CHAPTER 9

Bulk Inserting Data

In the last chapter, we saw how we can manipulate records using `INSERT INTO`, `UPDATE`, and `DELETE`, and we used these statements to populate our reference data tables. Now let's add some contacts and their related information to our database. We could do this with these three statements—and we will—but we can also use SQL Server's `BULK INSERT` statement to add records via external files.

We'll start by adding some data using the `BULK INSERT` statement, and then we'll create a script to manually add a contact. We'll cover a lot of SQL Server functionality in this chapter, so hang on to your hat!

The BULK INSERT Statement

The `BULK INSERT` statement imports a data file into a database table (you can also import into a view, but that's something we won't look at in this book).

The basic structure of the `BULK INSERT` statement is:

```
BULK INSERT TableName FROM 'Full Path To FileName' WITH (Specify Options Here);
```

We are only interested in a couple of the options the statement provides, but if you want to see the full list, visit `https://msdn.microsoft.com/en-us/library/ms188365.aspx`.

Two things need to exist for the `BULK INSERT` statement to work:

- a table, into which data can be inserted
- a file containing data that will be imported into the table

The number of columns in the data file must correlate to the number of columns in the data file. We're going to start by importing a file into the `Contacts` table.

Preparing the Data File

Figure 9-1 gives us a look at the columns in our `Contacts` table.

ContactId	FirstName	LastName	DateOfBirth	AllowContactByPhone	CreatedDate

Figure 9-1. *Columns in the* `Contacts` *table*

And Figure 9-2 shows our `Contacts` data file as a table:

ContactId	FirstName	LastName	DateOfBirth	AllowContactByPhone	CreatedDate
0	Stephen	Gerrard	1980-05-30	1	
0	Dennis	Potter	1935-05-17	0	
0	Richard	Adams	1920-05-09	0	
0	Bertie	McQuillan	2001-06-30	1	
0	Walt	Disney	1966-12-05	1	
0	Barbara	Gordon	1952-01-11	0	
0	Josephine	Bailey	1949-05-31	1	
0	Linda	Canoglu	1959-07-11	1	
0	Grace	McQuillan	1993-09-27	0	
0	Vera	Black	1984-08-03	0	
0	Angelica	Jones	1981-02-04	1	
0	Steve	Davis	1957-08-22	1	
0	Allison	Fisher	1968-02-24	1	
0	julius	Marx	1990-10-02	0	
0	george	formby	1944-05-26	1	
0	Alan	Partridge	1965-04-14	0	
0	Harper	Lee	1986-04-28	1	
0	Robert	Burns	1959-01-25	0	
0	Michael	Jackson	1967-06-31	0	
0	Roald Dahl		1916-09-13	1	

Figure 9-2. *`Contacts` data file contents*

The first row contains the column names, and these map to the columns held in our `Contacts` table. The other rows contain data. You can download this file from `www.mcqtech.com/books/introducingsql/files` (`http://www.mcqtech.com/books/introducingsql/files`). Save it to `c:\temp\sqlbasics\importfiles\01_Contacts.csv`. You may need to create the **importfiles** folder if it doesn't already exist.

Having the column names in the file is completely optional, and if they are specified they don't even have to have the same names as the table. We have them in the file purely so we can see to which table column the data file column should map. `BULK INSERT` uses the position of the data file columns to map to the table columns. So column 1 in the file is imported into table column 1, column 2 maps to table column 2, and so on.

If we look at the rows of data in our file, we can see that the `ContactId` column contains `0`. This is because the database will automatically calculate `ContactId` values for us, as `ContactId` is an `IDENTITY` column. `CreatedDate` is blank as this has a default value. We could have left `AllowContactByPhone` as a blank value, too, as this has a default value, but we want some of these values to be `TRUE`, so we specify the actual values as `1` (for `TRUE`) or `0` (for `FALSE`).

`FirstName` and `LastName` are self-explanatory. In the last row, you should note there is no `LastName` specified; don't worry about this. Remember that we have a Check constraint configured to prevent blank values from being inserted into the `LastName` column.

There's also an error in the `DateOfBirth` column: Michael Jackson has an invalid value in this column (there is no 31st day in June!). Note the format of the `DateOfBirth` column: `YYYY-MM-DD`. It is important you use the date format expected by SQL Server, otherwise the import will fail (there are ways around this, but they are outside of our scope).

The `BULK INSERT` statement uses text files. These can be created with a variety of tools, such as Notepad, Word, and Excel. Excel is quite often the best way of creating your data files, as it makes the construction of rows of data straightforward.

A text file may have any extension you care to give it, but these are the two most common types of text file you'll deal with:

- **TXT**: a text file
- **CSV**: a comma separated file

We are going to use a CSV file, which consists of rows, and column values separated by commas. Here's an example:

```
ContactId,FirstName,LastName,DateOfBirth,AllowContactByPhone,DateCreated
0,Stephen,Gerrard,1980-05-30,1,
```

You can use any type of separator you wish—commas are the most common, but you'll see tabs and pipes used regularly. The `BULK INSERT` statement allows you to specify what character you are using as the column terminator. Similarly, the newline character built into Windows is usually used as the row terminator (this is written as `\r\n`). If you receive files from other companies they may not follow these conventions, which is why you can specify your own characters if necessary.

The default column terminator is actually the tab character (**\t**). **\r\n** is the default row terminator.

To save a file in Excel as a comma-separated file, go to **File** ➤ **Save As**, and in the **Save as type** drop-down list, choose **CSV (Comma delimited) (*.csv)**. Figure 9-3 shows what the dialog looks like in Excel 2013.

