



Robert Sheldon 10 December 2009

Working with the bcp Command-line Utility

Even though there are many other ways to get data into a database, nothing works quite as fast as BCP, once it is set up with the right parameters and format file. Despite its usefulness, the art of using the command-line utility has always seemed more magic than method; but now along comes Robert Sheldon to shed light on the murky details.

The bcp utility is a command-line tool that uses the Bulk Copy Program (BCP) API to bulk copy data between an instance of SQL Server and a data file. By using the utility, you can export data from a SQL Server database into a data file, import data from a data file into a SQL Server database, and generate format files that support importing and exporting operations.

To use the bcp utility to perform these tasks, you can run a **bcp** command (along with the appropriate arguments) at a Command Prompt window. The command should conform to the following syntax:>

```
bcp {table|view|"query"}  
    {out|queryout|in|format}  
    {data_file|nul}  
    {[optional_argument]}...
```

As you can see, a **bcp** command requires three arguments. The first (*table|view|"query"*) represents the data source or destination in a SQL Server database. You can use the bcp utility to export data from a table or view or through a query. If you specify a query, you must enclose it in quotation marks. In addition, you can import data into a table or view. If you import into a view, all columns within the view must reference a single table. (Note that, when you specify a table or view, you must

qualify the name with the database or schema names as necessary.)

The second argument in a **bcp** command (`out|queryout|in|format`) determines the command's mode (direction). When you run a **bcp** command, you must specify one of the following four modes:

- **out**: The command exports data from a table or view into a data file.
- **queryout**: The command exports data retrieved through a query into a data file.
- **in**: The command imports data from a data file into a table or view.
- **format**: The command creates a format file based on a table or view. (Format files are explained later in the article.)

The third argument in a **bcp** command (`data_file|nul`) is the full path of the data file or, when a data file should not be specified, the **nul** value. If you're importing data, you must specify the file that contains the source data. If you're exporting data, you must specify the file that the data will be copied to. (If the file does not exist, it will be created.) When you're using the bcp utility to generate a format file, you do not specify a data file. Instead, you should specify **nul** in place of the data file name.

In addition to the three required arguments, you can include one or more optional arguments when you issue a **bcp** command. The bcp utility supports numerous optional arguments, and the ones you include often depend on the mode you specify in the second argument. The remainder of this article provides examples that demonstrate how many of these arguments work. For a description of all the arguments supported by the bcp utility, see the topic "bcp Utility" in SQL Server Books Online.

Exporting Data from a Table or View

As mentioned above, when you export data out of a table or view, you must specify the **out** option, along with the data source and destination file. The following **bcp** command copies data from the Employee table in the AdventureWorks2008 sample database and copies it to the EmployeeData.dat file.

```
bcp AdventureWorks2008.HumanResources.Employee out C:
```

As you would expect, the command includes the three required arguments: the source table (AdventureWorks2008.HumanResources.Employee), the mode (out), and the full path name of the destination data file (C:\Data\EmployeeData.dat). If the data file exists when you run the command, any data within the file will be overwritten with the exported Employee information. If the file does not exist, it will be created and the data will be exported.

The fourth argument in the example (-S localhost\SqLsrv2008) specifies the server and instance of SQL Server. You do not have to include the instance name if the source database is in the default instance. And if the source database is in the default instance on the local machine, you do not have to specify the **-S** argument at all, as in the following example:

```
bcp AdventureWorks2008.HumanResources.Employee out C:
```

The last argument in the preceding examples (-T) indicates that a trusted connection should be used to connect to SQL Server. You should use this option if the SQL Server instance uses integrated security. If integrated security is not used, you should instead specify the **-U** argument, along with the login ID of an account that can access the SQL Server instance. For example, the following **bcp** command specifies the login ID acct1 when accessing the SQL Server instance:

```
bcp AdventureWorks2008.HumanResources.Employee out C:
```

When you run this command, you will be prompted for a password. Alternatively, you can include the password in the command as well by specifying the **-P** argument, along with the account's password (pw123), as shown in the following example:

```
bcp AdventureWorks2008.HumanResources.Employee out C:
```

However, Microsoft generally recommends that you do not include the **-P** argument and instead wait to be prompted. But there might be circumstance when you want to pass in the password as part of the command.

When you run any of the commands shown in the preceding examples, you will be prompted for information about each column in the source

table or view. The following three prompts show you the type of data that you need to supply for each column:

```
Enter the file storage type of field BusinessEntityID
Enter prefix-length of field BusinessEntityID [0]:
Enter field terminator [none]:
```

These prompts are what you receive for the BusinessEntityID column in the Employee table. Notice that each prompt includes a recommended value, shown in the brackets. To accept the suggested setting, press Enter after you receive the prompt. Otherwise enter a value and then press Enter. Note, however, the recommended settings are generally your best options. (For more information about each prompt, see the topic “Specifying Data Formats for Compatibility by Using bcp” in SQL Server Books Online.)

