For Each Loop that iterates the results (which are placed in variables) in the recordset. The package then creates the folders, ftp's the file, unzips the files and retrieves the tables to load to. The load table information is returned within the loop because in each zip file you could have multiple files associated and thus multiple load tables. The load table information is retrieved using the following sproc:

```
CREATE PROCEDURE [dbo].[Package_GetLoadTable]

 @FileCode char(2)

AS


SET NOCOUNT ON


BEGIN


Select LoadTableName, ImportFileName

From dbo.[ETL_ImportFiles] mi

JOIN dbo.ETL_zipfiles mz on mi.FileCode = mz.FileCode
```

Then we loop again over all the load tables that are returned and Import the data into the associated Load_* table. Since all files will be set up the same way, the SSIS package is set up with a dynamic Flat File Source that loads to a dynamic ETL DB. The Data Flow task looks like this:



Zoom in | Open in new window

After the data is loaded into the Load_* tables, the last step is that we delete the files that were extracted.

This is the last step of the SSIS package and we are now ready to transform our load tables to our destination tables.

We call the following sproc which creates the destination tables by dropping and then recreating them.

```
CREATE PROCEDURE [dbo].[Package_CreateDestinationTables]

AS


SET NOCOUNT ON


BEGIN


DECLARE @SQL nvarchar(max),

 @DropSQL nvarchar(max)
DECLARE @DestTableCreate TABLE(ETLImportFileId int,

 BeginString nvarchar(2000), FieldString nvarchar(2000), EndStri
SET @SQL = ''
```

The following function is referenced in the above procedure. It creates Nullable fields based on the parameters in the ImportDictionary.

```
CREATE FUNCTION [dbo].[GetConcatenatedFieldsWithType]

(@ETLImportFileId int)

RETURNS nvarchar(2000)

AS

BEGIN

DECLARE @Fields nvarchar(2000)
SET @Fields = ''
SELECT @Fields = @Fields + '[' + FieldName + '] ' + DataTypeDesc

FROM ETL_ImportDictionary id

Where id.ETLImportFileId = @ETLImportFileId
```

Then we call the following sproc that creates the indexes:

```
CREATE PROCEDURE [dbo].[Package_CreateIndexes]

AS


SET NOCOUNT ON


BEGIN


DECLARE @IndexSQL nvarchar(max)
SET @IndexSQL = ''

--Create DROP TABLE SQL

SELECT @IndexSQL = @IndexSQL +
```

Now we are ready to transform the large string of data into the parsed/delimited string. The data at this point will be transformed and saved to our destination tables. We call this sproc:

```
CREATE PROCEDURE [dbo].[Package_PopulateDestination]

AS


SET NOCOUNT ON


BEGIN


DECLARE @SQL nvarchar(max)
SET @SQL = ''
IF Exists (Select 1

 From dbo.[ETL_ImportFiles] mi

 JOIN dbo.ETL_zipfiles mz on mi.FileCode = mz.FileCode
```

The following sproc transforms the data based on whether it is delimited or fixed length. The two main functions used are CreateDelimitedTableString and CreateParsedTableString. These functions generate the SQL to transform the data and are described here:

```
CREATE FUNCTION [dbo].[CreateDelimitedTableString]

(@ETLImportFileId int)

RETURNS nvarchar(max)

AS

BEGIN

DECLARE @DestTableName varchar(100),

 @LoadTableName varchar(100),

 @FieldNames varchar(2000),

 @Select varchar(5000),
```

This function is creating the sql to delimit the strings in a table. An example of SQL output for this is here:

```
Insert into [RN_UPD]([FieldA], [FieldB], [FieldC], [FieldD], [Fiel

SELECT [1],[2],[3],[4],[5],[6],[7],[8],[9]

FROM

 (Select d.Id, WordNumber, Word

 FROM dbo.[Load_RNFile] d

 Cross Apply dbo.DelimitedItem(d.EverythingElse,'|')) p

 PIVOT (MAX([Word]) FOR WordNumber in ([1],[2],[3],[4],[5],[6],[7]
```

So what we are doing here is pivoting on a Table Value Function that essentially is breaking down each word in the string so that it is a row. The following piece of code within this SQL Query:

```
Select d.Id, WordNumber, Word

 FROM dbo.[Load_RN] d

 Cross Apply dbo.DelimitedItem(d.EverythingElse,'|')
```

Returns the following sample output:

Id	WordNumber	Word
1	1	FieldA1Data
1	2	FieldB1Data
1	3	FieldC1Data
1	4	FieldD1Data
1	5	FieldE1Data
1	6	FieldF1Data
1	7	FieldG1Data
1	8	FieldH1Data
1	9	FieldI1Data
2	1	FieldA2Data
2	2	FieldB2Data
2	3	FieldC2Data
2	4	FieldD2Data
2	5	FieldE2Data
2	6	FieldF2Data
2	7	FieldG2Data
2	8	FieldH2Data
2	9	FieldI2Data

So we are essentially breaking up the field data into rows. The Id field represents the row number in the file, the word number represents the field number within the row, and the word represents the data in the file at that row and field number.

