


# The TSQL of CSV: Comma-Delimited of Errors

 [red-gate.com/simple-talk/sql/t-sql-programming/the-tsql-of-csv-comma-delimited-of-errors](http://red-gate.com/simple-talk/sql/t-sql-programming/the-tsql-of-csv-comma-delimited-of-errors)

April 13, 2012



Phil Factor

13 April 2012

79370 views

61

Despite the neglect of the basic ODBC drivers over the years, they still afford a neat way of reading from, and writing to, CSV files; and to be able to do so in SQL as if they were tables is somewhat magical. Just to prove it is possible, Phil creates a CSV version of AdventureWorks as a linked server.

## Introduction

This article is about using 'Comma-Separated Values' (CSV) format in SQL Server. You'll come across CSV if you're a SQL Server developer when you are importing, or exporting, tabular data from 'foreign' applications, data feeds, or the wilder fringe of databases. Exporting via CSV is useful, since CSV is generally supported by even the most archaic legacy systems. We'll demonstrate a few tricks such as that of saving the tables and views of AdventureWorks to CSV format, parsing a CSV file to a table, converting CSV to XML and JSON and so on.

Although we'll cover some of the many ways of manipulating CSV in this article, we'll be concentrating more on using ODBC, since it is pretty fast, versatile, and reliable, and is easy to access from TSQL.

## The CSV Format

There is nothing much wrong with the CSV file format, beyond the neglect it suffers. It's been around far longer than MSDOS. When properly implemented, it is the most efficient available way of representing the contents of a data table in a readable document, and is therefore very effective in transferring such data between different applications or platforms. Unlike JSON or XML, it is designed for the single purpose of representing tabular data. It doesn't do hierarchical data, and it was designed assuming that both ends of the data transfer process understood what each column represented, as well as the data type, collation and character set being used. If it is done properly, it can transfer any type of tabular data reliably.

We come across CSV files in SSIS, Excel, ODBC, SQLCMD and even in PowerShell. A faulty implementation, nicknamed 'comedy-limited', exists in BCP. The term 'CSV format' is not the correct description of the practice of using commas or other 'special' characters, to separate data elements in some lists, this generally applies only to 'private' data where there is a guarantee that the delimiting character does not appear within the data.

If CSV transfers fail, it is generally because of mistakes in the implementation. [RFC 4180](#) gives a good description of what it is supposed to do. It is easy to describe:

Each row, or tuple, is separated by a linefeed, and the last line can be terminated by a linefeed. Each line should contain the same number of fields, which are separated by a single character, usually a comma. Fields may be enclosed in double-quote characters. If they are, then fields may contain commas or linefeed characters. They can also contain double-quote characters if 'escaped' with a second adjacent double-quote character. Null data values are denoted by two delimiters in a row with no data between them. Character strings in a delimited text line can be enclosed in double quotation marks (""). No blanks can occur before or after delimited values

The first line of the file may be a header line which contains the name of each column. There may be an initial 'comment' line initiated by a hash (#) character, and this is either terminated by a linefeed character or, if followed by a double-quote, until a subsequent unescaped double-quote, followed by a linefeed character.

CSV is unusual in that it encloses the entire field in double quotes if it contains the separator. If the field contains double quotes, it must be 'escaped' by a second double quote so that all double quotes in the field are repeated twice to indicate that they don't end the field. However, if a repeated double-quote are the only two characters within the field, this indicates a field containing an empty string, which is quite different from NULL, signifying an unknown value.

Because the rules are underspecified, different implementations diverge in their handling of edge cases. You will find in some products that continuation lines are supported by starting the last field of the line with an un-terminated double-quote. Microsoft, surprisingly, sometimes has incompatible versions of CSV files between its own applications, and has even had incompatible CSV files in different versions of the same application

There is a good formal grammar for producing a CSV reader or writer [here](#).

## A quick tour of some of the alternatives

You are generally struck with one of four alternative requirements:

- Pushing data into a SQL Server table from a file via an external process.
- Pulling data out of a SQL Server table to a file via an external application
- pushing data from a SQL Server table to a file via TSQL

- Pulling data out of a file to a SQL Server table via TSQL.