After you respond to each prompt for each column, you will be asked whether you want to save the format information and, if so, the full path name of the format file, as shown in the following two prompts:

```
Do you want to save this format information in a file
Host filename [bcp.fmt]:
```

As you can imagine, it can get quite annoying having to supply format information each time you run a **bcp** command when exporting data. Fortunately, the bcp utility includes options that make this process much simpler. When defining your **bcp** command, you can include one of the following four arguments, which specify how the data should be formatted:

- **-n (native format):** The bcp utility retains the database native data types when bulk copying the data to the data file. Microsoft recommends that you use this format to bulk copy data between instances of SQL Server. However, you should use this format option only when the data file should not support extended or double-byte character set (DBCS) characters.
- **-N (Unicode native format):** The bcp utility uses the database native data types for non-character data and uses Unicode for character data for the bulk copy operation. Microsoft recommends that you use this format to bulk copy data between SQL Server instances when the data file should support extended or DBCS

characters.

- **-w (Unicode character format):** The bcp utility uses Unicode characters when bulk copying data to the data file. This format option is intended for bulk copying data between SQL Server instances. Note, however, that the Unicode native format (**-N**) offers a higher performance alternative.
- **-c (character format):** The bcp utility uses character data for the bulk copy operation. Microsoft recommends that you use this format to bulk copy data between SQL Server and other applications, such as Microsoft Excel.

When you include one of these options, you are not prompted for format information. The formatting is taken care of automatically.

Now let's look at an example that demonstrates how the format options work. The following **bcp** command includes the native format option (**-n**):

```
bcp AdventureWorks2008.HumanResources.Employee out C:
```

When you specify the **-n** argument, the data is automatically copied to the file in its native format. In the next example, the data is also automatically saved to the file without prompting for formatting information:

```
bcp AdventureWorks2008.HumanResources.Employee out C:
```

Notice that the **-c** argument is specified, rather than **-n**, as in the preceding example. As a result, the Employee information will now be saved as character data.

When the character format (**-c**) is used in a **bcp** command, each field, by default, is terminated with a tab character, and each row is terminated with a newline character. You can override the default behavior by using the **-t** argument to specify a field terminator and the **-r** argument to specify the row terminator. For example, the following **bcp** command specifies that each field be terminated with a comma:

```
bcp AdventureWorks2008.HumanResources.Employee out C:
```

As you can see, the command includes the **-t** argument following by a

comma, so each field in the data file will be terminated with a comma, rather than a tab. For a complete list of the types of terminators you can use and how to specify them, see the topic “Specifying Field and Row Terminators” in SQL Server Books Online.

Up to this point, the examples I’ve shown you have copied data from a table into a file. However, you can just as easily copy data from a view, as in the following example:

```
bcp AdventureWorks2008.HumanResources.vEmployee out C:
```

This command is identical to the preceding example, except that it now extracts data from the vEmployee view, rather than the Employee table.

When your **bcp** command retrieves data from a table or view, it copies all the data. However, you have some control over which rows are copied to the data file. In a **bcp** command, you can use the **-F** argument to specify the first row to be retrieved and the **-L** argument to specify the last row. In the following example, the first row I retrieve is 101 and the last row is 200:

```
bcp AdventureWorks2008.HumanResources.Employee out C:
```

Now the data file will include only the 100 rows that fall within the specified range.

The bcp utility also supports arguments that are not specific to the data itself. For example, you can use the **-o** argument to specify an output file. An output file captures the information normally returned to the command prompt after your run a **bcp** command. In the following example, I use the **-o** argument to specify that the output be saved to the EmployeeOutput.txt file:

```
bcp AdventureWorks2008.HumanResources.Employee out C:
```

When you run this command, any output that would have been displayed to the console is now saved to the output file.

Exporting Data Returned by a Query

All the examples up to this point have used the **out** argument to copy data from a table or view. Now let’s look at the **queryout** argument,

which retrieves data through a query. In the following example, I specify a SELECT statement, enclosed in quotation marks, and then specify the **queryout** argument:

```
bcp "SELECT * FROM AdventureWorks2008.Person.Person" queryout
```

As you can see, the command retrieves data from the Person table and uses the Unicode native format (**-N**) to save the data to the file. Notice that the command also includes the **-L** argument, which means that only 100 rows will be retrieved from the table. However, when you use the **queryout** option rather than the **out** option, you can be as specific in your query as necessary—you can include multiple tables and you can qualify your queries as necessary. For example, in the following statement, I include the TOP 100 clause in the SELECT statement, rather than include the **-L** argument:

```
bcp "SELECT TOP 100 * FROM AdventureWorks2008.Person.Person" queryout
```

Notice that the query also includes an ORDER BY clause so the data in the file will be ordered by BusinessEntityID. This, of course, is still a very simple query, but it does demonstrate how to use the **queryout** argument in a **bcp** command and how similar this mode is to a command that contains the **out** argument. You still specify the data file, the format, and any other applicable options. Now let's look at how to import data.