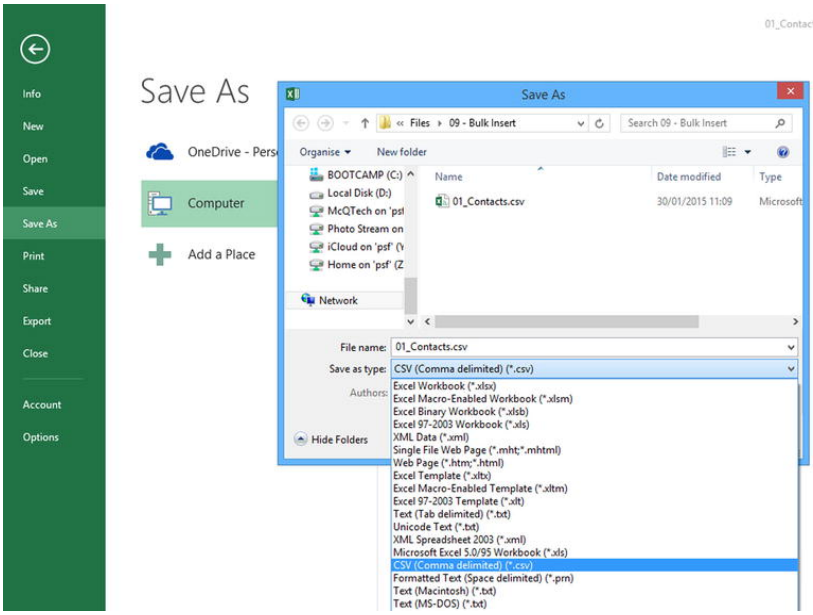

Figure 9-3. *Saving a CSV file in Excel*

If you have created the file manually rather than downloading it, make sure you save it as `c:\temp\sqlbasics\importfiles\01_Contacts.csv`.

To view the file, make sure you've closed Excel (if you have it open). Open Windows Explorer and navigate to `c:\temp\sqlbasics\importfiles`. Right-click `01_Contacts.csv` and choose **Open with**. If **Open with** has a submenu containing **Notepad** (as seen in Figure **9-4**), click Notepad to open the file in that program. If there is no submenu, click **Open with** and choose **Notepad** from the options that appear.

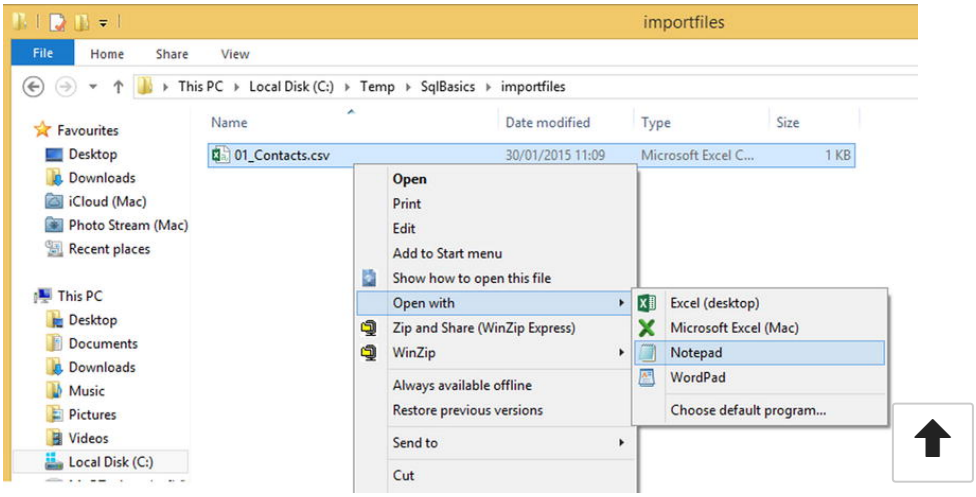

Figure 9-4. *Opening a CSV file with Notepad*

You'll see the file in all its comma-separated glory once you've opened Notepad, as Figure 9-5 demonstrates.

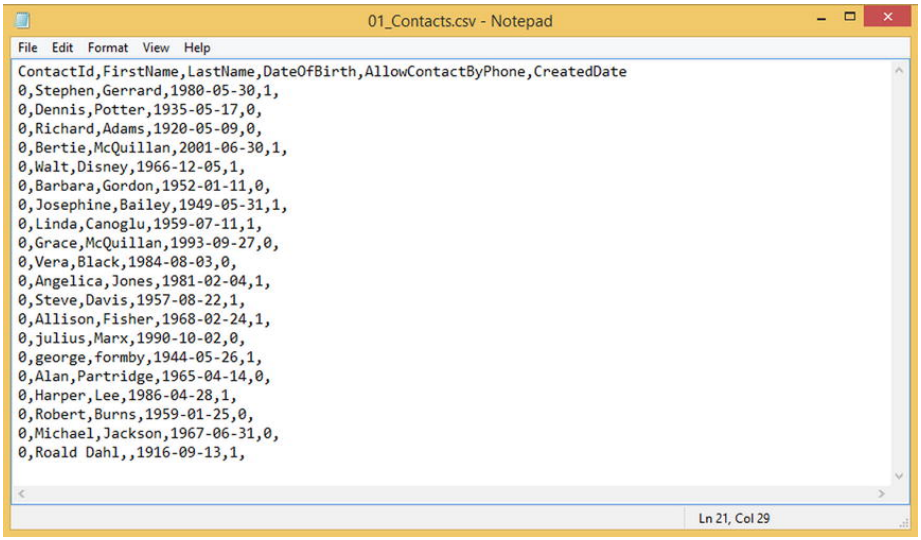

Figure 9-5. *A CSV file opened in Notepad*

The file is now ready for import. Time to type a `BULK INSERT` statement.

Importing the Data File

Open SSMS, connect to your server, and open up a New Query Window. Type in the `BULK INSERT` script:

```
USE AddressBook;

BULK INSERT dbo.Contacts FROM 'C:\temp\sqlbasics\importfiles\01_Contacts.csv';
GO
```

This is all pretty simple. We are saying to import the file `01_Contacts.csv` to the `Contacts` table. But when we hit F5 to run it, things go wrong (Figure 9-6)!

Figure 9-6. *A failed `BULK INSERT` statement*

Huh? THREE errors?! Well, the first error actually tells us the problem:

```
Msg 4832, Level 16, State 1, Line 3
Bulk load: An unexpected end of file was encountered in the data file.
```

"Unexpected end of file." This is a bit cryptic, but there are two possible causes:

- `BULK INSERT` is using the wrong row terminator for our file (`\r\n` instead of `\n`)
- `BULK INSERT` is using the wrong column terminator for our file (`\t` instead of `,`)

The upshot is `BULK INSERT` cannot figure out where our rows or columns start and end.