It isn't always as easy as you'd expect.

One obvious problem comes from values that aren't delimited with double-quotes. Often, programmers assume that numbers don't need delimiters. Unfortunately, parts of Europe use the comma as a 'decimal point' in money, and some representations of large integer values have a comma as 'thousands' separators. This sort of mistake is common to find, but It is not the CSV convention that is at fault, but the implementation. The easiest way to test, and often break, a poor implementation of CSV is to put the linefeed characters into a string value. With BCP, its defects are even less subtle. Even putting a comma into a properly delimited string breaks the import. For the demonstration of this, you'll need the `spSaveTextToFile` procedure from The TSQL of Text.

```

1  --Create a simple test for interpreting CSV (From Wikipedia)
2  DECLARE @TestCSVFileFromWikipedia VARCHAR(MAX)
3  --put CSV into a variable
4  SELECT @TestCSVFileFromWikipedia='Year,Make,Model,Description,Price
5  1997,Ford,E350,"ac, abs, moon",3000.00
6  1999,Chevy,"Venture ""Extended Edition""",",4900.00
7  1999,Chevy,"Venture ""Extended Edition, Very Large""",",5000.00
8  1996,Jeep,Grand Cherokee,"MUST SELL!
9  air, moon roof, loaded",4799.00'
10 --write it to disk (Source of procedure in the article 'The TSQL of Text'
11 EXECUTE phifactor.dbo.spSaveTextToFile
12 @TestCSVFileFromWikipedia,'d:\files\TestCSV.csv',0
13
14 --create a table to read it into
15 CREATE TABLE TestCSVImport ([Year] INT, Make VARCHAR(80), Model VARCHAR(80), [Description] VARCHAR(80), Price money)
16 BULK INSERT TestCSVImport FROM 'd:\files\TestCSV.csv'
17 WITH ( FIELDTERMINATOR = ',', ROWTERMINATOR = '\n', FirstRow=2)
18 --No way. This is merely using commas to delimit.
19 --BCP can't import a CSV file!
20 GO
21
22 --whereas
23 INSERT INTO TestCSVImport
24 SELECT *
25 FROM
26 OPENROWSET('MSDASQL',--provider name (ODBC)
27 'Driver={Microsoft Text Driver (*.txt; *.csv)};
28     DEFAULTDIR=d:\files;Extensions=CSV;',--data source

```

	Year	Make	Model	Description	Price
1	1997	Ford	E350	ac, abs, moon	3000.00
2	1999	Chevy	Venture "Extended Edition"	NULL	4900.00
3	1999	Chevy	Venture "Extended Edition, Very Large"	NULL	5000.00
4	1996	Jeep	Grand Cherokee	MUST SELL! air, moon roof, loaded	4799.00

Much better, but you'll see that we possibly suffer from the vagueness of the CSV spec, as I reckon that those descriptions should be blank. There is a great difference between a NULL and an empty string.

So let's try to read it in via TSQL . We'll get to the details of this code later on in the article.

```

1  Declare @CSVFileContents Varchar(MAX)
2  SELECT @CSVFileContents = BulkColumn
3  FROM OPENROWSET(BULK 'd:\files\TestCSV.csv', SINGLE_BLOB) AS x
4
5  CREATE TABLE AnotherTestCSVImport ([Year] INT, Make VARCHAR(80),
6      Model VARCHAR(80), [Description] VARCHAR(80), Price money)
7  INSERT INTO AnotherTestCSVImport
8      Execute CSVToTable @CSVFileContents

```

	Year	Make	Model	Description	Price
1	1997	Ford	E350	ac, abs, moon	3000.00
2	1999	Chevy	Venture "Extended Edition"		4900.00
3	1999	Chevy	Venture "Extended Edition, Very Large"		5000.00
4	1996	Jeep	Grand Cherokee	MUST SELL! air, moon roof, loaded	4799.00

Yes, this is correct.