Importing Data into a Table

When you use the bcp utility to import data into a SQL Server table, you must specify the **in** mode, rather than **out** or **queryout**. To demonstrate how to use the **in** argument in a **bcp** command, I used the following Transact-SQL to create the Employees table:

```
USE AdventureWorks2008;  
IF OBJECT_ID ('Employees', 'U') IS NOT NULL  
DROP TABLE dbo.Employees;  
  
CREATE TABLE dbo.Employees  
(  
    BusinessEntityID int NOT NULL IDENTITY PRIMARY KEY,  
    NationalIDNumber nvarchar(15) NOT NULL,  
    LoginID nvarchar(256) NOT NULL,  
    OrganizationNode hierarchyid NOT NULL,  
    OrganizationLevel smallint NULL,  
    JobTitle nvarchar(50) NOT NULL,br> BirthDate date NOT  
    MaritalStatus nchar(1) NOT NULL,
```

```
Gender nchar(1) NOT NULL,  
HireDate date NOT NULL,  
SalariedFlag Flag NOT NULL,  
VacationHours smallint NOT NULL,  
SickLeaveHours smallint NOT NULL,  
CurrentFlag Flag NOT NULL,br> rowguid uniqueidentifier  
ModifiedDate datetime NOT NULL  
);
```

I created the table in the AdventureWorks2008 sample database in a SQL Server 2008 instance. After I created the table, I ran the following **bcp** command to create a data file that contains employee test data:

```
bcp AdventureWorks2008.HumanResources.Employee out C:
```

We can now use this data file to demonstrate how to import data into the Employees table.

When you import data into a table, you must specify the table (or updatable view) and the **in** argument, as shown in the following example:

```
bcp AdventureWorks2008.dbo.Employees in C:\Data\Emplo
```

Notice that you must also specify the source data file (C:\Data\EmployeeData_c.dat) and the options that define the format of the data as it was saved to the file: character format (-c) and field terminator (-t). If you do not specify the correct format options, you will receive an error when you try to import the data.

You might have noticed that the BusinessEntityID column in the Employees table is configured as an IDENTITY column. As a result, when you import the data, the database engine, by default, ignores the BusinessEntityID values that are in the data file and generates its own IDs. However, you can override the default behavior by specifying the **-E** argument, as shown in the following example:

```
bcp AdventureWorks2008..Employees in C:\Data\Employee
```

Now when you import the data, the BusinessEntityID values in the data file will be loaded into the table, rather than new values being generated.

You can also order the data that you import into the table by using the **-h** argument along with the **ORDER** hint:

```
bcp AdventureWorks2008..Employees in C:\Data\Employee
```


After I specify **-h ORDER**, I provide the name of the column (in parentheses) whose values should be sorted. If you want to sort multiple columns, you must separate the columns with a comma.

The **-h** argument supports multiple hints. Another one, for example, is the **TABLOCK** hint, which specifies that a bulk update table-level lock should be acquired during the bulk operation. The following **bcp** command uses the **TABLOCK** hint:

```
bcp AdventureWorks2008..Employees in C:\Data\Employee
```

Because the command includes the **TABLOCK** hint, the database engine will hold the lock for the duration of the bulk load operation, which significantly improves performance over the default row-level locks.

When you run a **bcp** command, you can also specify the number of rows per batch of imported data. To specify the batch size, include the **-b** argument, along with the number of rows per batch. For example, the following **bcp** command limits each batch to 100 rows:

```
bcp AdventureWorks2008..Employees in C:\Data\Employee
```

The **bcp** utility also lets you specify an error file that stores any rows that the utility cannot copy from a data file into a table. To specify an error file, use the **-e** argument, followed by the full path name of the file, as shown in the following example:

```
bcp AdventureWorks2008..Employees in C:\Data\Employee
```

When you run this command, any rows that cannot be imported into the **Employees** table will be saved to the **EmployeeErrors.txt** file.

Using Format Files to Import and Export Data

The **bcp** utility lets you create format files that store format information that can be used to import and export data. Each format file contains formatting information about the data in the data file as well as information about the target database table. Essentially, the format file maps the fields in the data file to the columns in the table. By using format files, you have far more flexibility in the type of data files you can use, allowing you to work with data files that can be shared by other

applications or used for different SQL Server tables.