The table valued function used to rotate this data is here:

```
CREATE FUNCTION [dbo].[DelimitedItem]

(

 -- Add the parameters for the function here

 @String varchar(8000),

 @Delimiter char(1)

)

RETURNS

@ListTable TABLE

(
```

This function essentially uses a Recursive CTE to break the string into a list of parsed words. The anchor member starts with a word number of 1 and a start point of one. It determines the length of the first word and using the substring and charindex functions to parse out the word. The delimiter is added to the end of the string in case the string happens to be empty at which time it would return an ending point of 1 and a word of blank. The recursive member increments the word number and has a starting point at the previous ending point plus one. The ending point is the point of the next delimiter and through the substring and charindex functions we determine what the next word is as long as there is another delimiter. So each call to this TVF returns a table of WordNumbers, the starting point and ending point of that word and the word itself. The start and endpoints are great for debugging, but for my use I actually don't need them.
So going back to this code:

```
Insert into [RN_UPD]([FieldA], [FieldB], [FieldC], [FieldD], [Fiel

SELECT [1],[2],[3],[4],[5],[6],[7],[8],[9]

FROM

 (Select d.Id, WordNumber, Word

 FROM dbo.[Load_RNFile] d

 Cross Apply dbo.DelimitedItem(d.EverythingElse,'|')) p

 PIVOT (MAX([Word]) FOR WordNumber in ([1],[2],[3],[4],[5],[6],[7]
```

I use the cross apply to join the TVF results to my Load table. Then I pivot the data back so that we have a row per original row instead of a row for each field. The results come back looking like this:

1	2	3	4	5	6	7	8	9
FieldA1Data	FieldB1Data	FieldC1Data	FieldD1Data	FieldE1Data	FieldF1Data	FieldG1Data	FieldH1Data	FieldI1Data
FieldA2Data	FieldB2Data	FieldC2Data	FieldD2Data	FieldE2Data	FieldF2Data	FieldG2Data	FieldH2Data	FieldI2Data

With the dynamic SQL that is generated from CreateDelimitedTableString the destination table is populated with the delimited data.

The other piece of code for the delimited file piece is a bit simpler. This piece just uses substring to break apart the data row into it's respective piece based on the data dictionary's end points in your ETL_ImportDictionary.

```
CREATE FUNCTION [dbo].[CreateParsedTableString]

(@ETLImportFileId int)

RETURNS nvarchar(max)

AS

BEGIN

DECLARE @DestTableName varchar(100),

 @LoadTableName varchar(100),

 @FieldNames varchar(2000),

 @Select varchar(5000),
```

Once the destination tables have been populated we now truncate the Load_* tables by calling the following sproc:

```sql
CREATE PROCEDURE [dbo].[Package_TruncateLoadTables]

AS


SET NOCOUNT ON


BEGIN


DECLARE @DropSQL nvarchar(max)
SET @DropSQL = ''

--Create DROP TABLE SQL

SELECT @DropSQL = @DropSQL +
```

At this point all of our destination tables have been populated, our load tables have been removed and we are ready to kick off the business processes to convert the data into what we want to use it for. After my business processes end I will kick off one last sproc to set my package run to complete. Here is the sproc for that:

```
CREATE Procedure [dbo].[Package_SetRunCompletion]

 @FileCode char(2)

AS

SET NOCOUNT ON

Update m

Set RunEnd = GetDate(), Completed = 1, Duration = DATEDIFF(minute,

FROM ETL_Run m

WHERE m.Completed = 0 AND ContainsErrors = 0

AND FileCode = @FileCode
```

As new files come in, I only need to update my database tables with the new formats and configurations and write the business conversion pieces. I never have to change my package, and my testers don't have to worry about testing the package repeatedly. This is my dynamic ETL solution. I hope you enjoyed it.

Resources:

ETLFTPAndImport.dtsx

Rate this | Join the discussion | Add to briefcase | Print **Thank this author by sharing:** 6

Total article views: 17179 | Views in the last 30 days: 4073