As well as 'pulling' files into SQL Server via TSQL, we can 'push' them via an external app. In the old days we'd use DTS or SSIS but this is very over-engineered for the purpose, and PowerShell makes this very easy since one can read in a CSV file and then use `Data.SqlClient.SqlBulkCopy` to insert the data into a SQL Server database table.

```

1  #Thanks to Chad Miller and Marc van Orsouw
2  $CSVfilenameAndPath='D:\MyDirectory\MyFilename.csv'
3  $ServerInstance='MyServerName'
4  $Database='MyDatabase'
5  $TableToImportTo='MyTable'
6  #####
7  Trap {
8      # Handle the error
9      $err = $_.Exception
10     write-host $err.Message
11     while( $err.InnerException ) {
12         $err = $err.InnerException
13         write-host $err.Message
14     };
15     # End the script.
16     break
17 }
18 $CSV=import-csv $CSVfilenameAndPath #Read the CSV file to a powershell object
19 $datatable = new-object Data.datatable #we need a datatable to do a bulk copy with
20 $ThisIsTheHeaderRow = $true #we the first row is the header thanks to Import_CSV
21 #insert it row by agonizing row
22 foreach ($tuple in $CSV)
23 {
24     $CurrentRow = $datatable.NewRow() #create a new row
25     foreach($property in $tuple.PsObject.get_properties()) # for each column...
26     {
27         if ($ThisIsTheHeaderRow) #if so we need to create a column 'object'
28         {
29             $Col = new-object Data.DataColumn
30             $Col.ColumnName = $property.Name.ToString()

```

```

31 if ($property.value)
32 {
33 if ($property.value -isnot [System.DBNull])
34 { $Col.DataType = $property.value.gettype() }
35 }
36 $datatable.Columns.Add($Col) #and actually add it to the datatable columns
37 }
38 if ($property.IsArray)
39 { $CurrentRow.Item($property.Name)=$property.value | ConvertTo-XML -AS String -NoTypeInfoInformation -Depth 1 }
40 else { $CurrentRow.Item($property.Name) = $property.value }
41 }
42 $datatable.Rows.Add($CurrentRow) #and actually add it to the datatable rows
43 $ThisIsTheHeaderRow = $false
44 }
45 #assemble the connection string
46 $connectionString = "Data Source=$ServerInstance;Integrated Security=true;Initial Catalog=$Database;"
47 $bulkCopy = new-object ("Data.SqlClient.SqlBulkCopy") $connectionString #create the BulkCopy
48 $bulkCopy.DestinationTableName = $TableToImportTo
49 $bulkCopy.WriteToServer($datatable) #write it all out
50 'did I do well, master?'

```

The reverse process is even easier. Here we use SMO, but there are a number of alternatives

```

1  $ServerName='MyServer'# the server it is on
2  $Database='MyDatabase' # the name of the database you want to script as objects
3  $DirectoryToSaveTo='E:\MyScriptsDirectory' # the client-side directory where you want to store them
4  $TableOrView='TestCSVImport'
5  #requires -version 2.0
6  $v = [System.Reflection.Assembly]::LoadWithPartialName( 'Microsoft.SqlServer.SMO')
7  Trap {
8  # Handle the error
9  $err = $_.Exception
10 write-host $err.Message
11 while( $err.InnerException ){
12 $err = $err.InnerException
13 write-host $err.Message
14 };
15 # End the script.
16 break
17 }
18 $server = New-Object ( Microsoft.SqlServer.Management.Smo.Server ) $ServerName
19 $dbase = New-Object ( Microsoft.SqlServer.Management.Smo.Database ) ($server, $Database)
20 # what we do if there is a sql info message such as a PRINT message
21 $handler = [System.Data.SqlClient.SqlInfoMessageEventHandler] (param($sender, $event) Write-Host $event.Message);
22 $server.ConnectionContext.add_InfoMessage($handler);
23 $result=$dbase.ExecuteWithResults("Select * from $TableOrView")
24 $result.Tables[0] convertto-csv >"$DirectoryToSaveTo$TableOrView.csv"
25 "All done, Mighty One."