You can use the bcp utility to create a format file. Once you've created the file, you can then modify it as necessary to support bulk copy operations. To create a format file, you must specify **format** for the mode and **nul** for the data file. In addition, you must also include the **-f** argument, followed by the full path name of the format file, as shown in the following example:

```
bcp AdventureWorks2008.Person.Person format nul -c -t
```

As you can see, the format file is based on the Person table. In addition, the data will be saved to the data file with the character format and a comma as the field terminator. The format file produced by this command will look similar to the following:

```
10.0
13
1  SQLCHAR      0      12      ","      1  BusinessEntit
2  SQLCHAR      0       4      ","      2  PersonType
3  SQLCHAR      0       3      ","      3  NameStyle
4  SQLCHAR      0      16      ","      4  Title
5  SQLCHAR      0     100      ","      5  FirstName
6  SQLCHAR      0     100      ","      6  MiddleName
7  SQLCHAR      0     100      ","      7  LastName
8  SQLCHAR      0      20      ","      8  Suffix
9  SQLCHAR      0      12      ","      9  EmailPromotio
10 SQLCHAR      0       0      ","     10  AdditionalCon
11 SQLCHAR      0       0      ","     11  Demographics
12 SQLCHAR      0      37      ","     12  rowguid
13 SQLCHAR      0      24      "\r\n"   13  ModifiedDate
```

The file includes such details as the data types of the data in the data file (second column), the field and row terminators (fifth column), and a mapping between table columns and the fields in the data file (sixth column). Note, however, that a full explanation of format files is beyond the scope of this article, and you should refer to the topic “Format Files for Importing or Exporting Data” in SQL Server Books Online for a complete explanation of how the files work. The main focus of this section is to show you how to use the bcp utility to create format files and use them to export and import data.

Returning to the format file above, notice that it lists SQLCHAR as the data type for all fields in the data file. This is because the character format (**-c**) is used to create the format file. However, you can also use

one of the other format options to create the format file. For example, the following **bcp** command includes the native format (**-n**) argument, rather than the character format:

```
bcp AdventureWorks2008.Person.Person format nul -n -f
```

Now the data types for the data file will show the native data types, as shown in the following:

```
10.0
13
1  SQLINT      0      4      ""      1  BusinessEntit
2  SQLNCHAR    2      4      ""      2  PersonType
3  SQLBIT      1      1      ""      3  NameStyle
4  SQLNCHAR    2     16      ""      4  Title
5  SQLNCHAR    2    100      ""      5  FirstName
6  SQLNCHAR    2    100      ""      6  MiddleName
7  SQLNCHAR    2    100      ""      7  LastName
8  SQLNCHAR    2     20      ""      8  Suffix
9  SQLINT      0      4      ""      9  EmailPromotio
10 SQLNCHAR    8      0      ""     10  AdditionalCon
11 SQLNCHAR    8      0      ""     11  Demographics
12 SQLUNIQUEID 1     16      ""     12  rowguid
13 SQLDATETIME 0      8      ""     13  ModifiedDate
```

As you can see, because the native format is specified, the field and row terminators are no longer required, but the prefix length (the third column) is now specified. The prefix length indicates how much space the field requires. Again, refer to SQL Server Books Online for a complete description of format files.

The file created above is only one of two types of format files supported by SQL Server 2005 and 2008. The format file above is a non-XML format file, the original form of the file supported by earlier versions of SQL Server (and still supported in SQL Server 2005 and 2008). However, a newer type of format file-the XML format file-was introduced in SQL Server 2005. The XML format file is more flexible and powerful than the original non-XML format file.

To create an XML format file, you run a command similar to the preceding two examples; however, you must also include the **-x** argument, as shown in the following example:

```
bcp AdventureWorks2008.Person.Person format nul -c -t
```