We'll have to add a `WITH` clause to specify the row and column terminators. Column terminators are known as field terminators by the `BULK INSERT` statement. Replace the line:

```
BULK INSERT dbo.Contacts FROM 'C:\temp\sqlbasics\importfiles\01_Contacts.csv';
```

with:

```
BULK INSERT dbo.Contacts FROM 'C:\temp\sqlbasics\importfiles\01_Contacts.csv'
WITH (ROWTERMINATOR = '\n', FIELDTERMINATOR = ',');
```

There are a number of values you can provide for the `ROWTERMINATOR` clause, but `\r\n` and `\n` are the most common (`\r` is a carriage return, `\n` is a newline—these come from the C programming language).

Run this. Oh no, another error! Well, two actually.

```
Msg 4864, Level 16, State 1, Line 3
Bulk load data conversion error (type mismatch or invalid character for the specified codepage) for
Msg 8114, Level 16, State 10, Line 3
Error converting data type DBTYPE_DBDATE to date.
```

The first error informs us that the `ContactId` value in the first row is invalid. The second error tells us there's an invalid date in the file.

Concentrating on the first error, let's take a look at the first row in our file (Figure 9-7).

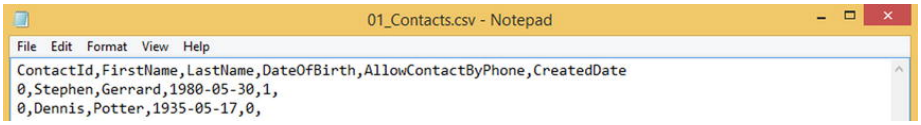

Figure 9-7. *The first row contains headers.*

The first line of data for Stephen Gerrard looks okay; it has a `0` in it. `ContactId` is an `IDENTITY` column; specifying a `0` should allow the `BULK INSERT` statement to ignore the `0` and just assign the next value. When specifying a value for an `IDENTITY` column in a `BULK INSERT` statement, you must specify a value in the correct format (i.e., an integer value). We've specified `0` for every row, but we could have put any number we wished to in there and it should still work. If you want `BULK INSERT` to use the actual values we provide for an `IDENTITY` column, then you need to specify the `KEEPIDENTITY` clause inside the `WITH` declaration.

The Stephen Gerrard row isn't raising the error; this has a valid `ContactId` value. Row 1 has the error, and that is the row containing the column headers. Unless we specifically tell it, `BULK INSERT` will assume any row containing column headers is a data row. To solve this issue, we need to tell `BULK INSERT` to ignore the first row, by starting the import at row 2. Replace the existing `BULK INSERT...WITH` statement with this new version:

```
BULK INSERT dbo.Contacts FROM 'C:\temp\sqlbasics\importfiles\01_Contacts.csv'
WITH (ROWTERMINATOR = '\n', FIELDTERMINATOR = ',', FIRSTROW = 2);
```

There is also a `LASTROW` option, which lets you specify the number of the last row you want to import. If you had a 20-row file but just wanted to import 10 rows, you could specify `LASTROW = 11` (assuming the first data row was row 2).

Run this and you should be left with just the `DBTYPE_DBDATE` error we saw earlier. This is being caused by our Michael Jackson row, which has a date of `1967-06-31`. As we mentioned earlier, June has only 30 days!

There is an `ERRORFILE` clause we can add; maybe that will help? This copies any rows that fail to import into the error file specified.

```
BULK INSERT dbo.Contacts FROM 'C:\temp\sqlbasics\importfiles\01_Contacts.csv'
WITH (ROWTERMINATOR = '\n', FIELDTERMINATOR = ',', FIRSTROW = 2, ERRORFILE = 'c:\temp\sqlbasics\imp
```

Still no joy:

```
Msg 8114, Level 16, State 10, Line 3
Error converting data type DBTYPE_DBDATE to date.
```

Unfortunately, no import can be attempted because the value specified doesn't meet the format of the column —a date in this case. To resolve the issue, we need to modify the file. Change Michael Jackson's `DateOfBirth` from `1967-06-31` to `1967-06-30`. Save the file and run the `BULK INSERT` statement again.

Hopefully all will go well, and you'll see this message:

```
(19 row(s) affected)
```

Excellent, we've imported some contacts. But hang on (dramatic music here)! We have 20 rows in our file. What's happened to one of our rows? Open the table editor for the `Contacts` table to display the rows, like in Figure 9-8.

BERTIE1\SQLEXPRES...ok - dbo.Contacts | SQLQuery1.sql - BE...ikemcquillan (54))*

ContactId	FirstName	LastName	DateOfBirth	AllowContactByPhone	CreatedDate
1	Stephen	Gerrard	1980-05-30	True	2015-01-30 14:23:41.857
2	Dennis	Potter	1935-05-17	False	2015-01-30 14:23:41.857
3	Richard	Adams	1920-05-09	False	2015-01-30 14:23:41.857
4	Bertie	McQuillan	2001-06-30	True	2015-01-30 14:23:41.857
5	Walt	Disney	1966-12-05	True	2015-01-30 14:23:41.857
6	Barbara	Gordon	1952-01-11	False	2015-01-30 14:23:41.857
7	Josephine	Bailey	1949-05-31	True	2015-01-30 14:23:41.857
8	Linda	Canoglu	1959-07-11	True	2015-01-30 14:23:41.857
9	Grace	McQuillan	1993-09-27	False	2015-01-30 14:23:41.857
10	Vera	Black	1984-08-03	False	2015-01-30 14:23:41.857
11	Angelica	Jones	1981-02-04	True	2015-01-30 14:23:41.857
12	Steve	Davis	1957-08-22	True	2015-01-30 14:23:41.857
13	Allison	Fisher	1968-02-24	True	2015-01-30 14:23:41.857
14	julius	Marx	1990-10-02	False	2015-01-30 14:23:41.857
15	george	formby	1944-05-26	True	2015-01-30 14:23:41.857
16	Alan	Partridge	1965-04-14	False	2015-01-30 14:23:41.857
17	Harper	Lee	1986-04-28	True	2015-01-30 14:23:41.857
18	Robert	Burns	1959-01-25	False	2015-01-30 14:23:41.857
19	Michael	Jackson	1967-06-30	False	2015-01-30 14:23:41.857

Figure 9-8. *Nineteen successful rows*

Who is missing? It's the last row, for Roald Dahl. If you recall, this row didn't have a `LastName` value.

```
0,Roald Dahl,,1916-09-13,1,
```

We added a Check constraint to prevent blank `FirstName` and `LastName` values in Chapter 7. This constraint has correctly stopped the bad data from coming in. Hurray for us!