```

This produces a file containing this.

```

1  #TYPE System.Data.DataRow
2  "Year","Make","Model","Description","Price"
3  "1997","Ford","E350","ac, abs, moon","3000.0000"
4  "1999","Chevy","Venture ""Extended Edition""","","4900.0000"
5  "1999","Chevy","Venture ""Extended Edition, Very Large""","","5000.0000"
6  "1996","Jeep","Grand Cherokee","MUST SELL!
7  air, moon roof, loaded","4799.0000"

```

Yes, it is correct. You can see that the people who devised the ConvertTo-CSV cmdlets played safe by delimiting every variable, which is fine but slightly more verbose than strictly necessary.

Another utility that is great for pushing CSV data into SQL Server is the LogParser. This has become a cult application with websites dedicated to its use. Basically, it can treat any text-based data file, whether XML, CSV, tab-delimited or whatever, as a single-table database. You can do SELECT statements on it, and output the results to a number of different formats, including a table in SQL: Server. This can be run at the command line, or automated via COM. [Chad Miller has published a number of solutions using it for importing CSV into SQL Server.](#) It is fast and reliable.

	<i>External process</i>	<i>TSQL</i>
From table to CSV file	PowerShell, SSIS, SQLCMD	OpenRowSet with ODBCtext driver (requiring Schema.ini and file creation), Linked Server (ditto)
From CSV file to table	LogParser , SSIS, PowerShell, Net via Data.SqlClient.SqlBulkCopy	OpenRowSet with ODBCtext driver), Linked Server (ditto)

## Reading from a CSV file into SQL Server via SQL

---

As we've seen, the CSV driver doesn't perform too badly. In fact using the ODBC Text Driver is usually the most convenient way to read a CSV file into, or out of, SQL Server. (we'd actually be using the Microsoft Jet IISAM driver under the covers). BCP as a means of reading CSV is broken. If you need to use BCP (or FILE IMPORT) use native mode for tabular data, but don't expect any other system to be able to read it.

The ODBC driver can use an external 'schema.ini' format file to control the details and, if necessary, specify the column headers and the data type formats and conversions. You will need to add 'DBQ=C:\;' to the connection string to specify the location of this, unless you store it in the same directory as the data.

If you are lucky enough to have ad-hoc distributed queries allowed on your server ..

```
1 EXEC sp_configure 'Ad Hoc Distributed Queries',
   '1';
2
   Reconfigure
```

..then you can treat a CSV file as a quasi-table.

```
1 SELECT *
2 FROM
3     OPENROWSET('MSDASQL',--provider name (ODBC)
4         'Driver={Microsoft Text Driver (*.txt; *.csv)};
5         DEFAULTDIR=C:\;Extensions=CSV;',--data source
6         'SELECT * FROM sample.csv')
```

This is fine, and you can alter the provider connection string in a number of ways to specify whether it is tab delimited or CSV, and whether there are headers in the file or not. If there aren't, you need to get the information about the column names from somewhere else, and that 'somewhere else' is the 'schema.ini' file. The ODBC text driver can use an external '[schema.ini](#)' format file to control the details and, if necessary, specify the column headers and the [data type formats and conversions](#). You will need to add 'DBQ=C:\;' to the connection string to specify the location of this, unless you store it in the same directory as the data. The driver will look in the directory you've specified, and look for a 'Schema.ini' file there. If it is there, it looks up a section with the same name as the file you want to read from and will pull out the details it needs.

This 'Schema.ini' file stores the metadata. It gives you a lot of control over the format. You can specify whether your text files are CSV, Tab delimited or fixed-column in format. You have a lot of control over date and currency formats too. If you haven't got a header line in your CSV file, you can specify the column names and also the data type. Unfortunately, these types are not SQL types, but JET types from way back, and it is tricky, but possible, to get a match for such things as varbinary or xml SQL Server Datatypes. The Text ODBC driver has a GUI way of creating such schema.ini files, but it uses only a subset of the JET data types that are available in the driver, and has left out the vital 'decimal, or numeric, type.