Now the data formatting information is stored as XML:

```
<?xml version="1.0"?>
<BCPFORMAT xmlns="http://schemas.microsoft.com/sqlser
<RECORD>
  <FIELD id="xsi:type=""CharTerm" TERMINATOR="," MAX_
  <FIELD id="xsi:type=""CharTerm" TERMINATOR="," MAX_
  <FIELD id="xsi:type=""CharTerm" TERMINATOR="," MAX_
  <FIELD id="xsi:type=""CharTerm" TERMINATOR="," MAX_
  <FIELD id="xsi:type=""CharTerm" TERMINATOR="," MAX_
  <FIELD id="xsi:type=""CharTerm" TERMINATOR="," MAX_
  <FIELD id="xsi:type=""CharTerm" TERMINATOR="," MAX_
  <FIELD id="xsi:type=""CharTerm" TERMINATOR="," MAX_
  <FIELD id="xsi:type=""CharTerm" TERMINATOR="," MAX_
  <FIELD id="xsi:type=""CharTerm" TERMINATOR="," />
  <FIELD id="xsi:type=""CharTerm" TERMINATOR="," />
  <FIELD id="xsi:type=""CharTerm" TERMINATOR="," MAX_
  <FIELD id="xsi:type=""CharTerm" TERMINATOR="\r\n" M
</RECORD>
<ROW>
  <COLUMN SOURCE="1" id="BusinessEntityID" xsi:type=
  <COLUMN SOURCE="2" id="PersonType" xsi:type="SQLNC
  <COLUMN SOURCE="3" id="NameStyle" xsi:type="SQLBIT
  <COLUMN SOURCE="4" id="Title" xsi:type="SQLNVARCHA
  <COLUMN SOURCE="5" id="FirstName" xsi:type="SQLNVA
  <COLUMN SOURCE="6" id="MiddleName" xsi:type="SQLNV
  <COLUMN SOURCE="7" id="LastName" xsi:type="SQLNVAR
  <COLUMN SOURCE="8" id="Suffix" xsi:type="SQLNVARCH
  <COLUMN SOURCE="9" id="EmailPromotion" xsi:type="S
  <COLUMN SOURCE="10" id="AdditionalContactInfo" xsi
  <COLUMN SOURCE="11" id="Demographics" xsi:type="SQ
  <COLUMN SOURCE="12" id="rowguid" xsi:type="SQLUNIQ
  <COLUMN SOURCE="13" id="ModifiedDate" xsi:type="SQ
</ROW>
</BCPFORMAT>
```

In an XML format file, the <RECORD> element lists the fields in the data file, and the <ROW> element lists the columns in the table. Notice that the xsi:type element in each field in the <RECORD> element lists the type as CharTerm, which is used for each field when the character format (-c) is specified. However, as with the non-XML format file, you can specify a different format, as shown in the following example:

```
bcp AdventureWorks2008.Person.Person format nul -n -f
```

Now the format file will include the native types and a prefix length, rather than a terminator:

```
<?xml version="1.0"?>
<BCPFORMAT xmlns="http://schemas.microsoft.com/sqlser
<RECORD>
  <FIELD id="xsi:type=""NativeFixed" LENGTH="4"/>
  <FIELD id="xsi:type=""NCharPrefix" PREFIX_LENGTH="2
  <FIELD id="xsi:type=""NativePrefix" PREFIX_LENGTH="
```

```

<FIELD id="xsi:type="NCharPrefix" PREFIX_LENGTH="2
<FIELD id="xsi:type="NCharPrefix" PREFIX_LENGTH="2
<FIELD id="xsi:type="NCharPrefix" PREFIX_LENGTH="2
<FIELD id="xsi:type="NCharPrefix" PREFIX_LENGTH="2
<FIELD id="xsi:type="NCharPrefix" PREFIX_LENGTH="2
<FIELD id="xsi:type="NativeFixed" LENGTH="4"/>
<FIELD id="xsi:type="NCharPrefix" PREFIX_LENGTH="8
<FIELD id="xsi:type="NCharPrefix" PREFIX_LENGTH="8
<FIELD id="xsi:type="NativePrefix" PREFIX_LENGTH="
<FIELD id="xsi:type="NativeFixed" LENGTH="8"/>
</RECORD>
<ROW>
<COLUMN SOURCE="1" id="BusinessEntityID" xsi:type=
<COLUMN SOURCE="2" id="PersonType" xsi:type="SQLNC
<COLUMN SOURCE="3" id="NameStyle" xsi:type="SQLBIT
<COLUMN SOURCE="4" id="Title" xsi:type="SQLNVARCHA
<COLUMN SOURCE="5" id="FirstName" xsi:type="SQLNVA
<COLUMN SOURCE="6" id="MiddleName" xsi:type="SQLNV
<COLUMN SOURCE="7" id="LastName" xsi:type="SQLNVAR
<COLUMN SOURCE="8" id="Suffix" xsi:type="SQLNVARCH
<COLUMN SOURCE="9" id="EmailPromotion" xsi:type="S
<COLUMN SOURCE="10" id="AdditionalContactInfo" xsi
<COLUMN SOURCE="11" id="Demographics" xsi:type="SQ
<COLUMN <COLUMN </ROW>
</BCPFORMAT>

```

After you create the format file, you can then reference it in your **bcp** commands by specifying the **-f** argument, along with the name of the format file, as shown in the following example:

```
bcp AdventureWorks2008.Person.Person out C:\Data\Pers
```

Notice that the command no longer needs a format option (such as **-c**) because all the formatting information is included in the format file. And you can specify any type of format file, as long as the formats of the specified fields are compatible with the columns in the table. In the following example, I use to the native format to export the data:

```
bcp AdventureWorks2008.Person.Person out C:\Data\Pers
```