BULK INSERT AND CHECK CONSTRAINTS

The behavior we've just encountered isn't SQL Server's expected behavior. The `BULK INSERT` documentation states that Check constraints will be ignored by the `BULK INSERT` statement, unless the `CHECK_CONSTRAINTS` clause is specified. So the Roald Dahl row should have been inserted. Any rows that fail because of a Check constraint will not be added to the `ERRORFILE`, either.

My view is this is a bug in `BULK INSERT`. To be on the safe side, you should specify `CHECK_CONSTRAINTS` if you want constraints to be enforced. You'll see this clause on the rest of the `BULK INSERT` statements in this chapter.

We've finished with our `BULK INSERT`, and have successfully inserted 19 contact rows. Our complete `BULK INSERT` script follows; save this as `c:\temp\sqlbasics\apply\12 - Bulk Insert Contacts.sql`.

```
USE AddressBook;

BULK INSERT dbo.Contacts FROM 'C:\temp\sqlbasics\importfiles\01_Contacts.csv'
WITH
(ROWTERMINATOR = '\n', FIELDTERMINATOR = ',', FIRSTROW = 2, ERRORFILE = 'c:\temp\sqlbasics\importfi
GO
```

Add a call to this file in the `00 - Apply.sql` script, so SQLCMD will execute the script whenever we rebuild the database. Put this above the `PRINT` statement near the bottom of the script.

```
:setvar currentFile "12 - Bulk Insert Contacts.sql"
PRINT 'Executing $(path)$(currentFile)';
:r $(path)$(currentFile)
```

Great; now for the other tables.

More Bulk Insert Escapades

We have five more tables we need to populate:

- `ContactAddresses`
- `ContactNotes`
- `ContactPhoneNumbers`
- `ContactRoles`
- `ContactVerificationDetails`

We'll use `BULK INSERT` to help us with the population. Except for `ContactVerificationDetails`, which extends the `Contacts` table, all of these are child tables, which presents us with a different set of problems.

Populating these tables is not as easy as you might expect. They all need a `ContactId` value to be provided. Without a `ContactId`, not one record can be created in any of these tables. Until we run script 12 to `BULK INSERT` the `Contacts` table, none of our records will have a `ContactId` value. We have two options here; assume the same `ContactId`s will always be assigned to the same contacts when the database is created, or create custom scripts to insert the values.

The custom script is tempting, but it is the long way around in this case. We'll take a look at creating a custom script to add our missing Roald Dahl record a bit later. For now, it's pretty safe to assume the same `ContactId`s will always be assigned to the same contact records when the database is created. After all, the scripts are always executed in the same order, against a clean database. So we'll import CSV files to all five tables. You can download the files from `www.mcqtech.com/books/introducingsql/files` (`http://www.mcqtech.com/books/introducingsql/files`). Save them to `c:\temp\sqlbasics\importfiles`. As displayed in Figure 9-9, you should finish up with six files in the folder.

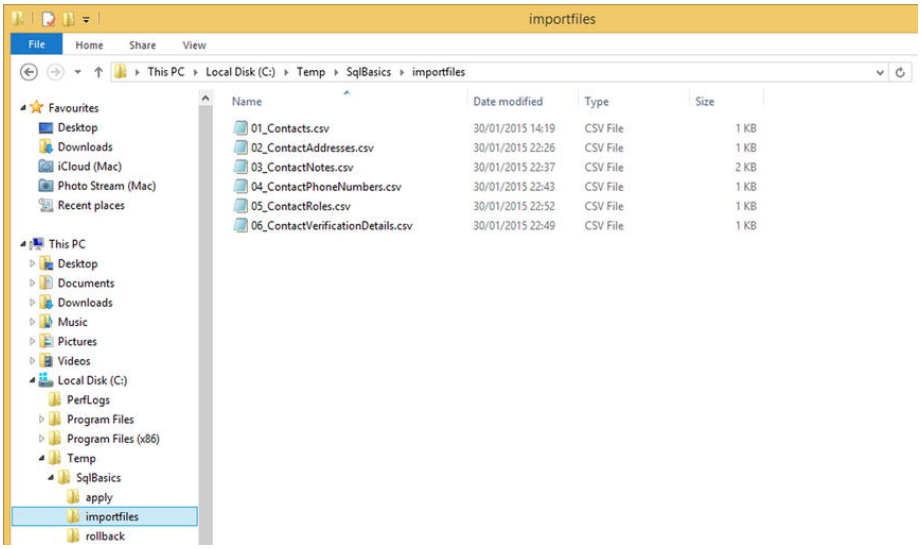

Figure 9-9. *A set of six import files*

The files we are about to import are going to include the ID values for records we assume will already exist, such as `Contacts`, `PhoneNumberTypes`, and `Roles`. To be on the safe side, rebuild your database before running these import files. This is a good checkpoint to ensure your `00 - Apply.sql` and `00 - Rollback.sql` files are working as expected (don't forget to turn on SQLCMD mode from the **Query** menu before trying to run these scripts). After you have rebuilt your database, create this script:

```
USE AddressBook;

BULK INSERT dbo.ContactAddresses FROM 'c:\temp\sqlbasics\importfiles\02_ContactAddresses.csv'
WITH
(ROWTERMINATOR = '\n', FIELDTERMINATOR = ',', FIRSTROW = 2,
ERRORFILE = 'c:\temp\sqlbasics\importfiles\02_ContactAddresses_Errors.csv',
CHECK_CONSTRAINTS);

GO
```