Fortunately, if we are using this ODBC driver to write out CSV files of tables or views, we can generate this file in TSQL, from the information schema for whatever database you are using. Each table or view can have an entry in the one .INI file, and we can store all the files in the one directory. We'll show you how to do this in this article. Once we have done this, we will have a text-based database that we can write to or read from.

## Using a linked Server

---

Usually, the best way to interact with any ODBC source is to treat it as a linked database. This takes a bit of effort with CSV, but is worth doing because it makes the process of creating CSV reports very easy, because you are dealing with the files as if they were tables. This means that you can use the CSV as the 'staging' tables for import, and merely use UPDATE statements to do the refreshing. For outputting reports, I generally create ordinary views that give the format I want, and then create the reports as linked tables in the 'CSV database'. This means that you can keep these reports up to date as a background task, and just make copies as and when you need them.

I've created a stored procedure that keeps all the messy details from you. You just set it going and get on with something else.

Here, we are writing a file out to a directory on the server, setting up a linked server, and writing out the table 'CrudeOilProduction' to it. We leave things with the linked server still in place.

```

1 Use MyDatabase
2 Execute SaveViewsOrTablesToCSV
3     @ServerDirectory='D:\files\OilAnalysis',
4     @NameOfLinkedServer='OilAnalysis',
5     @TablesOrViews='CrudeOilProduction'
6     @Database='GlobalWarming'
7

```

And we can then read it in with something like this.

```

1 Execute ReadViewsOrTablesFromAttachedCSVDATABASE
2     @NameOfLinkedServer='OilAnalysis',
3     @TablesOrViews='CrudeOilProduction'
4     @Database='GlobalWarming'

```

Generally, you'll just want to do this with views, but just to demonstrate what is possible, (and we squeezed out a few unexpected bugs in the process) we'll create the entire AdventureWorks CSV database, views and tables, and fill them

```

1 Execute SaveViewsOrTablesToCSV
2     @ServerDirectory = 'd:\textVersionOfAdventureWorks',
3     @NameOfLinkedServer= 'AdventureCSV',
4     @TablesOrViews = '%',
5     @Database='AdventureWorks'

```

This routine is setting up a linked server called 'AdventureCSV'. It is then finding out all the tables that conform to your wildcard specification, and the information about the columns within those tables. It is then creating all the tables as files within a directory on disk whose path is 'd:\textVersionOfAdventureWorks'. Having done that, it is then creating a SCHEMA.INI file which it writes to the same directory. In the schema, it puts all the metadata about the tables or views, mapping the SQL Server datatypes to the ODBC (Jet 4) datatypes. It then fills the CSV files with data, using SQL Expressions, and inserting into the files as if they were tables. Finally, it saves all the SQL to disk so that you can do an autopsy on the result to check for errors.

Here is part of the results, on a server directory

Name	Size	Type	Date Modified
Sales_ContactCreditCard.csv	596 KB	CSV File	10/04/2012 17:51
Sales_CountryRegionCurrency.csv	4 KB	CSV File	10/04/2012 17:51
Sales_CreditCard.csv	1,226 KB	CSV File	10/04/2012 17:51
Sales_Currency.csv	5 KB	CSV File	10/04/2012 17:51
Sales_CurrencyRate.csv	937 KB	CSV File	10/04/2012 17:51
Sales_Customer.csv	1,593 KB	CSV File	10/04/2012 17:51
Sales_CustomerAddress.csv	1,387 KB	CSV File	10/04/2012 17:51
Sales_Individual.csv	11,585 KB	CSV File	10/04/2012 17:51
Sales_justCreditcards.csv	469 KB	CSV File	10/04/2012 17:51
Sales_SalesOrderDetail.csv	12,933 KB	CSV File	10/04/2012 17:51
Sales_SalesOrderHeader.csv	7,516 KB	CSV File	10/04/2012 17:51
Sales_SalesOrderHeaderSalesReason.csv	785 KB	CSV File	10/04/2012 17:51
Sales_SalesPerson.csv	2 KB	CSV File	10/04/2012 17:51
Sales_SalesPersonQuotaHistory.csv	16 KB	CSV File	10/04/2012 17:51
Sales_SalesReason.csv	1 KB	CSV File	10/04/2012 17:51
Sales_SalesTaxRate.csv	3 KB	CSV File	10/04/2012 17:51
Sales_SalesTerritory.csv	2 KB	CSV File	10/04/2012 17:51
Sales_SalesTerritoryHistory.csv	2 KB	CSV File	10/04/2012 17:51
Sales_ShoppingCartItem.csv	1 KB	CSV File	10/04/2012 17:51
Sales_SpecialOffer.csv	3 KB	CSV File	10/04/2012 17:51
Sales_SpecialOfferProduct.csv	35 KB	CSV File	10/04/2012 17:51
Sales_Store.csv	352 KB	CSV File	10/04/2012 17:51
Sales_StoreContact.csv	52 KB	CSV File	10/04/2012 17:51
Sales_vIndividualCustomer.csv	13,823 KB	CSV File	10/04/2012 17:51
Sales_vIndividualDemographics.csv	2,078 KB	CSV File	10/04/2012 17:51
Sales_vSalesPerson.csv	4 KB	CSV File	10/04/2012 17:51
Sales_vSalesPersonSalesByFiscalYears.csv	2 KB	CSV File	10/04/2012 17:51
Sales_vStoreWithDemographics.csv	213 KB	CSV File	10/04/2012 17:51