And you can just as easily specify the XML format file. The following example uses the character format XML file to export data:

```
bcp AdventureWorks2008.Person.Person out C:\Data\Pers
```

And the next example uses the native format XML file to export data:

```
bcp AdventureWorks2008.Person.Person out C:\Data\Pers
```

Of course, you can also use format files to import data. To demonstrate

how to use the format files, I used the following Transact-SQL to create the Contacts1 table in the AdventureWorks2008 database:

```
USE AdventureWorks2008;

IF OBJECT_ID ('Contacts1', 'U') IS NOT NULL
DROP TABLE dbo.Contacts1;

SELECT *
INTO dbo.Contacts1
FROM AdventureWorks2008.Person.Person
WHERE 1 = 2;
```

Because I created the Contacts1 table based on the Person table, I can use one of the format files created above to import data into the Contact1 table. In the following example, I use the character format XML file to import data from the PersonData_c.dat file:

```
bcp AdventureWorks2008..Contacts1 in C:\Data\PersonDa
```

As you can see, I simply use the **-f** argument to specify the name of the format file. And I can do the same thing for the native format XML file:

```
bcp AdventureWorks2008..Contacts1 in C:\Data\PersonDa
```

In the examples we've looked at so far, the columns in the SQL Server tables have matched the fields in the data files. However, sometimes there are more columns in the data file than in the table, more columns in the table than in the data file, or the columns are in a different order between the two sources. Under such scenarios, format files are ideal for mapping the columns and fields to each other, so let's look at a few examples that demonstrate how this is done.

More Columns in Data File than in Table

When importing data from a file that contains more fields than there are columns in the target table, you can configure the format file to accommodate these differences. First, however, to demonstrate how to import the data, I used the following Transact-SQL to create the Contacts2 table:

```
USE AdventureWorks2008;

IF OBJECT_ID ('Contacts2', 'U') IS NOT NULL
DROP TABLE dbo.Contacts2;
```

```

SELECT BusinessEntityID AS ContactID,
FirstName,
LastName,
Demographics,
rowguid,
ModifiedDate
INTO dbo.Contacts2
FROM AdventureWorks2008.Person.Person
WHERE 1 = 2;

```

The new table is based on the Person table, but doesn't use all of the columns. As a result, the data file generated from the Person table will contain more fields than the Contacts2 table. To address this issue, I modified the character data XML file (PersonFormat_c.xml) so that only the columns in the Contacts2 table are included, as shown in the following file contents:

```

<?xml version="1.0"?>
<BCPFORMAT xmlns="http://schemas.microsoft.com/sqlser
<RECORD>
  <FIELD id="xsi:type=""CharTerm" TERMINATOR="," MAX_
  <FIELD id="xsi:type=""CharTerm" TERMINATOR="," MAX_
  <FIELD id="xsi:type=""CharTerm" TERMINATOR="," MAX_
  <FIELD id="xsi:type=""CharTerm" TERMINATOR="," MAX_
  <FIELD id="xsi:type=""CharTerm" TERMINATOR="," MAX_
  <FIELD id="xsi:type=""CharTerm" TERMINATOR="," MAX_
  <FIELD id="xsi:type=""CharTerm" TERMINATOR="," MAX_
  <FIELD id="xsi:type=""CharTerm" TERMINATOR="," MAX_
  <FIELD id="xsi:type=""CharTerm" TERMINATOR="," MAX_
  <FIELD id="xsi:type=""CharTerm" TERMINATOR="," />
  <FIELD id="xsi:type=""CharTerm" TERMINATOR="," />
  <FIELD id="xsi:type=""CharTerm" TERMINATOR="," MAX_
  <FIELD id="xsi:type=""CharTerm" TERMINATOR="\r\n" M
</RECORD>
<ROW>
  <COLUMN SOURCE="1" id="BusinessEntityID"" xsi:type=
  <COLUMN SOURCE="5" id="FirstName"" xsi:type="SQLNVA
  <COLUMN SOURCE="7" id="LastName"" xsi:type="SQLNVAR
  <COLUMN SOURCE="11" id="Demographics"" xsi:type="SQ
  <COLUMN SOURCE="12" id="rowguid"" xsi:type="SQLUNIQ
  <COLUMN SOURCE="13" id="ModifiedDate"" xsi:type="SQ
</ROW>
</BCPFORMAT>

```

Notice that the columns listed in the <ROW> element now match the Contacts2 table. To match the columns to the fields in the data file, you must make sure that the SOURCE element value matches the appropriate ID value in the FIELD elements. For example, the BusinessEntityID column shows a SOURCE value of 1. This matches the first field listed in the <RECORD> element, which is ID 1.