Save this as `c:\temp\sqlbasics\apply\13 - Bulk Insert Contact Addresses.sql`. Now copy this code into a New Query Window, and change the following two lines:

- `BULK INSERT dbo.ContactAddresses FROM 'c:\temp\sqlbasics\importfiles\02_ContactAddresses.csv'`
- `ERRORFILE = c:\temp\sqlbasics\importfiles\02_ContactAddresses_Errors.csv',`

To:

- `BULK INSERT dbo.ContactNote FROM 'c:\temp\sqlbasics\importfiles\03_ContactNotes.csv'`
- `ERRORFILE = 'c:\temp\sqlbasics\importfiles\03_ContactNotes_Errors.csv',`

Save this script as `c:\temp\sqlbasics\apply\14 - Bulk Insert ContactNotes.sql`. Follow these steps for the next three scripts:

- Change the preceding two lines to:

```
BULK INSERT dbo.ContactPhoneNumbers FROM 'c:\temp\sqlbasics\importfiles\04_ContactPhoneNumber
ERRORFILE = 'c:\temp\sqlbasics\importfiles\04_ContactPhoneNumbers_Errors.csv',
```

- Save the script as `c:\temp\sqlbasics\apply\15 - Bulk Insert ContactPhoneNumbers.sql`.
- Change the two lines to:

```
BULK INSERT dbo.ContactRoles FROM 'c:\temp\sqlbasics\importfiles\05_ContactRoles.csv'
ERRORFILE = 'c:\temp\sqlbasics\importfiles\05_ContactRoles_Errors.csv',
```

- Save the script as `c:\temp\sqlbasics\apply\16 - Bulk Insert ContactRoles.sql`
- Change the two lines to:

```
BULK INSERT dbo.ContactVerificationDetails FROM 'c:\temp\sqlbasics\importfiles\06_ContactVeri
ERRORFILE = 'c:\temp\sqlbasics\importfiles\06_ContactVerificationDetails_Errors.csv',
```

- Save the script as `c:\temp\sqlbasics\apply\17 - Bulk Insert ContactVerificationDetails.sql`

You should now have the 18 files shown in Figure 9-10 in your apply folder (including the `00 - Apply.sql` script).

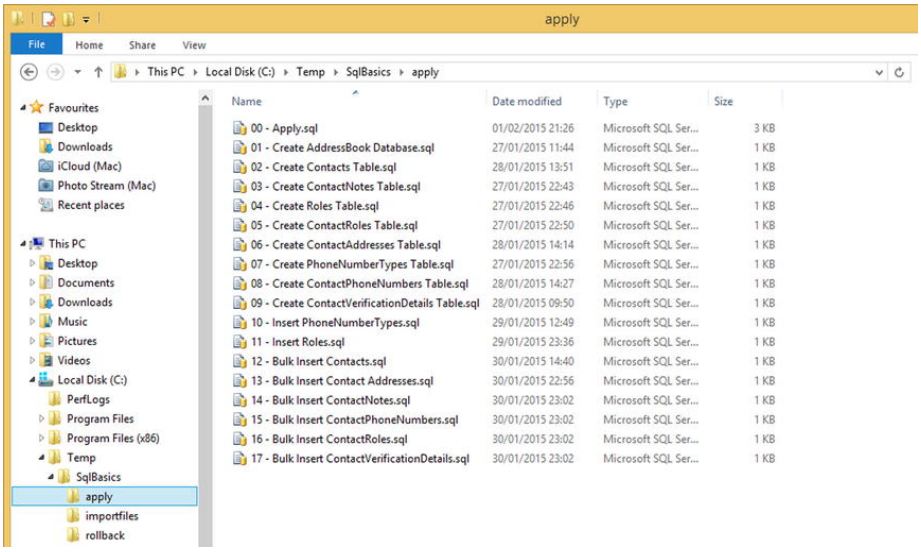

Figure 9-10. *The complete set of apply scripts so far*

You may have noted that scripts 12 to 17 all include the exact same code; the only things that change are the paths to the import file and the output error file, should any errors occur. We could have placed all of these calls into one script file, but adding them separately makes it easier for us to drop an import should we need to.

Let's quickly look at Figure 9-11, which displays the contents of the `02_ContactAddresses.csv` file.

AddressId	ContactId	HouseNumber	Street	City	Postcode
0	1	47	Madryn Avenue	Liverpool	L1 1PR
0	1	29	Formby Road	Formby	L21 1DD
0	2	169	Portobello Road	London	SW19 2AK
0	3	Nadallo	Los Ramblas	Barcelona	11223
0	4	2	Thornycroft	Chester	CH8 4PA
0	5	Walt Disney World	Florida	USA	45689
0	6	11	Knight Road	Gotham	99331
0	7	123	Gladwyn Street	Pottersville	PT9 5GA
0	8	265	Princes Road	Edinburgh	EH1 2EW
0	9	122	Stirling Crescent	Prescot	L32 9TY
0	10	198	Dartmoor Road	Yelverton	PL20 6RR
0	11	Maesdu Park	Builder Street West	Llandudno	LL30 1HH
0	12	10	South Street	Romford	RM1 6TV
0	13	1	St George's Avenue	Northampton	NN2 6JD
0	14	24	Mission Hill	Los Angeles County	78944
0	15	Beryldene	The Front	Lytham St Annes	BL1 1LX
0	16	Partridge House	Royal Arcade	Norwich	NR2 1NQ
0	17	Atticus Ranch	Maycomb County	Alabama	91210
0	18	Burns Cottage	Ayr	Alloway	KA1 1WK
0	19	Neverland Ranch	Santa Barbara	California	93441

Figure 9-11. *The* `ContactAddresses.csv` *file contents*

The last four columns contain address information, but the first two columns are a bit different. `AddressId` is the primary key of the `ContactAddresses` table, and it's also an `IDENTITY` column. So just as we did with the `ContactId` column in the `01_Contacts.csv` import file, we set this value to `0`. Doing this will cause the `BULK INSERT` statement to assign `IDENTITY` values in the `AddressId` column to the new records.