And here is an excerpt of the machine-generated .ini file...

```

1  [Sales_Store.csv]
2      ColNameHeader = False
3      Format = CSVDelimited
4      CharacterSet = ANSI
5      Col1=CustomerID Integer
6      Col2=Name Char width 50
7      Col3=SalesPersonID Integer
8      Col4=Demographics LongChar
9      Col5=rowguid char width 40
10     Col6=ModifiedDate DateTime
11 [Production_ProductPhoto.csv]
12     ColNameHeader = False
13     Format = CSVDelimited
14     CharacterSet = ANSI
15     Col1=ProductPhotoID Integer
16     Col2=ThumbNailPhoto Longchar
17     Col3=ThumbnailPhotoFileName Char width 50
18     Col4=LargePhoto Longchar
19     Col5=LargePhotoFileName Char width 50
20     Col6=ModifiedDate DateTime
21 [Production_ProductProductPhoto.csv]
22     ColNameHeader = False
23     Format = CSVDelimited
24     CharacterSet = ANSI
25     Col1=ProductID Integer
26     Col2=ProductPhotoID Integer
27     Col3=Primary Byte
28     Col4=ModifiedDate DateTime
29 [Sales_StoreContact.csv]
30     ColNameHeader = False
31     Format = CSVDelimited
32     CharacterSet = ANSI
33     Col1=CustomerID Integer
34     Col2=ContactID Integer
35     Col3=ContactTypeID Integer
36     Col4=rowguid char width 40
37     Col5=ModifiedDate DateTime
38

```

...along with some of the machine-generated insert statements!



```

1  INSERT INTO AdventureCSV...Sales_Store#csv([CustomerID],[Name],[SalesPersonID],[Demographics],[rowguid],[ModifiedDate])
2  Select [CustomerID], [Name], [SalesPersonID], convert(NVARCHAR(MAX),[Demographics]), [rowguid], [ModifiedDate]
3  FROM AdventureWorks.Sales.Store
4
5  INSERT INTO
AdventureCSV...Production_ProductPhoto#csv([ProductPhotoID],[ThumbNailPhoto],[ThumbnailPhotoFileName],[LargePhoto],[LargePhotoFile
6  Select [ProductPhotoID], convert(NVARCHAR(MAX),[ThumbNailPhoto]), [ThumbnailPhotoFileName], convert(NVARCHAR(MAX),[LargePhot
7  [LargePhotoFileName], [ModifiedDate]
8  FROM AdventureWorks.Production.ProductPhoto
9
10 INSERT INTO AdventureCSV...Production_ProductProductPhoto#csv([ProductID],[ProductPhotoID],[Primary],[ModifiedDate])
11 Select [ProductID], [ProductPhotoID], [Primary], [ModifiedDate]
    FROM AdventureWorks.Production.ProductProductPhoto

```

OK, it took five minutes to run, which is a bit extreme. BCP takes around two minutes on the same task using native mode, but, of course doesn't do CSV properly.