After I modified the PersonFormat_c.xml file, I renamed it PersonFormat_c2.xml. I can then use the file in my **bcp** command to import data into the Contacts2 table, as shown in the following example:

```
bcp AdventureWorks2008..Contacts2 in C:\Data\PersonDa
```

Because the format file has been configured to include only the Contacts2 column and those columns map the appropriate fields, only the correct data from the data file is imported into the table.

More Columns in Table than in Data File

In some cases, your target table will include more values than are in the data file, as is sometimes the case when you have nullable columns or columns with default or computed values. In such cases, you'll need to create a format file that reflects the difference between the source and target.

To demonstrate how this works, I used the following Transact-SQL to create the Contacts3 table and then to define a default value on the Suffix column:

```
USE AdventureWorks2008;

IF OBJECT_ID ('Contacts3', 'U') IS NOT NULL
DROP TABLE dbo.Contacts3;

SELECT BusinessEntityID AS ContactID,
FirstName,
LastName,
Suffix,
Demographics,
rowguid,
ModifiedDate
INTO dbo.Contacts3
FROM AdventureWorks2008.Person.Person
WHERE 1 = 2;

ALTER TABLE Contacts3
ADD CONSTRAINT suffix_def DEFAULT 'unknown' FOR Suffix;
```

After I created the table, I exported the data from Contacts2, the table I populated in the previous example:

```
bcp AdventureWorks2008..Contacts2 out C:\Data\PersonD
```

The result is that the Contacts3 table contains a Suffix column (with a

default), but the data file does not contain a matching field. Next, I create a format file based on the Contacts3 table:

```
bcp AdventureWorks2008..Contacts3 format nul -c -t, -
```

Notice that I created a non-XML format file. This is because the XML format files do not let you have more table columns than fields in the data file. Every column listed in the <ROW> element of the XML file must correspond to a field in the <RECORD> element. (You can get around this limitation by creating an updateable view against the target table and including only those columns that have corresponding fields in the data file.)

After you create the format file, you can modify the file to map the columns to the data fields. In this case, I deleted the Suffix row and updated the numbers in the first column (the numbers that represent the data file field order) so they're in consecutive order. The format file should now look similar to the following:

10.0						
6						
1	SQLCHAR	0	12	", "	1	ContactID
2	SQLCHAR	0	100	", "	2	FirstName
3	SQLCHAR	0	100	", "	3	LastName
4	SQLCHAR	0	0	", "	5	Demographics
5	SQLCHAR	0	37	", "	6	rowguid
6	SQLCHAR	0	24	"\r\n"	7	ModifiedDate

Once you've modified the format file, you can use the following **bcp** command to import the data:

```
bcp AdventureWorks2008..Contacts3 in C:\Data\PersonDa
```

Notice that I'm using the PersonData_c2.data data file and the PersonFormat_c3.fmt format file. When you run this command, the database engine will automatically insert the default value into the Suffix column.

Columns in Data File in Different Order than Table

In some cases, the columns in a table will be in a different order from the fields in a data file. You can modify either the non-XML or XML format files to accommodate these differences. In either case, you should make

certain that the columns map to the correct fields in the format files. To demonstrate how to map the columns, I used the following Transact-SQL to create the Contacts4 table:

```
USE AdventureWorks2008;

IF OBJECT_ID ('Contacts4', 'U') IS NOT NULL
DROP TABLE dbo.Contacts4;

SELECT BusinessEntityID AS ContactID,
       LastName,
       FirstName,
       Demographics,
       rowguid,
       ModifiedDate
INTO dbo.Contacts4
FROM AdventureWorks2008.Person.Person
WHERE 1 = 2;
```

In this case, I switched the FirstName and LastName fields from the source table (Person). Now let's look at the non-XML format file. For this example, I created a format file and data file based on the Person table, as we saw in earlier examples. I then modified the format file to support the Contacts4 table, as shown in the following:

10.0						
13						
1	SQLCHAR	0	12	", "	1	BusinessEntit
2	SQLCHAR	0	4	", "	0	PersonType
3	SQLCHAR	0	3	", "	0	NameStyle
4	SQLCHAR	0	16	", "	0	Title
5	SQLCHAR	0	100	", "	3	FirstName
6	SQLCHAR	0	100	", "	0	MiddleName
7	SQLCHAR	0	100	", "	2	LastName
8	SQLCHAR	0	20	", "	0	Suffix
9	SQLCHAR	0	12	", "	0	EmailPromotio
10	SQLCHAR	0	0	", "	0	AdditionalCon
11	SQLCHAR	0	0	", "	4	Demographics
12	SQLCHAR	0	37	", "	5	rowguid
13	SQLCHAR	0	24	"\r\n"	6	ModifiedDate

The first modification I made is to change the numbering in the sixth column of the format file (the numbering for the server column order). Any row that includes a field in the data file that is not a column in the table should be set to 0. In addition, the number should be modified to reflect the order of the columns in the table. For example, the FirstName column is now set to 3 because it is the third column in the table, and the LastName column is set to 2 because it is the second column in the table.