The really interesting column is `ContactId`. This is populated with a set of numbers. These are the `ContactId`s we know will be assigned to the contact records when they are created. We know what the IDs will be based on the record's location in the file. The Stephen Gerrard record is the first record in the `01_Contacts.csv` import file, so when the database has been rebuilt and is awaiting an import, it will be assigned the `ContactId` value of `1`. The first two addresses in `02_ContactAddresses.csv` are for `ContactId` 1, meaning they will both be linked to Stephen Gerrard. All other records work in a similar manner for their appropriate contacts.

GENERATING IMPORT FILES

Normally, you would not have ID values present in a file; you'd have a different piece of information, such as a customer number. You would create something called a *staging table* to which the data would be imported via the `BULK INSERT` statement. A staging table is just a table that exactly mirrors the contents of the import file. This is what we've just done—we've actually treated our data tables as staging tables.

Once you have data in the staging tables you would have another piece of code ready to run that could look up the correct `ContactId` for a record, then insert the record into the appropriate table (e.g., `ContactAddresses`). One of the exercises suggested in Appendix D asks you to modify the database to implement this type of import.

Open the `00 - Apply.sql` script, and above this line:

```
PRINT 'All apply scripts successfully executed.';
```

add the lines:

```
:setvar currentFile "13 - Bulk Insert Contact Addresses.sql"
PRINT 'Executing $(path)$(currentFile)';
:r $(path)$(currentFile)

:setvar currentFile "14 - Bulk Insert ContactNotes.sql"
PRINT 'Executing $(path)$(currentFile)';
:r $(path)$(currentFile)
```

```
:setvar currentFile "15 - Bulk Insert ContactPhoneNumbers.sql"
PRINT 'Executing $(path)$(currentFile)';
:r $(path)$(currentFile)

:setvar currentFile "16 - Bulk Insert ContactRoles.sql"
PRINT 'Executing $(path)$(currentFile)';
:r $(path)$(currentFile)

:setvar currentFile "17 - Bulk Insert ContactVerificationDetails.sql"
PRINT 'Executing $(path)$(currentFile)';
:r $(path)$(currentFile)
```

Save this, and then rebuild the database (run the `00 - Rollback.sql` script first, then the updated `00 - Apply.sql` script). If you take a look at any of the contact tables now, you'll find data in them. Figure 9-12 shows the ContactPhoneNumberstable:

BERTIE1\SQLEXPRE...tactPhoneNumbers × | 00 - Rollback.sql -...\mikemcquillan (54)) | 00 - Apply.sql - BE...mikemcquillan (51))

PhoneNumberId	ContactId	PhoneNumberTypeId	PhoneNumber
1	1	2	0151 264 2500
2	3	1	01565 100 100
3	7	3	07000 200 200
4	8	3	07000 300 300
5	12	1	01928 150 150
6	12	3	07500 350 350
7	14	1	01606 250 250
8	14	2	01606 260 260
9	16	1	01782 400 400
10	16	2	01782 410 410
11	16	3	07600 420 420
12	16	4	01782 430 430
13	18	3	07700 500 500
14	18	4	01244 520 520
15	19	4	0161 900 900
NULL	NULL	NULL	NULL

Figure 9-12. *Imported* `ContactPhoneNumbers` *records*

Operation Data Import successfully completed, Sergeant Major! Now for the rollback.

Truncating Tables

The rollbacks for these scripts are fairly simple. Each rollback script just needs to clear out the table it's corresponding apply script has populated, ensuring the `IDENTITY` value is reset back to `1`. This is actually a breeze for all tables except the `Contacts` table.

Here's the script; there isn't much to it:

```
USE AddressBook;

TRUNCATE TABLE dbo.ContactAddresses;

GO
```

The `TRUNCATE TABLE` is a special type of `DELETE` command. If you think back to our earlier look at the `DELETE` statement, we could delete an entire table if we didn't specify a `WHERE` clause. `TRUNCATE TABLE` does the same thing, the principal difference being `TRUNCATE TABLE` *always* clears out a table. You cannot tell a `TRUNCATE TABLE` statement which records you want to delete—it's all or nothing.

The preceding script will clear out the `ContactAddresses` table. `TRUNCATE TABLE` is a very useful alternative to `DELETE FROM` when you want to clear out a table:

- `TRUNCATE TABLE` resets `IDENTITY` columns back to the seed value; `DELETE FROM` doesn't do this, and `IDENTITY` will pick up from where it left off.

- `DELETE FROM` logs every single row deletion, which can quickly fill up your log. `TRUNCATE TABLE` logs just the `TRUNCATE TABLE` command, taking up less log space.
- `TRUNCATE TABLE` is faster than `DELETE FROM`.

One key thing to remember is `TRUNCATE TABLE` cannot be used on tables that are referenced by a foreign key. `ContactAddresses` participates in a foreign-key relationship with the `Contacts` table, but it is the child in that relationship; the `TRUNCATE TABLE` restriction applies to the parent tables. So we are good to use `ContactAddresses` with `TRUNCATE TABLE`.

As I said earlier, creating the rollback script is a breeze for all tables except the `Contacts` table. The foreign-key restriction is the reason we can't use `TRUNCATE TABLE` with the `Contacts` table. We'll come back to `Contacts` in a moment after we've created the other rollback scripts.