The source for these routines can be downloaded at the bottom of the article.

## Handling CSV Using TSQL

---

You can parse CSV using nothing other than SQL. I decided to demonstrate this with a parser that writes to a hierarchy file that I've already described in another [Consuming JSON Strings in SQL Server](#). Just occasionally, you'll find that you have to deal with data represented by a CSV representation within a string. Well, it has happened to me! I have created a function that returns a hierarchy table and another one that just converts it into a standard SQL result. The only real point of returning it as a hierarchy table is that the same table gets returned whatever the CSV, and also that you can then turn it into JSON or XML if you get the urge.

```

1  DECLARE @MyHierarchy Hierarchy, @XML XML
2  INSERT INTO @myHierarchy
3  Select * from parseCSV('Year,Make,Model,Description,Price
4  1997,Ford,E350,"ac, abs, moon",3000.00
5  1999,Chevy,"Venture ""Extended Edition""", "",4900.00
6  1999,Chevy,"Venture ""Extended Edition, Very
Large""", "",5000.00
7  1996,Jeep,Grand Cherokee,"MUST SELL!
8  air, moon roof, loaded",4799.00', Default,Default,Default)
9  SELECT dbo.ToXML(@MyHierarchy)

```

Which produces this...

```

1  <?xml version="1.0" ?>
2  <root>
3  <CSV>
4  <item Year="1997" Make="Ford" Model="E350" Description="ac, abs, moon" Price="3000.00" />
5  <item Year="1999" Make="Chevy" Model="Venture &quot;Extended Edition&quot;" Description="" Price="4900.00" />
6  <item Year="1999" Make="Chevy" Model="Venture &quot;Extended Edition, Very Large&quot;" Description="" Price="5000.00" />
7  <item Year="1996" Make="Jeep" Model="Grand Cherokee" Description="MUST SELL!
8  air, moon roof, loaded" Price="4799.00" />
9  </CSV>
10 </root>

```

```

1 DECLARE @MyHierarchy Hierarchy, @XML XML
2 INSERT INTO @myHierarchy
3   Select * from parseCSV('Year,Make,Model,Description,Price
4   1997,Ford,E350,"ac, abs, moon",3000.00
5   1999,Chevy,"Venture ""Extended Edition""", "",4900.00
6   1999,Chevy,"Venture ""Extended Edition, Very
   Large""", "",5000.00
7
8   1996,Jeep,Grand Cherokee,"MUST SELL!
   air, moon roof, loaded",4799.00', Default,Default,Default)
9   SELECT dbo.ToJSON(@MyHierarchy)

```

...which produces this...

```

1  {
2  "CSV" : [
3    {
4      "Year" : 1997,
5      "Make" : "Ford",
6      "Model" : "E350",
7      "Description" : "ac, abs, moon",
8      "Price" : 3000.00
9    },
10   {
11     "Year" : 1999,
12     "Make" : "Chevy",
13     "Model" : "Venture "Extended Edition"",
14     "Description" : "",
15     "Price" : 4900.00
16   },
17   {
18     "Year" : 1999,
19     "Make" : "Chevy",
20     "Model" : "Venture "Extended Edition, Very Large"",
21     "Description" : "",
22     "Price" : 5000.00
23   },
24   {
25     "Year" : 1996,
26     "Make" : "Jeep",
27     "Model" : "Grand Cherokee",
28     "Description" : "MUST SELL!\nair, moon roof, loaded",
29     "Price" : 4799.00
30   }
31 ]
32 }