After I made these changes, I saved the format file as PersonFormat_c4.fmt. You can now use the format file to load data into the Contacts4 table, as shown in the following **bcp** command:

```
bcp AdventureWorks2008..Contacts4 in C:\Data\PersonDa
```

You can achieve the same results by creating an XML format file. The key is to include only those columns in the <ROW> element that are also contained in the Contacts4 table and to list those columns in the same order as they appear in the table, as shown in the following:

```
<?xml version="1.0"?>
<BCPFORMAT xmlns="http://schemas.microsoft.com/sqlser
<RECORD>
  <FIELD id="xsi:type=""CharTerm" TERMINATOR="," MAX_
  <FIELD id="xsi:type=""CharTerm" TERMINATOR="," MAX_
  <FIELD id="xsi:type=""CharTerm" TERMINATOR="," MAX_
  <FIELD id="xsi:type=""CharTerm" TERMINATOR="," MAX_
  <FIELD id="xsi:type=""CharTerm" TERMINATOR="," MAX_
  <FIELD id="xsi:type=""CharTerm" TERMINATOR="," MAX_
  <FIELD id="xsi:type=""CharTerm" TERMINATOR="," MAX_
  <FIELD id="xsi:type=""CharTerm" TERMINATOR="," MAX_
  <FIELD id="xsi:type=""CharTerm" TERMINATOR="," />
  <FIELD id="xsi:type=""CharTerm" TERMINATOR="," />
  <FIELD id="xsi:type=""CharTerm" TERMINATOR="," MAX_
  <FIELD id="xsi:type=""CharTerm" TERMINATOR="\r\n" M
</RECORD>
<ROW>
  <COLUMN SOURCE="1" id="BusinessEntityID" xsi:type=
  <COLUMN SOURCE="7" id="LastName" xsi:type="SQLNVAR
  <COLUMN SOURCE="5" id="FirstName" xsi:type="SQLNVA
  <COLUMN SOURCE="11" id="Demographics" xsi:type="SQ
  <COLUMN SOURCE="12" id="rowguid" xsi:type="SQLUNIQ
  <COLUMN SOURCE="13" id="ModifiedDate" xsi:type="SQ
</ROW>
</BCPFORMAT>
```

To create this file, I generated an XML format file based on the Person table, modified the file, and saved it as PersonFormat_c4.xml. Note that, after I eliminated the extra columns from the <ROW> element and ensured that they were in the correct order, I modified the SOURCE attribute as necessary to map the columns to the fields. For example, the LastName column has a SOURCE value of 7 to match the field with an ID value of 7.

After you've created the format file, you can run a **bcp** command similar to the following to import the data into the Contact4 table:

```
bcp AdventureWorks2008..Contacts4 in C:\Data\PersonDa|
```

When you run this command, the data will be imported from the PersonData_c.dat file into the Contacts4 table, and the import operation will be based on the PersonFormat_c4.xml format file.

Using the bcp Utility

As you've seen from the examples in this article, the bcp command-line utility provides a great deal of flexibility when importing and exporting data. You can copy data in its native format in order to transfer data between instances of SQL Server, or you can copy it as character data in order to work with applications other than SQL Server. And the bcp utility supports numerous options that let you fine-tune your import and export operations. In addition, you can use the utility to generate format files and then use those files for your bulk copy operations. Indeed, once you've learned to work with the bcp utility, you should be able to handle most of your bulk copy needs.



✉ Monthly newsletter



Join over 150,000 data professionals

Get the latest best
practices, insight, and
product news from our
industry experts

Get the newsletter

Products

Automate

Monitor

Standardize

Test Data

Management

Support

Forums

Contact product
support

Find my serial
numbers

Download older
versions

Solutions

Overview

Test Data
Management

Automate

Monitor

Maturity Assessment

Our Company

Careers

Contact us

Redgate Blog

Our values

Community & Learning

[Product Learning](#)

[University](#)

[Events & Friends](#)

[Simple Talk](#)

[Books](#)

[Forums](#)

Partners

[SQL Server Central](#)

[Resellers](#)

[Consulting partners](#)

Trust Center

[License agreement](#)

[Privacy and cookies](#)

[Modern slavery](#)

[CCPA - Do not sell my data](#)

Follow us



Copyright 1999 - 2023 Red Gate Software Ltd