Enter the script for `ContactAddresses` given earlier and save it as `c:\temp\sqlbasics\rollback\13 - Bulk Insert Contact Addresses Rollback.sql`. Now you can create the rollback scripts for the other tables:

- `c:\temp\sqlbasics\rollback\14 - Bulk Insert ContactNotes Rollback.sql`

```
USE AddressBook;

TRUNCATE TABLE dbo.ContactNotes;

GO
```

- `c:\temp\sqlbasics\rollback\15 - Bulk Insert ContactPhoneNumbers Rollback.sql`

```
USE AddressBook;

TRUNCATE TABLE dbo.ContactPhoneNumbers;

GO
```

- `c:\temp\sqlbasics\rollback\16 - Bulk Insert ContactRoles Rollback.sql`

```
USE AddressBook;

TRUNCATE TABLE dbo.ContactRoles;

GO
```

- `c:\temp\sqlbasics\rollback\17 - Bulk Insert ContactVerificationDetails Rollback.sql`

```
USE AddressBook;

TRUNCATE TABLE dbo.ContactVerificationDetails;

GO
```

We're missing rollback script 12! We need this to clear out the `Contacts` table. As mentioned, we can't use `TRUNCATE TABLE` here, so we'll replace this with `DELETE FROM` instead.

```
USE AddressBook;

DELETE FROM dbo.Contacts;

GO
```

Hold on, though—this only fulfills half of our requirement! Yes, it will clear the table out. But it won't reset the `ContactId IDENTITY` column to `1`. To do that, we need to add something called a `DBCC` command.

```
USE AddressBook;

DELETE FROM dbo.Contacts;
DBCC CHECKIDENT('dbo.Contacts', RESEED, 1);
```

```
GO
```

Save this script as `c:\temp\sqlbasics\rollback\12 - Bulk Insert Contacts Rollback.sql`.

The `DBCC` command—`DBCC CHECKIDENT`—resets the `IDENTITY` column to its seed value. In brackets, we specify the table name, the `RESEED` command, and the value we want to reseed to—`1`, in this case. `DBCC` stands for Database Console Command, and a number of these commands are available to you in SQL Server; look up what is on offer at `https://msdn.microsoft.com/en-us/library/ms188796.aspx`.

DBCC COMMANDS

Database Console Commands are often used by DBAs. They can be used to check the status of a database (`DBCC CHECKDB`) or to shrink a file (`DBCC SHRINKFILE`). It's well worth taking a look at what can be done with `DBCC` commands; you never know when they'll come in handy.

We've successfully created a full set of rollback scripts. As with the apply folder, the rollback folder should now contain 18 files (you can see these in Figure 9-13).

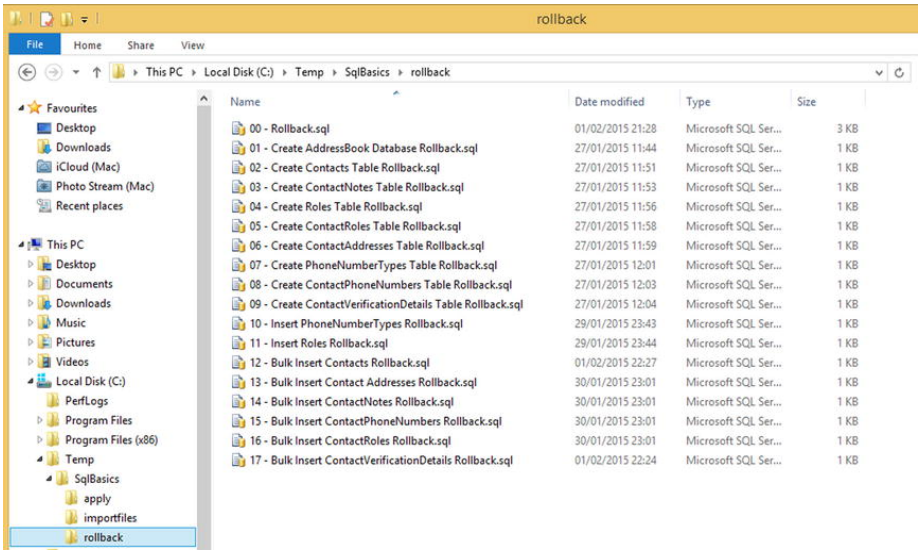

Figure 9-13. *The complete set of rollback scripts so far*

The final step is to include our new files in the `00 - Rollback.sql` SQLCMD script. Add these lines to the top of the script, underneath the `:setvar path` line.

```
:setvar currentFile "17 - Bulk Insert ContactVerificationDetails Rollback.sql"
PRINT 'Executing $(path)$(currentFile)';
:r $(path)$(currentFile)

:setvar currentFile "16 - Bulk Insert ContactRoles Rollback.sql"
PRINT 'Executing $(path)$(currentFile)';
:r $(path)$(currentFile)

:setvar currentFile "15 - Bulk Insert ContactPhoneNumbers Rollback.sql"
PRINT 'Executing $(path)$(currentFile)';
:r $(path)$(currentFile)

:setvar currentFile "14 - Bulk Insert ContactNotes Rollback.sql"
PRINT 'Executing $(path)$(currentFile)';
:r $(path)$(currentFile)

:setvar currentFile "13 - Bulk Insert Contact Addresses Rollback.sql"
PRINT 'Executing $(path)$(currentFile)';
:r $(path)$(currentFile)

:setvar currentFile "12 - Bulk Insert Contacts Rollback.sql"
PRINT 'Executing $(path)$(currentFile)';
:r $(path)$(currentFile)
```

Run the rollback and ensure everything completes successfully (make sure you close all other windows first). If things don't complete successfully, SQL Server has kept something open in the background—just close and reopen SSMS, then run the scripts again. Once you've executed without any issues, take a look at some of the output generated. You can see some of this in Figure 9-14.

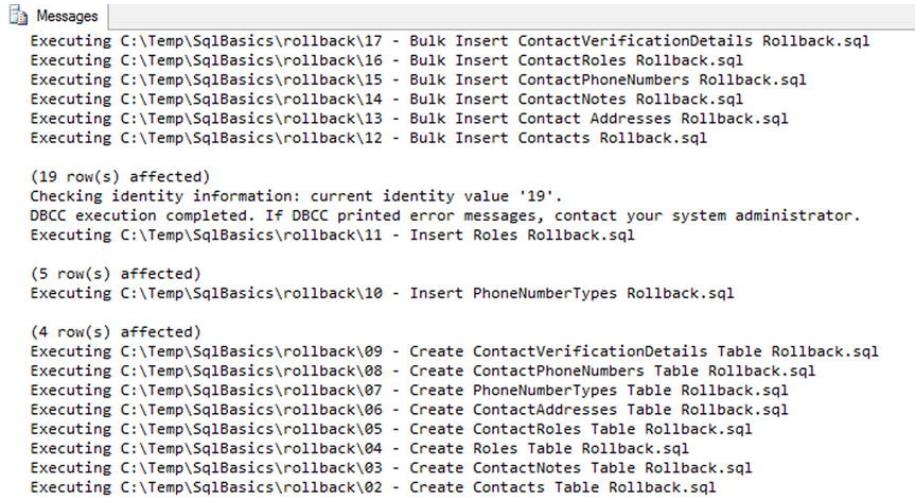

Figure 9-14. *Output of the rollback script*

All scripts should execute as normal, but look at the lines that show `row(s) affected`. This tells you how many rows were deleted from the appropriate tables. There is also a message telling you that the `DBCC` command has executed, and what the current value was before the `DBCC CHECKIDENT` command ran.