```

The procedure that produces a table seems far more useful than it turns out to be in practicality. Although one can do this (thanks for the sample, Timothy)

```

1  Execute CSVToTable ""REVIEW_DATE","AUTHOR","ISBN","DISCOUNTED_PRICE"
2  "1985/01/21","Douglas Adams",0345391802,5.95
3  "1990/01/12","Douglas Hofstadter",0465026567,9.95
4  "1998/07/15","Timothy ""The Parser"" Campbell",0968411304,18.99
5  "1999/12/03","Richard Friedman",0060630353,5.95
6  "2001/09/19","Karen Armstrong",0345384563,9.95
7  "2002/06/23","David Jones",0198504691,9.95
8  "2002/06/23","Julian Jaynes",0618057072,12.50
9  "2003/09/30","Scott Adams",0740721909,4.95
10 "2004/10/04","Benjamin Radcliff",0804818088,4.95
11 "2004/10/04","Randel Helms",0879755725,4.50'

```

... To produce ..

	REVIEW_DATE	AUTHOR	ISBN	DISCOUNTED_PRICE
1	1985/01/21	Douglas Adams	0345391802	5.95
2	1990/01/12	Douglas Hofstadter	0465026567	9.95
3	1998/07/15	Timothy "The Parser" Campbell	0968411304	18.99
4	1999/12/03	Richard Friedman	0060630353	5.95
5	2001/09/19	Karen Armstrong	0345384563	9.95
6	2002/06/23	David Jones	0198504691	9.95
7	2002/06/23	Julian Jaynes	0618057072	12.50
8	2003/09/30	Scott Adams	0740721909	4.95
9	2004/10/04	Benjamin Radcliff	0804818088	4.95
10	2004/10/04	Randel Helms	0879755725	4.50

..it is more useful for debugging and developing than in a production system.

Naturally, it would be good to have some of the features available to XML markup built-into SQL Server. It wouldn't, surely, take much to add a FOR CSV to the FOR XML. Sadly for those of us who still need to deal with CSV, we are left with an ODBC driver that doesn't seem to have changed much in years.

## Conclusions

Whereas there are plenty of ways of placing CSV data into staging tables, there are fewer ways of accessing tabular information in CSV format more directly. The ODBC text driver, for all its faults and limitations, still provides the best single means for both reading and writing CSV file data. If it was made more correct in its interpretation of CSV, and allowed ODBC SQL CREATE statements as other ODBC drivers have, then it would be far more useful. Naturally, if CSV format in SQL Serve had been accorded just a small fraction of the effort put into XML, then I wouldn't have needed to write this article at all!

## References

- Creativyst "The Comma Separated Value (CSV) File Format" <http://www.creativyst.com/Doc/Articles/CSV/CSV01.htm>
- Edoceo, Inc., "CSV Standard File Format", 2004, <http://www.edoceo.com/utilis/csv-file-format.php>.
- Factor, P. "Consuming JSON Strings in SQL Server" November 2010  
<https://www.red-gate.com/simple-talk/sql/t-sql-programming/consuming-json-strings-in-sql-server/>
- MSDN "Schema.ini File (Text File Driver)"  
<http://msdn.microsoft.com/en-us/library/windows/desktop/ms709353%28v=vs.85%29.aspx>
- MSDN "Text File Format (Text File Driver)",  
<https://docs.microsoft.com/en-us/sql/odbc/microsoft/text-file-format-text-file-driver>
- Repici, J., "HOW-TO: The Comma Separated Value (CSV) File Format", 2004,  
<http://www.creativyst.com/Doc/Articles/CSV/CSV01.htm>.
- PowerShell Basics #1: Reading and parsing CSV <http://www.heikniemi.net/hardcoded/2010/01/powershell-basics-1-reading-and-parsing-csv/>
- Raymond, E., "The Art of Unix Programming, Chapter 5", September 2003,  
<http://www.catb.org/~esr/writings/taoup/html/ch05s02.html>.
- Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.
- Python Documentation "CSV File Reading and Writing" <http://docs.python.org/library/csv.html>
- Shafranovich Y. Draft RFC 4180 "Common Format and MIME Type for Comma-Separated Values (CSV) Files"  
<http://tools.ietf.org/html/rfc4180>
- Wikipedia "Comma-separated values" [http://en.wikipedia.org/wiki/Comma-separated\\_values](http://en.wikipedia.org/wiki/Comma-separated_values)