After rollback has completed, run the `00 - Apply.sql` script to recreate the database. Make sure the scripts have completed successfully. Once completed, you'll see we have a **Results** tab displayed, instead of the **Messages** tab. Figure 9-15 shows the contents of the **Results** tab.

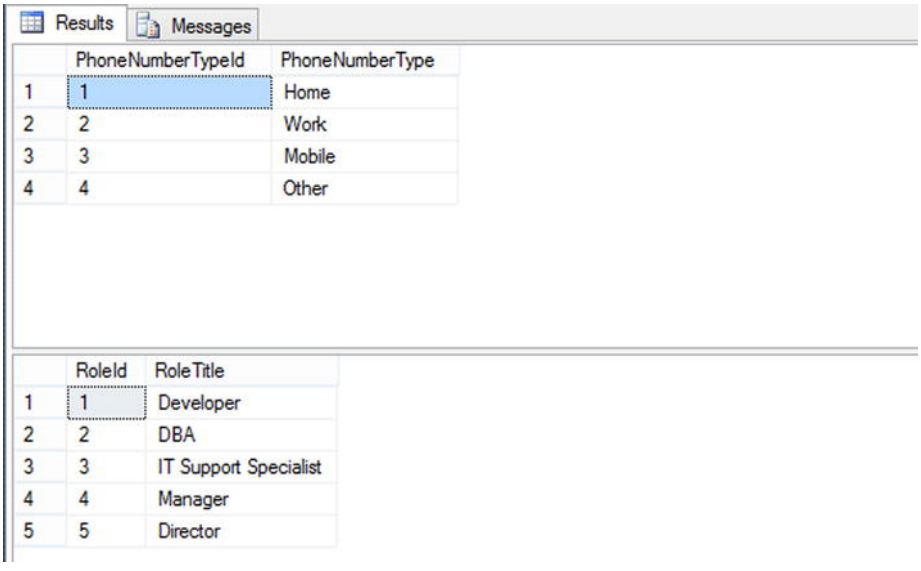

Figure 9-15. *Results output of the apply script*

This is because we left `SELECT` statements in at the end of the `PhoneNumberTypes` and Roles record creation scripts—the **Results** tab shows us the results of those `SELECT` statements. Click the **Messages** tab to see the output generated by the scripts (shown in Figure 9-16):

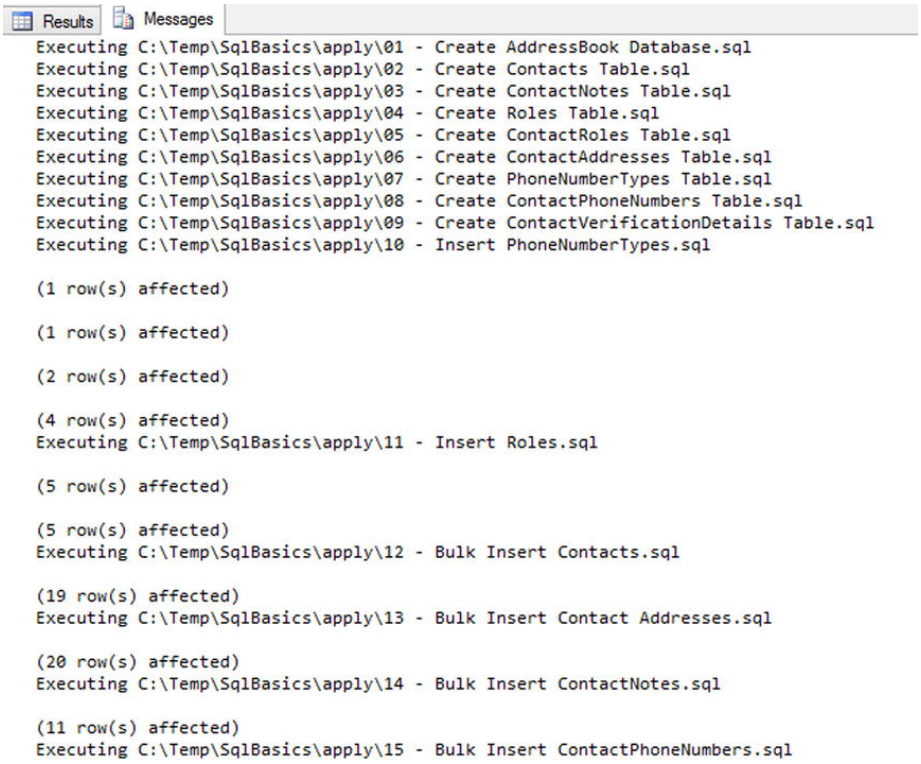

Figure 9-16. *Messages output of the apply script*

Again, we can see lots of `row(s) affected` messages, telling us how many rows were added to each table by the appropriate script. Not bad; now we have a database with some data in it!

Summary

That was an epic journey, much like Bilbo's trek to Dale in *The Hobbit*. Of course, Bilbo's adventures don't end when he reaches Dale, and our adventures in data import haven't finished with this chapter, either.

Do you remember Roald Dahl? Not in the sense that he was an author who wrote some fantastic books, but rather in the sense that we had to take his record out of our `Contacts` import file. Our last data task is to create an import script to bring Roald Dahl to life in our database, and we'll look at doing this in the very next chapter. Keep walking!

PREV
Chapter 8 : DML (or Inserts, Upd…
NEXT
Chapter 10 : Creating Data Impo…